SAPIENZA
UNIVERSITÀ DI ROMA

"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

7 gennaio 2024

Indice

Informazioni e Contatti

Prerequisiti consigliati:

- Corso di *Progettazione di Algoritmi*.

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).

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 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)**.** $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{x}, \texttt{y}, \texttt{z}\}$ 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; inoltre

- data una stringa w su Σ, $|w|$ 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 $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{x}, \texttt{y}, \texttt{z}\}$ un alfabeto; allora una sua possibile stringa è $w = \texttt{x1y0z}$.

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 $x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_n$ 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 $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ un alfabeto; allora la stringa $x = \texttt{ab}$ è prefisso della stringa $y = \texttt{abc}$, poiché esiste una stringa $z = \texttt{c}$ tale per cui $xz = \texttt{abc} = y$.

1.1.2 Linguaggi

Definizione 1.1.2.1: Linguaggio

Sia Σ un alfabeto; si definisce **linguaggio** un insieme di stringhe di Σ; dunque, un linguaggio L su Σ è un elemento $L \in \mathcal{P}(\Sigma)$.

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:

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

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 $x = \texttt{1011101}$ ed $y = \texttt{1001001}$ due stringhe sull'alfabeto $\Sigma = \{\texttt{0}, \texttt{1}\}$; poiché differiscono per 2 caratteri, si ha che $d_H(x, y) = 2$.

Definizione 1.1.3.2: Peso di Hamming

Sia $\Sigma = \{0, \dots, 9\}$ 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 0. In simboli, data una stringa $x = x_1 \cdots x_n$, con $n \in \mathbb{N}$, si ha che

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

Osservazione 1.1.3.1: Peso di Hamming di stringhe binarie

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

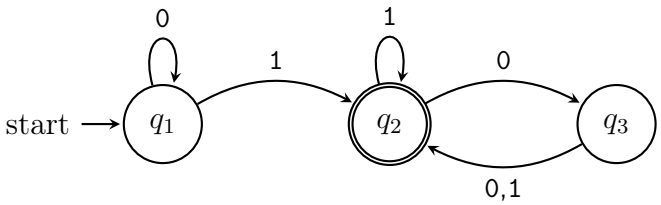


Figura 1.1: Un DFA.

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

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

- 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, indicato con $L(M)$, contenente tutte le stringhe accettate da M; simmetricamente, si dice che M **riconosce** $L(M)$.

Esempio 1.2.1.2 (Linguaggi di DFA)**.** Si consideri il seguente DFA M_1:

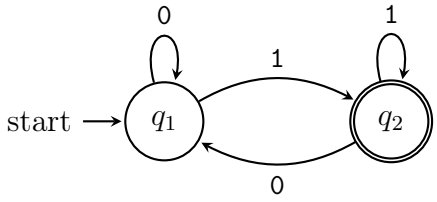


Figura 1.2: Un DFA M_1.

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 definito come segue:

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

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:

$$L := \{w \mid w = 0^n 1, n \in \mathbb{N} - \{0\}\}$$

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

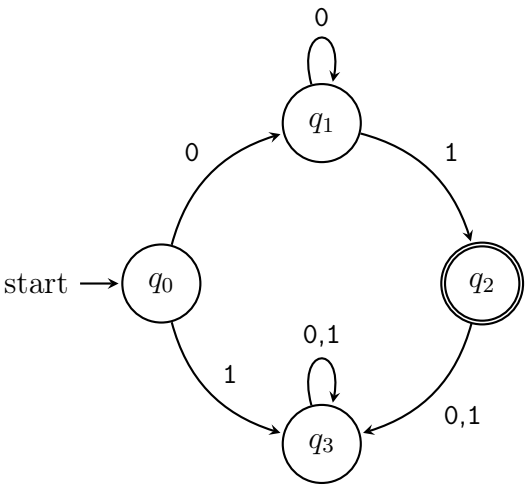


Figura 1.3: Un DFA che riconosce L.

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 della computazione; infatti, ogni volta che si presenta una scelta, la computazione si *ramifica*, ed ognuno degli automi nei vari rami computa tali 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 corrente in una data ramificazione di computazione, tale ramo 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.

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

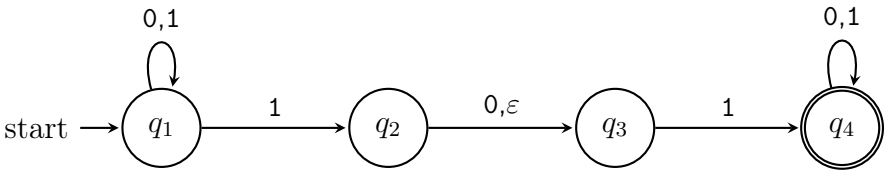


Figura 1.4: L'NFA N.

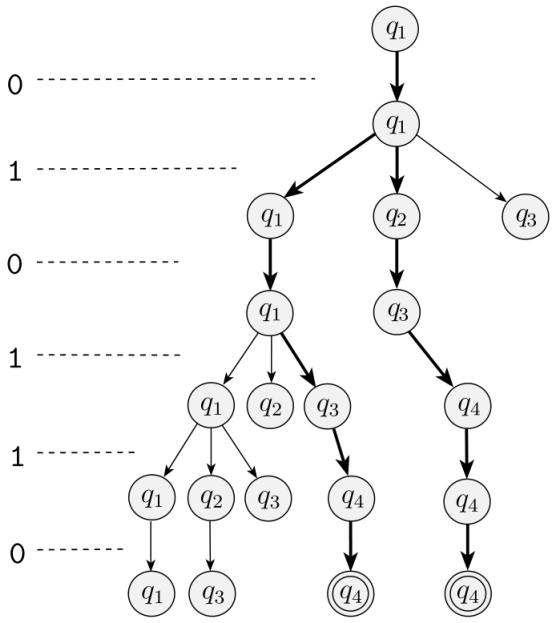
esso può essere descritto secondo la quintupla $N = (Q, \Sigma, \delta, q_0, F)$ come segue:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- 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, dunque *senza produrre un passo ulteriore*, producendo inoltre un *nuovo ramo di computazione*.

Osservazione 1.3.1.1: Determinismo e non determinismo

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.

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

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

Dimostrazione.

Prima implicazione. Per l'Osservazione 1.3.1.1 si ha che

$$\mathsf{DFA} \subseteq \mathsf{NFA} \implies \mathcal{L}(\mathsf{DFA}) \subseteq \mathcal{L}(\mathsf{NFA})$$

Seconda 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 $E(R)$ l'insieme degli stati raggiungibili da stati in R, attraverso 0 o più ε-archi. 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;

- $F' := \{R \in Q' \mid R \cap F \neq \varnothing\}$, 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.

□

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 $\Sigma = \{\texttt{a}, \ldots, \texttt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\texttt{uno}, \texttt{due}\}$ e $B = \{\texttt{tre}, \texttt{quattro}\}$ due linguaggi su Σ. Allora, si ha che

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

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. In simboli

$$\forall A, B \in \mathsf{REG} \quad A \cup B \in \mathsf{REG}$$

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 accettante, poiché altrimenti non si accetterebbero delle stringhe accettate o da M_1, o ed M_2.

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 definito come segue:

- $Q := \{q_0\} \cup Q_1 \cup Q_2$, dove q_0 è un nuovo stato — dunque Q è scelto tale da includere gli stati sia di N_1 che di 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.

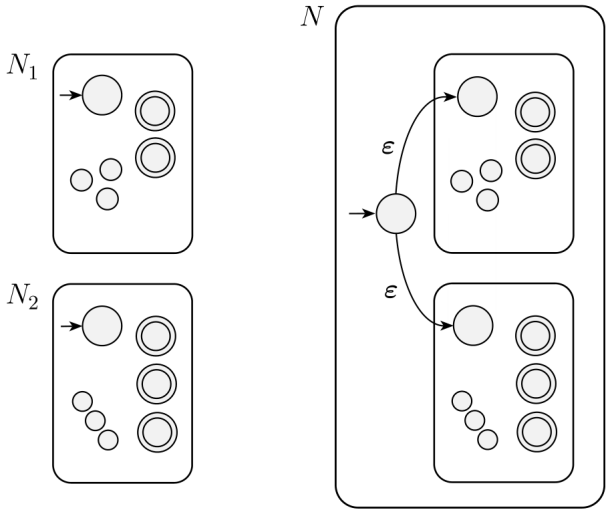


Figura 1.5: Rappresentazione dell'NFA N descritto.

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 il Teorema 1.3.2.1. □

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 $\Sigma = \{\texttt{a}, \ldots, \texttt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\texttt{uno}, \texttt{due}\}$ e $B = \{\texttt{uno}, \texttt{tre}\}$ due linguaggi su Σ. Allora, si ha che

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

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. In simboli

$$\forall A, B \in \mathsf{REG} \quad A \cap B \in \mathsf{REG}$$

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:

- $Q := Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \land r_2 \in Q_2\}$, 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;
- $F := F_1 \times F_2 = \{(r_1, r_2) \mid r_1 \in F_1 \land r_2 \in F_2\}$, 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:

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

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 $\Sigma = \{\texttt{a}, \ldots, \texttt{z}\}$ l'alfabeto composto da 26 lettere, e siano $A = \{\texttt{uno}, \texttt{due}\}$ e $B = \{\texttt{tre}, \texttt{quattro}\}$ due linguaggi su Σ. Allora, si ha che

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

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. In simboli

$$\forall A, B \in \mathsf{REG} \quad A \circ B \in \mathsf{REG}$$

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 gli stati di 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;
- $\forall q \in Q, a \in \Sigma_\varepsilon \quad \delta(q,a) := \begin{cases} \delta_1(q,a) & q \in Q_1 - F_1 \lor (q \in F_1 \land a \neq \varepsilon) \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \land a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$,

  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.

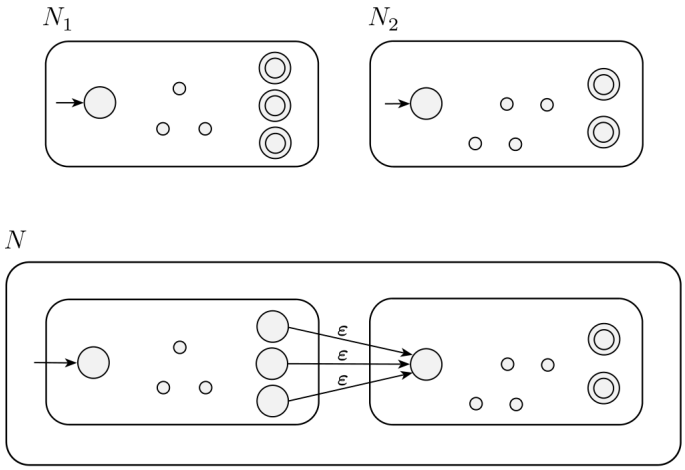


Figura 1.6: Rappresentazione dell'NFA N descritto.

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 il Teorema 1.3.2.1. □

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:

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

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. In simboli

$$\forall A \in \mathsf{REG}, n \in \mathbb{N} \quad A^n \in \mathsf{REG}$$

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 sia 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.

□

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:

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

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

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

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

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. In simboli

$$\forall A \in \mathsf{REG} \quad A^* \in \mathsf{REG}$$

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.

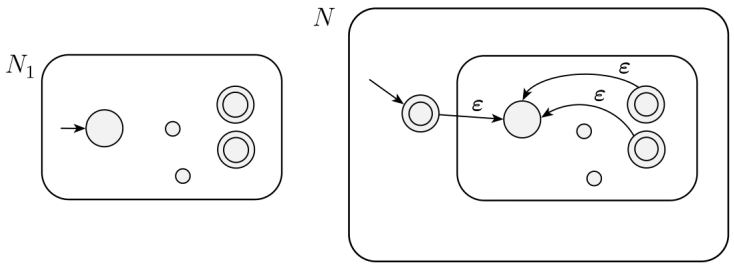


Figura 1.7: Rappresentazione dell'NFA N descritto.

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 il Teorema 1.3.2.1. □

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:

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

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

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

Proposizione 1.4.6.1: Chiusura sul complemento (REG)

Sia A un linguaggio regolare su un alfabeto Σ; allora $\overline{A}$ è regolare. In simboli

$$\forall A \in \mathsf{REG} \quad \overline{A} \in \mathsf{REG}$$

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 $\overline{A}$; allora, $\overline{A}$ 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) $\overline{A \cup B} = \overline{A} \cap \overline{B}$

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

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 $\text{re}(\Sigma)$. 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:

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

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

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

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 ad $\mathtt{a}$ 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 lunghez-

za 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 rappresenta tutte 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

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

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

Spesso si usa la notazione $R^+ := RR^*$, che rappresenta dunque tutte le stringhe che si ottengono attraverso la concatenazione di almeno una stringa 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:

- $L(\mathtt{0}^*\mathtt{10}^*) = \{w \mid w \text{ contiene un solo } \mathtt{1}\}$;
- $L(\Sigma^*\mathtt{001}\Sigma^*) = \{w \mid w \text{ contiene la stringa } \mathtt{001} \text{ come sottostringa}\}$;
- $L(\mathtt{0}\Sigma^*\mathtt{0} \cup \mathtt{1}\Sigma^*\mathtt{1} \cup \mathtt{0} \cup \mathtt{1}) = \{w \mid w \text{ inizia e termina con lo stesso carattere}\}$, 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:

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

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:

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

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; due configurazioni di M si dicono essere **in relazione** se e solo se dall'una è possibile passare all'altra. In simboli:

$$\forall (p, x), (q, y) \in Q \times \Sigma^* \quad (p, x) \vdash_M (q, y) \iff \exists a \in \Sigma \mid x = ay \wedge \delta(p, a) = q$$

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:

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

1.7 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 termine della computazione.

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 né con altri stati, né con sé stesso;
- ogni altro stato collegato con ogni altro stato (a meno di quello iniziale), anche con loro 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 $\Sigma = \{\texttt{a}, \texttt{b}\}$:

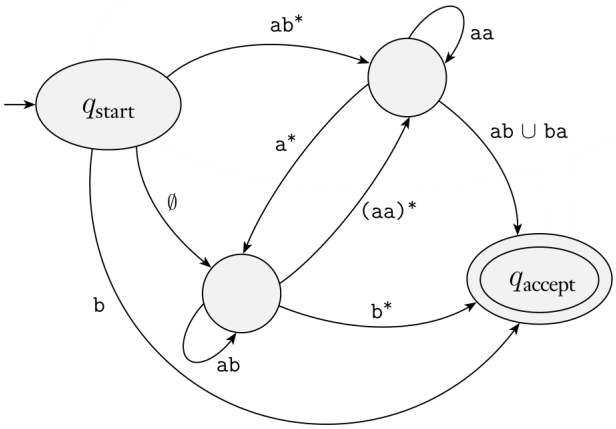


Figura 1.8: Un GNFA.

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

- $q_0 = q_{\text{start}}$
- $\forall i \in [0, n-1] \quad w_{i+1} \in L(\delta(q_i, q_{i+1}))$, 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 $M = (Q, \Sigma, \delta, q_0, F)$ un DFA; allora per costruire un GNFA ad esso equivalente, è sufficiente:

- aggiungere un nuovo stato iniziale q_{start}, con un ε-arco entrante sul vecchio stato iniziale;
- aggiungere un nuovo stato accettante q_{accept}, con ε-archi entranti provenienti dai vecchi stati accettanti; rendere i vecchi stati accettanti come *non* 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 fra loro, *omettendo* gli archi
  - $(q_{\text{accept}}, q_{\text{accept}})$

- $\forall q \in Q \cup \{q_{\text{start}}, q_{\text{accept}}\} \quad (q, q_{\text{start}})$

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:

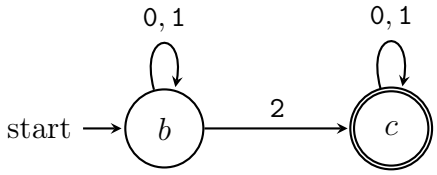


Figura 1.9: Il DFA D.

Allora, il suo GNFA equivalente è il seguente:

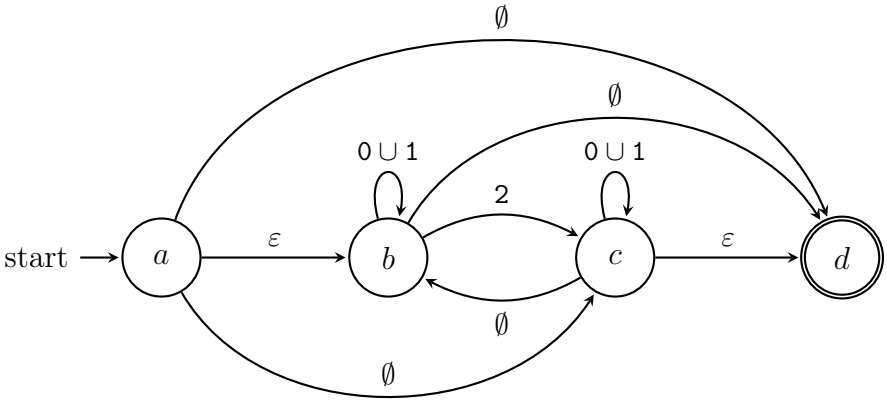


Figura 1.10: Il GNFA di D.

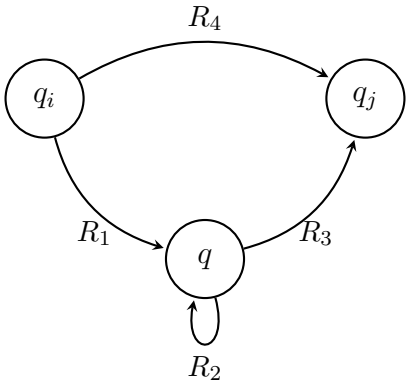
Algoritmo 1.7.2.1: Espressione regolare di un GNFA

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

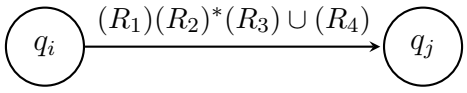
```
1:  function convertGNFAtoRegEx(G)
2:      if |Q| == 2 then
3:          return δ(q_start, q_accept)
4:      else if |Q| > 2 then
5:          q ∈ Q − {q_start, q_accept}
6:          Q' := Q − {q}
7:          for q_i ∈ Q' − {q_accept} do
8:              for q_j ∈ Q' − {q_start} do
9:                  δ'(q_i, q_j) := δ(q_i, q)δ(q, q)*δ(q, q_j) ∪ δ(q_i, q_j)
10:             end for
11:         end for
12:         G' := (Q', Σ, δ', q_start, q_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 `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 `convertGNFAtoRegEx`(G) è un'espressione regolare equivalente a G.

Passo induttivo. È necessario dimostrare che, se G è un GNFA con k stati, allora `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':

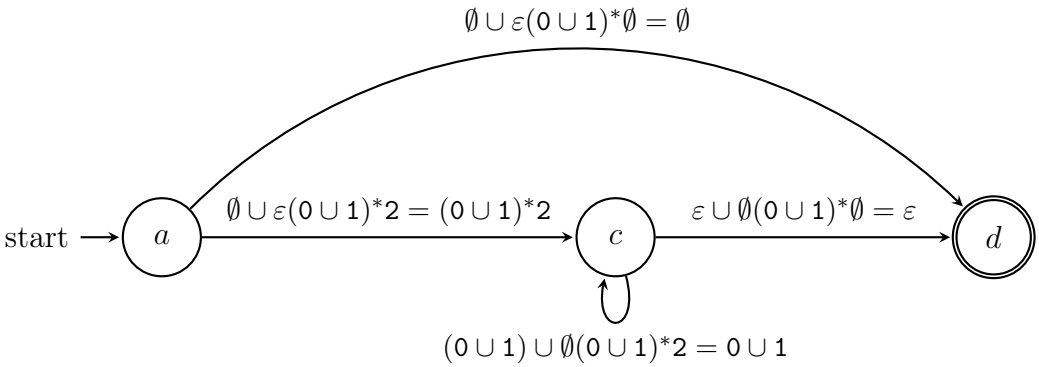


Figura 1.11: G', ovvero il GNFA G, dopo aver rimosso b.

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

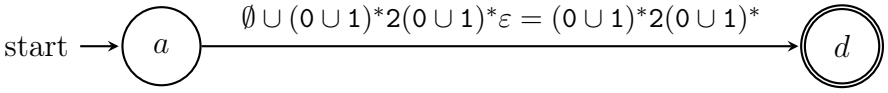


Figura 1.12: G'', ovvero il GNFA G', dopo aver rimosso c.

- 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

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

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 $L(R)$:

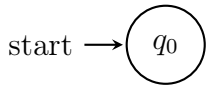


Figura 1.13: Un NFA in grado di riconoscere $L(\emptyset)$.

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 $L(R)$:

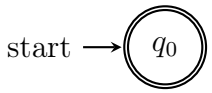


Figura 1.14: Un NFA in grado di riconoscere $L(\varepsilon)$.

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 $L(R)$:

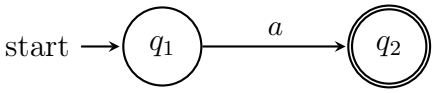


Figura 1.15: Un NFA in grado di riconoscere $L(a)$.

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

- 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 Teorema 1.3.2.1, A è regolare.

□

Corollario 1.7.2.1: Equivalenze tra classi di automi

Si verifica che

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

Dimostrazione. Dal Teorema 1.3.2.1, dal Teorema 1.7.2.1, dall'Algoritmo 1.7.2.1, e dal Metodo 1.7.2.1, si ha che

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

e dunque segue la tesi. □

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 $f : A \to B$.

Questo problema è noto in letteratura "principio della piccionatia" per via della seguente analogia: avendo una piccionaia con m caselle, non è possibile inserire più di m piccioni al suo interno; infatti, 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:

- $\forall i \geq 0 \quad xy^i z \in A$
- $|y| > 0$ (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 $r_1 := q_1$ e siano

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

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

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

in quanto devono essere attraversati n archi, e dunque $n+1$ stati. Inoltre, si noti che

$$n \geq p \implies n+1 \geq p+1 \geq p$$

e dunque per il Principio 1.8.1.1 due dei primi $p+1$ stati della sequenza devono necessariamente essere lo stesso stato (poiché $p := |Q|$); siano r_j il primo ed r_l il secondo, con $j < l$ ed $r_j = r_l$. 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 ad $r_l = r_j$ — dunque y porta M da r_j ad r_j stesso — ed infine z porta M da r_l ad r_{n+1}, e poiché $r_{n+1} \in F$, M accetta sicuramente xy^iz per ogni $i \geq 0$; questo dimostra la prima condizione della tesi;
- $j \neq l \implies |y| > 0$, segue la seconda condizione della tesi;
- $l \leq p+1 \iff l-1 \leq p$, e poiché $|xy| = l-1$, segue la terza condizione della tesi.

Allora, sono soddisfatte tutte le condizioni della tesi. □

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

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

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.

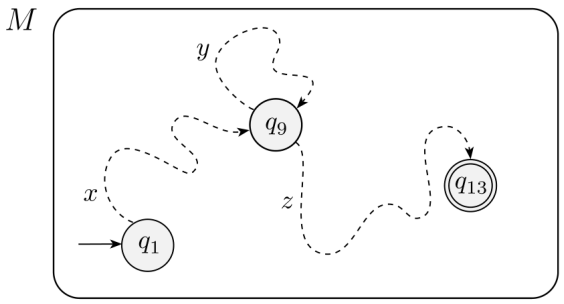


Figura 1.16: L'automa M descritto

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

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

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

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

avente lunghezza sicuramente maggiore di p. Siano inoltre x, y, z tali da soddisfare il pumping lemma, ed in particolare $|xy| \leq p$, 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 `0`, ed inoltre $|y| > 0$ implica che y ha almeno uno `0`. Siano allora

- $k := |x| \implies x = \mathtt{0}^k$, poiché x è composta solo da `0`
- $m := |y| > 0 \implies y = \mathtt{0}^m$, poiché y è composta solo da `0` (ed almeno uno)
- $|xy| \leq p \implies k + m \leq p \implies z = \mathtt{0}^{p-m-k}\mathtt{1}^p$ per gli `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 `1` di questa stringa è pari a p, mentre il numero di `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 $\mathtt{0}^n\mathtt{1}^n$ ↯.

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

$$\alpha A \beta \quad \rightarrow \quad \gamma \beta$$

Definizione 2.1.1.2: Acontestualità

Si definisce **acontestualità** la caratteristica per la quale 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

$$X \quad \to \quad Y$$

dove $X \in V$, $Y \in (V \cup \Sigma_\epsilon)^*$ 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

$$X \quad \to \quad Y \mid Z$$

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 come segue

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

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

$$\begin{aligned} A &\to \texttt{0}A\texttt{1} \\ A &\to B \\ B &\to \texttt{\#} \end{aligned}$$

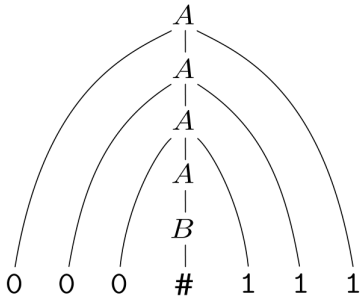
In essa, si hanno:

$$\begin{aligned} V &:= \{A, B\} \\ \Sigma &:= \{\texttt{0}, \texttt{1}, \texttt{\#}\} \\ S &:= A \in V \end{aligned}$$

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

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

Tale catena di produzioni può essere espressa anche attraverso il seguente *albero di derivazione*:



Definizione 2.1.1.4: Linguaggio di una grammatica

Data una grammatica G, il suo **linguaggio** è l'insieme delle stringhe che tale grammatica 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, il linguaggio di tale grammatica 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:

$$\begin{array}{rcl} S_1 & \to & X\mathtt{1}X\mathtt{1}X \\ X & \to & \varepsilon \mid \mathtt{0}X \mid \mathtt{1}X \end{array}$$

- il linguaggio $L_2 := \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w = w^{\mathcal{R}} \wedge |w| \equiv 0 \pmod 2\}$ 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:

$$\begin{array}{rcl} S & \to & \mathtt{a}S\mathtt{c} \mid X \\ X & \to & \mathtt{b}S\mathtt{c} \mid \varepsilon \end{array}$$

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, dove $q_i \in Q$;
- porre $S := V_0$;
- introdurre una regola $V_i \to aV_j$ se vale $\delta(q_i, a) = q_j$, dove $q_i, q_j \in Q$;
- aggiungere la regola $V_i \to \varepsilon$ se $q_i \in F$.

Corollario 2.1.1.1: Linguaggi regolari e context-free

Si verifica che

$$\mathsf{REG} \subsetneq \mathsf{CFL}$$

Dimostrazione. Sia $L \in \mathsf{REG}$, e dunque per definizione esiste un DFA M tale che $L(M) = L$; dunque, attraverso il Metodo 2.1.1.1 è possibile convertire M in una CFG G, tale che $L(G) = L(M) = L$; allora, segue la tesi. □

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.

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**.

Equivalentemente, una stringa è detta essere generata **ambiguamente** se per essa esistono molteplici alberi di derivazione a sinistra.

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

Esempio 2.1.2.1 (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 seguente 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 stessa 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

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

dove $A, B, C \in V$, $a \in \Sigma$ ed S deve essere la variabile alla sinistra della prima regola di G. Si noti che la regola $S \to \varepsilon \in R$ 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: CFG in CNF

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 $A \to u_1 \dots u_k$ con $k \geq 3$ — dove $u_1, \dots, u_k \in (V \cup \Sigma)^*$ — 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 $A_1, \dots, A_{k-2} \in V$, 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 $U_i \to u_i \in R$.

Corollario 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. □

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 o meno 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 *in una sola operazione* — 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:

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

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:

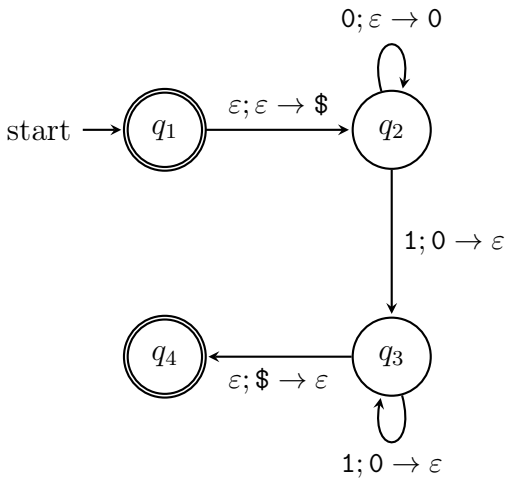


Figura 2.1: Il PDA P.

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
- $\forall i \in [0, n-1] \quad \exists a, b \in \Gamma_\varepsilon, t \in \Gamma^* \mid (r_{i+1}, b) \in \delta(r_i, w_{i+1}, a) \wedge \begin{cases} s_i = at \\ s_{i+1} = bt \end{cases}$, 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 raggiunta 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), implicando necessariamente che l'input è terminato e che lo stack è vuoto.

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 `$` all'interno dello stack;
- fintanto che viene letto `0`, viene inserito `0` nello stack;
- non appena viene letto `1`, e fintanto che viene letto `1`, viene rimosso `0` dallo stack;
- se all'interno dello stack è presente `$`, 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

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

il quale non è regolare per l'Esempio 1.8.1.2.

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

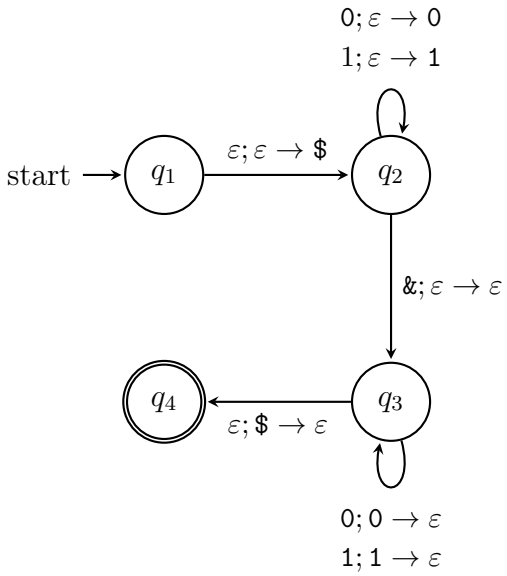


Figura 2.2: Un PDA.

Esso è in grado di riconoscere il linguaggio

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

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 posti *al contrario* 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 $u = u_1 \cdots u_k \in \Gamma^*$ all'interno dello stack di P, alla lettura di un certo carattere $a \in \Sigma_\varepsilon$ della stringa di input e di un certo carattere $b \in \Gamma$ sulla cima dello *stack* di P, devono esistere stati $r_1, \ldots, r_{k-1} \in Q$ tali per cui

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

In simboli, per indicare l'inserimento di tale stringa all'interno dello stack di P verrà utilizzato il seguente simbolismo

$$a; b \to u \equiv a; b \to u_1 \cdots u_k$$

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

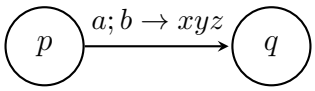


Figura 2.3: Un PDA che inserisce una stringa nello stack.

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

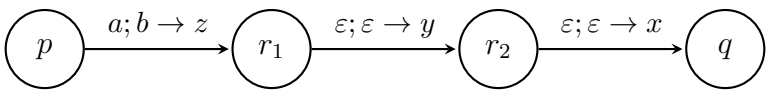


Figura 2.4: La formalizzazione del PDA precedente.

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 *non deterministicamente* una delle regole di G in grado di sostituire A;
- $\forall a \in \Sigma \quad \delta(q', a, a) := \{(q', \varepsilon)\}$ per rimuovere i terminali dallo stack, evitando che possano essere ulteriormente rimpiazzati (si noti che se un simbolo a è nella stringa di input, allora è sicuramente un terminale di G);
- $\delta(q', \varepsilon, \$) := \{(q, \varepsilon)\}$ per sfruttare il marcatore $.

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

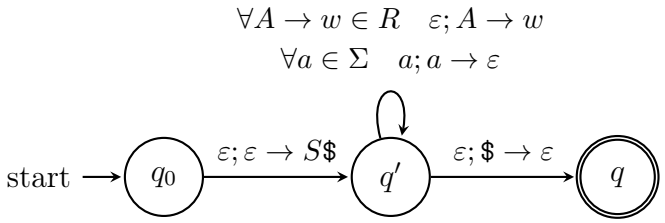


Figura 2.5: Il PDA P appena costruito.

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

$$\begin{aligned} G: \quad S &\to \mathtt{a}T\mathtt{b} \mid \mathtt{b} \\ T &\to T\mathtt{a} \mid \varepsilon \end{aligned}$$

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

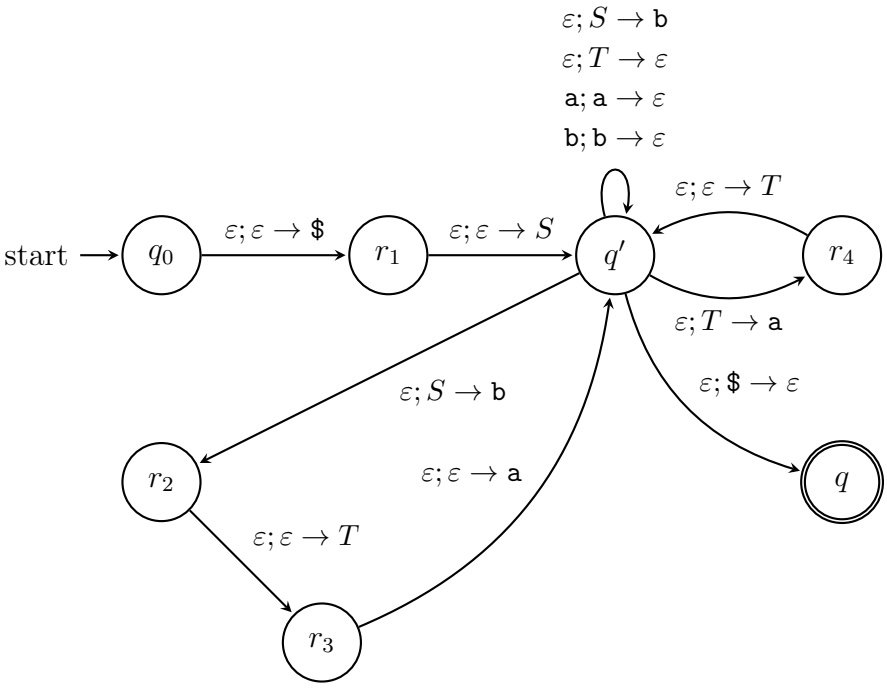


Figura 2.6: Un PDA in grado di riconoscere $L(G)$.

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

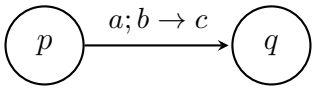
$$P' := (Q', \Sigma, \Gamma, \delta', q_0, F')$$

tale che:

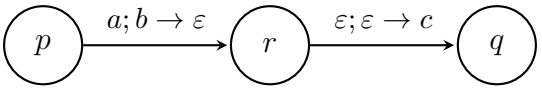
- ogni transizione di stati possa effettuare solamente un'operazione di *push* o *pop*, *non simultaneamente*, dunque avente δ' definita come segue:

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

  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:

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

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 $A_{p,q} \to u_1 \dots u_h$ dove $u_1, \dots u_h, \in \Sigma$, 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, mantenendo 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 $A_{p,q} \Rightarrow aA_{r,s}b \overset{*}{\Rightarrow} w$ per certi stati $r, s \in Q'$ e terminali $a, b \in \Sigma_\varepsilon$, oppure $A_{p,q} \Rightarrow A_{p,r}A_{r,q} \overset{*}{\Rightarrow} w$ per un qualche stato $r \in Q'$, e dunque:

- nel primo caso, sia y la sottostringa di w generata da $A_{r,s}$, dunque $A_{r,s} \overset{*}{\Rightarrow} y$, quindi $w = ayb$ 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 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 è mai stato vuoto durante l'intera computazione — se non all'inizio ed alla fine — oppure esistono alcuni passaggi — diversi dal primo e dall'ultimo — 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 $(q, \varepsilon) \in \delta'(s, b, u)$, e sia y la stringa che permette di portare P' dallo stato r allo stato s; inoltre, poiché
  - la computazione da p a q deve lasciare invariato lo stack
  - $u \in \Gamma$ viene posto all'interno dello stack all'inizio della computazione
  - $u \in \Gamma$ viene rimosso dallo stack alla fine della computazione

  segue necessariamente che y deve lasciare lo stack invariato anche'essa; questo dimostra che y 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, poiché $A_{p,q} \to aA_{r,s}b \in R$ per costruzione di G, si ha che

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

- differentemente, nel secondo caso si assuma esista uno 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 $y, z \in \Sigma^*$ 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

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

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

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

implicando che $L(G) = L$.

□

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:

- $\forall i \geq 0 \quad uv^i xy^i z \in A$
- $|vy| > 0$ (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. Inoltre, in tale albero si consideri un cammino che parta da S e abbia lunghezza h; allora, per quanto detto, il cammino attraversa k nodi, dove $k \geq |V| + 2$, dei quali solo l'ultimo è un terminale, dunque il cammino contiene $k-1$ variabili, dove

$$k \geq |V| + 2 \implies k - 1 \geq |V| + 1 \geq |V|$$

dunque per il Principio 1.8.1.1 due delle ultime $|V| + 1$ variabili del cammino devono necessariamente essere la stessa variabile; siano A_j la prima ed A_l la seconda, con $j < l$ e $A_j = A_l$, e dunque sicuramente

$$j \geq k - 1 - (|V| + 1) + 1 = k - |V| - 1$$

Si pongano dunque

$$\begin{cases} u, z \mid S \overset{*}{\Rightarrow} uA_j z \\ v, y \mid A_j \overset{*}{\Rightarrow} vA_l y \\ x \mid A_l \overset{*}{\Rightarrow} x \end{cases}$$

Allora, si ha che:

- poiché $A_j = A_l$, in ogni derivazione $A_j \overset{*}{\Rightarrow} vA_ly$ è possibile sostituire A_l con A_j, anche ricorsivamente, infatti

  $$A_j \overset{*}{\Rightarrow} vA_ly = vA_jy \overset{*}{\Rightarrow} vvA_lyy = v^2A_ly^2 = v^2A_jy^2 \overset{*}{\Rightarrow} \dots$$

  di conseguenza

  $$\forall i > 0 \quad S \overset{*}{\Rightarrow} uA_jz \overset{*}{\Rightarrow} uv^iA_ly^iz \overset{*}{\Rightarrow} uv^ixy^iz$$

  ed inoltre per $i = 0$ si ha che $uv^ixy^iz = uxy$ che è derivabile come segue

  $$S \overset{*}{\Rightarrow} uA_jz = uA_lz \overset{*}{\Rightarrow} uxz$$

  questo dimostra la prima condizione della tesi;

- poiché G è in CNF in ipotesi, la derivazione $A_j \overset{*}{\Rightarrow} vA_ly$ deve aver necessariamente utilizzato una regola della forma $A_j \to BC$ dove $B \overset{*}{\Rightarrow} vA_l$ e $C \overset{*}{\Rightarrow} y$ oppure $B \overset{*}{\Rightarrow} v$ e $C \overset{*}{\Rightarrow} A_ly$; allora, poiché la CNF non ammette ε-regole, segue che $v \neq \varepsilon$ oppure $y \neq \varepsilon$; questo dimostra la seconda condizione della tesi;

- poiché A_j si trova tra le ultime $|V|+1$ variabili del cammino considerato, chiamando h' il suo sottoalbero, esso deve avere altezza al massimo $|V| + 1$; allora, poiché

  $$h' \leq |V| + 1 \iff h' - 1 \leq |V|$$

  per la Proposizione 2.3.1.1 segue che

  $$|vxy| \leq 2^{h'-1} \leq 2^{|V|} =: p$$

  dimostrando dunque la terza condizione della tesi.

□

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

$$L := \{0^n1^n2^n \mid n \in \mathbb{N}\}$$

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

$$w := 0^p1^p2^p \implies |w| = 3p > p$$

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 $w := 0^p1^p2^p = uvxyz$, può verificarsi solo uno dei seguenti:

- vxy contiene solamente 0;
- vxy contiene solamente 1;
- vxy contiene solamente 2;
- vxy contiene soltanto 0 e 1;

- vxy contiene soltanto $\mathtt{1}$ e $\mathtt{2}$;

Allora, prendendo in esame ad esempio il primo caso — poiché è possibile ripetere il ragionamento analogo per ogni altro — si ha che

$$vxy = \mathtt{0}^{|vxy|} \implies w = uvxyz = \mathtt{0}^{|u|}\mathtt{0}^{|vxy|}\mathtt{0}^{p-|u|-|vxy|}\mathtt{1}^p\mathtt{2}^p$$

e poiché per il pumping lemma $\forall i \geq 0 \quad uv^ixy^iz \in L$, ad esempio per $i = 0$, si ha che

$$uv^ixy^iz = uv^0xy^0z = uxz = \mathtt{0}^{|u|}\mathtt{0}^{|x|}\mathtt{0}^{p-|u|-|vxy|}\mathtt{1}^p\mathtt{2}^p$$

e dunque

$$uxz \in L \iff |u| + |x| + p - |u| - |vxy| = p \iff -|v| - |y| = 0 \iff |vy| = 0$$

ma per le condizioni del pumping lemma $|vy| > 0$ ↯.

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 $|vxy| \leq p$, ma poiché $w := \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p = uvxyz$, si può verificarsi 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 $uv^1xy^1z \in L$ ↯.

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 la loro unione è un CFL. In simboli

$$\forall L_1, \ldots, L_n \in \mathsf{CFL} \quad \bigcup_{i=1}^{n} L_i \in \mathsf{CFL}$$

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, e siano queste 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

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

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:

$$G_1 : \quad S_1 \to \mathtt{0}S_1\mathtt{1} \mid \varepsilon$$

$$G_2 : \quad S_2 \to \mathtt{1}S_2\mathtt{0} \mid \varepsilon$$

Allora, la seguente CFG G

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

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

2.4.2 Concatenazione

Proposizione 2.4.2.1: Chiusura sulla concatenazione (CFL)

Siano $L_1, \ldots, L_n$ dei CFL; allora la loro concatenazione è un CFL. In simboli

$$\forall L_1, \ldots, L_n \in \mathsf{CFL} \quad \bigcirc_{i=1}^{n} L_i \in \mathsf{CFL}$$

Dimostrazione. Omessa. □

2.4.3 Star

Proposizione 2.4.3.1: Chiusura sull'operazione star (CFL)

Sia L un CFL; allora L^* è un CFL. In simboli

$$\forall L \in \mathsf{CFL} \quad L^* \in \mathsf{CFL}$$

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. In simboli

$$\exists L_1, L_2 \in \mathsf{CFL} \mid L_1 \cap L_2 \notin \mathsf{CFL}$$

Dimostrazione. Siano G_1 e G_2 le due seguenti CFG:

$$\begin{array}{lrcl} G_1: & S & \to & TU \\ & T & \to & \mathtt{0}T\mathtt{1} \mid \varepsilon \\ & U & \to & \mathtt{2}U \mid \varepsilon \end{array} \qquad \begin{array}{lrcl} G_2: & S & \to & UT \\ & T & \to & \mathtt{1}T\mathtt{2} \mid \varepsilon \\ & U & \to & \mathtt{0}U \mid \varepsilon \end{array}$$

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 $\mathtt{0}T\mathtt{1}$ o terminare — e dunque produce stringhe della forma $\mathtt{0}^n\mathtt{1}^n$ — mentre U è solo in grado di diventare una combinazione di $\mathtt{2}U$ o terminare — e dunque produce stringhe della forma $\mathtt{2}^k$; allora, il suo linguaggio è

$$L(G_1) = \{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^k \mid n, k \in \mathbb{N}\}$$

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

$$L(G_2) = \{\mathtt{0}^k\mathtt{1}^n\mathtt{2}^n \mid n, k \in \mathbb{N}\}$$

Infine, si noti che

$$L(G_1) \cap L(G_2) = \{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \in \mathbb{N}\}$$

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, $\{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \in \mathbb{N}\} \notin \mathsf{CFL}$. Allora, poiché $L(G_1), L(G_2) \in \mathsf{CFL}$ 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 $\overline{L}$ non è necessariamente un CFL. In simboli

$$\exists L \in \mathsf{CFL} \mid \overline{L} \notin \mathsf{CFL}$$

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

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

e si consideri la seguente grammatica G:

$$\begin{array}{rlcl} G: & S & \to & A \mid B \mid AB \mid BA \\ & A & \to & \mathtt{0} \mid \mathtt{0}A\mathtt{0} \mid \mathtt{0}A\mathtt{1} \mid \mathtt{1}A\mathtt{0} \mid \mathtt{1}A\mathtt{1} \\ & B & \to & \mathtt{1} \mid \mathtt{0}B\mathtt{0} \mid \mathtt{0}B\mathtt{1} \mid \mathtt{1}B\mathtt{0} \mid \mathtt{1}B\mathtt{1} \end{array}$$

si noti che le stringhe derivate da A sono tutte le stringhe aventi lunghezza dispari ed uno `0` al centro, mentre le stringhe derivate da B sono tutte le stringhe aventi lunghezza dispari ed un `1` al centro.

Dunque, si ha che:

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

  $$\forall i \in [1,n] \quad x_i = x_{\frac{n}{2}+i} \iff \exists w \in \{\mathtt{0},\mathtt{1}\}^* \mid x = ww \iff x \in L$$

  siano inoltre

  $$u := x_1 \cdots x_{2i-1} \implies |u| = 2i - 1 - 1 + 1 = 2i - 1$$
  $$v := x_{2i} \cdots x_n \implies |v| = n - 2i + 1$$

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

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

allora, poiché $x_i \neq x_{\frac{n}{2}+i}$, 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 u e v possono essere derivate da A e B, ed in G sono presenti le regole $S \to AB \mid BA$; questo dimostra che $\overline{L} \subseteq L(G)$;

- presa una stringa x tale che $S \overset{*}{\Rightarrow} x$, e ponendo $n := |x|$, se n è dispari allora sicuramente $x \notin L \iff x \in \overline{L}$; 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 $S \to AB$, si ha che

$$S \Rightarrow AB \overset{*}{\Rightarrow} x \iff \exists u, v \in \{\mathtt{0}, \mathtt{1}\}^* \mid \begin{cases} x = uv \\ A \overset{*}{\Rightarrow} u \\ B \overset{*}{\Rightarrow} v \\ |u| \equiv |v| \equiv 1 \pmod 2 \end{cases}$$

dunque u e v devono avere lunghezza dispari per costruzione di G; allora, chiamando $l := |u| \implies |v| = n - l$, si ottiene che

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

e 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 $x \notin L \iff x \in \overline{L}$; questo dimostra che $L(G) \subseteq \overline{L}$;

Dunque, poiché $\overline{L} = L(G) \in \mathsf{CFL}$, ed il complemento di $\overline{L}$ è $L \notin \mathsf{CFL}$, segue la tesi. □

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, le cui celle sono sovrascrivibili anch'esse — 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).

Formalmente, una **TM** è una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ 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 $\sqcup \in \Gamma$;
- $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$ è la **funzione di transizione**, che definisce la relazione tra gli stati;
- $q_0 \in Q$ è lo **stato iniziale**;
- $q_{\text{accept}} \in Q$ è lo **stato di accettazione**;
- $q_{\text{reject}} \in Q$ è lo **stato di rifiuto**, tale che $q_{\text{accept}} \neq q_{\text{reject}}$.

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 le TM in questione siano costituite da un nastro limitato a sinistra.

Inizialmente, il nastro di una macchina di Turing contiene solamente una stringa $w = w_1 \cdots w_n \in \Sigma^*$ in input, sulle n celle più a sinistra del nastro, e le celle restanti contengono il simbolo $\sqcup \in \Gamma$ (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:

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

e δ segue dal suo diagramma:

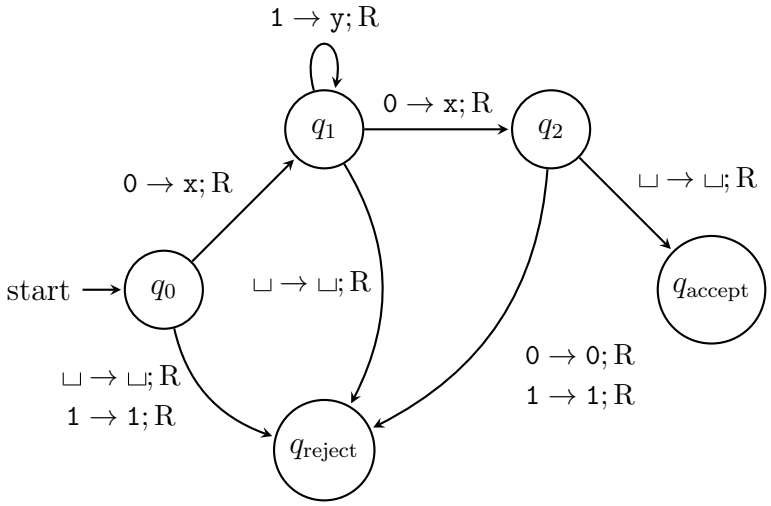


Figura 3.1: La TM M.

La notazione $a \to b; c$ presente sugli archi di questo diagramma sta ad indicare che se 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 $c \in \{\text{L}, \text{R}\}$. 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)

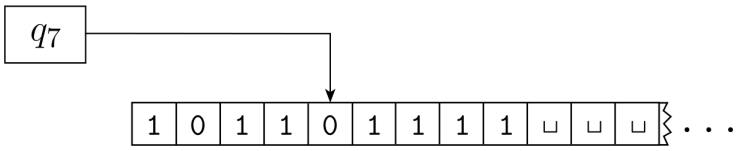
Sia $M = (Q, \Sigma, \Gamma, q_0, q_{\text{accept}}, q_{\text{reject}})$ una TM; con il simbolismo

$$u\ q\ v$$

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 $\sqcup$);
- la testina punta al 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

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

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 del 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 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. In simboli

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

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 $L \in \mathsf{REC}$.

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*.

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 3.1.4.1: Turing-decidibilità

Si noti che $\mathsf{DEC} \subset \mathsf{REC}$, 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:

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

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 $\{\text{L}, \text{R}, \text{S}\}$, 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

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

per un certo $\gamma \in \Gamma$, dove $r \in Q'$ è un 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 $\{\text{L}, \text{R}\}$, e poiché

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

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 di M verrà utilizzata una versione marcata dei caratteri in Γ. Dunque, si ha che:

- inizialmente M' trasforma il nastro nel formato descritto, e dunque viene effettuato 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:

$$\# \dot{w}_1 \, w_2 \cdots w_n \, \# \, \dot{x}_1 \, x_2 \cdots x_m \, \# \, \ldots \, \# \, \dot{y}_1 \, y_2 \cdots y_j \, \# \sqcup \ldots$$

- per simulare una singola mossa di M, la testina di M' scansiona tutto il suo

nastro, aggiornando oppurtunamente i simboli marcati con • per simulare lo spostamento di ogni testina su ogni nastro; successivamente, viene effettua una seconda scansione, per simulare i rimpiazzi dei simboli sulle celle, descritti dalla funzione di transizione 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 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' *a partire 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 $k = 1$ nastri.

□

3.2.3 Macchine di Turing non deterministiche

Definizione 3.2.3.1: NTM

Una **NTM** (*Nondeterministic Turing Machine*) è una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ 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:

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

poiché una NTM computa non deterministicamente.

Osservazione 3.2.3.1: Computazioni di NTM

Si noti che, una NTM *riconoscitore*:

- accetta una stringa se *almeno un ramo di computazione non deterministica accetta*, dunque ogni altro ramo può *sia accettare, sia rifiutare, sia andare in loop*;
- rifiuta una stringa se *ogni ramo di computazione non deterministica o rifiuta o va in loop*.

Differentemente, una NTM *decisore*:

- accetta una stringa se *almeno un ramo di computazione non deterministica accetta*, e ogni altro ramo può *solo accettare o rifiutare*;
- rifiuta una stringa se *ogni ramo di computazione non deterministica rifiuta*.

Proposizione 3.2.3.1: Equivalenza delle NTM

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

Dimostrazione.

Prima implicazione. Sia $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una NTM, e si consideri l'albero di computazione che N produce elaborando gli input, associando un *indirizzo* ad ogni nodo dell'albero, come segue:

- alla radice r dell'albero viene assegnato l'indirizzo ε;
- ad ogni altro nodo v diverso dalla radice, viene assegnato l'indirizzo xa, dove x è l'indirizzo del padre di v, ed a è un numero in $[1, b]$, dove b è il numero di figli del nodo dell'albero avente maggior numero di figli.

Ad esempio, l'indirizzo 132 indica il secondo figlio del terzo figlio del primo figlio della radice dell'albero di computazione di N. Si consideri allora una TM M a 3 nastri, dove

- il primo nastro contiene l'input di N;
- il secondo nastro mantiene una copia del primo nastro, corrispondente a qualche diramazione della sua computazione non deterministica;
- il terzo nastro contiene l'indirizzo del nodo dell'albero di computazione non deterministica corrente.

Dunque, data in input una stringa w, M computa come segue:

- inizialmente, il primo nastro di M contiene w, il secondo è vuoto ed il terzo contiene ε;
- M ripete i seguenti passi:
  - M copia il primo nastro sul secondo;

– M simula la ramificazione della computazione non deterministica di N correntemente sul secondo nastro, controllando prima di eseguire ogni passo quale scelta fare tra quelle consentite dalla funzione di transizione δ; se la simulazione di N accetta la stringa, M accetta;

– se non rimangono più simboli sul terzo nastro, o la simultazione rifiuta la stringa, M sostituisce la stringa sul terzo nastro con il prossimo indirizzo da controllare.

Si noti che la macchina M deve simulare N esplorandone l'albero di computazione in BFS (*Breadth-First Search*), poiché se la visita avvenisse in DFS si rischierebbe di entrare in un ramo di computazione potenzialamente infinito, senza trovare uno dei rami che avrebbe potuto eventualmente accettare l'input. Allora, per la Proposizione 3.2.2.1, segue la tesi.

Seconda implicazione. Sia M una TM; allora, poiché una macchina di Turing è una particolare macchina di Turing non deterministica, segue la tesi.

□

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 — che non prende nessun input, ed è connessa ad una stampante (ad esempio un nastro secondario) sulla quale stampa stringhe del proprio linguaggio in ordine casuale, anche ripetutamente.

Proposizione 3.2.4.1: Equivalenza degli enumeratori

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

Dimostrazione.

Prima implicazione. Sia $E = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ un enumeratore; allora, sia M la seguente TM– avente E come sua sottoprocedura — che, su input w, computa come segue:

- M simula E;
- ogni volta che E stampa una stringa x, M compara x con w
- M accetta se w appare almeno una volta nell'output di E.

Allora, per costruzione M accetta una stringa w se e solo se E la stampa, e poiché E stampa le stringhe del proprio linguaggio per definizione, segue la tesi.

Seconda implicazione. Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una TM; allora, sia E il seguente enumeratore — avente M come sua sottoprocedura — che, dato nulla in input, computa come segue:

- per ogni $i \in \mathbb{N}$, e per ogni $j \in [1, i]$, E simula $M(w_j)$, dove w_j è la j-esima stringa di Σ^*, ordinato secondo il criterio di ordinamento $\prec$ definito all'interno della dimostrazione del Lemma 3.3.2.1;
- E stampa w_j se $M(w_j)$ accetta.

Allora, per costruzione E stampa w_j se e solo se M la accetta, e poiché E controlla tutte le stringhe in Σ^*, E è in grado di stampare tutte e sole le stringhe riconosciute da M; allora, segue la tesi.

□

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 dato 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. Inoltre, a meno di specifica, si assume che le TM considerate, data loro in input una qualche codifica, controllino la validità di quest'ultime come prima istruzione, e rifiutino nel caso in cui si presentino codifiche non valide.

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 di $\mathcal{M}$

Sia consideri il seguente insieme:

$$\mathcal{M} := \{\langle M \rangle \mid M \in \mathsf{TM}\}$$

Allora, si verifica che $\mathcal{M}$ è numerabile.

Dimostrazione. Sia Σ un alfabeto, e sia $\prec$ 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).

Si consideri un alfabeto Σ in grado di rappresentare ogni sua possibile codifica; allora, poiché $\mathcal{M} \subseteq \Sigma^*$, e Σ^* è numerabile per quanto detto in precedenza, segue che $\mathcal{M}$ è necessariamente nuerabile anch'esso. □

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

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

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

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:

$$\begin{array}{lcccccc} f(0) & = & 0 & 1 & 0 & 0 & 1 & \dots \\ f(1) & = & 1 & 0 & 0 & 1 & 0 & \dots \\ f(2) & = & 1 & 0 & 1 & 0 & 0 & \dots \\ f(3) & = & 0 & 1 & 0 & 1 & 1 & \dots \\ f(4) & = & 1 & 0 & 1 & 0 & 1 & \dots \\ \vdots & & & & \vdots & & & \end{array}$$

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

$$b := 11000\cdots$$

Essa dunque è definita come segue:

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

(si noti che si stanno considerando gli indici delle stringhe che partono da 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 $b_k = f(k)_k$, in quanto b_k è la k-esima cifra di $b = f(k)$, ma per definizione di b si avrebbe anche che $b_k = \overline{f(k)_k}$ e dunque

$$f(k) = b_k = \overline{f(k)_k}$$

che è chiaramente impossibile ↯. □

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 segue la tesi per doppia implicazione.

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, si definisce *sequenza caratteristica* di L una stringa binaria χ_L in cui l'i-esimo carattere di χ_L è `1` se e solo se l'i-esima stringa di Σ^* — ordinato secondo l'ordinamento $\prec$ definito all'interno della dimostrazione del Lemma 3.3.2.1 — è presente in L. Ad esempio, dato

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

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

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

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.

Allora, poiché $\mathcal{M} \subseteq \Sigma^*$ (dove Σ un il linguaggio che permette di codificare ogni macchina di Turing) è numerabile per il Lemma 3.3.2.1, e $\mathcal{P}(\Sigma^*)$ non lo è, deve necessariamente esistere un linguaggio $L \in \mathcal{P}(\Sigma^*)$ — dunque definito su Σ stesso — che non può essere riconosciuto da nessuna macchina di Turing descritta da una codifica in $\mathcal{M}$. □

3.3.3 Problema dell'accettazione

Definizione 3.3.3.1: Linguaggi di accettazione

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

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

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. 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 almeno 2, 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 G è in CNF e non presenta ϵ-regole. 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 z 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; in caso affermativo accetta, 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

$$\mathsf{CFL} \subsetneq \mathsf{DEC}$$

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 input, esegua M_G su $\langle G, w \rangle$ per stabilire se $w \in L(G)$. Allora M è un decisore di $L = L(G)$ 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 a 2 nastri; 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 codifica 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 $\lightning$. □

Corollario 3.3.3.2: Gerarchia dei linguaggi di Chomsky

Dato un alfabeto Σ, si ha che

$$\mathsf{REG} \subsetneq \mathsf{CFL} \subsetneq \mathsf{DEC} \subsetneq \mathsf{REC} \subsetneq \mathcal{P}(\Sigma^*)$$

Dimostrazione. La tesi segue direttamente dal Corollario 2.1.1.1, dal Corollario 3.3.3.1, dal'Osservazione 3.1.4.1, dal Teorema 3.3.3.6 e dal Lemma 3.3.2.2. □

Definizione 3.3.3.3: Co-Turing-riconoscibilità

Un linguaggio è detto **co-Turing-riconoscibile** se questo è il complemento di un linguaggio Turing-riconoscibile. In simboli

$$\mathsf{coREC} := \{L \mid \overline{L} \in \mathsf{REC}\}$$

Teorema 3.3.3.7: Co-Turing-riconoscibilità

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

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

Dimostrazione.

Prima implicazione. Sia $L \in \mathsf{DEC}$; per l'Osservazione 3.1.4.1 si ha che $\mathsf{DEC} \subset \mathsf{REC}$, 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 $\overline{L}$, 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 $L(M') = \overline{L}$, si ha che $\overline{L} \in \mathsf{DEC}$ per definizione stessa, ed in particolare

$$\overline{L} \in \mathsf{DEC} \implies \overline{L} \in \mathsf{REC} \iff L \in \mathsf{coREC}$$

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 $\overline{L}$; inoltre, sia Σ l'alfabeto su cui sono definiti L ed $\overline{L}$, e si consideri una stringa $w \in \Sigma^*$. Si noti che, per definizione stessa di complemento, si ha che

$$L \cup \overline{L} = \Sigma^*$$
$$L \cap \overline{L} = \varnothing$$

e dunque

$$w \in L \iff w \notin \overline{L}$$

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

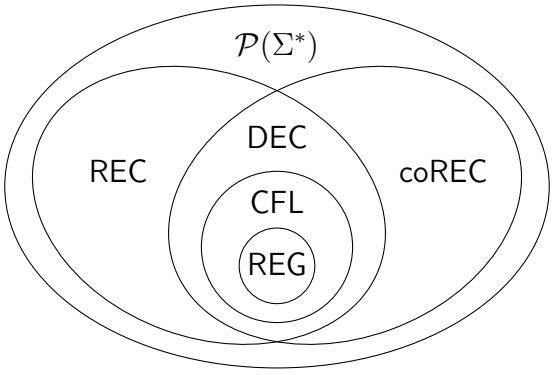
$$M(w) = \begin{cases} \text{accetta} & M_1 \text{ accetta } w \\ \text{rifiuta} & M_2 \text{ accetta } w \end{cases}$$

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 $\overline{L}$ — di fatto M decide L, e dunque $L \in \mathsf{DEC}$.

□

Osservazione 3.3.3.1: Gerarchia dei linguaggi di Chomsky

Dal Corollario 3.3.3.2, e dal Teorema 3.3.3.7, segue il seguente diagramma:



Corollario 3.3.3.3: Irriconoscibilità di $\overline{A_{\mathsf{TM}}}$

A_{TM} non è co-Turing-riconoscibile; in simboli

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

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

$$A_{\mathsf{TM}} \in \mathsf{REC} - \mathsf{DEC} \implies A_{\mathsf{TM}} \notin \mathsf{coREC}$$

□

3.3.4 Problema del vuoto

Definizione 3.3.4.1: Linguaggi del vuoto

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

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

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 nastro 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, sfruttandole "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:

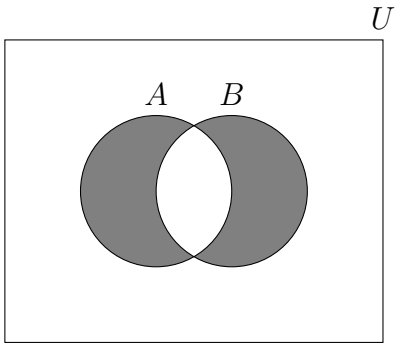
$$EQ_{\mathcal{C}} := \{\langle A, B\rangle \mid A, B \in \mathcal{C} : L(A) = L(B)\}$$

Definizione 3.3.5.2: Differenza simmetrica

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

$$A \,\Delta\, B := (A \cap \overline{B}) \cup (\overline{A} \cap B)$$

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



Proposizione 3.3.5.1: Differenza simmetrica vuota

Dati due insiemi A e B, si ha che

$$A \,\Delta\, B = \varnothing \iff A = B$$

Dimostrazione.

Prima implicazione. Per assurdo, sia $A \Delta B = \varnothing$ 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

- $x \in A - B \implies x \in A \wedge x \notin B \implies x \in A \cap \overline{B} \implies A \Delta B \neq \varnothing$ ↯
- $x \in B - A \implies x \in B \wedge x \notin A \implies x \in \overline{A} \cap B \implies A \Delta B \neq \varnothing$ ↯

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

$$A = B \implies A \cap \overline{B} = \overline{A} \cap B = \varnothing \implies A \Delta B = \varnothing$$

□

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' sul 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 Funzioni calcolabili

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 $w \in \Sigma^*$, se si ferma, termina avendo solo $f(w)$ sul suo nastro.

Definizione 4.1.1.3: Riduzione mediante funzione

Siano A e B due linguaggi definiti su un alfabeto Σ; A è detto essere **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

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

e la funzione f prende il nome di **riduzione di** A **a** B.

Lemma 4.1.1.1: Riducibilità dei complementi

Siano A e B due linguaggi definiti su un alfabeto Σ; allora, si ha che

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

Dimostrazione. Si assuma $A \leq_{\mathrm{m}} B$, senza perdita di generalità; allora, per definizione, esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ tale che $\forall w \in \Sigma^* \quad w \in A \iff f(w) \in B$. Allora, si ha che

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

e dunque f, poiché calcolabile, è anche la riduzione da $\overline{A}$ a $\overline{B}$, dunque per definizione segue che $\overline{A} \leq_{\mathrm{m}} \overline{B}$. □

Lemma 4.1.1.2: Transitività della riducibilità

Siano A, B e C tre linguaggi definiti su un alfabeto Σ; allora, si ha che

$$\begin{cases} A \leq_{\mathrm{m}} B \\ B \leq_{\mathrm{m}} C \end{cases} \implies A \leq_{\mathrm{m}} C$$

Dimostrazione. Per definizione stessa, poiché $A \leq_{\mathrm{m}} B$ e $B \leq_{\mathrm{m}} C$, esistono funzioni calcolabili $f, g : \Sigma^* \to \Sigma^*$ tali che

$$\begin{aligned} \forall w \in \Sigma^* \quad w \in A \iff f(w) \in B \\ \forall w \in \Sigma^* \quad w \in B \iff g(w) \in C \end{aligned}$$

dunque, segue che

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

ed è quindi possibile definire una funzione $F := (g \circ f)$ tale che

$$\forall w \in \Sigma^* \quad w \in A \iff F(w) \in C$$

Si consideri infine la seguente macchina di Turing M che, data in input una stringa w, computa come segue:

- M calcola $f(w)$ attraverso la TM che rende f calcolabile;
- M calcola $g(f(w))$ attraverso la TM che rende g calcolabile;
- M restituisce $g(f(w))$ sul proprio nastro.

Allora, per costruzione di M, se M si ferma, termina avendo solo $F(w) = g(f(w))$ sul suo nastro; di conseguenza, F risulta essere una funzione calcolabile, e per le proprietà di F precedentemente discusse, segue la tesi. □

4.2 Turing-decidibilità mediante riduzione

4.2.1 Teoremi

Teorema 4.2.1.1: 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:

- N computa $f(w)$ attraverso la TM che rende f calcolabile;
- N 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 4.2.1.1: 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 4.2.1.1, si ha che $A \in \mathsf{DEC}$ ↯. □

4.2.2 Problema della terminazione

Definizione 4.2.2.1: Linguaggi di terminazione

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

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

Teorema 4.2.2.1: 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 I. 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:

- D esegue $H(\langle M, w\rangle)$;
- se H rifiuta, allora M va in loop avendo w come input, e dunque D rifiuta;
- se H accetta, allora M si ferma avendo w come input, e dunque D 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. Allora, l'esistenza di D dimostrerebbe che $A_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo è assurdo per il Teorema 3.3.3.6 $\lightning$. □

Dimostrazione II. Si consideri la macchina di Turing F che, data in input una codifica $\langle M, w\rangle$, computa come segue:

- F costruisce una macchina di Turing M' che, data in input una stringa x, computa come segue:
  - M' esegue $M(x)$
  - se M accetta, M' accetta;
  - se M rifiuta, M' cicla (ad esempio muovendo la testina verso una direzione per sempre);
- F restituisce $\langle M', w\rangle$ sul suo nastro.

Si noti dunque che, poiché F si ferma avendo solo $\langle M', w\rangle$ sul nastro, per qualsiasi input $\langle M, w\rangle$, la funzione $f : \Sigma^* \to \Sigma^* : \langle M, w\rangle \mapsto \langle M', w\rangle$ che F descrive è calcolabile per definizione. Allora, si noti che

$$\langle M, w\rangle \in A_{\mathsf{TM}} \iff w \in L(M) \iff w \in L(M')$$

per definizione di A_{TM} e per costruzione di M' rispettivamente; inoltre

$$w \in L(M') \iff f(\langle M, w\rangle) = \langle M', w\rangle \in HALT_{\mathsf{TM}}$$

per definizione di $HALT_{\mathsf{TM}}$; dunque

$$\langle M, w\rangle \in A_{\mathsf{TM}} \iff f(\langle M, w\rangle) \in HALT_{\mathsf{TM}}$$

Allora, per definizione, segue che

$$A_{\mathsf{TM}} \leq_{\mathrm{m}} HALT_{\mathsf{TM}}$$

e poiché $A_{\mathsf{TM}} \notin \mathsf{DEC}$ per il Teorema 3.3.3.6, è verificata la tesi per cui $HALT_{\mathsf{TM}} \notin \mathsf{DEC}$ per il Corollario 4.2.1.1. □

4.2.3 Problema del vuoto

Teorema 4.2.3.1: 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:

- D costruisce una macchina di Turing M' che, data in input una stringa x, computa come segue:
  - se l'input x di M è proprio w, M' accetta se M accetta;
  - se l'input x di M non è w, M' rifiuta;

  (si noti che questa macchina è realizzabile poiché è sufficiente aggiungere ad M degli stati che controllano l'input fornito, e computare come descritto);
- D esegue $H(\langle M' \rangle)$;
- se H accetta, allora $L(M') = \varnothing$, e dunque D rifiuta;
- se H rifiuta, allora $L(M') \neq \varnothing$, e dunque D 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. Allora, l'esistenza di D dimostrerebbe che $A_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo non è possibile per il Teorema 3.3.3.6 ↯. □

4.2.4 Problema della regolarità

Definizione 4.2.4.1: Linguaggi di regolarità

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

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

Teorema 4.2.4.1: 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:

- D 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 $n \in \mathbb{N}$), M' accetta;
  - se l'input x di M' non ha tale forma, M' accetta se $M(w)$ accetta (dunque ignorando x);
- D esegue $H(\langle M' \rangle)$;
- se H accetta, allora $L(M') \in \mathsf{REG}$, e dunque D accetta;
- se H rifiuta, allora $L(M') \in \mathsf{REG}$, e dunque D rifiuta.

Di conseguenza, D è in grado di stabilire, data una macchina di Turing M ed un input w, se $w \in L(M)$, 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 □

4.2.5 Problema dell'uguaglianza

Teorema 4.2.5.1: 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 I. 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:

- D 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 D accetta;
- se H rifiuta, allora $L(M) \neq L(M_{\text{reject}}) = \varnothing$, e dunque D 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. Allora, l'esistenza di D dimostrerebbe che $E_{\mathsf{TM}} \in \mathsf{DEC}$, ma questo non è possibile per il Teorema 4.2.3.1 ↯. □

Dimostrazione II. Si consideri la macchina di Turing F che, data in input una codifica $\langle M \rangle$, computa come segue:

- F costruisce una macchina di Turing M' che, data in input una stringa x, rifiuta sempre — e dunque $L(M') = \varnothing$;
- F restituisce $\langle M, M' \rangle$ sul suo nastro.

Si noti dunque che, poiché F si ferma avendo solo $\langle M, M' \rangle$ sul natro, per qualsiasi input $\langle M \rangle$, la funzione $f : \Sigma^* \to \Sigma^* : \langle M \rangle \mapsto \langle M, M' \rangle$ che F descrive è calcolabile per definizione. Allora, si noti che

$$\langle M \rangle \in E_{\mathsf{TM}} \iff L(M) = \varnothing = L(M')$$

per definizione di E_{TM} e per costruzione di M' rispettivamente; inoltre

$$L(M') = L(M) \iff f(\langle M \rangle) = \langle M, M' \rangle \in EQ_{\mathsf{TM}}$$

per definizione di EQ_{TM}; dunque

$$\langle M \rangle \in E_{\mathsf{TM}} \iff f(\langle M \rangle) \in EQ_{\mathsf{TM}}$$

Allora, per definizione, segue che

$$E_{\mathsf{TM}} \leq_{\text{m}} EQ_{\mathsf{TM}}$$

e poiché $E_{\mathsf{TM}} \notin \mathsf{DEC}$ per il Teorema 4.2.3.1, è verificata la tesi per cui $EQ_{\mathsf{TM}} \notin \mathsf{DEC}$ per il Corollario 4.2.1.1. □

4.3 Turing-riconoscibilità mediante riduzione

4.3.1 Teoremi

Teorema 4.3.1.1: Turing-riconoscibilità mediante riduzione

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

$$\begin{cases} A \leq_{\text{m}} B \\ B \in \mathsf{REC} \end{cases} \implies A \in \mathsf{REC}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 4.2.1.1, tranne per il fatto che le macchine di Turing considerate sono riconoscitori e non decisori; verrà dunque omessa. □

Corollario 4.3.1.1: Turing-irriconoscibilità mediante riduzione

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

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

Dimostrazione. Per assurdo, sia $B \in \mathsf{REC}$; allora, poiché $A \leq_{\mathrm{m}} B$, per il Teorema 4.3.1.1 si ha che $A \in \mathsf{REC}$ ↯. □

Corollario 4.3.1.2: Co-riconoscibilità mediante riduzione

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

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

Dimostrazione. Per il Lemma 4.1.1.1 si ha che

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

e dunque, per il Teorema 4.3.1.1, si ha che

$$\begin{cases} \overline{A} \leq_{\mathrm{m}} \overline{B} \\ B \in \mathsf{coREC} \iff \overline{B} \in \mathsf{REC} \end{cases} \implies \overline{A} \in \mathsf{REC} \iff A \in \mathsf{coREC}$$

□

Corollario 4.3.1.3: Co-irriconoscibilità mediante riduzione

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

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

Dimostrazione. Per assurdo, sia $B \in \mathsf{coREC}$; allora, poiché $A \leq_{\mathrm{m}} B$, per il Corollario 4.3.1.2, si ha che $A \in \mathsf{coREC}$ ↯. □

Corollario 4.3.1.4: Riducibilità al proprio complemento

Sia A un linguaggio tale da essere riducibile al suo complemento; allora, se A è Turing-riconoscibile, è anche decidibile. In simboli

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

Dimostrazione. Per il Lemma 4.1.1.1 si ha che

$$A \leq_{\mathrm{m}} \overline{A} \iff \overline{A} \leq_{\mathrm{m}} \overline{\overline{A}} = A$$

e dunque, per il Teorema 4.3.1.1, si ha che

$$\begin{cases} \overline{A} \leq_{\mathrm{m}} A \\ A \in \mathsf{REC} \end{cases} \implies \overline{A} \in \mathsf{REC} \iff A \in \mathsf{coREC}$$

e poiché $\mathsf{DEC} = \mathsf{REC} \cap \mathsf{coREC}$ per il Teorema 3.3.3.7, segue che $A \in \mathsf{DEC}$. □

4.3.2 Problema dell'uguaglianza

Teorema 4.3.2.1: Irriconoscibilità di EQ_{TM}

EQ_{TM} non è Turing-riconoscibile; in simboli

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

Dimostrazione. Si consideri la macchina di Turing F che, data in input una codifica $\langle M, w\rangle$, computa come segue:

- F costruisce una macchina di Turing M_1 che, data in input una stringa x, rifiuta sempre — e dunque $L(M_1) = \varnothing$;
- F costruisce una macchina di Turing M_2 che, data in input una stringa x, computa come segue:
  - M_2 esegue $M(w)$ (dunque ignorando x);
  - se M accetta, M_2 accetta, altrimenti M_2 rifiuta;

  dunque, se M accetta w, M_2 riconosce Σ^*, altrimenti M_2 riconosce $\varnothing$;
- F restituisce $\langle M_1, M_2\rangle$ sul suo nastro.

Si noti dunque che, poiché F si ferma avendo solo $\langle M_1, M_2\rangle$ sul nastro, per qualsiasi input $\langle M, w\rangle$, la funzione $f : \Sigma^* \to \Sigma^* : \langle M, w\rangle \mapsto \langle M_1, M_2\rangle$ che F descrive è calcolabile per definizione. Allora, si noti che

$$\langle M, w\rangle \in A_{\mathsf{TM}} \iff w \in L(M) \iff L(M_2) = \Sigma^*$$

per definizione di A_{TM} e per costruzione di M_2 rispettivamente; inoltre

$$L(M_1) = \varnothing \neq L(M_2) \iff f(\langle M, w\rangle) = \langle M_1, M_2\rangle \notin EQ_{\mathsf{TM}} \iff \langle M_1, M_2\rangle \in \overline{EQ_{\mathsf{TM}}}$$

per definizione di EQ_{TM}; dunque

$$\langle M, w\rangle \in A_{\mathsf{TM}} \iff f(\langle M, w\rangle) \in \overline{EQ_{\mathsf{TM}}}$$

Allora, per definizione, segue che

$$A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{EQ_{\mathsf{TM}}}$$

e poiché $A_{\mathsf{TM}} \notin \mathsf{coREC}$ per il Corollario 3.3.3.3, è verificata la tesi per cui $\overline{EQ_{\mathsf{TM}}} \notin \mathsf{coREC} \iff EQ_{\mathsf{TM}} \notin \mathsf{REC}$, per il Corollario 4.3.1.3. □

Teorema 4.3.2.2: Irriconoscibilità di $\overline{EQ_{\mathsf{TM}}}$

EQ_{TM} non è co-Turing-riconoscibile; in simboli

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

Dimostrazione. La dimostrazione è analoga al Teorema 4.3.2.1, pertanto verrà omessa. □

5 Complessità di tempo

5.1 Analisi asintotica

5.1.1 O-grande ed o-piccolo

Definizione 5.1.1.1: O-grande

Siano $f, g : \mathbb{N} \to \mathbb{R}^+$; $f(n)$ è detta essere **in O-grande di** $g(n)$ se e solo se

$$\exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}_{>0} \mid \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$$

ed è indicato con il simbolismo

$$f(n) = O(g(n))$$

Viceversa, $g(n)$ è detto essere **limite superiore asintotico** per $f(n)$.

Esempio 5.1.1.1 (O-grande)**.** Siano $f(n) = 100n^2$ e $g(n) = n^3$; poiché esistono $c = 10$ ed $n_0 = 10$ tali che

$$\forall n \geq n_0 = 10 \quad f(n) = 100n^2 \leq 10 \cdot n^3 = c \cdot g(n)$$

si ha che $f(n) = O(g(n))$ per definizione.

Definizione 5.1.1.2: o-piccolo

Siano $f, g : \mathbb{N} \to \mathbb{R}^+$; $f(n)$ è detta essere **in o-piccolo di** $g(n)$ se e solo se

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

o equivalentemente, se e solo se

$$\exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}_{>0} \mid \forall n \geq n_0 \quad f(n) < c \cdot g(n)$$

ed è indicato con il simbolismo

$$f(n) = o(g(n))$$

Esempio 5.1.1.2 (o-piccolo)**.** Si consideri l'Esempio 5.1.1.1; poiché sono stati scelti $c = n_0 = 10$, per $n = n_0$ si ottiene che

$$f(n_0) = f(10) = 100 \cdot 10^2 = 10000 = 10 \cdot 1000 = 10 \cdot 10^3 = 10 \cdot g(10) = c \cdot g(n_0)$$

Allora, scegliendo n_0 tale da essere maggiore di 10 — dunque ad esempio 11 — si ha che

$$\forall n \geq n_0 = 11 \quad f(n) = 100n^2 < 10 \cdot n^3 = c \cdot g(n)$$

e dunque $f(n) = o(g(n))$ per definizione.

Osservazione 5.1.1.1: O-grande ed o-piccolo

Si noti che, date due funzioni $f, g : \mathbb{N} \to \mathbb{R}^+$, per definizione stessa si ha che

$$f(n) = o(g(n)) \implies f(n) = O(g(n))$$

Proposizione 5.1.1.1: Algebra asintotica

Siano $f, g, h : \mathbb{N} \to \mathbb{R}^+$; allora, si ha che:

- $\forall c \in \mathbb{R} \quad f(n) = c \cdot o(g(n)) \implies f(n) = o(g(n))$
- $f(n) = o(g(n)) + o(h(n)) \implies f(n) = o(\max(g(n), h(n)))$
- $f(n) = o(g(n)) \cdot o(h(n)) \implies f(n) = o(g(n) \cdot h(n))$

Si noti che tali regole valgono anche per O-grande, per l'Osservazione 5.1.1.1.

Osservazione 5.1.1.2: Funzioni logaritmiche

Per convenzione, le funzioni logaritmiche si esprimono in termini di $\log_2$, ed è possibile farlo grazie alle proprietà dei logaritmi, infatti

$$\forall k \in \mathbb{R} \quad f(n) = \log_k(n) \implies f(n) = O(\log_k(n)) = O\left(\frac{\log_2(n)}{\log_2(k)}\right) = O(\log_2(n))$$

Osservazione 5.1.1.3: Funzioni esponenziali

Per convenzione, le funzioni esponenziali si esprimono in termini di potenze di 2, ed è possibile farlo grazie alle proprietà delle potenze, infatti

$$\forall k \in \mathbb{R} \quad f(n) = k^n = 2^{\log_2(k^n)} = 2^{n \cdot \log_2(k)} = 2^{O(n)}$$

Inoltre, una funzione $f(n) = O\left(2^{O(n)}\right)$, per definizione, ha come limite superiore asintotico $2^{O(n)}$, ed è dunque superata asintoticamente da una funzione $c \cdot 2^{O(n)}$ per qualche $c \in \mathbb{N} - \{0\}$; allora, per proprietà delle potenze, si ha che

$$c \cdot 2^{O(n)} = 2^{\log_2(c)} \cdot 2^{O(n)} = 2^{\log_2(c)+O(n)} = 2^{O(n)}$$

implicando che $f(n) = O\left(2^{O(n)}\right) = 2^{O(n)}$.

Proposizione 5.1.1.2: Transitività di o-piccolo

Siano $f, g, h : \mathbb{N} \to \mathbb{R}^+$; allora, si verifica che

$$\begin{cases} f(n) = o(g(n)) \\ g(n) = o(h(n)) \end{cases} \implies f(n) = o(h(n))$$

Dimostrazione. Per definizione, si ha che

$$f(n) = o(g(n)) \iff \lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

e che

$$g(n) = o(h(n)) \iff \lim_{n \to +\infty} \frac{g(n)}{h(n)} = 0$$

allora, segue che

$$\lim_{n \to +\infty} \frac{f(n)}{h(n)} = \lim_{n \to +\infty} \frac{f(n)}{g(n)} \cdot \frac{g(n)}{h(n)} = 0 \iff f(n) = o(h(n))$$

□

5.2 Complessità di tempo di macchine di Turing

5.2.1 Macchine di Turing

Definizione 5.2.1.1: Complessità di tempo di una TM

Sia D un decisore; si definisce **complessità di tempo** (o **tempo di esecuzione**) di D la funzione $f : \mathbb{N} \to \mathbb{N}$ tale che $f(n)$ sia il numero massimo di passi che D utilizza per processare una stringa di lunghezza n.

Esempio 5.2.1.1 (Complessità di tempo di TM)**.** Sia D un decisore che, data in input una stringa x, sposta la testina verso destra finché non legge $\sqcup$, e quando legge quest'ultimo accetta. Allora, se in input è presente una stringa avente lunghezza n, D impiega esattamente n passi a processarla, e dunque $f(n) = n \implies f \equiv \mathrm{id}$.

5.2.2 Macchine di Turing multinastro

Teorema 5.2.2.1: Relazione tra TM e TM multinastro

Sia $t(n)$ una funzione tale che $t(n) \geq n$, e sia M una macchina di Turing multinastro avente tempo $t(n)$; allora, esiste una TM M', equivalente ad M, avente tempo $O(t^2(n))$.

Dimostrazione. Sia M una macchina di Turing a k-nastri avente tempo di esecuzione $t(n) \geq n$, e si consideri la TM M' costruita all'interno della prima implicazione della dimostrazione della Proposizione 3.2.2.1. Dunque, si ha che:

- il numero di nastri k non dipende dall'input, ed è dunque costante;
- si assuma n essere la lunghezza della stringa più lunga tra le stringhe presenti sui nastri di M; allora, per trasformare il nastro nel formato descritto all'interno della dimostrazione — ovvero effettuando uno *shift* verso destra, anteponendo # all'inizio e inserendo i caratteri marcati per segnare le testine virtuali — sono necessari $O(n)+O(n)+O(n) = O(n)$ passi;
- si considerino le parti attive dei nastri di M: nel caso peggiore (ovvero se la testina di M non fa altro che spostarsi a destra), ognuna di queste ha lunghezza $t(n)$, poiché M utilizza $t(n)$ celle del nastro in $t(n)$ passi; di conseguenza, una scansione della parte attiva del nastro di M' impiega $k \cdot O(t(n)) = O(t(n))$ passi;
- per simulare una singola mossa di M, la testina di M' scansiona tutto il suo nastro, aggiorna le testina virtuali, e successivamente effettua una seconda scansione per simulare i rimpiazzi; allora, un singolo passo di M impiega $2 \cdot O(t(n)) = O(t(n))$;
- allora, poiché il numero di passi da simulare è $t(n)$, la complessità temporale di M risulta essere
$$O(n) + t(n) \cdot O(t(n)) = O(t^2(n))$$

□

5.2.3 Macchine di Turing non deterministiche

Teorema 5.2.3.1: Relazione tra TM ed NTM

Sia $t(n)$ una funzione tale che $t(n) \geq n$, e sia N una macchina di Turing non deterministica avente tempo $t(n)$; allora, esiste una TM M, equivalente ad N, avente tempo $2^{O(t(n))}$.

Dimostrazione. Sia N una NTM avente tempo di esecuzione $t(n) \geq n$, e si consideri la TM M costruita all'interno della prima implicazione della dimostrazione della Proposizione 3.2.3.1. Dunque, si ha che:

- b non dipende dalla dimensione dell'input;
- TODO

□

5.3 Classi di complessità di tempo

5.3.1 Classe DTIME

Definizione 5.3.1.1: Classe DTIME

Data una funzione $t : \mathbb{N} \to \mathbb{R}^+$, si definisce **classe di complessità di tempo** $\mathsf{DTIME}(t(n))$ l'insieme dei linguaggi decidibili da una macchina di Turing — *varianti escluse* — in $O(t(n))$.

Esempio 5.3.1.1 (Classe di complessità di tempo)**.** Si consideri una macchina di Turing M — in grado di decidere $L = \{\texttt{0}^n\texttt{1}^n \mid n \in \mathbb{N}\}$ — che, data in input una stringa x, computa come segue:

1. scorre il nastro: se è presente uno `0` alla destra di un `1`, rifiuta;
2. torna all'inizio del nastro;
3. fintanto che sono presenti almeno uno `0` ed almeno un `1`, ripete i seguenti passi:
    4. scorre il nastro: se trova uno `0`, lo rimpiazza con `x`;
    5. continua a scorrere il nastro: se trova un `1`, lo rimipazza con `x`;
    6. torna all'inizio del nastro;
7. scorre il nastro: se sono presenti solo `x` sul nastro, accetta.

Se uno qualsiasi dei passaggi menzionati non è effettuabile, M rifiuta; dunque ci si convince facilmente che M decide L come richiesto.

Analizzando la complessità temporale di M, si ottiene che:

- l'istruzione 1 richiede n passi, ed ha dunque costo $O(n)$;
- l'istruzione 2 richiede n passi, ed ha dunque costo $O(n)$;
- il tempo dell'istruzione 3 dipende dalle istruzioni che ripete, e dunque
    - l'istruzione 4 richiede n passi, ed ha dunque costo $O(n)$;
    - analogamente all'istruzione 4, l'istruzione 5 ha costo $O(n)$;
    - analogamente all'istruzione 2, l'istruzione 6 ha costo $O(n)$;

inoltre, si noti che l'istruzione 3 ripete le istruzioni 4, 5 e 6 al più $\frac{n}{2}$ volte, poiché ad ogni iterazione vengono rimossi uno `0` ed un `1` insieme; allora, per via dei costi delle istruzioni che ripete, l'istruzione 3 ha costo pari a

$$\frac{n}{2} \cdot [O(n) + O(n) + O(n)] = \frac{n}{2} \cdot O(\max(n, n, n)) =$$
$$= \frac{n}{2} \cdot O(n) = \frac{1}{2} \cdot n \cdot O(n) = \frac{1}{2} \cdot O(n \cdot n) = \frac{1}{2} \cdot O(n^2) = O(n^2)$$

(si veda la Proposizione 5.1.1.1 per chiarimenti);

- analogamente all'istruzione 2, l'istruzione 7 ha costo $O(n)$.

Allora, il costo temporale di M risulta essere

$$O(n) + O(n) + O(n^2) + O(n) = O(\max(n, n, n^2, n)) = O(n^2) \implies L \in \mathsf{DTIME}\left(n^k\right)$$

5.3.2 Classe P

Definizione 5.3.2.1: Classe P

Si definisce **classe dei linguaggi decidibili in tempo polinomiale** da una TM a singolo nastro il seguente insieme

$$\mathsf{P} := \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}\left(n^k\right)$$

Osservazione 5.3.2.1: Classe P

La classe di problemi descritti dai linguaggi in P è particolarmente importante, poiché P corrisponde approssimativamente alla classe dei problemi che sono realisticamente risolvibili da un computer in tempi ragionevoli.

Definizione 5.3.2.2: Funzione polinomialmente calcolabile

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

Definizione 5.3.2.3: Riduzione polinomiale mediante funzione

Siano A e B due linguaggi definiti su un alfabeto Σ; A è detto essere **riducibile polinomialmente mediante funzione a** B se esiste una funzione polinomialmente calcolabile $f : \Sigma^* \to \Sigma^*$ *che termina su ogni input* tale che

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

Tale relazione è indicata attraverso il simbolismo

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

e la funzione f prende il nome di **riduzione di tempo polinomiale di** A **a** B.

Lemma 5.3.2.1: Riducibilità polinomiale dei complementi

Siano A e B due linguaggi definiti su un alfabeto Σ; allora, si ha che

$$A \leq_{\mathrm{P}} B \iff \overline{A} \leq_{\mathrm{P}} \overline{B}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Lemma 4.1.1.1, pertanto verrà omessa. □

Lemma 5.3.2.2: Transitività della riducibilità polinomiale

Siano A, B e C tre linguaggi definiti su un alfabeto Σ; allora, si verifica che

$$\begin{cases} A \leq_{\mathrm{P}} B \\ B \leq_{\mathrm{P}} C \end{cases} \implies A \leq_{\mathrm{P}} C$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Lemma 4.1.1.2, pertanto verrà omessa. □

Teorema 5.3.2.1: P mediante riduzione

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

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

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 4.2.1.1, con la sola differenza che le macchine di Turing considerate computano in tempo polinomiale; verrà dunque omessa. □

Corollario 5.3.2.1: P mediante riduzione

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

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

Dimostrazione. Per assurdo, sia $B \in \mathsf{P}$; allora, poiché $A \leq_{\mathrm{P}} B$, per il Teorema 5.3.2.1, si ha che $A \in \mathsf{P}$ ↯. □

Osservazione 5.3.2.2: Rappresentazione di grafi

Nelle prossime dimostrazioni si assumerà che sia possibile definire una codifica coerente per rappresentare grafi $G = (V, E)$, al fine di fornire questi in input a macchine di Turing; un esempio potrebbe essere quello della matrice di adiacenza, in cui la generica cella è definita come segue:

$$\forall i, j \in [1, |V|] \quad m_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E \end{cases}$$

dove $v_i, v_j \in V$.

Teorema 5.3.2.2: $PATH$ in P

Sia $PATH$ il linguaggio definito come segue:

$$PATH := \{\langle G, s, t\rangle \mid G = (V, E) \text{ grafo diretto con un cammino } s \to t\}$$

Allora, si verifica che

$$PATH \in \mathsf{P}$$

Dimostrazione. Sia M una macchina di Turing che, data in input una codifica $\langle G, s, t\rangle$, una volta controllata la validità della codifica in input, computa come segue:

- cerca il vertice s tra i vertici di G;
- una volta trovato, marca s;
- finche non è più possibile marcare vertici in V:
  - marca ogni vertice $v \in V$ tale per cui esiste un arco $(u, v) \in E$ con $u \in V$ già marcato;
- se tra i vertici marcati è presente t, allora M accetta.

Dunque, poiché $|E|$ è sicuramente minore di n (in quanto il numero di archi è sicuramente inferiore del numero di simboli che servono per rappresentare tutto l'input), sicuramente $|E| = O(n)$, e poiché nel caso peggiore M marca tutti i vertici in V, si ha che la sua

complessità temporale risulta essere (rispettivamente per ogni istruzione):

$$O\left(n^k\right) + O(1) + |E| \cdot O\left(n^k\right) + O\left(n^k\right) = O(n) \cdot O\left(n^k\right) = O(n^{k+1}) \implies PATH \in \mathsf{P}$$

□

5.3.3 Classe coP

Definizione 5.3.3.1: Classe coP

La classe coP è definita come segue:

$$\mathsf{coP} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{P}\}$$

Teorema 5.3.3.1: Relazione tra P e coP

Si verifica che

$$\mathsf{P} = \mathsf{coP}$$

Dimostrazione. Si consideri un linguaggio $L \in \mathsf{P}$, e dunque per definizione esiste una macchina di Turing M_L che decide L in tempo polinomiale; allora, si consideri la seguente TM $M_{\overline{L}}$ che, data in input una stringa w, computa come segue:

- $M_{\overline{L}}$ esegue $M_L(w)$;
- $M_{\overline{L}}$ accetta se e solo se $M_L(w)$ rifiuta.

Allora, per costruzione, si ha che

$$w \in L(M_{\overline{L}}) \iff w \notin L(M_L) = L$$

e di conseguenza $L(M_{\overline{L}}) = \overline{L}$. Inoltre, poiché M_L è un decisore di tempo polinomiale, anche $M_{\overline{L}}$ è un decisore polinomiale per sua stessa costruzione, implicando che $\overline{L} \in \mathsf{P}$. Questo dimostra che $\mathsf{P} \subseteq \mathsf{coP}$.

Infine, poiché è possibile ripetere il ragionamento analogo per $\mathsf{coP} \subseteq \mathsf{P}$, segue la tesi per doppia implicazione. □

5.3.4 Classe EXP

Definizione 5.3.4.1: Classe EXP

Si definisce **classe dei linguaggi decidibili in tempo esponenziale** da una TM a singolo nastro il seguente insieme

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

Definizione 5.3.4.2: Cammino hamiltoniano

Sia G un grafo; un **cammino hamiltoniano** su G è un cammino che passa una sola volta per ogni nodo di G.

Teorema 5.3.4.1: $HAMPATH$ in EXP

Sia $HAMPATH$ il linguaggio definito come segue:

$$HAMPATH := \{\langle G, s, t\rangle \mid G = (V, E) \text{ grafo diretto con un cammino hamiltoniano } s \to t\}$$

Allora, si verifica che

$$HAMPATH \in \mathsf{EXP}$$

Dimostrazione. Si consideri una macchina di Turing M che, data in input una codifica $\langle G, s, t\rangle$ in input, computa come segue:

- per ogni possibile cammino c della forma $s \to t$ in G, M controlla se c è hamiltoniano; se lo è, M accetta;
- se M non ha mai accettato, M rifiuta.

Allora M, per costruzione, è in grado di decidere $HAMPATH$. Si noti che i lnumero di cammini possibili in un grafo è $O\left(2^{n^k}\right)$ per qualche $k \in \mathbb{N}$ — più precisamente, sarebbe $\left(\max_{v \in V} \deg(v)\right)^{O(n)}$ — e poiché controllare che un cammino sia hamiltoniano richiede tempo $O(n)$, la complessità temporale di M risulta essere

$$O(n) \cdot O\left(2^{n^k}\right) = O\left(n \cdot 2^{n^k}\right)$$

e poiché $n^k \geq \log(n)$ (per k sufficientemente grande) si ha che

$$O\left(n \cdot 2^{n^k}\right) = O\left(2^{n^k}\right) \implies HAMPATH \in \mathsf{EXP}$$

□

5.3.5 Classe coEXP

Definizione 5.3.5.1: Classe coEXP

La classe coEXP è definita come segue:

$$\mathsf{coEXP} := \{L \in \mathsf{DEC} \mid \overline{A} \in \mathsf{EXP}\}$$

Teorema 5.3.5.1: Relazione tra EXP e coEXP

Si verifica che

$$\mathsf{EXP} = \mathsf{coEXP}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.3.1, pertanto verrà omessa. □

5.3.6 Classe NTIME

Definizione 5.3.6.1: Classe NTIME

Data una funzione $t : \mathbb{N} \to \mathbb{R}^+$, si definisce **classe di complessità di tempo** $\mathsf{NTIME}(t(n))$ l'insieme dei linguaggi decidibili da una macchina di Turing non deterministica in $O(t(n))$.

5.3.7 Classe NP

Definizione 5.3.7.1: Classe NP

Si definisce **classe dei linguaggi decidibili non deterministicamente in tempo polinomiale** da una NTM il seguente insieme

$$\mathsf{NP} := \bigcup_{k \in \mathbb{N}} \mathsf{NTIME}\left(n^k\right)$$

Proposizione 5.3.7.1: Relazione tra NP ed EXP

Si verifica che

$$\mathsf{NP} \subseteq \mathsf{EXP}$$

Dimostrazione. Sia $L \in \mathsf{NTIME}\left(n^k\right)$ per qualche $k \in \mathbb{N}$, e dunque per definizione esiste una NTM che decide L in $O\left(n^k\right)$; allora, per il Teorema 5.2.3.1, si ha che $L \in \mathsf{DTIME}\left(2^{n^k}\right)$. □

Teorema 5.3.7.1: NP mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{P}} B \\ B \in \mathsf{NP} \end{cases} \implies A \in \mathsf{NP}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 4.2.1.1, tranne per il fatto che le macchine di Turing considerate computano non deterministicamente in tempo polinomiale; verrà dunque omessa. □

Corollario 5.3.7.1: NP mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{P}} B \\ A \notin \mathsf{NP} \end{cases} \implies B \notin \mathsf{NP}$$

Dimostrazione. Per assurdo, sia $B \in \mathsf{NP}$; allora, poiché $A \leq_{\mathrm{P}} B$, per il Teorema 5.3.7.1, si ha che $A \in \mathsf{NP}$ ↯. □

Definizione 5.3.7.2: Verificatore

Sia L un linguaggio decidibile definito su un certo alfabeto Σ, e V un decisore; V è detto **verificatore di** L se

$$L = \{w \in \Sigma^* \mid \exists c \in \Sigma^* : \langle w, c \rangle \in L(V)\}$$

Data una codifica $\langle w, c \rangle \in L(V)$, c prende il nome di **certificato di** w.

Osservazione 5.3.7.1: Complessità temporale di verificatori

Dato un verificatore V, ed una codifica $\langle w, c \rangle \in L(V)$, il tempo di computazione di V è misurato solamente in termini della lunghezza di w

Definizione 5.3.7.3: Linguaggio polinomialmente verificabile

Un linguaggio è detto essere **polinomialmente verificabile** se ammette un verificatore in tempo polinomiale.

Osservazione 5.3.7.2: Certificati di verificatori

Sia V un verificatore che computa in $O(f(n))$ per qualche f, e sia $\langle w, c \rangle \in L(V)$; si noti che c, per definizione di V, può avere qualsiasi lunghezza, ma poiché V deve computare in $O(f(n))$, la porzione di c che verrà utilizzata da parte di V deve comunque essere in $O(f(n))$, poiché $O(f(n))$ è proprio il tempo di cui V dispone per computare.

Teorema 5.3.7.2: Linguaggi polinomialmente verificabili

NP è la classe dei linguaggi polinomialmente verificabili.

Dimostrazione.

Prima implicazione. Sia $L \in \mathsf{NP}$, e dunque per la Definizione 5.3.7.1 esiste una NTM N che decide $L = L(N)$ non deterministicamente in tempo polinomiale; sia allora

V un decisore che, data in input una configurazione $\langle w, c\rangle$, computa come segue:

- V interpreta c come un indirizzo (come descritto nella dimostrazione della Proposizione 3.2.3.1);
- V simula N su input w, eseguendo le scelte dettate dall'indirizzo c;
- V accetta se la simulazione di N accetta.

Allora, per definizione $w \in L = L(N)$ se e solo se esiste un ramo di computazione di N accettante, e per come V interpreta il certificato in input, ciò avviene se e solo se $\exists c \in \Sigma^* \mid \langle w, c\rangle \in L(V)$. Inoltre, poiché $L \in \mathsf{NP}$, la lunghezza dei rami dell'albero di computazione di N è necessariamente polinomiale, e dunque se un certificato c descrive una serie di scelte tali che N accetti w, la lunghezza di c deve necessariamente essere in polinomiale anch'essa. Di conseguenza, poiché la lunghezza di c determina la computazione di V stessa, V risulta essere un verificatore in tempo polinomiale di L.

Seconda implicazione. Sia L un linguaggio polinomialmente verificabile, e dunque esiste un verificatore V che verifica L polinomialmente. Sia allora N una NTM— avente V come sottoprocedura — che, data una stringa w in input, computa come segue:

- N sceglie casualmente una stringa c avente lunghezza polinomiale;
- N simula $V(\langle w, c\rangle)$
- N accetta se V accetta.

Poiché N computa non deterministicamente, il primo passo genererà un albero di computazione in cui ogni ramo elaborerà un solo certificato c avente lunghezza polinomiale (si noti che i certificati aventi lunghezza polinomiale sono *finiti*); inoltre, poiché V verifica L in tempo polinomiale, deve esistere un certificato di lunghezza polinomiale tale per cui il suo ramo di computazione sia accettante (si noti l'Osservazione 5.3.7.2). Dunque, poiché per costruzione N riconosce tutte e sole le stringhe w tali per cui esiste un certificato c tale che $\langle w, c\rangle \in L(V)$, segue che $L = L(N) \implies L \in \mathsf{NP}$.

□

Definizione 5.3.7.4: Grafo tricolorabile

Un grafo è detto **tricolorabile** se ad ogni suo nodo è possibile assegnare un colore diverso dai suoi nodi adiacenti, utilizzando al più 3 colori.

Teorema 5.3.7.3: $3COL$ in NP

Sia $3COL$ il linguaggio definito come segue:

$$3COL := \{\langle G\rangle \mid G = (V_G, E_G) \text{ grafo tricolorabile}\}$$

Allora, si verifica che

$$3COL \in \mathsf{NP}$$

Dimostrazione. Sia V una macchina di Turing che, data una codifica $\langle w, c\rangle$ in input, computa come segue:

1. V interpreta w come $\langle G\rangle$, dove $G = (V_G, E_G)$ è un grafo;
2. V interpreta c come $\langle c_1, \ldots, c_{|V_G|}\rangle$, e $\forall k \in [1, |V_G|] \quad c_k \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$, dunque c viene interpretato come possibile colorazione di G;
3. se esiste un arco $(v_i, v_j) \in E_G$, con $i, j \in [1, |V_G|] \mid i \neq j$ tale che $c_i = c_j$, V rifiuta; se tali archi non esistono, V accetta.

Dunque, poiché V accetta $\langle w, c\rangle$ se e solo se la colorazione descritta da c è una tricolorazione del grafo in input, segue che V verifica $3COL$.

Per quanto riguarda il costo temporale di V, si ha che

- le istruzioni 1 e 2 hanno costo $O(|w|) = O(n)$
- l'istruzione 3 è costituita da un ciclo, il cui corpo ha costo $O(1)$, e viene eseguito $|E_G|$ volte, e dunque ha costo $O(|E_G|) \cdot O(1) = O(|E_G|)$; si noti però che, nel caso peggiore, ogni nodo del grafo è connesso con ogni altro nodo, e dunque $O(|E_G|) = O(|V_G|^2) = O(n^2)$

dunque V ha costo pari a

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

di conseguenza V è un verificatore polinomiale, e per il Teorema 5.3.7.2 segue che $3COL \in \mathsf{NP}$. □

Definizione 5.3.7.5: k-clique

Sia G un grafo non diretto; una **k-clique** di G è un sottografo di G in cui ogni nodo è collegato con ogni altro nodo.

Teorema 5.3.7.4: $CLIQUE$ in NP

Sia $CLIQUE$ il linguaggio definito come segue:

$$CLIQUE := \{\langle G, k\rangle \mid G = (V_G, E_G) \text{ grafo non diretto contenente una } k\text{-clique}\}$$

Allora, si verifica che $CLIQUE \in \mathsf{NP}$.

Dimostrazione. Sia V una macchina di Turing che, data una codifica $\langle w, c\rangle$ in input, computa come segue:

1. V interpreta w come $\langle G, k\rangle$, dove $G = (V_G, E_G)$ è un grafo, e k è un intero;
2. V interpreta c come sottoinsieme di V_G;
3. V controlla se esiste un arco in E_G tra ogni coppia di nodi dell'insieme descritto dal certificato c;
4. se gli archi sono tutti presenti, V accetta, altrimenti rifiuta.

Allora, V risulta essere un verificatore di $CLIQUE$, poiché per costruzione V accetta se e solo se esiste un sottoinsieme di nodi $c \in \Sigma^*$ tale che $\langle\langle G, k\rangle\, c\rangle \in L(V)$; inoltre V è un verificatore polinomiale, poiché l'istruzione 2 richiede tempo $O(n^2)$. Allora, per il Teorema 5.3.7.2, segue la tesi. □

5.3.8 Classe NP-Complete

Definizione 5.3.8.1: NP-difficoltà

Un linguaggio L è detto essere **NP-difficile** se e solo se ogni altro linguaggio in NP è polinomialmente riducibile ad L. In simboli

$$\text{NP-Hard} := \{L \mid \forall L' \in \text{NP} \quad L' \leq_{\text{P}} L\}$$

Definizione 5.3.8.2: NP-completezza

Un linguaggio è detto essere **NP-completo** se e solo se è in NP ed è NP-difficile. In simboli

$$\text{NP-Complete} := \text{NP} \cap \text{NP-Hard}$$

Osservazione 5.3.8.1: "P versus NP problem"

Uno dei problemi del millennio consiste nello stabilire se P coincide o meno con NP.

Intuitivamente, è possibile interpretare tale problema aperto attraverso il seguente quesito: *per tutti i problemi per cui un algoritmo può verificare efficientemente la correttezza di una data soluzione, esiste un algoritmo che è anche in grado di trovare tale soluzione efficientemente?* Infatti, si ritiene che $\text{P} \neq \text{NP}$.

Proposizione 5.3.8.1: Implicazioni di P = NP

Si verifica che

$$\text{P} \cap \text{NP-Complete} \neq \varnothing \iff \text{P} = \text{NP}$$

Dimostrazione.

Prima implicazione. Per la Definizione 5.3.8.2, e per il Teorema 5.3.2.1, si ha che

$$\begin{cases} L \in \mathsf{NP\text{-}Complete} \\ L \in \mathsf{P} \end{cases} \iff \begin{cases} L \in \mathsf{NP} \\ \forall L' \in \mathsf{NP} \quad L' \leq_{\mathrm{P}} L \implies \forall L' \in \mathsf{NP} \quad L' \in \mathsf{P} \iff \mathsf{NP} \subseteq \mathsf{P} \\ L \in \mathsf{P} \end{cases}$$

Inoltre, poiché $\mathsf{P} \subseteq \mathsf{NP}$ per definizione stessa, segue la tesi.

Seconda implicazione. Trivialmente, dalle ipotesi e dalla Definizione 5.3.8.2 segue che

$$L \in \mathsf{NP\text{-}Complete} \implies L \in \mathsf{NP} = \mathsf{P}$$

□

Osservazione 5.3.8.2: $\emptyset$ e Σ^*

Si noti che, per un dato alfabeto Σ, si ha che $\emptyset, \Sigma^* \in \mathsf{P} \subseteq \mathsf{NP}$, ma nessuno dei due può essere NP-Hard, poiché:

- per assurdo, sia $L \leq_{\mathrm{P}} \emptyset$, e dunque esiste una funzione f polinomialmente calcolabile tale che
$$w \in L \iff f(w) \in \emptyset$$
allora, in particolre
$$w \in L \implies f(w) \in \emptyset$$
ma questo è impossibile perche $\emptyset$ non contiene alcuna stringa $\lightning$;
- per assurdo, sia $L \leq_{\mathrm{P}} \Sigma^*$, e dunque esiste una funzione f polinomialmente calcolabile tale che
$$w \in L \iff f(w) \in \Sigma^*$$
allora, in particolare
$$w \notin L \implies f(w) \notin \Sigma^*$$
ma questo è impossibile perché Σ^* contiene qualsiasi stringa definita su Σ $\lightning$.

Di conseguenza $\emptyset, \Sigma^* \notin \mathsf{NP\text{-}Complete}$, ed è dunque possibile ridefinire la classe dei linguaggi NP-completi come segue:

$$\mathsf{NP\text{-}Complete} := (\mathsf{NP} - \{\emptyset, \Sigma^*\}) \cap \mathsf{NP\text{-}Hard}$$

Teorema 5.3.8.1: Implicazioni di P = NP

Per un dato alfabeto Σ, si verifica che

$$\mathsf{P} = \mathsf{NP} \implies \mathsf{NP} - \{\emptyset, \Sigma^*\} = \mathsf{NP\text{-}Complete}$$

Dimostrazione. Si noti che

$$\mathsf{NP} - \{\emptyset, \Sigma^*\} = \mathsf{NP\text{-}Complete} := (\mathsf{NP} - \{\emptyset, \Sigma^*\}) \cap \mathsf{NP\text{-}Hard} \iff \mathsf{NP} - \{\emptyset, \Sigma^*\} \subseteq \mathsf{NP\text{-}Hard}$$

(si veda l'Osservazione 5.3.8.2 per chiarimenti) e dunque si ha che

$$\mathsf{NP} - \{\emptyset, \Sigma^*\} \subseteq \mathsf{NP\text{-}Hard} \iff \forall L, L' \in \mathsf{NP} - \{\emptyset, \Sigma^*\} \quad L \leq_{\mathrm{P}} L'$$

dunque la tesi equivale a dimostrare che tra ogni coppia di problemi in $\mathsf{NP} - \{\emptyset, \Sigma^*\}$ esiste una riduzione polinomiale; di conseguenza, assumendo che $\mathsf{P} = \mathsf{NP}$, per dimostrare la tesi è sufficiente mostrare che tale proprietà è verificata in $\mathsf{P} - \{\emptyset, \Sigma^*\}$.

Siano dunque $L, L' \in \mathsf{P} - \{\emptyset, \Sigma^*\}$; per loro stessa definizione, esistono due TM M ed M' tali da deciderli in tempo polinomiale rispettivamente. Allora, siano $x \in L'$ e $y \notin L'$ due stringhe, e si consideri una macchina di Turing $M_{x,y}$ che, data in input una stringa $w \in \Sigma^*$, computa come segue:

- $M_{x,y}$ computa $M_1(w)$;
- se M accetta, $M_{x,y}$ scrive x sul suo nastro e termina;
- se M' rifiuta, $M_{x,y}$ scrive y sul suo nastro e termina.

Allora, chiamando $f : \Sigma^* \to \Sigma^*$ la funzione computabile che $M_{x,y}$ descrive per sua stessa costruzione, si ha che

$$\begin{aligned} w \in L &\implies f(w) = x \in L' \\ w \notin L &\implies f(w) = y \notin L' \end{aligned}$$

e dunque

$$w \in L = L(M) \iff f(w) \in L' = L(M')$$

Infine, poiché il costo temporale di $M_{x,y}$ è interamente determinato da M, la quale computa in tempo polinomiale in ipotesi, segue che f è una riduzione di tempo polinomiale di L a L', dimostrando che $L \leq_{\mathrm{P}} L'$.

Si noti che non sarebbe possibile dimostrare questa proprietà per P, poiché se L e/o L' fossero $\emptyset$ e/o Σ^*, non sarebbe possibile prendere le stringhe x ed y utilizzate all'interno della dimostrazione. □

Teorema 5.3.8.2: NP-Complete mediante riduzione

Siano A e B due linguaggi definiti su un alfabeto Σ; se $A \leq_{\mathrm{P}} B$, A è NP-completo, e B è in NP, allora B è NP-completo. In simboli

$$\begin{cases} A \leq_{\mathrm{P}} B \\ A \in \mathsf{NP\text{-}Complete} \\ B \in \mathsf{NP} \end{cases} \implies B \in \mathsf{NP\text{-}Complete}$$

Dimostrazione. Per il Lemma 5.3.2.2, si ha che

$$\begin{cases} A \leq_{\mathrm{P}} B \\ A \in \mathsf{NP\text{-}Complete} \subseteq \mathsf{NP\text{-}Hard} \\ B \in \mathsf{NP} \end{cases} \implies \begin{cases} A \leq_{\mathrm{P}} B \\ \forall C \in \mathsf{NP} \quad C \leq_{\mathrm{P}} B \\ B \in \mathsf{NP} \end{cases} \implies B \in \mathsf{NP\text{-}Complete}$$

□

Definizione 5.3.8.3: Formula soddisfacibile

Una formula booleana è detta **soddisfacibile** se esiste un assegnamento dei suoi letterali tale per cui il valore della formula è pari a `1`.

Esempio 5.3.8.1 (Formule soddisfacibili)**.** Sia ϕ la seguente formula booleana:

$$(\overline{x}_1 \wedge x_2) \vee \overline{x}_4 \vee (x_3 \wedge \overline{x}_2 \wedge x_4) \vee x_5$$

è soddisfacibile attraverso ad esempio il seguente assegnamento

$$x_1 = 0,\ x_2 = 1,\ x_3 = 0,\ x_4 = 0,\ x_5 = 1$$

Teorema 5.3.8.3: Teorema di Cook-Levin

Sia SAT il linguaggio definito come:

$$SAT := \{\langle\phi\rangle \mid \phi \text{ formula soddisfacibile}\}$$

Allora, si verifica che

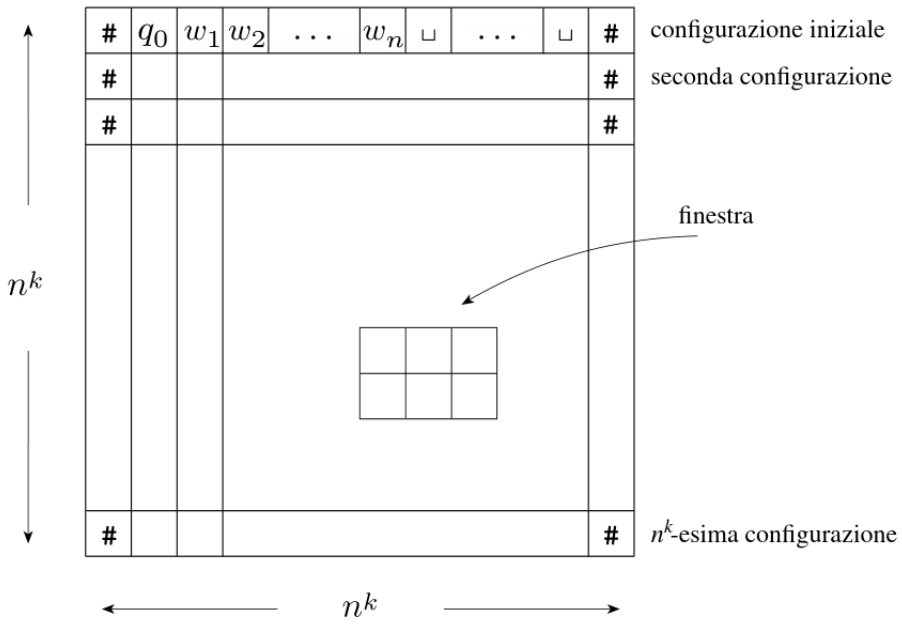
$$SAT \in \mathsf{NP\text{-}Complete}$$

Dimostrazione. Sia V una macchina di Turing che, data una codifica $\langle w, c\rangle$ in input, computa come segue:

- V interpreta w come $\langle\phi\rangle$, dove ϕ è una formula booleana, e c come assegnamento alle variabili di ϕ;
- V accetta se e solo se $\phi(c)$ è soddisfatta.

Allora, V risulta essere un verificatore di SAT, poiché per costruzione V accetta se e solo se $c \in \Sigma^*$ è un assegnamento per ϕ tale che $\langle\langle\phi\rangle, c\rangle \in L(V)$; inoltre, V è un verificatore polinomiale, e dunque per il Teorema 5.3.7.2 si ha che $SAT \in \mathsf{NP}$.

Sia $A \in \mathsf{NP}$, e dunque esiste una NTM $N = (Q, \Sigma, \Gamma, q_0, q_{\text{accept}}, q_{\text{reject}})$ che decide A in tempo polinomiale. Sia consideri ora una stringa w in input ad N, ed un *tableau* — *per ognuno dei rami di computazione di* N — definito come mostrato in figura:



Dunque, sull'i-esima riga del tableau è presente l'i-esima configurazione che la macchina assume durante la computazione di w, all'interno del ramo di computazione non deterministica scelto. Si noti che, poiché N computa in tempo polinomiale, il nastro non può avere più di n^k celle occupate, e non può rappresentare più di n^k configurazioni, dunque la tabella ha dimensione $n^k \times n^k$ (si assume che il tableau considerato sia quadrato anche se il nastro fosse utilizzato per meno di n^k celle, andando a considerare celle vuote nella porzione di nastro non utilizzata). Un tableau è detto *accettante* se almeno una sua riga contiene una configurazione accettante.

Verrà ora descritto un tableau accettante attraverso formule booleane, come segue:

- dato $C := Q \cup \Gamma \cup \{\#\}$, si definiscano variabili

$$\forall i, j \in \left[1, n^k\right], s \in C \quad x_{i,j,s} := \begin{cases} 1 & T_{i,j} = s \\ 0 & T_{i,j} \neq s \end{cases}$$

  si noti che è possibile immaginare $x_{i,j}$ come stringa booleana che attraverso codifica *one-hot encoding* mostra il carattere presente sulla cella $T_{i,j}$ del tableau considerato; si consideri allora la seguente formula booleana:

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

  per sua costruzione, se questa formula è pari a 1, la struttura dalla quale tale formula è stata generata rappresenta proprio un tableau definito come descritto in precedenza, poiché:

  - la prima parte della formula garantisce che almeno uno dei caratteri delle stringhe $x_{i,j}$ sia un $\mathtt{1}$;
  - la seconda parte della formula garantisce che sia presente solamente un $\mathtt{1}$ tra i caratteri delle stringhe $x_{i,j}$ (più precisamente, controlla che per ogni coppia di simboli $x_{i,j,s}$ ed $x_{i,j,t}$ con $s \neq t$, non si verifichi che $x_{i,j,s} = x_{i,j,t} = \mathtt{1}$);

- si consideri la seguente formula booleana:

$$\begin{gathered}\phi_{\text{start}} := x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\end{gathered}$$

  per sua costruzione, se questa formula è pari a $\mathtt{1}$, la struttura dalla quale tale formula è stata generata contiene la configurazione iniziale di N sulla sua prima riga;

- si consideri la seguente formula booleana:

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

  per sua costruzione, se questa formula booleana è pari a $\mathtt{1}$, la struttura dalla quale tale formula è stata generata contiene almeno una cella con q_{accept}, dunque almeno una configurazione accettante;

- una *finestra* 2×3 di un tableau è una porzione di tableau, composta da 6 celle di 2 righe diverse, in cui la prima riga contiene una porzione di configurazione corrente, e la seconda riga contiene una porzione della prossima configurazione; una finestra è detta *lecita* se non viola le azioni specificate dalla funzione di transizione δ di N; ad esempio, se la δ ammette le seguenti configurazioni

$$\begin{gathered}\delta(q_1, \mathtt{a}) = \{(q_1, \mathtt{b}, \mathrm{R})\} \\ \delta(q_1, \mathtt{b}) = \{(q_2, \mathtt{c}, \mathrm{L}), (q_2, \mathtt{a}, \mathrm{R})\}\end{gathered}$$

  i seguenti sono esempi di finestre lecite:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

  Si noti che, se un tableau T contiene la configurazione iniziale di N nella sua prima riga, ed ogni finestra di T è lecita, ogni riga di T è una configurazione che segue legittimamente dalla precedente (*è trivialmente verificabile per induzione, pertanto non*

ne verrà presentata una prova). Allora, si consideri la seguente formula booleana:

$$\phi_{\text{move}} := \bigwedge_{\substack{i\in\left[1,n^k\right)\\ j\in\left(1,n^k\right)}} \left(\bigvee_{\substack{(a_1,\ldots,a_6)\in C^6\\ \text{finestra lecita}}} \begin{array}{c} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{array} \right)$$

per sua costruzione, se questa formula booleana è pari a `1`, la struttura dalla quale tale formula è stata generata contiene esclusivamente finestre lecite (rispetto alla δ).

Si consideri ora la seguente formula booleana:

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

per costruzione ϕ è vera se e solo se la struttura considerata rappresenta un tableau accettante.

Dunque, si consideri una versione modificata di N, che su input w, ogni suo ramo di computazione non deterministica costruisce il tableau che rappresenta la propria computazione; inoltre, sia F una TM tale che, data in input w, computa come segue:

- se N ha accettato w, F restituisce in output uno dei tableau accettanti delle computazioni accettanti di N;
- se N ha rifiutato w — si noti che N è un decisore — F restituisce un tableau di una qualsiasi computazione.

Di conseguenza, per costruzione si ha che

$$w \in L(N) = A \iff \exists T \text{ tableau accettante} \iff f(w) = \langle\phi\rangle \in SAT$$

dimostrando che F descrive una riduzione $f : \Sigma^* \to \Sigma^*$ di A a SAT.

Si consideri ora il tempo impiegato da M per calcolare $f(w)$:

- ϕ_{cell} richiede tempo $O\left(n^k\right) \cdot O\left(n^k\right) \cdot |C| = O\left(n^{2k}\right)$ (poiché $|C| = O(1)$ perché non dipende da n);
- ϕ_{start} richiede tempo $O\left(n^k\right)$, poiché rappresenta una sola riga del tableau;
- ϕ_{accept} richiede tempo $O\left(n^{2k}\right)$, poiché richiede che sia presente q_{accept} all'interno del tableau;
- ϕ_{move} richiede tempo $O\left(n^k\right) \cdot O\left(n^k\right) \cdot O\left(\mathcal{W}\right) = O\left(n^{2k}\right)$, dove $\mathcal{W} := \binom{|C|}{6}$ poiché viene considerata ogni possibile finestra lecita per ogni cella del tableau.

Allora, f risulta essere una riduzione polinomiale di A a SAT, dunque $A \leq_{\text{P}} SAT$, provando che $SAT \in \mathsf{NP\text{-}Hard}$ poichè la riduzione non dipende dalla scelta di A. Di conseguenza, per quanto dimostrato in precedenza, segue la tesi, poiché

$$SAT \in \mathsf{NP} \cap \mathsf{NP\text{-}Hard} =: \mathsf{NP\text{-}Complete}$$

□

Definizione 5.3.8.4: Formula in 3CNF

Una formula booleana è detta essere in **3CNF**, se risulta essere l'*and logico* tra clausole, le quali devono essere costituite dall'*or logico* di 3 letterali.

Esempio 5.3.8.2 (Formule in 3CNF)**.** La seguente formula booleana è in 3CNF:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4} \vee x_6) \wedge (\overline{x_5} \vee \overline{x_2} \vee x_1) \wedge (x_3 \vee x_4 \vee x_1)$$

Teorema 5.3.8.4: $3SAT$ polinomialmente riducibile a $CLIQUE$

Sia $3SAT$ il linguaggio definito come segue:

$$3SAT := \{\langle\phi\rangle \mid \phi \text{ formula in 3CNF soddisfacibile}\}$$

Allora, si verifica che

$$3SAT \leq_{\mathrm{P}} CLIQUE$$

Dimostrazione. Si consideri la seguente TM F, che su input w, computa come segue:

- F interpreta w come $\langle\phi\rangle$, dove ϕ è una formula booleana in 3CNF;
- F costruisce un grafo $G = (V, E)$ come segue:
  - F crea nodi $x_i, \overline{x_i} \in V$ per ogni variabile x_i in ϕ (creando duplicati nel caso in cui la variabile compare più volte in ϕ);
  - F organizza i nodi in V in k-triple — dove k è il numero di clausole di ϕ — tali che ogni tripla rappresenti una clasuola di ϕ;
  - per ogni coppia di nodi x_i ed x_j, F crea l'arco $(x_i, x_j) \in E$ se e solo se $x_j \neq \overline{x_i}$, e x_i ed x_j appartengono a triple di V — e dunque clausole di ϕ — diverse;
- F restituisce $\langle G, k\rangle$ sul suo nastro e termina.

Sia $\langle\phi\rangle \in 3SAT$; allora ϕ è in 3CNF, e dunque essa è soddisfacibile se e solo se almeno un letterale di ogni clasuola è vero. Sia dunque $C = \{x_1, \ldots, x_k\} \subseteq V$ tale che ogni x_i sia il nodo corrispondente ad un letterale vero dell'i-esima clausola; allora, si noti che:

- i nodi in C nodi rappresentano letterali appartenenti a clausole diverse, dunque si troveranno necessariamente in triple di V differenti;
- in C non possono esistere due nodi x_i, x_j tali che $x_j = \overline{x_i}$ (ovvero che uno rappresenti il letterale opposto dell'altro), poiché C contiene solamente letterali veri, dunque $\overline{x_i}$ sarebbe falso.

Di conseguenza, per ogni coppia di nodi $x_i, x_j \in C$ esiste un arco $(x_i, x_j) \in E$ per costruzione di G. Questo dimostra che C è una k-clique, e dunque

$$\langle\phi\rangle \in 3SAT \implies f(\langle\phi\rangle) = \langle G, k\rangle \in CLIQUE$$

Sia $f(\langle\phi\rangle) = \langle G, k\rangle \in CLIQUE$, e dunque in G esiste una k-clique; allora, per costruzione di G, una sua k-clique non può contenere due vertici all'interno della stessa tripla, e se contiene x_i non può contenere $\overline{x_i}$. Di conseguenza, è possibile definire un assegnamento dei letterali rappresentati dai vertici della k-clique, affinché la formula ϕ che G rappresenta sia soddisfacibile, poiché i vertici della clique si trovano in triple differenti e non possono interferire tra loro. Questo dimostra che

$$f(\langle\phi\rangle) = \langle G, k\rangle \in CLIQUE \implies \langle phi\rangle \in 3SAT$$

In conclusione, si ha che

$$\langle\phi\rangle \in 3SAT \iff f(\langle\phi\rangle) = \langle G, k\rangle \in CLIQUE$$

provando che f rappresenta una riduzione da $3SAT$ a $CLIQUE$, e poiché F esegue esclusivamente operazioni in tempo polinomiale, segue che

$$3SAT \leq_{\mathrm{P}} CLIQUE$$

□

Teorema 5.3.8.5: $3SAT$ in NP-Complete

Si verifica che

$$3SAT \in \mathsf{NP\text{-}Complete}$$

Dimostrazione. Omessa. □

Corollario 5.3.8.1: $CLIQUE$ in NP-Complete

Si verifica che

$$CLIQUE \in \mathsf{NP\text{-}Complete}$$

Dimostrazione. Per il Teorema 5.3.8.5, il Teorema 5.3.7.4, ed il Teorema 5.3.8.2, segue la tesi. □

5.3.9 Classe coNP

Definizione 5.3.9.1: Classe coNP

La classe coNP è definita come segue:

$$\mathsf{coNP} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{NP}\}$$

Proposizione 5.3.9.1: Relazione tra P, coNP ed EXP

Si verifica che

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

Dimostrazione. Per il Teorema 5.3.3.1, la Proposizione 5.3.7.1, il Teorema 5.3.5.1 si ha che

$$A \in \mathsf{P} \iff \overline{A} \in \mathsf{P} \implies \overline{A} \in \mathsf{NP} \iff A \in \mathsf{coNP} \iff \overline{A} \in \mathsf{NP} \implies \overline{A} \in \mathsf{EXP} \iff A \in \mathsf{EXP}$$

□

Teorema 5.3.9.1: Implicazioni di P = NP

Si verifica che

$$\mathsf{P} = \mathsf{NP} \implies \mathsf{P} = \mathsf{coNP}$$

Dimostrazione. Per il Proposizione 5.3.9.1, si ha che $\mathsf{P} \subseteq \mathsf{coNP}$; inoltre, preso un linguaggio $L \in \mathsf{coNP}$, si ha che

$$L \in \mathsf{coNP} \iff \overline{L} \in \mathsf{NP} = \mathsf{P}$$

e dunque segue la tesi per doppia inclusione. □

Osservazione 5.3.9.1: "NP versus coNP "

Un altro noto problema aperto riguarda stabilire se NP coincide o meno con coNP.

Corollario 5.3.9.1: Implicazioni di P = NP

Si verifica che

$$\mathsf{P} = \mathsf{NP} \implies \mathsf{NP} = \mathsf{coNP}$$

Dimostrazione. Per il Teorema 5.3.9.1, si ha che

$$\mathsf{P} = \mathsf{NP} \implies \mathsf{NP} = \mathsf{P} = \mathsf{coNP}$$

□

Teorema 5.3.9.2: coNP mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{P}} B \\ B \in \mathsf{coNP} \end{cases} \implies A \in \mathsf{coNP}$$

Dimostrazione. Per il Lemma 5.3.2.1, e per il Teorema 5.3.7.1, si ha che

$$\begin{cases} A \leq_{\mathrm{P}} B \\ B \in \mathsf{coNP} \end{cases} \iff \begin{cases} \overline{A} \leq_{\mathrm{P}} \overline{B} \\ \overline{B} \in \mathsf{NP} \end{cases} \implies \overline{A} \in \mathsf{NP} \iff A \in \mathsf{coNP}$$

□

Corollario 5.3.9.2: coNP mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{P}} B \\ A \notin \mathsf{coNP} \end{cases} \implies B \notin \mathsf{coNP}$$

Dimostrazione. Per assurdo, sia $B \in \mathsf{coNP}$; allora, poiché $A \leq_{\mathrm{P}} B$, per il Teorema 5.3.9.2 si ha che $A \in \mathsf{coNP}$ ↯. □

Corollario 5.3.9.3: Implicazioni di NP = coNP

Si verifica che

$$\mathsf{coNP} \cap \mathsf{NP\text{-}Complete} \neq \varnothing \iff \mathsf{NP} = \mathsf{coNP}$$

Dimostrazione.

Prima implicazione. Sia $L \in \mathsf{coNP} \cap \mathsf{NP\text{-}Complete}$; allora, per il Teorema 5.3.9.2, si ha che

$$\begin{cases} \forall L' \in \mathsf{NP} \quad L' \leq_{\mathrm{P}} L \\ L \in \mathsf{coNP} \end{cases} \implies \forall L' \in \mathsf{NP} \quad L' \in \mathsf{coNP} \iff \mathsf{NP} \subseteq \mathsf{coNP}$$

viceversa, per il Lemma 5.3.2.1 e per il Teorema 5.3.7.1, si ha che

$$\begin{cases} \forall L' \in \mathsf{NP} \quad L' \leq_{\mathrm{P}} L \\ L \in \mathsf{coNP} \end{cases} \iff \begin{cases} \forall \overline{L'} \in \mathsf{coNP} \quad \overline{L'} \leq_{\mathrm{P}} \overline{L} \\ \overline{L} \in \mathsf{NP} \end{cases} \implies \forall \overline{L'} \in \mathsf{coNP} \quad \overline{L'} \in \mathsf{NP} \iff \mathsf{coNP} \subseteq \mathsf{NP}$$

dunque segue la tesi per doppia inclusione.

Seconda implicazione. La tesi segue dal fatto che $\mathsf{NP\text{-}Complete} \subseteq \mathsf{NP} = \mathsf{coNP}$ per definizione di NP-Complete.

□

5.3.10 Classe coNP-Complete

Definizione 5.3.10.1: coNP-difficoltà

Un linguaggio L è detto essere **coNP-difficile** se e solos e ogni altro linguaggio in coNP è polinomialmente riducibile ad L. In simboli

$$\mathsf{coNP\text{-}Hard} := \{L \mid \forall L' \in \mathsf{coNP} \quad L' \leq_{\mathrm{P}} L\}$$

Teorema 5.3.10.1: Definizione alternativa di coNP-Hard

Si verifica che

$$\mathsf{coNP\text{-}Hard} = \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{NP\text{-}Hard}\}$$

Dimostrazione. Si noti che, per dimostare la tesi, è sufficiente dimostrare che

$$L \in \text{coNP-Hard} \iff \overline{L} \in \text{NP-Hard}$$

e dunque, per il Lemma 5.3.2.1, si ha che

$$L \in \text{coNP-Hard} \iff \forall L' \in \text{coNP} \quad L' \leq_{\text{P}} L \iff$$
$$\iff \forall \overline{L'} \in \text{NP} \quad \overline{L'} \leq_{\text{P}} \overline{L} \iff \overline{L} \in \text{NP-Hard}$$

□

Definizione 5.3.10.2: coNP-completezza

Un linguaggio è detto essere **coNP-completo** se e solo se è in coNP ed è coNP-difficile. In simboli

$$\text{coNP-Complete} := \text{coNP} \cap \text{coNP-Hard}$$

Teorema 5.3.10.2: Definizione alternativa di coNP-Complete

Si verifica che

$$\text{coNP-Complete} = \{L \in \text{DEC} \mid \overline{L} \in \text{NP-Complete}\}$$

Dimostrazione. Si noti che, per dimostrare la tesi, è sufficiente dimostrare che

$$L \in \text{coNP-Complete} \iff \overline{L} \in \text{NP-Complete}$$

e dunque, per il Teorema 5.3.10.1, si ha che

$$L \in \text{coNP-Complete} \iff L \in \text{coNP} \cap \text{coNP-Hard} \iff \overline{L} \in \text{NP} \cap \text{NP-Hard} \iff \overline{L} \in \text{NP-Complete}$$

□

5.3.11 Classe NEXP

Definizione 5.3.11.1: Classe NEXP

Si definisce **classe dei linguaggi decidibili non deterministicamente in tempo esponenziale** da una NTM il seguente insieme

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

5.3.12 Teorema di gerarchia di tempo

Definizione 5.3.12.1: Funzione tempo-costruibile

Una funzione $f : \mathbb{N} \to \mathbb{N}$ è detta essere **tempo-costruibile** se $f(n)$ è almeno $O(n \log(n))$, e se la funzione che mappa 1^n nella rappresentazione binaria di $f(n)$ è computabile in tempo $O(f(n))$.

Funzioni a valori non interi vengono arrotondate all'intero successivo più piccolo al fine di renderle tempo-costruibili.

Esempio 5.3.12.1 (Funzioni tempo-costruibili)**.** Si consideri la funzione $f(n) = n^2$, la quale è almeno $O(n \log(n))$; inoltre, sia M una TM che, data in input la stringa 1^n, computa come segue:

- M conta il numero di 1 presenti sul suo nastro;
- M restituisce $n \cdot n = n^2$ sul suo nastro.

Allora, poiché entrambe le operazioni che M effettua richiedono tempo $O(n)$, e poiché M termina avendo n^2 sul suo nastro, essa è una TM tale che la funzione $g : \Sigma^* \to \Sigma^* : 1^n \mapsto n^2$ sia computabile in tempo $O(n)$, che è certamente $O(n^2)$. Questo dimostra che $f(n) = n^2$ è tempo-costruibile.

Teorema 5.3.12.1: Teorema di gerarchia di tempo

Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione tempo-costruibile; allora, esiste un linguaggio L decidibile in tempo $O(f(n))$ ma non in tempo $o\left(\frac{f(n)}{\log(f(n))}\right)$.

Dimostrazione. Omessa. □

Corollario 5.3.12.1: Teorema di gerarchia di tempo

Siano $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ due funzioni, dove $f_1(n) = o\left(\frac{f_2(n)}{\log(f_2(n))}\right)$ ed f_2 è tempo-costruibile; allora, si verifica che

$$\mathsf{DTIME}(f_1(n)) \subsetneq \mathsf{DTIME}(f_2(n))$$

Dimostrazione. Siano $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ due funzioni, tali che

$$f_1(n) = o\left(\frac{f_2(n)}{\log(f_2(n))}\right) \implies f_1(n) = O\left(\frac{f_2(n)}{\log(f_2(n))}\right)$$

(si veda la Osservazione 5.1.1.1) ed f_2 sia tempo-costruibile; allora, per definizione di f_1 ed f_2, e per il Teorema 5.3.12.1, si ha che

$$\mathsf{DTIME}(f_1(n)) \subsetneq \mathsf{DTIME}\left(\frac{f_2(n)}{\log(f_2(n))}\right) \subset \mathsf{DTIME}(f_2(n))$$

□

Proposizione 5.3.12.1: Relazione tra P e EXP

Si verifica che

$$\mathsf{P} \subsetneq \mathsf{EXP}$$

Dimostrazione. Si consideri la funzione $f(n) = 2^n$, la quale è almeno $O(n\log(n))$; inoltre, sia M una TM data in input la stringa 1^n, computa come segue:

- M conta il numero di 1 presenti sul nastro;
- M calcola 2^n
- M termina restituendo il risultato sul nastro.

Allora, poiché calcolare 2^n richiede tempo al massimo $O(2^n)$ (si noti che l'efficienza varia in base all'algoritmo utilizzato), segue che la funzione $g : \Sigma^* \to \Sigma^* : 1^n \mapsto 2^n$ che M rappresenta è computabile in tempo $O(2^n)$. Questo dimostra che $f(n) = 2^n$ è tempo-costruibile.

Si noti che

$$\forall k \in \mathbb{N} \quad \lim_{n\to+\infty} \frac{n^k}{\frac{2^n}{\log(2^n)}} = \lim_{n\to+\infty} \frac{n^k}{\frac{2^n}{n}} = \lim_{n\to+\infty} \frac{n^{k+1}}{2^n} = \lim_{n\to+\infty} \frac{n^{k+1}}{n^{\log(n)}} \cdot \frac{n^{\log(n)}}{2^n} = \lim_{n\to+\infty} \frac{n^{k+1}}{n^{\log(n)}} \cdot \frac{2^{\log^2(n)}}{2^n} = 0$$

e dunque si ha che

$$\forall k \in n^k \quad n^k = o\left(\frac{2^n}{\log(2^n)}\right)$$

e poiché 2^n è tempo-costruibile per quanto detto in precedenza, per il Corollario 5.3.12.1 segue che

$$\forall k \in \mathbb{N} \quad \mathsf{DTIME}\left(n^k\right) \subsetneq \mathsf{DTIME}\left(2^n\right) \subsetneq \mathsf{EXP}$$

implicando che

$$\mathsf{P} := \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(n^k\right) \subsetneq \mathsf{DTIME}\left(2^n\right) \subsetneq \bigcup_{h\in\mathbb{N}} \mathsf{DTIME}\left(2^{n^h}\right) =: \mathsf{EXP}$$

□

6 Complessità di spazio

6.1 Complessità di spazio di macchine di Turing

6.1.1 Macchine di Turing

Osservazione 6.1.1.1: Modello di TM per compl. di spazio

Per il calcolo della complessità di spazio, verrà utilizzato un modello di TM differente dalle TM utilizzate fin'ora, al fine di non penalizzare le macchine considerate rispetto allo spazio utilizzato per salvare l'input sul proprio nastro; di conseguenza, verranno considerate TM fornite di 2 nastri, dove

- il primo nastro conterrà esclusivamente l'input, verrà utilizzato in modalità *read-only*, e prenderà il nome di *input tape*
- il secondo nastro verrà utilizzato per effettuare le computazioni (dunque di default conterrà solo $\sqcup$), e prenderà il nome di *working tape*

Definizione 6.1.1.1: Configurazione (TM per compl. di spazio)

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ una macchina di Turing definita come descritto nell'Osservazione 6.1.1.1; con il simbolismo

$$u\ q\ v;\ i$$

per certi $u, v, \in \Gamma^*, q \in Q, i \in [1, |w|]$ — dove w è l'input della TM, posto sull'*input tape* — si denota una **configurazione di** M, dove:

- q rappresenta lo stato attuale della computazione dell'input w;
- uv è la stringa che descrive il *working tape* (si assume che, dopo l'ultimo simbolo di v, il *working tape* contenga solo $\sqcup$);

- la testina del *working tape* punta al primo simbolo di v;
- la testina dell'*input tape* punta all'i-esimo carattere di w, dunque si trova sulla cella dell'*input tape* contenente w_i.

Definizione 6.1.1.2: Complessità di spazio di una TM

Sia D un decisore definito come descritto nell'Osservazione 6.1.1.1; si definisce **complessità di spazio** di D la funzione $f : \mathbb{N} \to \mathbb{N}$ tale che $f(n)$ sia il numero massimo di celle che D utilizza per processare una stringa di lunghezza n.

6.1.2 Macchine di Turing multinastro

Teorema 6.1.2.1: Relazione tra TM e TM multinastro

Sia $s(n)$ una funzione tale che $s(n) \geq n$, e sia M una macchina di Turing multinastro avente spazio $s(n)$; allora, esiste una TM M', equivalente ad M, avente spazio $O(s(n))$.

Dimostrazione. Data una TM multinastro M che computa in $s(n) \geq n$, è possibile realizzare una TM M' ad essa equivalente utilizzando la costruzione definita all'interno della dimostrazione del Proposizione 3.2.2.1, e poiché il numero di nastri di M è costante, segue la tesi. □

6.2 Classi di complessità di spazio

6.2.1 Classe DSPACE

Definizione 6.2.1.1: Classe DSPACE

Data una funzione $s : \mathbb{N} \to \mathbb{R}^+$, si definisce **classe di complessità di spazio** $\mathsf{DSPACE}(s(n))$ l'insieme dei linguaggi decidibili da una macchina di Turing definita come descritto nell'Osservazione 6.1.1.1 in $O(s(n))$.

Teorema 6.2.1.1: Limite di spazio

Si verifica che

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$$

Dimostrazione. Per definizione, si ha che $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$; si consideri dunque un linguaggio in $\mathsf{NTIME}(f(n))$ per qualche funzione $f : \mathbb{N} \to \mathbb{R}^+$; di conseguenza, il decisore di tale linguaggio effettua al massimo $f(n)$ passi, e poiché nel caso peggiore la macchina non fa altro che spostare la testina verso destra, tale macchina potrebbe

effettuare solamente $f(n)$ spostamenti verso destra, e dunque potrebbe lavorare con al massimo $f(n)$ celle; allora, segue la tesi. □

Teorema 6.2.1.2: Limite di tempo

Si verifica che

$$\mathsf{DSPACE}(f(n)) \subseteq \mathsf{DTIME}\left(n \cdot 2^{O(f(n))}\right)$$

Dimostrazione. Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ un decisore definito come descritto nell'Osservazione 6.1.1.1, che computa in spazio $f(n)$; allora, si ha che:

- il numero di stati possibili di una configurazione di M è $|Q|$;
- il numero di posizioni possibili che la testina dell'*input tape* di M può assumere è n;
- il numero di posizioni possibili che la testina del *working tape* di M può assumere è $f(n)$;
- il numero di stringhe possibili sul *working tape* di M è pari a $|\Gamma|^{f(n)}$;

e dunque il numero di configurazioni di M possibili è pari a

$$\begin{aligned} n \cdot |Q| \cdot f(n) \cdot |\Gamma|^{f(n)} = n \cdot O\left(f(n) \cdot |\Gamma|^{f(n)}\right) = n \cdot O\left(f(n) \cdot 2^{O(f(n))}\right) = \\ = n \cdot O\left(2^{\log(f(n)) + O(f(n))}\right) = n \cdot O\left(2^{O(f(n))}\right) = n \cdot 2^{O(f(n))} \end{aligned}$$

(si veda l'Osservazione 5.1.1.3 per chiarimenti). Si noti che il numero di configurazioni di M possibili coincide con il numero massimo di passi possibili che M può effettuare durante una sua esecuzione, poiché M è un decisore e dunque non può ripetere le configurazioni, altrimenti andrebbe in loop; di conseguenza, segue la tesi.

Infine, si noti che non è possibile accorpare n nell'O-grande, poiché

$$f(n) < \log(n) \iff 2^{f(n)} < 2^{\log(n)} = n$$

e dunque se $f(n)$ è asintoticamente inferiore a $\log(n)$, n è asintoticamente superiore a $2^{f(n)}$. □

Corollario 6.2.1.1: Limite di tempo

Si verifica che

$$f(n) \geq \log(n) \implies \mathsf{DSPACE}(f(n)) \subseteq \mathsf{DTIME}\left(2^{O(f(n))}\right)$$

Dimostrazione. La tesi segue direttamente dal Teorema 6.2.1.2, poiché

$$f(n) \geq \log(n) \iff 2^{f(n)} \geq 2^{\log(n)} = n \implies n \cdot 2^{O(f(n))} = 2^{O(f(n))}$$

□

6.2.2 Classe L

Definizione 6.2.2.1: Classe L

Si definisce **classe dei linguaggi decidibili in spazio logaritmico** da una TM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{L} := \mathsf{DSPACE}(\log(n))$$

Esempio 6.2.2.1 (Linguaggi in L)**.** Si considerino il linguaggio

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

e una macchina di Turing M — definita come descritto nell'Osservazione 6.1.1.1 — che, data in input una stringa w, computa come segue:

- M controlla che non siano presenti `1` seguiti da `0` all'interno dell'*input tape*;
- M definisce un contatore binario sul *working tape*;
- M scorre l'input, e se legge `0` incrementa il contatore, mentre se legge `1` lo decrementa;
- M accetta se e solo se, alla fine dell'input, il contatore è a 0.

Dunque, per costruzione M decide L, e poiché un numero a in base k ha $O(\log_k(a)) = O(\log(a))$ cifre (si veda l'Osservazione 5.1.1.2 per chiarimenti), M decide in L con spazio

$$O(\log(|w|)) = O(\log(n)) \implies L \in \mathsf{L}$$

Osservazione 6.2.2.1: Classe L

La classe di problemi descritti dai linguaggi in L è particolarmente interessante, poiché le funzioni logaritmiche sono tali per cui $f(n) < n$.

Proposizione 6.2.2.1: Relazione tra L e P

Si verifica che

$$\mathsf{L} \subseteq \mathsf{P}$$

Dimostrazione. La tesi segue direttamente dal Corollario 6.2.1.1, poiché

$$f(n) = \log(n) \implies \mathsf{L} := \mathsf{DSPACE}(\log(n)) \subseteq \mathsf{DTIME}\left(2^{O(\log(n))}\right) = \mathsf{DTIME}(O(n)) \subseteq \mathsf{P}$$

□

Definizione 6.2.2.2: Trasduttore di spazio logaritmico

Un **trasduttore di spazio logaritmico** è una macchina di Turing definita come descritto nell'Osservazione 6.1.1.1, che presenta un ulteriore nastro, detto *output tape*, utilizzato esclusivamente in modalità *read-once* per scrivere l'output.

Definizione 6.2.2.3: Funzione calcolabile in spazio logaritmico

Una funzione $f : \Sigma^* \to \Sigma^*$ è detta **funzione calcolabile in spazio logaritmico** se esiste un trasduttore di spazio logaritmico che, su qualsiasi input $w \in \Sigma^*$, se si ferma, termina avendo solo $f(w)$ sul suo *output tape*.

Osservazione 6.2.2.2: Trasduttori di spazio logaritmico

L'uso di trasduttori di spazio logaritmico all'interno della Definizione 6.2.2.3 permette di non penalizzare le TM considerate sullo spazio per rappresentare l'output.

Definizione 6.2.2.4: Riduzione in spazio log. mediante funzione

Siano A e B due linguaggi definiti su un alfabeto Σ; A è detto essere **riducibile in spazio logaritmico mediante funzione a** B se esiste una funzione calcolabile in spazio logaritmico $f : \Sigma^* \to \Sigma^*$ *che termina su ogni input* tale che

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

Tale relazione è indicata attraverso il simbolismo

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

e la funzione f prende il nome di **riduzione in spazio logaritmico di** A **a** B.

Lemma 6.2.2.1: Riducibilità in spazio log. dei complementi

Siano A e B due linguaggi definiti su un alfabeto Σ; allora, si ha che

$$A \leq_{\mathrm{L}} B \iff \overline{A} \leq_{\mathrm{L}} \overline{B}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Lemma 4.1.1.1, pertanto verrà omessa. □

Lemma 6.2.2.2: Riducibilità di spazio log. e polinomiale

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

$$A \leq_{\mathrm{L}} B \implies A \leq_{\mathrm{P}} B$$

Dimostrazione. Siano A e B due linguaggi definiti su uno stesso alfabeto, tali che $A \leq_{\mathrm{L}} B$, e dunque per definizione esiste una funzione $f : \Sigma^* \to \Sigma^*$ calcolabile in spazio logaritmico; allora, per definizione di f, esiste un trasduttore di spazio logaritmico T che, su qualsiasi input $w \in \Sigma^*$, se si ferma, termina avendo solo $f(w)$ sul suo *output tape*. Si noti che T computa in spazio logaritmico, e dunque poiché $\mathsf{L} \subseteq \mathsf{P}$ per il Proposizione 6.2.2.1, esiste una macchina di Turing T' di tempo polinomiale equivalente a T, che di conseguenza è

tale per cui f risulta dunque essere anche polinomialmente calcolabile; in conclusione, f è tale che $A \leq_{\mathrm{P}} B$ per definizione di riduzione polinomiale mediante funzione. □

Teorema 6.2.2.1: L mediante riduzione

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

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

Dimostrazione. Poiché $B \in \mathsf{L}$ in ipotesi, esiste un decisore M — definito come descritto nell'Osservazione 6.1.1.1 — tale che M decide B in spazio logaritmico; inoltre, poiché $A \leq_{\mathrm{L}} B$, esiste una funzione calcolabile in spazio logaritmico $f : \Sigma^* \to \Sigma^*$ tale per cui A riducibile in spazio logaritmico a B. Sia dunque N una macchina di Turing — definita come descritto nell'Osservazione 6.1.1.1 — che, data in input una stringa $w \in \Sigma^*$, computa come segue:

- N computa $f(w)$, attraverso il trasduttore di spazio logaritmico tale per cui f è calcolabile, *ma salvando un simbolo alla volta sul suo working tape*;
- N simula M avente i vari simboli di $f(w)$ uno ad uno come input, tenendo traccia di dove si trova la testina di M, ed ogni qual volta che M necessita di conoscere il prossimo simbolo di $f(w)$, N pulisce il proprio *working tape* e calcola il $f(w)$ dall'inizio fino ad ottenere il prossimo simbolo necessario;
- N accetta se e solo se M accetta.

Allora, per costruzione di N, e per riducibilità in spazio logaritmico 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 in spazio logaritmico, poiché f è calcolabile in spazio logaritmico, e M è un decisore di B in spazio logaritmico anch'esso.

Si noti che il motivo per cui è necessario utilizzare una macchina definita come descritto, e non è sufficiente sfruttrare l'approccio di dimostrazione utilizzato per il Teorema 4.2.1.1, è che la calcolabilità in spazio logaritmico di f implica che esista un trasduttore di spazio logaritmico T tale da calcolare f in spazio logaritmico. Di conseguenza, lo spazio utilizzato per rappresentare $f(w)$ (per qualche $w \in \Sigma^*$) stesso non è considerato nella sua definizione, ma N deve invece essere una TM definita come descritto nell'Osservazione 6.1.1.1 per definizione di appartenenza ad L. Dunque, calcolare preventivamente $f(w)$ potrebbe eccedere lo spazio logaritmico per rappresentare $f(w)$ sul nastro di N, ed è dunque necessario computare come descritto. □

Corollario 6.2.2.1: L mediante riduzione

Siano A e B due linguaggi definiti su un alfabeto Σ; se $A \leq_{\mathsf{L}} B$, e A non è in L, allora B non è in L. In simboli

$$\begin{cases} A \leq_{\mathsf{L}} B \\ A \notin \mathsf{L} \end{cases} \implies B \notin \mathsf{L}$$

Dimostrazione. Per assurdo, si $B \in \mathsf{L}$; allora, poiché $A \leq_{\mathsf{L}} B$, per il Teorema 6.2.2.1, si ha che $A \in \mathsf{L}$ ↯. □

Lemma 6.2.2.3: Transitività della riducibilità in spazio log.

Siano A, B e C tre linguaggi definiti su un alfabeto Σ; allora, si ha che

$$\begin{cases} A \leq_{\mathsf{L}} B \\ B \leq_{\mathsf{L}} C \end{cases} \implies A \leq_{\mathsf{L}} C$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Lemma 4.1.1.2, con la differenza che la TM M considerata deve essere definita come descritto all'interno dell'Osservazione 6.1.1.1, e deve cmputare come descritto nella dimostrazione del Teorema 6.2.2.1, poiché anche in questo caso rappresentare $f(w)$ (per qualche $w \in \Sigma^*$) potrebbe richiedere più spazio di $O(\log(n))$. □

Proposizione 6.2.2.2: Operazioni in spazio logaritmico

In spazio logaritmico, è possibile effettuare le seguenti operazioni:

- stabilire la lunghezza dell'input:
  - per calcolare la lunghezza dell'input sull'*input tape* è sufficiente utilizzare un contatore sul *working tape* e contare di quanti simboli è composto l'input, dunque richiede spazio $O(\log(n))$
- salvare il valore di variabili;
- effettuare somme tra 2 numeri:
  - grazie al Teorema 6.1.2.1, si può assumere che la macchina sia dotata di più *working tape* (ammesso che il numero di nastri sia in $O(1)$, e dunque non dipenda dalla dimensione dell'input);
  - dunque, per effettuare una somma tra 2 numeri a e b, è sufficiente avere salvato a su un nastro, b salvato su un altro, e successivamente incrementare il nastro contenente a e decrementare il nastro contenente b, un passo alla volta, fino a che sul nastro di b non vi è 0;
  - il risultato della somma sarà contenuto sul nastro di a;

  - lo spazio richiesto dalla somma è dunque lo spazio richiesto per rappresentare $a + b$, ovvero $O(\log(a + b))$;
- effettuare differenze tra 2 numeri:
  - l'operazione è analoga alla somma (assumendo che $a > b$), ma richiede di decrementare il nastro del primo numero al posto di incrementare;
  - come per la somma, il risultato della differenza sarà contenuta sul nastro di b;
  - lo spazio richiesto dalla differenza è dunque lo spazio richiesto per rappresentare a, ovvero $O(\log(a))$;
- effettuare prodotti tra 2 numeri:
  - la logica è analoga alla somma (sono necessari 4 nastri), pertanto verrà omessa;
  - lo spazio richiesto dal prodotto è dunque lo spazio richiesto per rappresentare $a \cdot b$, ovvero $O(\log(a \cdot b)) = O(\log(a) + \log(b)) = O(\max(\log(a), \log(b)))$
- effettuare potenze tra 2 numeri:
  - la logica è analoga al prodotto (sono necessari 5 nastri), pertanto verrà omessa;
  - lo spazio richiesto dalla potenza è dunque lo spazio richiesto per rappresentare a^b, ovvero $O\left(\log\left(a^b\right)\right) = O\left(b \cdot \log(a)\right)$

Teorema 6.2.2.2: *PALINDROMES* **in L**

Sia $PALINDROMES$ il linguaggio definito come segue:

$$PALINDROMES := \{w \mid \forall i \in [1, |w|] \quad w_i = w_{n-i+1}\}$$

Allora, si verifica che

$$PALINDROMES \in \mathsf{L}$$

Dimostrazione. Sia M una TM definita come descritto nell'Osservazione 6.1.1.1 che, data in input una stringa w, computa come segue:

- M determina $n := |w|$ attraverso un contatore;
- per ogni $i \in [1, n]$, se $w_i \neq w_{n-i+1}$, M rifiuta; si noti che, dato i, M può computare $n - i + 1$ come segue:
  - M scrive n su un primo nastro;
  - M scrive i su un secondo nastro;
  - M calcola $n - i$ sul primo nastro, portando il secondo a 0 (come descritto nella Proposizione 6.2.2.2);

  - M incrementa il primo nastro di 1;
- se M ha raggiunto la fine dell'input, accetta.

Dunque, per la Proposizione 6.2.2.2, M richiede spazio logaritmico, e poiché per costruzione decide $PALINDROMES$, segue la tesi. □

Teorema 6.2.2.3: $PATH$ in DSPACE$(\log^2(n))$

Si verifica che

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

Dimostrazione. Si consideri il seguente algoritmo:

```
1: function findPath(G, x, y, k)
2:     if k == 0 then
3:         if (x, y) ∈ E ∨ x == y then
4:             return True
5:         else
6:             return False
7:         end if
8:     else if k > 0 then
9:         for w ∈ V do
10:            if { findPath(G, x, w, k − 1) == True
                   { findPath(G, w, y, k − 1) == True   then
11:                return True
12:            end if
13:        end for
14:        return False
15:    end if
16: end function
```

Si consideri una TM M in grado di eseguire l'algoritmo `findPath`, la quale esiste per l'Osservazione 3.3.1.1. Allora, sia M' una TM— avente M come sua sottoprocedura — che, data in input una codifica $\langle G, s, t\rangle$, computa come segue:

- M' interpreta $G = (V, E)$ come un grafo diretto, ed $s, t \in V$ come due suoi nodi;
- M' esegue `findPath` $(G, s, t, \lceil \log(|V|)\rceil)$ su M;
- M' accetta se e solo se M restituisce `True` sul suo nastro.

L'algoritmo `findPath`, per sua definizione, dato un grafo diretto G, due suoi nodi $x, y \in G$, ed un intero $k \in \mathbb{N}$, restituisce `True` se e solo se in G esiste un cammino $x \to y$ lungo al massimo 2^k, in quanto `findPath` effettua 2 chiamate ricorsive per ogni livello di ricorsione, e dunque l'albero di chiamate ricorsive ha altezza al massimo k e base al massimo 2^k (si noti che non è *esattamente* 2^k poiché w potrebbe essere anche x o y). Dunque M' utilizza `findPath` per decidere $PATH$, ponendo $k = \lceil \log(|V|)\rceil$ poiché un cammino in un grafo ha lunghezza $O(|V|)$ ed infatti

$$k = \lceil \log(|V|)\rceil \implies O\left(2^k\right) = O\left(2^{\lceil \log(|V|)\rceil}\right) = O(|V|)$$

Allora, poiché in ogni livello di ricorsione è necessario salvare un numero costante di variabili, ovvero $|V|, s, t, x, y, k$, salvarle ha costo $O(\log(n))$ (si veda l'Proposizione 6.2.2.2); inoltre, il numero di livelli di ricosione è pari all'altezza dell'albero di chiamate ricorsive, ovvero $O(k) = O(\log(|V|)) = O(\log(n))$. Dunque, segue che lo spazio che la TM M' utilizza per computare è pari a

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

□

6.2.3 Classe coL

Definizione 6.2.3.1: Classe coL

La classe coL è definita come segue:

$$\mathsf{coL} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{L}\}$$

Teorema 6.2.3.1: Relazione tra L e coL

Si verifica che

$$\mathsf{L} = \mathsf{coL}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.3.1, pertanto verrà omessa. □

6.2.4 Classe PSPACE

Definizione 6.2.4.1: Classe PSPACE

Si definisce **classe dei linguaggi decidibili in spazio polinomiale** da una TM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{PSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{DSPACE}\left(n^k\right)$$

6.2.5 Classe coPSPACE

Definizione 6.2.5.1: Classe coPSPACE

La classe coPSPACE è definita come segue:

$$\mathsf{coPSPACE} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{PSPACE}\}$$

Teorema 6.2.5.1: Relazione tra PSPACE e coPSPACE

Si verifica che

$$\mathsf{PSPACE} = \mathsf{coPSPACE}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.3.1, pertanto verrà omessa. □

6.2.6 Classe PSPACE-Complete

Definizione 6.2.6.1: PSPACE-difficoltà

Un linguaggio L è detto essere **PSPACE-difficile** se e solo se ogni altro linguaggio in PSPACE è riducibile in spazio logaritmico ad L. In simboli

$$\mathsf{PSPACE\text{-}Hard} := \{L \mid \forall L' \in \mathsf{PSPACE} \quad L' \leq_{\mathrm{L}} L\}$$

Definizione 6.2.6.2: PSPACE-completezza

Un linguaggio è detto essere **PSPACE-completo** se e solo se è in PSPACE ed è PSPACE-difficile. In simboli

$$\mathsf{PSPACE\text{-}Complete} := \mathsf{PSPACE} \cap \mathsf{PSPACE\text{-}Hard}$$

Proposizione 6.2.6.1: Proprietà di PSPACE-Hard

Si verifica che

$$\mathsf{NP\text{-}Hard} \subseteq \mathsf{PSPACE\text{-}Hard} \implies \mathsf{PSPACE} = \mathsf{NP}$$

Dimostrazione. Si noti che, per definizione, si ha che NP-Complete $\subseteq$ NP-Hard dunque se NP-Hard $\subseteq$ PSPACE-Hard, in particolare si ha che

$$\mathsf{NP\text{-}Complete} \subseteq \mathsf{PSPACE\text{-}Hard}$$

Sia dunque un linguaggio $L \in$ NP-Complete, e quindi $L \in$ PSPACE-Hard per ipotesi; allora, per il Lemma 6.2.2.2 ed il Teorema 5.3.7.1 segue che

$$\begin{cases} \forall L' \in \mathsf{PSPACE} \quad L' \leq_{\mathrm{L}} L \implies L' \leq_{\mathrm{P}} L \\ L \in \mathsf{NP\text{-}Complete} \subseteq \mathsf{NP} \end{cases} \implies \forall L' \in \mathsf{PSPACE} \quad L' \in \mathsf{NP}$$

provando che

$$\mathsf{PSPACE} \subseteq \mathsf{NP}$$

Viceversa, per il Teorema 6.2.1.1, si ha che

$$\mathsf{NP} \subseteq \mathsf{PSPACE}$$

e dunque segue la tesi per doppia implicazione. □

6.2.7 Classe EXPSPACE

Definizione 6.2.7.1: Classe EXPSPACE

Si definisce **classe dei linguaggi decidibili in spazio esponenziale** da una TM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{EXPSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{DSPACE}\left(2^{n^k}\right)$$

Proposizione 6.2.7.1: Relazione tra NEXP ed EXPSPACE

Si verifica che

$$\mathsf{NEXP} \subseteq \mathsf{EXPSPACE}$$

Dimostrazione. La tesi segue direttamente dal Corollario 6.2.1.1, e la dimostrazione è analoga alla dimostrazione della Proposizione 6.2.2.1, pertanto verrà omessa. □

6.2.8 Classe coEXPSPACE

Definizione 6.2.8.1: Classe coEXPSPACE

La clase coEXPSPACE è definita come segue:

$$\mathsf{coEXPSPACE} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{EXPSPACE}\}$$

Teorema 6.2.8.1: Relazione tra EXPSPACE e coEXPSPACE

Si verifica che

$$\mathsf{EXPSPACE} = \mathsf{coEXPSPACE}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.3.1, pertanto verrà omessa. □

6.2.9 Classe NSPACE

Definizione 6.2.9.1: Classe NSPACE

Data una funzione $s : \mathbb{N} \to \mathbb{R}^+$, si definisce **classe di complessità di spazio** $\mathsf{NSPACE}(s(n))$ l'insieme dei linguaggi decidibili da una macchina di Turing non deterministica definita come descritto nell'Osservazione 6.1.1.1 in $O(s(n))$.

Teorema 6.2.9.1: Teorema di Savitch

Si verifica che

$$f(n) \geq \log(n) \implies \mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}(f^2(n))$$

Dimostrazione. Sia $L \in \mathsf{NSPACE}(f(n))$, e dunque per definizione esiste una NTM $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ che decide L in spazio $f(n)$. Si consideri un grafo $G_{N,w} = (V_{N,w}, E_{N,w})$, costruito in base a δ, che descrive l'albero di configurazioni che possono essere assunte da N su input w; in particolare, esiste un arco $(C, C') \in E_{N,w}$ se e solo se è possibile passare dalla configurazione C alla configurazione C' attraverso δ. Dunque, la radice dell'albero sarà sempre

$$C_0 := q_0 \sqcup;\ 1$$

mentre per quanto riguarda la configurazione di accettazione, poiché questa potrebbe non essere ben definita — in particolare, non è nota a priori come è definita la configurazione accettante, poiché generalmente dipende da come computa la macchina di Turing presa in considerazione — viene introdotta una configurazione

$$C_{\text{accept}} := C_0$$

e dunque si assume che, al termine della computazione, N pulisca il *working tape* e sposti entrambe le testine sulle prime celle dei due suoi nastri rispettivamente. Dunque, per costruzione di $G_{N,w}$, $N(w)$ accetta se e solo se esiste un cammino $C_0 \to C_{\text{accept}}$.

Si consideri la procedura `findPath` definita all'interno della dimostrazione del Teorema 6.2.2.3; è possibile modificarla affinché il grafo $G_{N,w}$ descritto venga costruito durante la ricorsione stessa, preservando lo stesso costo di spazio, in quanto tale modifica richiede di salvare solamente qualche altra variabile d'appoggio, preservando dunque il costo di spazio $O(\log^2(n))$; sia `findPathGraph` tale versione modificata di `findPath`.

Sia allora M una macchina di Turing definita come descritto nell'Osservazione 6.1.1.1 che, data in input una stringa w, computa come segue:

- M esegue $\texttt{findPathGraph}\,(G_{N,w}, C_0, C_{\text{accept}}, \lceil \log(|V|) \rceil)$;
- M accetta se e solo se l'istruzione precedente restituisce `True`.

Allora, per costruzione di M, si ha che

$$w \in L(M) \iff \texttt{findPathGraph}\,(G_{N,w}, C_0, C_{\text{accept}}, \lceil \log(|V|) \rceil) = \texttt{True} \iff w \in L(N)$$

per costruzione di $G_{N,w}$, dunque N ed M sono equivalenti. Infine, si noti che, per ragionamento analogo a quello proposto nella dimostrazione del Teorema 6.2.1.2, poiché il numero massimo di configurazioni possibili di N è $n \cdot 2^{O(f(n))} = 2^{O(f(n))}$ (poiché $f(n) \geq \log(n)$ in ipotesi), questo è anche il numero di nodi massimo che $G_{N,w}$ può avere, e dunque $|V| = 2^{O(f(n))}$. Allora, segue che il costo di spazio di M è pari a

$$O\left(\left\lceil \log^2\left(2^{O(f(n))}\right)\right\rceil\right) = O(f^2(n))$$

il che dimostra la tesi. □

6.2.10 Classe NL

Definizione 6.2.10.1: Classe NL

Si definisce **classe dei linguaggi decidibili non deterministicamente in spazio logaritmico** da una NTM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{NL} := \mathsf{NSPACE}(\log(n))$$

Teorema 6.2.10.1: NL mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{L}} B \\ B \in \mathsf{NL} \end{cases} \implies A \in \mathsf{NL}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 6.2.2.1, pertanto verrà omessa. □

Corollario 6.2.10.1: NL mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{L}} B \\ A \notin \mathsf{NL} \end{cases} \implies B \notin \mathsf{NL}$$

Dimostrazione. Per assurdo, sia $B \in \mathsf{NL}$; allora, poiché $A \leq_{\mathrm{L}} B$, per il Teorema 6.2.10.1 si ha che $A \in \mathsf{NL}$ ↯. □

Proposizione 6.2.10.1: Relazione tra NL e DSPACE$(\log^2(n))$

Si verifica che

$$\mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2(n))$$

Dimostrazione. La tesi segue immediatamente dal Teorema 6.2.9.1, pertanto ne verrà omessa la dimostrazione. □

Definizione 6.2.10.2: Complessità di spazio di verificatori

Dato un verificatore V definito come descritto nell'Osservazione 6.1.1.1, ed una codifica $\langle w, c \rangle \in L(V)$, lo spazio di computazione di V è misurato solamente in termini della lunghezza di w.

Definizione 6.2.10.3: Verificatore di spazio logaritmico

Un linguaggio è detto essere **verificabile in spazio logaritmico** se ammette un verificatore in spazio logaritmico.

Teorema 6.2.10.2: Linguaggi verificabili in spazio logaritmico

NL è la classe dei linguaggi verificabili in spazio logaritmico.

Dimostrazione. Omessa. □

6.2.11 Classe NL-Complete

Definizione 6.2.11.1: NL-difficoltà

Un lingauggio L è detto essere **NL-difficile** se e solo se ogni altro linguaggio in NL è riducibile in spazio logaritmico ad L. In simboli

$$\text{NL-Hard} := \{L \mid \forall L' \in \text{NL} \quad L' \leq_{\text{L}} L\}$$

Definizione 6.2.11.2: NL-completezza

Un linguaggio è detto essere **NL-completo** se e solo se è in NL ed è NL-difficile. In simboli

$$\text{NL-Complete} := \text{NL} \cap \text{NL-Hard}$$

Osservazione 6.2.11.1: "L versus NL problem"

Un altro noto problema aperto riguarda stabilire se L coincide o meno con NL.

Proposizione 6.2.11.1: Implicazioni di L = NL

Si verifica che

$$\text{L} \cap \text{NL-Complete} \neq \varnothing \iff \text{L} = \text{NL}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione della Proposizione 5.3.8.1, pertanto verrà omessa. □

Teorema 6.2.11.1: NL-Complete mediante riduzione

Siano A e B due linguaggi definiti su un alfabeto Σ; se $A \leq_{\mathrm{L}} B$, A è NL-completo è B è in NL, allora B è NL-completo. In simboli

$$\begin{cases} A \leq_{\mathrm{L}} B \\ A \in \mathsf{NL\text{-}Complete} \\ B \in \mathsf{NL} \end{cases} \implies B \in \mathsf{NL\text{-}Complete}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.8.2, pertanto verrà omessa □

Teorema 6.2.11.2: $PATH$ in NL-Complete

Si verifica che

$$PATH \in \mathsf{NL\text{-}Complete}$$

Dimostrazione. Si consideri il seguente algoritmo:

```
1:  function findPathNL(G, x, y)
2:      if x == y then
3:          accept
4:      end if
5:      u := x
6:      for _ ∈ [1, |V|] do
7:          v ∈ V                          ▷ scelto non deterministicamente
8:          if (u, v) ∈ E then
9:              u := v
10:             if u == y then
11:                 accept
12:             end if
13:         else
14:             reject                     ▷ si noti il non determinismo
15:         end if
16:     end for
17:     reject                             ▷ si noti il non determinismo
18: end function
```

Si consideri una NTM N che, su input $\langle G, s, t \rangle$, esegua l'algoritmo `findPathNL`(G, s, t) — la quale esiste per l'Osservazione 3.3.1.1. Dunque, per costruzione stessa dell'algoritmo, poiché N computa non deterministicamente, N accetta se e solo se esiste un cammino $s \to t$, quindi N decide $PATH$. Inoltre, poiché è necessario salvare un numero costante di variabili, ovvero $|V|, s, t, x, y, u, v$, salvarle ha costo $O(\log(n))$ (si veda la Proposizione 6.2.2.2), e dunque l'algoritmo risulta avere costo di spazio $O(\log(n))$, implicando che $PATH \in \mathsf{NL}$.

Sia $L \in \mathsf{NL}$, e dunque per esso esiste un decisore N_L non deterministico che computa in $O(\log(n))$; si consideri inoltre la seguente funzione:

$$r : \Sigma^* \to \Sigma^* : w \mapsto \langle G_{N_{L,w}}, C_0, C_{\text{accept}} \rangle$$

dove $G_{N_L,w}$, C_0 e C_{accept} sono definiti nella dimostrazione del Teorema 6.2.9.1. Dunque, per costruzione di r, si ha che

$$w \in L = L(N_L) \iff r(w) = \langle G_{N_L,w}, C_0, C_{\text{accept}} \rangle \in PATH$$

Si consideri un trasduttore T che, data in input una stringa $w \in \Sigma^*$, computa come segue:

- determina C_0 in base dalla funzione di transizione di N;
- T elenca *una alla volta* tutte le possibili stringhe di lunghezza $O(\log(n))$;
- per ognuna di queste, T controlla che siano configurazioni possibili di N_L (attraverso la funzione di transizione di quest'ultimo); sia $V_{N_L,w}$ l'insieme di tali configurazioni filtrate;
- data una coppia di configurazioni $C, C' \in V_{N_L,w}$, dove solo una delle due è scritta sul *working tape* di T, è sufficiente che T controlli che sia possibile produrre C' a partire da C (attraverso la funzione di transizione di N) per stabilire se (C, C') debba essere un arco in $E_{N_L,w}$, dunque *senza* dover scrivere C' stessa sul suo *working tape*;
- T restituisce $\langle G_{N_L} = (V_{N_L,w}, E_{N_L,w}), C_0, C_{\text{accept}} \rangle$ sul suo *output tape*.

Dunque, per costruzione stessa, T ha costo di spazio logaritmico, sia per come computa, sia perché ognuna delle configurazioni di N_L deve occupare spazio logaritmico per definizione di L. Inoltre, se T si ferma, termina avendo solo $r(w)$ sul suo *output tape* per definizione di r; allora, r risulta essere una funzione calcolabile in spazio logaritmico, che per sua stessa definizione costituisce dunque una riduzione in spazio logaritmico di L a $PATH$. In simboli, si ha che

$$\forall L \in \mathsf{NL} \quad L \leq_{\mathrm{L}} PATH \iff PATH \in \mathsf{NL\text{-}Hard}$$

allora segue la tesi. □

Proposizione 6.2.11.2: Relazione tra NL e P

Si verifica che

$$\mathsf{NL} \subseteq \mathsf{P}$$

Dimostrazione. Per il Teorema 6.2.11.2, il Teorema 5.3.2.2, il Lemma 6.2.2.2 ed il Teorema 5.3.2.1, segue che

$$\begin{cases} \forall L \in \mathsf{NL} \quad L \leq_{\mathrm{L}} PATH \implies L \leq_{\mathrm{P}} PATH \\ PATH \in \mathsf{P} \end{cases} \implies \forall L \in \mathsf{NL} \quad L \in \mathsf{P} \iff \mathsf{NL} \subseteq \mathsf{P}$$

□

6.2.12 Classe coNL

Definizione 6.2.12.1: Classe coNL

La classe coNL è definita come segue:

$$\mathsf{coNL} := \{L \in \mathsf{DEC} \mid \overline{L} \in \mathsf{NL}\}$$

Teorema 6.2.12.1: coNL mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{L}} B \\ B \in \mathsf{coNL} \end{cases} \implies A \in \mathsf{coNL}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione del Teorema 5.3.9.2, grazie al Lemma 6.2.2.1 ed al Teorema 6.2.10.1, pertanto verrà omessa. □

Corollario 6.2.12.1: coNL mediante riduzione

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

$$\begin{cases} A \leq_{\mathrm{L}} B \\ A \notin \mathsf{coNL} \end{cases} \implies B \notin \mathsf{coNL}$$

Dimostrazione. Per assurdo, sia $B \in \mathsf{coNL}$; allora, poiché $A \leq_{\mathrm{L}} B$, per il Teorema 6.2.12.1 si ha che $A \in \mathsf{coNL}$ ↯. □

Teorema 6.2.12.2: $PATH$ in coNL

Si verifica che

$$PATH \in \mathsf{coNL}$$

Dimostrazione. Per la Definizione 6.2.12.1, la tesi può procedere dimostrando equivalentemente che $\overline{PATH} \in \mathsf{NL}$; inoltre, per il Teorema 6.2.10.2, dimostrare che $\overline{PATH} \in \mathsf{NL}$ è equivalente a dimostrare che $\overline{PATH}$ e verificabile in spazio logaritmico. Infine, si noti che all'interno di $\overline{PATH}$ sono presenti non solo codifiche $\langle G, s, t\rangle$ tali per cui non esistono cammini $s \to t$ in G, ma anche ogni altra stringa di simboli dell'alfabeto considerato che non corrisponde ad una valida codifica della forma $\langle G, s, t\rangle$.

Si consideri dunque una TM V definita come descritto all'interno dell'Osservazione 6.1.1.1 che, data una codifica $\langle w, c\rangle$ in input, computa come segue:

- V interpreta w come $\langle G, s, t\rangle$, dove $G = (V_G, E_G)$ è un grafo diretto, ed $s, t \in V$ come due suoi nodi; se non è possibile effettuare tale interpretazione per via di una codifica incorretta, V accetta;

- V interpreta dunque c come $\langle c_0, \ldots, c_{|V_G|} \rangle$, dove ogni c_i viene interpretato da V come segue:

$$\forall i \in [0, |V_G|) \quad c_i = \langle s \to v_1, \ldots, s \to v_k \rangle$$

  per qualche $k \in \mathbb{N}$, dove $v_1, \ldots, v_k \in V_G$ sono tutti i nodi raggiungibili da s attraverso cammini lunghi al massimo i; inoltre, gli elementi all'interno di ogni c_i devono essere ordinati lessicograficamente, al fine di poter controllare efficientemente — in termini di spazio — se rappresentano cammini tutti distinti; se non è possibile effettuare tale interpretazione per via di una codifica incorretta, oppure i cammini rappresentati da qualche c_i non sono distinti, V accetta;

- V esegue la seguente procedura:

```
1:  function checkPath(G, s, t, c)
2:      for s → v ∈ c_0 do                          ▷ c_0 contiene solo s → s
3:          if v == t then
4:              reject
5:          end if
6:      end for
7:      for i ∈ [0, |V_G|) do
8:          for s → v' ∈ c_{i+1} − c_i do
9:              isExt = False
10:             for s → v ∈ c_i do
11:                 if (v, v') ∈ E_G then
12:                     isExt = True
13:                     break
14:                 end if
15:             end for
16:             if isExt == False then
17:                 accept
18:             end if
19:         end for
20:         if v' == t then
21:             reject
22:         end if
23:     end for
24:     accept
25: end function
```

Sia R_l l'insieme dei vertici raggiungibili da s con al massimo l archi, e sia $r_l := |R_l|$; dunque, la dimostrazione della correttezza della procedura `checkPath` di V procede per induzione sui vertici raggiungibili da s.

Caso base. Per definizione, si ha $R_0 = \{s\}$ e $c_0 = \langle s \to s \rangle$; allora, per stabilire se $t \notin R_0$, `checkPath` controlla se $t \neq s \in R_0$ alla riga 3 (si noti che il ciclo della riga 2 scorrerà un solo elemento, ovvero $s \to s \in c_0$).

Ipotesi induttiva. Dato un $l \in [0, |V_G|)$, `checkPath` è in grado di stabilire correttamente

- gli elementi in c_l rappresentano cammini della forma $s \to v$ per qualche $v \in V_G$;
- se $t \notin R_l$.

Passo induttivo. È necessario dimostare che per $l + 1$, `checkPath` è in grado di stabilire correttamente

- gli elementi in c_{l+1} rappresentano cammini della forma $s \to v$ per qualche $v \in V_G$;
- se $t \notin R_{l+1}$.

Il ciclo della riga 8 controlla che ogni $s \to v' \in c_{l+1} - c_l$ rappresenti realmente un camnino che parte da s e raggiunge un certo $v' \in V$, e poiché i cammini $s \to v \in c_l$ rappresentano cammini della forma $s \to v$ per ipotesi induttiva, il ciclo 8 di fatto controlla che c_{l+1} sia un'estensione di c_l; si noti inoltre che è possibile considerare solo gli ultimi $|c_{l+1} - c_l|$ elementi, poiché i primi $r_l := |R_l| = |c_l|$ combacieranno avendo assunto che gli elementi in ogni c_i siano ordinati lessicograficamente (si assume che sia sempre possibile etichettare i vertici di G, in modo che ogni c_{i+1} sia un estensione di c_i, e che ogni c_i sia ordinato lessicograficamente). Questo dimostra la prima parte della tesi.

Infine, per definizione stessa, si ha che

$$\forall h \in [0, |V_G|) \quad R_h \subseteq R_{h+1}$$

dunque segue che

$$t \notin R_l \implies \forall h \in [0, l] \quad t \notin R_h$$

di conseguenza, per controllare che $t \notin R_{l+1}$ è sufficiente verificare che i soli elementi $s \to v' \in c_{l+1} - c_l$ non abbiano $v' = t$, poiché $t \notin R_l$ per ipotesi induttiva. Questo dimostra la seconda parte della tesi.

Dunque, questo dimostra in particolare che `checkPath` è in grado di stabilire correttamente se $t \in R_{|V_G|}$; allora, attraverso tale procedura, V stabilisce correttamente se t sia raggiungibile o meno da s. Di conseguenza, se non esistono cammini $s \to t \in G$, esiste una $c \in \Sigma^*$ tale che $\langle \langle G, s, t \rangle, c \rangle \in L(V)$, e se esiste un cammino $s \to t$ allora `checkPath` è in grado di stabilire che $t \in R_{|V_G|}$ correttamente, e dunque V rifiuta. Questo dimostra che V è un verificatore per $\overline{PATH}$.

Inoltre, si noti che:

- per controllare che le codifiche siano corrette, non è necessario scrivere nulla sul *working tape*; invece, per controllare che gli elementi in ogni c_i siano tutti distinti, richiede spazio $O(\log(n))$ poiché si è assunto che ogni c_i è ordinato lessicograficamente;
- per effettuare il ciclo della riga 2 (che di fatto corrisponde ad un solo controllo), ed il controllo della riga 20, V deve solo scrivere t sul *working tape*, dunque sono entrambe operazioni che richiedono costo di spazio $O(\log(n))$;
- per effettuare il ciclo della riga 8 è sufficiente scrivere v e v' sul *working tape*, e dunque richiede costo di spazio $O(2 \cdot \log(n)) = O(\log(n))$

Allora, V computa in spazio logaritmico, e dunque segue la tesi per il Teorema 6.2.10.2. □

Teorema 6.2.12.3: Teorema di Immerman-Szelepcsényi

Si verifica che

$$\mathsf{NL} = \mathsf{coNL}$$

Dimostrazione. Per il Teorema 6.2.11.2, il Teorema 6.2.12.2, il Lemma 6.2.2.1, il Teorema 6.2.12.1, ed il Teorema 6.2.10.1 si ha che

$$\begin{cases} \forall L \in \mathsf{NL} \quad L \leq_{\mathrm{L}} PATH \\ PATH \in \mathsf{coNL} \end{cases} \implies \forall L \in \mathsf{NL} \quad L \in \mathsf{coNL} \iff \mathsf{NL} \subseteq \mathsf{coNL}$$

ed inoltre

$$\begin{cases} \forall L \in \mathsf{NL} \quad L \leq_{\mathrm{L}} PATH \iff \overline{L} \leq_{\mathrm{L}} \overline{PATH} \\ PATH \in \mathsf{coNL} \iff \overline{PATH} \in \mathsf{NL} \end{cases} \implies \forall \overline{L} \in \mathsf{coNL} \quad \overline{L} \in \mathsf{NL} \iff \mathsf{coNL} \subseteq \mathsf{NL}$$

dunque segue la tesi. □

6.2.13 Classe NPSPACE

Definizione 6.2.13.1: Classe NPSPACE

Si definisce **classe dei linguaggi decidibili non deterministicamente in spazio polinomiale** da una NTM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{NPSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{NSPACE}\left(n^k\right)$$

Proposizione 6.2.13.1: Relazione tra PSPACE ed NPSPACE

Si verifica che

$$\mathsf{PSPACE} = \mathsf{NPSPACE}$$

Dimostrazione. Trivialmente si ha che PSPACE $\subseteq$ NPSPACE, poiché una TM è una particolare NTM, inoltre, per il Teorema 6.2.9.1, si ha che

$$\forall k, h \in \mathbb{N} \mid h = 2k \quad \mathsf{NSPACE}\left(n^k\right) \subseteq \mathsf{DSPACE}\left(\left(n^k\right)^2\right) = \mathsf{DSPACE}\left(n^{2k}\right) = \mathsf{PSPACE}$$

dunque segue la tesi per doppia implicazione. □

Proposizione 6.2.13.2: Relazione tra NPSPACE ed EXP

Si verifica che

$$\mathsf{NPSPACE} \subseteq \mathsf{EXP}$$

Dimostrazione. Per il Teorema 6.2.9.1, e per il Corollario 6.2.1.1, segue che

$$\forall k, h \in \mathbb{N} \mid h = 2k \quad \mathsf{NSPACE}\left(n^k\right) \subseteq \mathsf{DSPACE}\left(n^{2k}\right) = \mathsf{DSPACE}\left(n^h\right) \subseteq \mathsf{DTIME}\left(2^{O\left(n^h\right)}\right)$$

dunque segue immediatamente la tesi. □

6.2.14 Classe NEXPSPACE

Definizione 6.2.14.1: Classe NEXPSPACE

Si definisce **classe dei linguaggi decidibili non deterministicamente in spazio esponenziale** da una NTM definita come descritto nell'Osservazione 6.1.1.1 il seguente insieme

$$\mathsf{NEXPSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{NSPACE}\left(2^{n^k}\right)$$

Proposizione 6.2.14.1: Relaz. tra EXPSPACE e NEXPSPACE

Si verifica che

$$\mathsf{EXPSPACE} = \mathsf{NEXPSPACE}$$

Dimostrazione. La dimostrazione è analoga alla dimostrazione della Proposizione 6.2.13.1, pertanto verrà omessa. □

6.2.15 Teorema di gerarchia di spazio

Definizione 6.2.15.1: Funzione spazio-costruibile

Una funzione $f : \mathbb{N} \to \mathbb{N}$ è detta essere **spazio-costruibile** se $f(n)$ è almeno $O(\log(n))$, e se la funzione che mappa $\mathtt{1}^n$ nella rappresentazione binaria di $f(n)$ è computabile in spazio $O(f(n))$.

Funzioni a valori non interi vengono arrotondate all'intero successivo più piccolo al fine di renderle spazio-costruibili.

Esempio 6.2.15.1 (Funzioni spazio-costruibili)**.** Si consideri la funzione $f(n) = n^2$, la quale è almeno $O(\log(n))$; inoltre, sia T un trasduttore che, data in input la stringa $\mathtt{1}^n$, computa come segue:

- T conta il numero di $\mathtt{1}$ presenti sull'*input tape*;
- T calcola $n \cdot n = n^2$;
- T restituisce n^2 sul suo *output tape*.

Allora, poiché entrambe le operazioni che T effettua richiedono spazio logaritmico per la Proposizione 6.2.2.2, T risulta essere un trasduttore di spazio logaritmico per la funzione $g : \Sigma^* \to \Sigma^* : \mathtt{1}^n \mapsto n^2$, implicando che g sia computabile in spazio $O(\log(n))$, che è certamente $O(n^2)$. Questo dimostra che $f(n) = n^2$ è spazio-costruibile.

Teorema 6.2.15.1: Teorema di gerarchia di spazio

Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione spazio-costruibile; allora, esiste un linguaggio L decidibile in spazio $O(f(n))$ ma non in spazio $o(f(n))$.

Dimostrazione. Si consideri il seguente decisore D definito come descritto all'interno dell'Osservazione 6.1.1.1 che, data in input una stringa w, computa come segue:

- D marca la $f(n)$-esima cella del suo *working tape* (si noti che $f(n)$ è calcolabile in $O(f(n))$ per spazio-costruibilità di f); se in fasi successive avviene un tentativo di usare più di $f(n)$ celle, D rifiuta;
- se w non è della forma $\langle M \rangle \, \mathtt{10}^*$, dove $M \in \mathsf{TM}$, D rifiuta;
- D simula $M(w)$, limitandole lo spazio ad $f(n)$ per quanto detto in precedenza, e contando contemporaneamente il numero di passi che M impiega per computare; se il conto eccede $2^{f(n)}$, D rifiuta;
- D accetta se e solo se M rifiuta.

È facile convincersi che D è un decisore, poiché anche se M dovesse andare in loop infinito entro le $f(n)$ celle di spazio che D le fornisce, dopo $2^{f(n)}$ passi D ne interromperebbe la simulazione e rifiuterebbe

Per quanto riguarda lo spazio utilizzato da D, si ha che:

- per come D computa, la sua esecuzione non può eccedere l'uso dell'$f(n)$-esima cella del suo *working tape*;
- poiché $O\left(\log\left(2^{f(n)}\right)\right) = O(f(n))$, il contatore di passi di $M(w)$ ha costo di spazio $O(f(n))$.
- siano Γ_D e Γ_M rispettivamente gli alfabeti di nastro di D ed M; si noti che, nel caso in cui $\Gamma_M - \Gamma_D \neq \varnothing$, D può rappresentare i simboli di M introducendo un fattore costante nel suo costo di spazio, poiché Γ_D è fissato.

Questo dimostra che D è un decisore di spazio $O(f(n))$, e sia $L := L(D)$.

Per giustificare la scelta della codifica di $w = \langle M \rangle \, \mathtt{10}^*$ in input a D, si ipotizzi che M computi in spazio $o(f(n))$ (dunque utilizzi meno di $f(n)$ celle); per definizione di $o(f(n))$, l'uso di spazio di M potrebbe essere superiore ad $f(n)$ per valori piccoli di n, quando il comportamento asintotico non è ancora stato raggiunto. Di conseguenza, potrebbe verificarsi che D non abbia abbastanza spazio per eseguire M fino al completamento, dunque rifiutando prematuramente, rischiando inoltre di decidere lo stesso lingauggio che decide M, condizione per cui il teorema non risulterebbe più verificato. Allora, per garantire che D decida il complemento del linguaggio riconosciuto da M, è necessario assicurarsi che M computi sempre in $o(f(n))$, e dunque viene inserita una stringa $\mathtt{10}^k$ al termine della codifica $\langle M \rangle$ nell'input di D, con un numero k di $\mathtt{0}$ tale che $w = \langle M \rangle \, \mathtt{10}^k$ permetta di raggiungere la soglia tale per cui M computa realmente in $o(f(n))$ — si noti che D simula $M\left(\langle M \rangle \, \mathtt{10}^k\right)$.

Per assurdo, sia M_L una macchina tale da decidere L in spazio $g(n)$, con $g(n) = o(f(n))$, e dunque per definizione esistono $c, n_0 \in \mathbb{N} - \{0\}$ tali per cui $\forall n \geq n_0 \quad g(n) < c \cdot f(n)$. Allora, se $k = n_0$, e dunque $w = \langle M_L \rangle \, \mathtt{1}\mathtt{0}^{n_0}$ si ha che $|w| > n_0$ (poiché ci sono almeno n_0 simboli $\mathtt{0}$); di conseguenza si è superata la soglia per cui M_L computa in $o(f(n))$ (per quanto detto in precedenza) e dunque D avrà spazio a sufficienza per simulare interamente M_L. Allora, per sua costruzione, D invertirà il risultato di M_L al termine della simulazione, e dunque M_L non può decidere $L := L(D)$ $\lightning$. □

Corollario 6.2.15.1: Teorema di gerarchia di spazio

Siano $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ due funzioni, dove $f_1(n) = o(f_2(n))$ ed f_2 è spazio-costruibile; allora, si verifica che

$$\mathsf{DSPACE}(f_1(n)) \subsetneq \mathsf{DSPACE}(f_2(n))$$

Dimostrazione. Siano $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ due funzioni, tali che

$$f_1(n) = o(f_2(n)) \implies f_1(n) = O(f_2(n))$$

(si veda la Osservazione 5.1.1.1) ed f_2 sia spazio-costruibile; allora, per definizione di f_1 ed f_2, e per il Teorema 6.2.15.1, si ha che

$$\mathsf{DSPACE}(f_1(n)) \subsetneq \mathsf{DSPACE}(f_2(n))$$

□

Proposizione 6.2.15.1: Relazione tra NL e PSPACE

Si verifica che

$$\mathsf{NL} \subsetneq \mathsf{PSPACE}$$

Dimostrazione. Si consideri la funzione $f(n) = n$, la quale è almeno $O(\log(n))$; inoltre, sia T un trasduttore che, data in input la stringa $\mathtt{1}^n$, computa come segue:

- T conta il numero di $\mathtt{1}$ presenti sull'*input tape*;
- T restituisce n sul suo *output tape*.

Allora, poiché l'operazione che T effettua richiede spazio logaritmico per la Proposizione 6.2.2.2, T risulta essere un trasduttore di spazio logaritmico per la funzione $g : \Sigma^* \to \Sigma^* : \mathtt{1}^n \mapsto n$, implicando che g sia computabile in spazio $O(\log(n))$, che è certamente $O(n)$. Questo dimostra che $f(n) = n$ è spazio-costruibile.

Di conseguenza, per il Teorema 6.2.15.1, si ha che esiste un linguaggio decidibile in $O(n)$ ma non in $o(n)$; in particolare, poiché $\log^2(n) = o(n)$, si ha che

$$\begin{cases} \mathsf{DSPACE}(n) - \mathsf{DSPACE}(\log^2(n)) \neq \varnothing \\ \mathsf{DSPACE}(\log^2(n)) \subset \mathsf{DSPACE}(n) \end{cases} \implies \mathsf{DSPACE}(\log^2(n)) \subsetneq \mathsf{DSPACE}(n) \subsetneq \mathsf{PSPACE}$$

Allora, per il Proposizione 6.2.10.1, segue che

$$\mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2(n)) \subsetneq \mathsf{PSPACE} \implies \mathsf{NL} \subsetneq \mathsf{PSPACE}$$

□

Proposizione 6.2.15.2: Relazione tra PSPACE e EXPSPACE

Si verifica che

$$\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$$

Dimostrazione. Si consideri la funzione $f(n) = 2^n$, la quale è almeno $O(\log(n))$; inoltre, sia T un trasduttore che, data in input la stringa $\mathtt{1}^n$, computa come segue:

- T conta il numero di $\mathtt{1}$ presenti sull'*input tape*;
- T calcola 2^n;
- T restituisce 2^n sul suo *output tape*.

Allora, poiché entrambe le operazioni che T effettua richiedono spazio logaritmico per la Proposizione 6.2.2.2, T risulta essere un trasduttore di spazio logaritmico per la funzione $g : \Sigma^* \to \Sigma^* : \mathtt{1}^n \mapsto 2^n$, implicando che g sia computabile in spazio $O(\log(n))$, che è certamente in $O\left(2^n\right)$. Questo dimostra che $f(n) = 2^n$ è spazio-costruibile.

Si noti che

$$\forall k \in \mathbb{N} \quad \lim_{n\to+\infty} \frac{n^k}{n^{\log(n)}} = \lim_{n\to+\infty} n^{k-\log(n)} = 0 \implies \forall k \in \mathbb{N} \quad n^k = o\left(n^{\log(n)}\right)$$

ed inoltre, si noti che

$$\lim_{n\to+\infty} \frac{n^{\log(n)}}{2^n} = \lim_{n\to+\infty} \frac{2^{\log^2(n)}}{2^n} = 0 \implies n^{\log(n)} = o\left(2^n\right)$$

allora, per la Proposizione 5.1.1.2, si ha che

$$\forall k \in n^k \quad n^k = o\left(2^n\right)$$

e poiché 2^n è spazio-costruibile per quanto detto in precedenza, per il Corollario 6.2.15.1, segue che

$$\forall k \in \mathbb{N} \quad \mathsf{DSPACE}\left(n^k\right) \subsetneq \mathsf{DSPACE}\left(2^n\right) \subsetneq \mathsf{EXPSPACE}$$

implicando che

$$\mathsf{PSPACE} := \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}\left(n^k\right) \subsetneq \mathsf{DSPACE}\left(2^n\right) \subsetneq \bigcup_{h\in\mathbb{N}} \mathsf{DSPACE}\left(2^{n^h}\right) =: \mathsf{EXPSPACE}$$

□

Osservazione 6.2.15.1: Gerarchia delle classi

TODO NOMINA I TEOREMI USATI