

Automi

Alessandro Dori

November 13, 2024

Contents

1 Linguaggi regolari	3
1.1 Automi finiti	3
1.1.1 Definizione formale di computazione	3
1.1.2 Le operazioni regolari	4
1.1.3 Definizione 1.23	4
1.1.4 Teorema 1.25	4
1.1.5 Teorema 1.26	4
1.2 Non determinismo	4
1.2.1 Definizione formale di automa finito non deterministico	5
Definizione 1.37	6
1.2.2 Equivalenza tra gli NFA e i DFA	6
Teorema 1.39(orale)	6
Corollario 1.40	7
Chiusura rispetto alle operazioni regolari	7
Teorema 1.45(orale)	7
Teorema 1.47(orale)	8
Teorema 1.49(orale)	9
1.3 Espressioni regolari	10
1.3.1 Definizione formale di espressioni regolari	11
1.3.2 Equivalenza con gli automi finiti	12
Teorema 1.54	12
Lemma 1.55(orale)	12
Lemma 1.60(orale 24-30)	14
Definizione 1.64(orale 24-30)	16
1.3.3 Linguaggi non regolari	18
Il pumping lemma	18
Teorema 1.70(orale)	18
2 Linguaggi context-free	22
2.1 Grammatiche context-free	22
Definizione formale di grammatica context-free (CFG):	22
Forma normale di Chomsky	24
Teorema 2.9(orale 24-30)	24
2.2 Automi a pila	26
Definizione formale di automa a pila	27
Equivalenza con le grammatiche context-free	29
Teorema 2.20	29
Lemma 2.21(orale)	30
Lemma 2.27(orale 24-30)	32
2.3 Linguaggi non context-free	35
Il pumping lemma per i linguaggi context-free Teorema 2.34 orale(24-30)	35
3 La Tesi di Church-Turing	38
3.1 Macchine di Turing	38

Definizione formale di macchina di Turing	39
3.2 Varianti delle Macchine di Turing	41
Macchine di Turing multinastro	41
Teorema 3.13 orale(24-30)	41
Corollario 3.15	42
Macchine di Turing non deterministiche	42
Teorema 3.16 orale(24-30)	42
Corollario 3.18	43
Corollario	44
Enumeratori	44
Teorema 3.21 (orale)	44
3.3 La definizione di algoritmo	45
I problemi di Hilbert	45
Terminologia per la descrizione di macchine di Tuning	46
4 Decidibilità	47
4.1 Linguaggi Decidibili	47
4.1.1 Problemi decidibili relativi a linguaggi regolari	47
Teorema 4.1(orale)	47
Teorema 4.2(orale)	48
Teorema 4.3(orale)	48
Teorema 4.4(orale)	48
Teorema 4.5(orale)	49
4.1.2 Problemi decidibili relativi a linguaggi context-free	49
Teorema 4.7(orale)	49
Teorema 4.8(orale)	50
Teorema 4.9	51
4.2 Indecidibilità	51

1 Linguaggi regolari

1.1 Automi finiti

Definizione formale di un automa finito

Un automa finito è una quintupla $M = (Q, \Sigma, \delta, q_0, F)$, dove:

- Q è un insieme finito chiamato l'insieme degli stati;
- Σ è l'insieme finito chiamato l'alfabeto;
- $\delta : Q \times \Sigma \rightarrow Q$ è la funzione di transizione;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati accettanti.

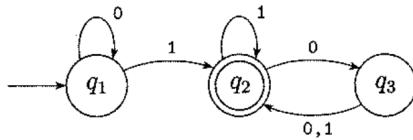


Figure 1: Esempio di automa finito

Possiamo descrivere M formalmente ponendo:

- $Q = \{q_1, q_2, \dots, q_n\}$
- $\Sigma = \{a, b, \dots, z\}$
- La funzione di transizione δ è descritta come segue:

$$\delta(q_i, a) = q_j \quad \text{dove } i, j \in Q \text{ e } a \in \Sigma$$

- q_0 è lo stato iniziale
- F è l'insieme degli stati accettanti

Se A è un insieme di tutte le stringhe che la macchina M accetta, diciamo che A è il **linguaggio della macchina** M e scriviamo $L(M) = A$. M riconosce A .

Nel nostro esempio, sia

$$A = \{w \mid w \text{ contiene almeno un } 1 \text{ e un numero pari di } 0 \text{ segue l'ultimo } 1\}.$$

1.1.1 Definizione formale di computazione

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un automa finito e sia $w = w_1 w_2 \dots w_n$ una stringa, dove ogni w_i è un elemento dell'alfabeto Σ . Allora M accetta w se esiste una sequenza di stati r_0, r_1, \dots, r_n in Q tale che:

- $r_0 = q_0$ (la macchina inizia nello stato iniziale),
- $r_{i+1} = \delta(r_i, w_{i+1})$ per ogni $i = 0, \dots, n-1$ (la macchina passa da uno stato all'altro in base alla funzione di transizione),
- $r_n \in F$ (la macchina accetta il suo input se termina la lettura in uno stato accettante).

Diciamo che M riconosce il linguaggio A se $L(M) = A$. Un linguaggio è chiamato un linguaggio regolare se un automa finito lo riconosce.

1.1.2 Le operazioni regolari

Definiamo tre operazioni sui linguaggi, chiamate operazioni regolari, e le usiamo per studiare le proprietà dei linguaggi regolari.

Operazioni Regolari

1.1.3 Definizione 1.23

Siano A e B linguaggi. Definiamo le operazioni regolari di unione, concatenazione e star come segue:

- Unione: $A \cup B = \{x \mid x \in A \text{ oppure } x \in B\}$
- Concatenazione: $A \cdot B = \{xy \mid x \in A \text{ e } y \in B\}$
- Star: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0, x_i \in A\}$
- Unione: prende tutte le stringhe sia in A che in B e le raggruppa insieme in un linguaggio.
- Concatenazione: antepone una stringa di A ad una stringa di B in tutti i modi possibili per ottenere le stringhe nel nuovo linguaggio.
- L'operazione di star è un'operazione unaria. Essa opera concatenando un numero qualsiasi di stringhe in A insieme per ottenere una stringa nel nuovo linguaggio.

Importante

La stringa vuota ϵ è sempre un elemento di A^* , indipendentemente da chi sia A .

Una classe di oggetti è chiusa rispetto ad un'operazione se l'applicazione di questa operazione a elementi della classe restituisce un oggetto ancora nella classe.

1.1.4 Teorema 1.25

La classe dei linguaggi regolari è chiusa rispetto all'operazione di unione. In altre parole, se A e B sono linguaggi regolari, lo è anche $A \cup B$. Costruiamo M da M_1 e M_2 , dove:

- M_1 riconosce A
- M_2 riconosce B
- M riconosce $A \cup B$
- Gli stati accettanti in M sono quelle coppie tali che $M_1 \circ M_2$ è in uno stato accettante.

1.1.5 Teorema 1.26

La classe dei linguaggi regolari è chiusa rispetto all'operazione di concatenazione. Se A e B sono linguaggi regolari, allora lo è anche $A \cdot B$.

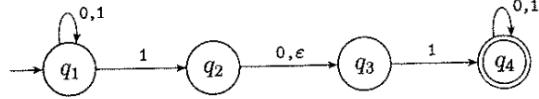
- M_1 riconosce A
- M_2 riconosce B
- M deve riconoscere $M_1 \circ M_2$, ma M non sa dove dividere il suo input.

Per risolvere questo problema introduciamo una nuova tecnica chiamata **non determinismo**.

1.2 Non determinismo

In una macchina non deterministica, possono esistere diverse scelte per lo stato successivo in ogni punto. Il non determinismo è una generalizzazione del determinismo, quindi ogni automa finito deterministico è autonomamente anche non deterministico.

Esempio:



Nota

DFA = Automa finito deterministico

NFA = Automa finito non deterministico

Ogni stato di un **DFA** ha sempre esattamente un arco di transizione uscente per ogni simbolo dell'alfabeto. In un **NFA** uno stato può avere zero, uno, o più archi uscenti per ogni simbolo dell'alfabeto. In un DFA le etichette sugli archi appartengono all'alfabeto, in un NFA troviamo anche ϵ ; zero, uno o più archi possono uscire da ciascuno stato con l'etichetta ϵ . In un NFA, nel caso in cui abbiamo due strade diverse con lo stesso simbolo, la "computazione" si divide. Se una di queste strade termina in uno stato non accettante, essa "muore"; se una qualsiasi di queste strade (copie) accetta, allora l'NFA accetta la stringa di input. Se incontriamo ϵ senza leggere alcun input, la macchina si divide in più copie multiple.

Importante

Ogni NFA può essere trasformato in un DFA equivalente, e costruire un NFA a volte è più semplice che costruire direttamente un DFA.

ESEMPIO 1.30

Sia A il linguaggio che consiste di tutte le stringhe su $\{0,1\}$ contenenti un 1 nella terza posizione dalla fine (ad esempio, 000100 è in A ma 0011 non lo è). Il seguente NFA N_2 , con quattro stati, riconosce A .

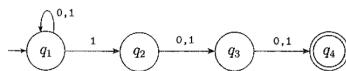
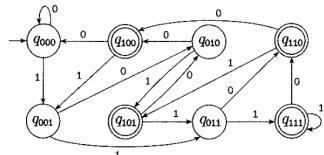


FIGURA 1.31
L'NFA N_2 che riconosce A

Un buon modo di vedere la computazione di questo NFA è dire che esso resta nello stato iniziale q_1 finché "ipotizza" che è a tre posizioni dalla fine. A questo punto, se il simbolo di input è un 1, passa nello stato q_2 e usa q_3 e q_4 per "controllare" se la sua ipotesi era corretta.

Come menzionato, ogni NFA può essere trasformato in un DFA equivalente; ma a volte questo DFA può avere molti più stati. Il più piccolo DFA per A contiene otto stati. Inoltre, capire il funzionamento dell'NFA è molto più facile, come si può vedere esaminando la figura seguente del DFA.



ESEMPIO 1.33

Il seguente NFA N_3 ha un alfabeto $\{0\}$ di simboli input che consiste di un solo simbolo. Un alfabeto contenente solo un simbolo è chiamato un *alfabeto unario*.

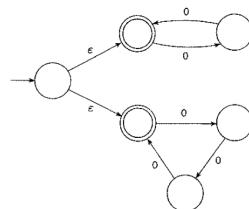


FIGURA 1.34
L'NFA N_3

Questa macchina mostra l'utilità di avere ϵ archi. Essa accetta tutte le stringhe della forma 0^k dove k è un multiplo di 2 o 3. (Ricorda che l'esponente denota ripetizione, non elevazione a potenza numerica.) Per esempio, N_3 accetta le stringhe ϵ , 00, 000, 0000, ma non 0 o 00000.

Puoi vedere che la macchina opera inizialmente provando a indovinare se testare per un multiplo di 2 o un multiplo di 3, andando nel ciclo superiore o nel ciclo inferiore, e poi verificando se la propria ipotesi era corretta. Naturalmente, potremmo sostituire questa macchina con una che non ha ϵ -archi o perfino del tutto priva di non determinismo, ma la macchina mostrata è quella più facile da capire per questo linguaggio.

1.2.1 Definizione formale di automa finito non deterministico

Per un qualsiasi insieme Q denotiamo con $P(Q)$ la collezione di tutti i sottoinsiemi di Q .

- $P(Q) = \text{insieme potenza di } Q$.
- Per ogni alfabeto Σ , scriviamo Σ_ϵ per denotare $\Sigma \cup \{\epsilon\}$.

Ora possiamo descrivere formalmente il tipo di funzione di transizione in un NFA come:

$$\delta : Q \times \Sigma_\epsilon \rightarrow P(Q)$$

Definizione 1.37

Definizione 1.37

Un automa finito non deterministico è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove:

1. Q è un insieme finito di stati.
2. Σ è un alfabeto finito.
3. $\delta : Q \times \Sigma \rightarrow P(Q)$ è la funzione di transizione.
4. $q_0 \in Q$ è lo stato iniziale.
5. $F \subseteq Q$ è l'insieme degli stati di accettazione.

1.2.2 Equivalenza tra gli NFA e i DFA

Teorema 1.39(orale)

Teorema 1.39 (orale)

Ogni automa finito non deterministico può essere convertito in un automa finito deterministico equivalente.

IDEA: Se un linguaggio è riconosciuto da un NFA, dobbiamo dimostrare l'esistenza di un DFA che lo riconosce. L'idea è di trasformare l'NFA in un DFA equivalente che simula l'NFA. Se k è il numero degli stati dell'NFA, il DFA avrà 2^k stati, corrispondenti ai sottoinsiemi degli stati di Q . Ora dobbiamo determinare lo stato iniziale, gli stati accettanti del DFA e la sua funzione di transizione.

DIMOSTRAZIONE: Sia $N = (Q, \Sigma, \delta, q_0, F)$ l'NFA che riconosce un linguaggio A . Costruiamo un DFA $M = (Q', \Sigma, \delta', q'_0, F')$ che riconosce A . Prima di tutto consideriamo il caso più semplice in cui N non ha ε -archi. In seguito considereremo gli ε -archi.

1. $Q' = P(Q)$.

- Ogni stato di M è un insieme di stati di N . Ricorda che $P(Q)$ è l'insieme dei sottoinsiemi di Q .

2. Per $R \in Q'$ e $a \in \Sigma$, sia

$$\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ per qualche } r \in R\}.$$

- Se R è uno stato di M , esso è anche un insieme di stati di N . Quando M legge un simbolo a nello stato R , mostra dove a porta ogni stato in R . Poiché da ogni stato si può andare in un insieme di stati, prendiamo l'unione di tutti questi insiemi. Un altro modo per scrivere questa espressione è

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

ovvero l'unione di tutti gli insiemi $\delta(r, a)$ per ogni possibile $r \in R$.

3. $q'_0 = \{q_0\}$.

- M inizia nello stato corrispondente alla collezione che contiene solo lo stato iniziale di N .

4. $F' = \{R \in Q' \mid R \text{ contiene uno stato accettante di } N\}$.

- La macchina M accetta se uno dei possibili stati in cui N potrebbe essere a quel punto è uno stato accettante.

Ora dobbiamo considerare gli ε -archi. Introduciamo qualche ulteriore notazione. Per ogni stato R di M , definiamo $E(R)$ come la collezione di stati che possono essere raggiunti dagli elementi di R proseguendo solo con ε -archi, includendo gli stessi elementi di R . Formalmente, per $R \subseteq Q$ sia

$$E(R) = \{q \mid q \text{ può essere raggiunto da } R \text{ attraverso } 0 \text{ o più } \varepsilon\text{-archi}\}.$$

Poi modifichiamo la funzione di transizione di M ponendo dita supplementari su tutti gli stati che possono essere raggiunti proseguendo attraverso ε -archi dopo ogni passo. Sostituendo $\delta(r, a)$ con $E(\delta(r, a))$ realizziamo questo effetto. Quindi

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ per qualche } r \in R\}.$$

Inoltre, dobbiamo modificare lo stato iniziale di M per muovere inizialmente la dita su tutti i possibili stati che possono essere raggiunti dallo stato iniziale di N attraverso gli ε -archi. Cambiare q'_0 in $E(\{q_0\})$ realizza questo risultato. Ora abbiamo completato la costruzione del DFA M che simula l'NFA N .

Ovviamente la costruzione di M funziona correttamente. A ogni passo nella computazione di M su un input, chiaramente entra in uno stato che corrisponde al sottoinsieme di stati in cui N potrebbe essere a quel punto. Quindi la nostra prova è completa.

Corollario 1.40

Corollario 1.40

Un linguaggio è regolare se e solo se qualche automa finito non deterministico lo riconosce.

ESEMPIO 1.41

Illustriamo la procedura che abbiamo dato nella prova del Teorema 1.39 per trasformare un NFA in un DFA usando la macchina N_4 che appare (presente) nell'Esempio 1.35. Per chiarezza, abbiamo rinominato gli stati di N_4 in $\{1, 2, 3\}$. Quindi, nella descrizione formale di $N_4 = (Q, \{a, b\}, \delta, 1, \{1\})$, l'insieme degli stati Q è $\{1, 2, 3\}$ come mostrato nella Figura 1.42.

Per costruire un DFA D equivalente a N_4 , innanzitutto determiniamo gli stati di D . N_4 ha tre stati, $\{1, 2, 3\}$, quindi costruiamo D con otto stati, uno per ogni sottoinsieme dell'insieme degli stati di N_4 . Etichettiamo ognuno degli stati di D con il corrispondente sottoinsieme. Quindi l'insieme degli stati di D è

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

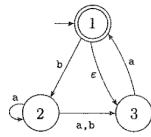


FIGURA 1.42
L'NFA N_4

Poi, determiniamo lo stato iniziale e gli stati accettanti di D . Lo stato iniziale è $E(\{1\})$, l'insieme degli stati che sono raggiungibili da 1 passando attraverso ε -archi, più 1 stesso. Un ε -arco va da 1 a 3, quindi $E(\{1\}) = \{1, 3\}$. I nuovi stati accettanti sono quelli che contengono lo stato accettante di N_4 ; quindi $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.

Infine, determiniamo le funzioni di transizione di D . Ciascuno degli stati di D va in una posizione sull'input a e in una posizione sull'input b . Illustriamo il processo di determinare la posizione degli archi di transizione di D con pochi esempi.

In D , lo stato $\{2\}$ va in $\{2, 3\}$ sull'input a perché in N_4 , lo stato 2 va in 2 che in 3 sull'input a e non possiamo andare più lontano da 2 o 3 attraverso ε -archi. Lo stato $\{2\}$ va nello stato $\{3\}$ sull'input b perché in N_4 , lo stato 2 va solo nello stato 3 sull'input b e da 3 non possiamo andare più lontano attraverso ε archi.

Lo stato $\{1\}$ va in \emptyset sull'input a perché nessun arco etichettato con a esce da esso. Va in $\{2\}$ sull'input b . Nota che la procedura nel Teorema 1.39 specifica che seguiamo gli ε archi *dopo* ogni simbolo di input che viene letto. Una procedura alternativa basata sul seguire gli ε archi prima di leggere ogni simbolo funziona ugualmente bene, ma questo metodo non è illustrato in questo esempio.

Lo stato $\{3\}$ va in $\{1, 3\}$ sull'input a perché in N_4 , lo stato 3 va in 1 sull'input a e 1 a sua volta va in 3 con un ε arco. Lo stato $\{3\}$ va in \emptyset sull'input b .

Lo stato $\{1, 2\}$ va in $\{2, 3\}$ sull'input a perché 1 non punta ad alcuno stato con a archi, 2 punta sia a 2 che a 3 con a archi, e nessuno dei due punta altrove con ε archi. Lo stato $\{1, 2\}$ va in \emptyset sull'input b . Continuando in questo modo, otteniamo il diagramma per D nella Figura 1.43.

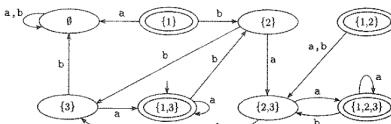


FIGURA 1.43
Un DFA D che è equivalente all'NFA N_4

Possiamo semplificare questa macchina osservando che nessun arco punta agli stati $\{1\}$ e $\{1, 2\}$, quindi essi possono essere tolti senza ripercussioni sulla prestazione della macchina. Facendo così si ottiene la figura seguente.

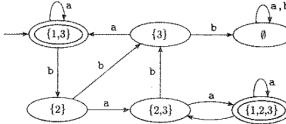


FIGURA 1.44
Il DFA D dopo aver tolto gli stati non necessari

Figure 2: Conversione da NFA a DFA

Chiusura rispetto alle operazioni regolari Il nostro scopo è provare che l'unione, la concatenazione e lo star di linguaggi regolari sono ancora regolari. L'uso del non determinismo rende queste prove molto più semplici. Innanzitutto consideriamo la chiusura rispetto all'unione.

La classe dei linguaggi regolari è chiusa rispetto all'operazione di unione.

Teorema 1.45(oramai)

Teorema 1.45 (oramai)

La classe dei linguaggi regolari è chiusa rispetto all'operazione di unione.

IDEA: Abbiamo i linguaggi regolari A_1 e A_2 e vogliamo provare che $A_1 \cup A_2$ è regolare. L'idea è prendere due NFA, N_1 e N_2 , che riconoscono rispettivamente A_1 e A_2 , e comporli in un nuovo NFA, N .

La macchina N deve accettare il suo input se N_1 o N_2 accettano questo input. La nuova macchina ha un nuovo stato iniziale che si dirama negli stati iniziali delle vecchie macchine tramite ε -archi. In questo modo, se una delle due macchine accetta l'input, anche N lo accetterà.

Rappresentiamo questa costruzione nella figura seguente. Sulla sinistra, indichiamo lo stato iniziale e gli stati accettanti delle macchine N_1 ed N_2 con cerchi grandi e alcuni ulteriori stati con cerchi piccoli. Sulla destra, mostriamo come comporre N_1 ed N_2 in N aggiungendo archi di transizione supplementari.

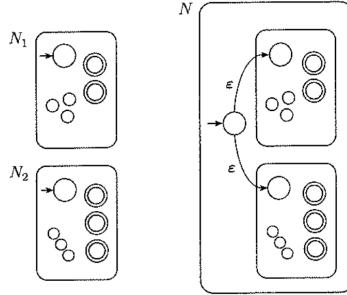


Figure 3: Esempio di NFA che riconosce l'unione di due linguaggi regolari

DIMOSTRAZIONE: Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ che riconosce A_2 . Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ per riconoscere $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
 - Gli stati di N sono tutti gli stati di N_1 e N_2 , con l'aggiunta di un nuovo stato iniziale q_0 .
2. Lo stato q_0 è lo stato iniziale di N .
3. L'insieme degli stati accettanti $F = F_1 \cup F_2$.
 - Gli stati accettanti di N sono tutti gli stati accettanti di N_1 e N_2 . In questo modo, N accetta se N_1 accetta o N_2 accetta.
4. Definiamo δ in modo che per ogni $q \in Q$ e per ogni $a \in \Sigma_\varepsilon$,

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

Teorema 1.47(orale)

Teorema 1.47 (orale)

La classe dei linguaggi regolari è chiusa rispetto all'operazione di concatenazione.

IDEA: Abbiamo i linguaggi regolari A_1 e A_2 e vogliamo provare che $A_1 \circ A_2$ è regolare. L'idea è sempre quella di prendere due NFA e combinarli in uno solo, ma questa volta in modo diverso. Poniamo come stato iniziale di N lo stato iniziale di N_1 . Gli stati accettanti di N_1 hanno ulteriori ε -archi che non deterministicamente permettono di diramarsi in N_2 ogni volta che N_1 è in uno stato accettante, indicando che ha trovato un pezzo iniziale dell'input che costituisce una stringa in A_1 . Gli stati accettanti di N sono solo gli stati accettanti di N_2 . Quindi, esso accetta quando l'input può essere diviso in due parti, la prima accettata da N_1 e la seconda da N_2 .

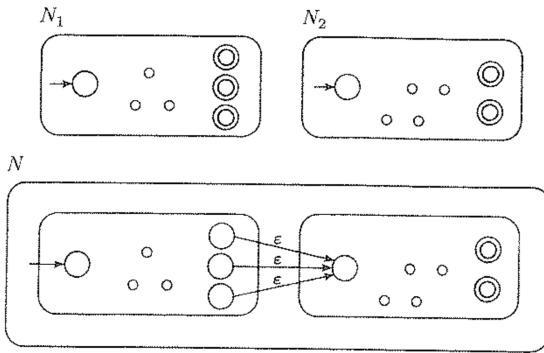


Figure 4: Esempio di concatenazione di due linguaggi regolari

DIMOSTRAZIONE: Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ che riconosce A_2 . Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ per riconoscere $A_1 \cdot A_2$.

1. $Q = Q_1 \cup Q_2$.
 - Gli stati di N sono tutti gli stati di N_1 e N_2 .
2. Lo stato q_1 è uguale allo stato iniziale di N_1 .
3. $F = F_2$.
 - Gli stati accettanti di N sono gli stati accettanti di N_2 .
4. Definiamo δ in modo che per ogni $q \in Q$ e ogni $a \in \Sigma_\varepsilon$,

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

Teorema 1.49(ocale)

Teorema 1.49 (ocale)

La classe dei linguaggi regolari è chiusa rispetto all'operazione star.

IDEA: Abbiamo un linguaggio regolare A_1 e vogliamo provare che anche A_1^* è regolare. Prendiamo un NFA N_1 per A_1 e lo modifichiamo per riconoscere A_1^* . L'NFA N risultante accetterà il suo input quando esso può essere diviso in varie parti ed N_1 accetta ogni parte. Possiamo costruire N come N_1 con ε -archi supplementari che dagli stati accettanti ritornano allo stato iniziale. In questo modo, quando l'elaborazione giunge alla fine di una parte che N_1 accetta, la macchina N ha a scelta di tornare indietro allo stato iniziale per provare a leggere un'altra parte che N_1 accetta. Inoltre, dobbiamo modificare N in modo che accetti ε , che è sempre un elemento di A_1^* . L'idea è di aggiungere un nuovo stato iniziale, che è anche uno stato accettante, e che ha un ε -arco entrante nel vecchio stato iniziale.

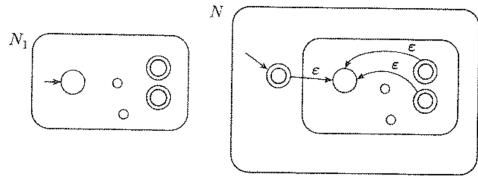


Figure 5: Esempio di operazione star su un linguaggio regolare

DIMOSTRAZIONE: Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ che riconosce A_1 . Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ per riconoscere A_1^* .

1. $Q = \{q_0\} \cup Q_1$.
 - Gli stati di N sono gli stati di N_1 più un nuovo stato iniziale.
2. Lo stato q_0 è il nuovo stato iniziale.
3. $F = \{q_0\} \cup F_1$.
 - Gli stati accettanti sono i vecchi stati accettanti più il nuovo stato iniziale.
4. Definiamo δ in modo che per ogni $q \in Q$ e ogni $a \in \Sigma_\varepsilon$,

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

1.3 Espressioni regolari

In aritmetica, possiamo usare le operazioni $+$, \times per costruire espressioni come

$$(5 + 3) \times 4.$$

Analogamente, possiamo usare le operazioni regolari per costruire espressioni che descrivono linguaggi, che sono chiamate **espressioni regolari**. Un esempio è:

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

Il valore dell'espressione aritmetica è il numero 32, mentre il valore di un'espressione regolare è un linguaggio. In questo caso, il valore è il linguaggio che consiste di tutte le stringhe che iniziano con uno 0 o un 1 seguito da un qualsiasi numero di simboli uguali a 0. Le espressioni regolari hanno un ruolo importante nelle applicazioni dell'informatica. Nelle applicazioni che coinvolgono testo, gli utenti possono voler cercare stringhe che soddisfano alcuni schemi.

ESEMPIO 1.51

Un altro esempio di espressione regolare è

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

Essa inizia con il linguaggio $(0 \cup 1)$ a cui applica l'operatore $*$. Il valore di questa espressione è il linguaggio che consiste di tutte le possibili stringhe di simboli 0 e 1. Se $\Sigma = \{0, 1\}$, possiamo usare

Σ come abbreviazione per l'espressione regolare $(0 \cup 1)$. Più in generale, se Σ è un qualsiasi alfabeto, l'espressione regolare Σ descrive il linguaggio che consiste di tutte le stringhe di lunghezza 1 su questo alfabeto e Σ^* descrive il linguaggio che consiste di tutte le stringhe su quell'alfabeto. Analogamente, Σ^*1 è il linguaggio che contiene tutte le stringhe che terminano con 1. Il linguaggio $(0\Sigma^*) \cup (\Sigma^*1)$ consiste di tutte le stringhe che iniziano con uno 0 o terminano con un 1.

IMPORTANTE!

Nelle espressioni regolari, l'operazione star è eseguita per prima, seguita dalla concatenazione e infine dall'unione, a meno che delle parentesi non cambino l'ordine usuale.

1.3.1 Definizione formale di espressioni regolari

Definizione

Diciamo che R è un'**espressione regolare** se R è

1. a per qualche a nell'alfabeto Σ ,
2. ε ,
3. 0,
4. $(R_1 \cup R_2)$, dove R_1 ed R_2 sono espressioni regolari,
5. $(R_1 \circ R_2)$, dove R_1 ed R_2 sono espressioni regolari, o
6. (R_1^*) , dove R_1 è un'espressione regolare.

Nei punti 1 e 2, le espressioni regolari a e ε rappresentano i linguaggi $\{a\}$ e $\{\varepsilon\}$, rispettivamente. Nel punto 3, l'espressione regolare 0 rappresenta il linguaggio vuoto. Nei punti 4, 5, e 6, le espressioni rappresentano i linguaggi ottenuti prendendo l'unione o la concatenazione dei linguaggi R_1 ed R_2 , o lo star del linguaggio R_1 , rispettivamente.

ATTENZIONE!

Non confondere le espressioni regolari ε e 0. L'espressione ε rappresenta il linguaggio che contiene una sola stringa - ossia, la stringa vuota - mentre 0 rappresenta il linguaggio che non contiene alcuna stringa.

Apparentemente, sembra che stiamo definendo la nozione di espressione regolare in termini di sé stessa. Se fosse vero, avremmo una definizione circolare, che non sarebbe valida. Comunque, R_1 , ed R_2 sono sempre più piccole di R . Quindi in realtà stiamo definendo le espressioni regolari in termini di espressioni regolari più piccole, evitando in tal modo la circolarità. Una definizione di questo tipo è chiamata una **definizione induttiva**.

Le parentesi in un'espressione possono essere omesse. Se lo sono, la valutazione è fatta nell'ordine della precedenza: star, poi concatenazione, poi unione. Per comodità, usiamo R^+ come abbreviazione per RR^* . In altre parole, mentre R^* contiene tutte le stringhe che sono concatenazione di 0 o più stringhe di R , il linguaggio R^+ contiene tutte le stringhe che sono concatenazione di 1 o più stringhe di R . Quindi $R^+ \cup \varepsilon = R^*$. Inoltre, denotiamo con R^k l'abbreviazione della concatenazione di k copie di R insieme.

Quando vogliamo distinguere tra un'espressione regolare R e il linguaggio che descrive, denotiamo con $L(R)$ il linguaggio di R .

ESEMPIO 1.53

Negli esempi seguenti, assumiamo che l'alfabeto Σ sia $\{0,1\}$.

1. $0^*10^* = \{w \mid w \text{ contiene un solo } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ ha almeno un } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contiene la stringa } 001 \text{ come sottostringa}\}$.
4. $1^*(01^+)^* = \{w \mid \text{ogni } 0 \text{ in } w \text{ è seguito da almeno un } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ è una stringa di lunghezza pari}\}^5$.
6. $(\Sigma\Sigma)^* = \{w \mid \text{la lunghezza di } w \text{ è un multiplo di } 3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ inizia e termina con lo stesso simbolo}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
L'espressione $0 \cup \varepsilon$ descrive il linguaggio $\{0, \varepsilon\}$, quindi l'operazione di concatenazione aggiunge 0 o ε prima di ogni stringa in 1^* .
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
Concatenando l'insieme vuoto a un qualsiasi insieme si ottiene l'insieme vuoto.
12. $\emptyset^* = \{\varepsilon\}$.
L'operazione star concatena un numero qualsiasi di stringhe del linguaggio per ottenere una stringa nel risultato. Se il linguaggio è vuoto, l'operazione star può concatenare 0 stringhe, dando solo la stringa vuota.

Se R è un'espressione regolare qualsiasi, abbiamo le seguenti identità. Esse sono un buon test per controllare se hai capito la definizione.

La lunghezza di una stringa è il numero di simboli che essa contiene.

Figure 6: Esempio di espressione regolare

$R \cup 0 = R$. Aggiungere il linguaggio vuoto a un qualsiasi altro linguaggio non lo cambia.

$R \circ \varepsilon = R$. Concatenare la stringa vuota a una qualsiasi stringa non la cambia.

Tuttavia, scambiare 0 e ε nelle precedenti identità può far sì che le uguaglianze non siano vere. - $R \cup \varepsilon$ può non essere uguale a R . - Per esempio, se $R = 0$, allora $L(R) = \{0\}$, ma $L(R \cup \varepsilon) = \{0, \varepsilon\}$. - $R \circ 0$ può non essere uguale a R . - Per esempio, se $R = 0$, allora $L(R) = \{0\}$ ma $L(R \circ 0) = 0$.

Gli oggetti elementari in un linguaggio di programmazione, chiamati *token*, come i nomi delle variabili e le costanti, possono essere descritti con espressioni regolari. Per esempio, una costante numerica che può avere una parte decimale e/o un segno può essere descritta come un elemento del linguaggio

$$(+ \cup - \cup \varepsilon)(D^+ \cup D^+.D^* \cup D^*.D^+)$$

dove $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ è l'alfabeto delle cifre decimali. Esempi di stringhe generate sono: 72, 3.14159, +7., e -.01.

1.3.2 Equivalenza con gli automi finiti

Le espressioni regolari e gli automi finiti sono equivalenti rispetto alla loro potenza descrittiva. Ogni espressione regolare può essere trasformata in un automa finito che riconosce il linguaggio che essa descrive, e viceversa. Ricorda che in linguaggio regolare è un linguaggio che è riconosciuto da qualche automa finito.

Teorema 1.54

Teorema 1.54

Un linguaggio è regolare se e solo se qualche espressione regolare lo descrive.

Questo teorema deve essere dimostrato in entrambe le direzioni. Noi enunciamo e proviamo ciascuna direzione in un lemma separato.

Lemma 1.55(orale)

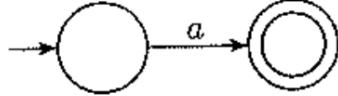
Lemma 1.55 (orale)

Se un linguaggio è descritto da un'espressione regolare, allora esso è regolare.

IDEA: Supponiamo di avere un'espressione regolare R che descrive un linguaggio A . Mostriamo come trasformare R in un NFA che riconosce A . Per il [Corollario 1.40](#), se un NFA riconosce A allora A è regolare.

DIMOSTRAZIONE: Trasformiamo R in un NFA N . Consideriamo i sei casi nella definizione di espressione regolare.

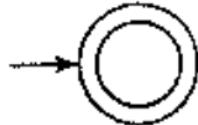
1. $R = a$ per qualche $a \in \Sigma$. Allora $L(R) = \{a\}$ e il seguente NFA riconosce $L(R)$.



Nota che questa macchina soddisfa la definizione di NFA ma non quella di DFA perché ha qualche stato con nessun arco uscente per ogni possibile simbolo di input. Naturalmente, qui avremmo potuto presentare un DFA equivalente; ma un NFA è tutto quello di cui abbiamo bisogno per ora, ed è più facile da descrivere.

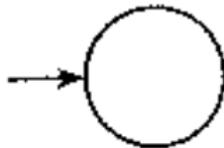
Formalmente, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, dove descriviamo δ dicendo che $\delta(q_1, a) = \{q_2\}$ e $\delta(r, b) = \emptyset$ per $r \neq q_1$ o $b \neq a$.

2. $R = \varepsilon$. Allora $L(R) = \{\varepsilon\}$ e il seguente NFA riconosce $L(R)$.



Formalmente, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ dove $\delta(r, b) = \emptyset$ per ogni r e b .

3. $R = 0$. Allora $L(R) = 0$, e il seguente NFA riconosce $L(R)$.



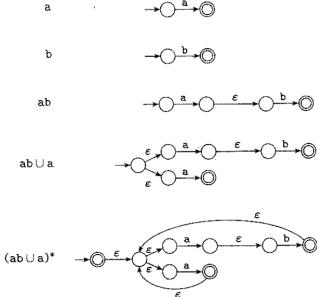
Formalmente, $N = (\{q\}, \sigma, \delta, q, 0)$, dove $\delta(r, b) = \emptyset$ per ogni r e b .

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

Per gli ultimi tre casi, usiamo le costruzioni date nelle prove che la classe dei linguaggi regolari è chiusa rispetto alle operazioni regolari. In altre parole, costruiamo l'NFA per R dagli NFA per R_1 ed R_2 (o solo R_1 nel caso 6) e mediante l'appropriata costruzione della chiusura.

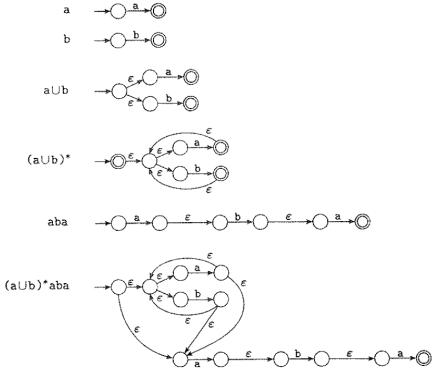
ESEMPIO 1.56

Trasformiamo l'espressione regolare $(ab \cup a)^*$ in un **NFA** in una sequenza di passi. Costruiamo l'automa dalle sottoespressioni più piccole alle sottoespressioni più grandi finché abbiamo un **NFA** per l'espressione iniziale, come mostrato nel diagramma seguente. Nota che questa procedura generalmente non dà l'**NFA** con il minor numero di stati. In questo esempio, la procedura dà un **NFA** con otto stati, ma il più piccolo **NFA** equivalente ha solo due stati. Riesci a trovarlo?



ESEMPIO 1.58

Nella Figura 1.59, trasformiamo l'espressione regolare $(a \cup b)^*aba$ in un **NFA**. Alcuni dei passi meno importanti non sono mostrati.



Lemma 1.60(orale 24-30)

Lemma 1.60 (orale 24-30)

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

IDEA.

Dobbiamo mostrare che se un linguaggio A è regolare, allora un'espressione regolare lo descrive. Poiché A è regolare, esso è accettato da un **DFA**. Descriviamo una procedura per trasformare i **DFA** in espressioni regolari equivalenti. Dividiamo questa procedura in due parti, usando un nuovo tipo di automa finito chiamato **automa finito non deterministico generalizzato**, **GNFA**. Prima mostriamo come trasformare un **DFA** in un **GNFA** e poi come trasformare un **GNFA** in un'espressione regolare.

Gli automi finiti non deterministici generalizzati sono semplicemente automi finiti non deterministici nei quali gli archi delle transizioni possono avere espressioni regolari come etichette, invece che solo elementi dell'alfabeto o ϵ . Il **GNFA** legge blocchi di simboli dall'input, non necessariamente solo un simbolo alla volta come in un comune **NFA**. Il **GNFA** si muove lungo un arco di transizione che collega due stati leggendo un blocco di simboli dall'input che formano una stringa descritta dall'espressione regolare su quell'arco. Un **GNFA** è non deterministico e quindi può avere diversi modi di elaborare la stessa stringa di input. Esso accetta il suo input se la sua elaborazione può far sì che il **GNFA** sia in uno stato accettante alla fine dell'input. La figura seguente presenta un esempio di un **GNFA**.

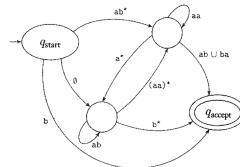


Figure 7: Esempio di GNFA

Per comodità, richiediamo che i **GNFA** abbiano sempre una forma speciale che soddisfi le seguenti condizioni.

- Lo stato iniziale ha archi di transizione uscenti verso un qualsiasi altro stato ma nessun arco entrante proveniente da un qualsiasi altro stato.
- Esiste un solo stato accettante, ed esso ha archi entranti provenienti da un qualsiasi altro stato ma nessun arco uscente verso un qualsiasi altro stato. Inoltre, lo stato accettante non è uguale allo stato iniziale.
- Eccetto che per lo stato iniziale e lo stato accettante, un arco va da ogni stato ad ogni altro stato e anche da ogni stato in se stesso.

Possiamo facilmente trasformare un DFA in un GNFA nella forma speciale. Aggiungiamo semplicemente un nuovo stato iniziale con un ε -arco che entra nel vecchio stato iniziale e un nuovo stato accettante con ε -archi entranti, provenienti dai vecchi stati accettanti. Se alcuni archi hanno più etichette (o se ci sono più archi che collegano gli stessi due stati nella stessa direzione), sostituiamo ognuno di essi con un solo arco la cui etichetta è l'unione delle precedenti etichette. Infine, aggiungiamo archi con etichetta 0 tra stati che non hanno archi. Questo ultimo passo non cambierebbe il linguaggio riconosciuto perché una transizione etichettata con 0 non può mai essere usata. Ora mostriamo come trasformare un GNFA in un'espressione regolare. Supponiamo che il GNFA abbia k stati. Allora, poiché un GNFA deve avere uno stato iniziale e uno accettante ed essi devono essere diversi tra loro, sappiamo che $k \geq 2$. Se $k > 2$, costruiamo un GNFA equivalente con $k - 1$ stati. Questo passo può essere ripetuto sul nuovo GNFA fino a quando esso è ridotto a 2 stati. Se $k = 2$ il GNFA ha un solo arco che va dallo stato iniziale allo stato accettante. L'etichetta di questo arco è l'espressione regolare equivalente. Per esempio, le fasi per trasformare un DFA con 3 stati in un'espressione regolare equivalente sono mostrati nella figura seguente.

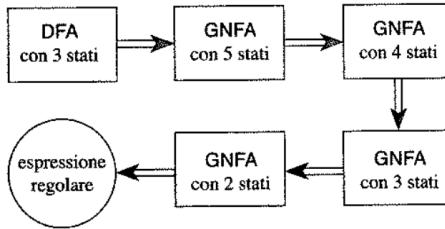


Figure 8: Trasformazione di un GNFA in un'espressione regolare

Il passo cruciale è costruire un GNFA equivalente con uno stato in meno quando $k \geq 2$. Lo facciamo scegliendo uno stato, estraendolo dalla macchina, e modificando il resto in modo che sia ancora riconosciuto lo stesso linguaggio. Ogni stato può essere scelto, purchè non sia lo stato iniziale o lo stato accettante. Siamo certi che un tale stato esiste perchè $k > 2$. Chiamiamo q_{rip} lo stato rimosso.

Dopo aver rimosso q_{rip} modifichiamo la macchina cambiando le espressioni regolari che etichettano ciascuno dei restanti archi. Le nuove etichette controbilanciano l'assenza di q_{rip} reintegrando le computazioni perse. La nuova etichetta che va da uno stato q_i a uno stato q_j è un'espressione regolare che descrive tutte le stringhe che porterebbero la macchina da q_i a q_j o direttamente o tramite q_{rip} . Illustriamo questa strategia nella seguente figura.

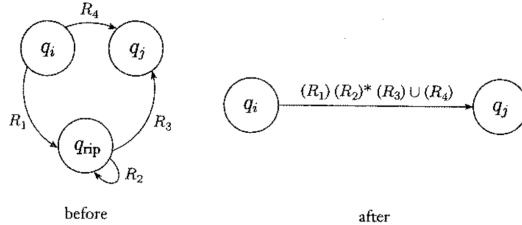


Figure 9: Esempio di trasformazione di un GNFA

Nella vecchia macchina, se

1. q_i va a q_{rip} con un arco etichettato R_1 ,
2. q_{rip} va in se stesso con un arco etichettato R_2 ,
3. q_{rip} va a q_j con un arco etichettato R_3 ,
4. q_i va a q_j con un arco etichettato R_4 ,

allora nella nuova macchina, l'arco da q_i a q_j riceve l'etichetta

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

Facciamo questa modifica per ogni arco che va da un qualsiasi stato q_i a un qualsiasi stato q_j , includendo il caso un cui $q_i = q_j$. La nuova macchina riconosce il linguaggio di partenza.

DIMOSTRAZIONE.

Formalizziamo questa idea. In primo luogo, per rendere più facile la prova, definiamo formalmente il nuovo tipo di automa introdotto. Un GNFA è simile a un automa finito non deterministico tranne che per la funzione di transizione, che ha la forma

$$\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow R.$$

Il simbolo R è la collezione di tutte le espressioni regolari sull'alfabeto Σ , e q_{start} e q_{accept} sono gli stati iniziale e accettante. Se $\delta(q_i, q_j) = R$, l'arco dallo stato q_i allo stato q_j ha come etichetta l'espressione regolare R . Il dominio della funzione di transizione è $(Q - \{q_{accept}\}) \times (Q - \{q_{start}\})$ perché un arco collega ogni stato a un qualsiasi altro stato, tranne che nessun arco è uscente da q_{accept} o è entrante in q_{start} .

Definizione 1.64(oramai 24-30)

Definizione 1.64 (oramai 24-30)

Un automa finito non deterministico generalizzato è una quintupla, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, dove:

1. Q è l'insieme finito degli stati,
2. Σ è l'alfabeto di input,
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow R$ è la funzione di transizione,
4. q_{start} è lo stato iniziale e
5. q_{accept} è lo stato accettante.

Un GNFA accetta una stringa w in Σ^* se $w = w_1w_2\dots w_k$, dove ogni w_i è in Σ^* ed esiste una sequenza di stati q_0, q_1, \dots, q_k tale che:

1. $q_0 = q_{start}$ è lo stato iniziale,
2. $q_k = q_{accept}$ è lo stato accettante, e
3. per ogni i , risulta $w_i \in L(R_i)$, dove $R_i = \delta(q_{i-1}, q_i)$; in altre parole, R_i è l'espressione sull'arco da q_{i-1} a q_i .

Tornando alla prova del Lemma 1.60, sia M il DFA per il linguaggio A. Allora trasformiamo M in un GNFA G aggiungendo un nuovo stato iniziale e un nuovo stato accettante e i necessari archi di transizione supplementari. Usiamo la procedura $CONVERT(G)$, che prende un GNFA e restituisce un'espressione regolare equivalente. Questa procedura usa la **ricorsione**, che significa che essa chiama se stessa. Evitiamo un ciclo infinito perché la procedura chiama se stessa solo per elaborare un GNFA che ha uno stato in meno. Il caso un cui il GNFA ha due stati è trattato senza ricorsione.

$CONVERT(G)$:

1. Sia k il numero di stati di G .
2. Se $k = 2$, allora G deve consistere di uno stato iniziale, uno stato accettante, e un singolo arco che li collega ed è etichettato con un'espressione regolare R . Restituisce l'espressione R .
3. Se $k > 2$, scegliamo un qualsiasi stato $q_{rip} \in Q$ diverso da q_{start} e q_{accept} e sia G' il GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$, dove

$$Q' = Q - \{q_{rip}\},$$

e per ogni $q_i \in Q' - \{q_{accept}\}$ e ogni $q_j \in Q' - \{q_{start}\}$, poniamo

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

dove $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$ ed $R_4 = \delta(q_i, q_j)$.

4. Calcola $CONVERT(G')$ e restituisce questo valore.

Di seguito proviamo che $CONVERT$ restituisce un valore corretto.

FATTO 1.65

Per ogni GNFA G , $CONVERT(G)$ è equivalente a G .

Proviamo quest'affermazione per induzione su k , il numero di stati del GNFA.

Base: Proviamo che l'affermazione è vera per $k = 2$ stati. Se G ha solo due stati, esso può avere solo un singolo arco, che va dallo stato iniziale allo stato accettante. L'espressione regolare che è l'etichetta di quest'arco descrive tutte le stringhe che permettono a G di raggiungere lo stato accettante. Quindi quest'espressione è equivalente a G .

Passo induttivo: Assumiamo che l'affermazione sia vera per $k - 1$ stati e usiamo questa ipotesi per provare che l'affermazione è vera per k stati. In primo luogo mostriamo che G e G' riconoscono lo stesso linguaggio. Supponiamo che G accetti un input w . Allora in un ramo accettante della computazione, G entra in una sequenza di stati:

$$q_{\text{start}}, q_1, q_2, q_3, \dots, q_{\text{accept}}$$

Se nessuno di essi è lo stato rimosso q_{rip} , evidentemente anche G' accetta w . La ragione è che ciascuna delle nuove espressioni regolari che etichettano gli archi di G' contiene le vecchie espressioni regolari come parte di un'unione.

Se q_{rip} è presente, eliminando ogni sequenza di stati q_{rip} consecutivi creiamo una computazione accettante per G' . Gli stati q_i e q_j che raggruppano una tale sequenza hanno una nuova espressione regolare sull'arco tra essi, la quale descrive tutte le stringhe che portano da q_i a q_j attraverso q_{rip} in G . Quindi G' accetta w .

Viceversa, supponiamo che G' accetti un input w . Poiché ciascun arco tra due qualsiasi stati q_i e q_j in G' descrive la collezione delle stringhe che portano da q_i a q_j in G , o direttamente o attraverso q_{rip} , anche G deve accettare w . Quindi G e G' sono equivalenti.

L'ipotesi induttiva afferma che quando l'algoritmo chiama sé stesso ricorsivamente sull'input G' , il risultato è un'espressione regolare che è equivalente a G' perché G' ha $k - 1$ stati. Quindi anche questa espressione regolare è equivalente a G , e abbiamo provato che l'algoritmo è corretto.

Questo conclude la prova del Claim 1.65, del Lemma 1.60, e del Teorema 1.54.

Esempio 1.66

In questo esempio, usiamo il precedente algoritmo per trasformare un DFA in un'espressione regolare. Iniziamo con il DFA con due stati nella Figura 1.67(a). Nella Figura 1.67(b), creiamo un GNFA con quattro stati aggiungendo un nuovo stato iniziale e un nuovo stato accettante, chiamati s e a invece di q_{start} e q_{accept} in modo da poterli disegnare in modo conveniente. Per evitare di ingombrare la figura, non disegniamo gli archi etichettati 0, sebbene essi vi siano. Nota che sostituiamo le etichette a, b sul ciclo nello stato 2 del DFA con l'etichetta $a \cup b$ nel corrispondente punto del GNFA. Facciamo in questo modo perché le etichette del DFA rappresentano due transizioni, una per a e l'altra per b , mentre il GNFA può avere solo una singola transizione che va da 2 a sé stesso. Nella Figura 1.67(c), eliminiamo lo stato 2 e aggiorniamo le etichette degli archi restanti. In questo caso, la sola etichetta che cambia è quella da 1 ad a . Nella parte (b) era 0, ma nella parte (c) essa è $b(a \cup b)^*$. Otteniamo questo risultato seguendo il passo 3 della procedura $CONVERT$. Lo stato q_i è lo stato 1, lo stato q_j è a , e q_{rip} è 2, quindi $R_1 = b$, $R_2 = a \cup b$, $R_3 = \varepsilon$ ed $R_4 = 0$. Pertanto, la nuova etichetta sull'arco da 1 ad a è $(b)(a \cup b)^*(\varepsilon) \cup 0$. Semplifichiamo quest'espressione regolare in $b(a \cup b)^*$. Nella Figura 1.67(d), eliminiamo lo stato 1 dalla parte (c) e seguiamo la stessa procedura. Poiché restano solo lo stato iniziale e lo stato accettante, l'etichetta sull'arco che li collega è l'espressione regolare equivalente al DFA iniziale.

ESEMPIO 1.68

In questo esempio, iniziamo con un DFA con tre stati. I passi nella trasformazione sono mostrati nella figura seguente.

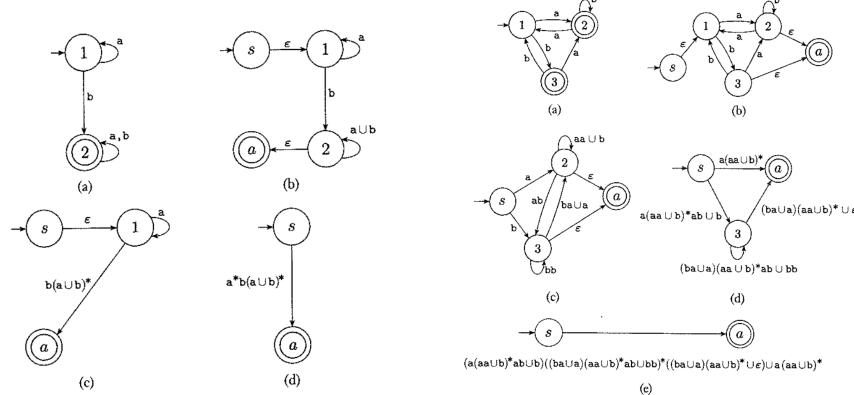


Figure 10: Trasformazione di un DFA in un'espressione regolare

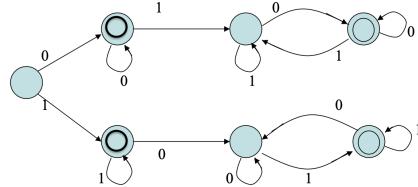
1.3.3 Linguaggi non regolari

In questa sezione, mostriamo come provare che alcuni linguaggi non possono essere riconosciuti da alcun automa finito.

Consideriamo il linguaggio $B = \{0^n 1^n \mid n \geq 0\} = \{\epsilon, 01, 0011, 000111, \dots\}$. Intuitivamente un DFA che riconosce B dovrebbe ricordare quanti 0 ha visto fin quando legge l'input, ma non è possibile con un numero finito di stati. Di seguito presentiamo un metodo per provare che linguaggi come B non sono regolari. Solo perché il linguaggio sembra richiedere memoria non limitata, non significa che non sia regolare. Per esempio, consideriamo due linguaggi sull'alfabeto $\Sigma = \{0, 1\}$:

- $C = \{w \mid w \text{ ha lo stesso numero di simboli uguali a } 0 \text{ e simboli uguali a } 1\}$ non è regolare;
- $D = \{w \mid w \text{ ha un numero uguale di occorrenze di } 01 \text{ e } 10 \text{ come sottostringhe}\}$.

Sorprendentemente, D è regolare ed è accettato dal seguente DFA:



Il pumping lemma

Questo teorema afferma che tutti i linguaggi regolari hanno una proprietà speciale. Se noi possiamo mostrare che un linguaggio non ha questa proprietà, siamo sicuri che esso non è regolare. La proprietà afferma che tutte le stringhe nel linguaggio possono essere "replicate" se la loro lunghezza raggiunge almeno uno specifico valore speciale, chiamato la **lunghezza del pumping**. Questo significa che ogni tale stringa contiene una parte che può essere ripetuta un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

Teorema 1.70(orale)

Teorema 1.70

Se A è un linguaggio regolare, allora esiste un numero p (la lunghezza del pumping) tale che se s è una qualsiasi stringa in A di lunghezza almeno p , allora s può essere divisa in tre parti, $s = xyz$, soddisfacenti le seguenti condizioni:

1. per ogni $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, e
3. $|xy| \leq p$.

IMPORTANTE!

Ricordiamo che $|s|$ rappresenta la lunghezza della stringa s , y^i indica i copie di y concatenate insieme, e y^0 è uguale a ε .

- Quando s è divisa in xyz , x o z potrebbe essere ε , ma la condizione 2 dice che $y \neq \varepsilon$ (senza la condizione 2 il teorema sarebbe banalmente vero).
- La condizione 3 afferma che le parti x e y insieme hanno lunghezza al più p ; questa è una condizione tecnica supplementare che ogni tanto troviamo utile quando dimostriamo che alcuni linguaggi non sono regolari.
- Scritto più precisamente il pumping lemma è:

PUMPING LEMMA:

$$A \text{ regolare} \Rightarrow \exists p \in \mathbb{N} \forall s \in A (|s| \geq p \Rightarrow \exists x, y, z \text{ t.c. } (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \in \mathbb{N}. xy^i z \in A))$$

CONTRAPPOSTA: (il prof dice di usare questa, daje G. !)

$$\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall x, y, z (s \neq xyz \vee |y| = 0 \vee |xy| > p \vee \exists i \in \mathbb{N}. xy^i z \notin A)) \Rightarrow A \text{ non regolare}$$

EQUIVALENTEMENTE:

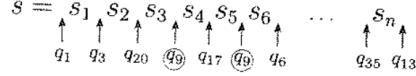
$$\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall x, y, z ((s = xyz \wedge |y| > 0 \wedge |xy| \leq p))) \Rightarrow A \text{ non regolare}$$

Utilizzo pratico (provare che A non è regolare):

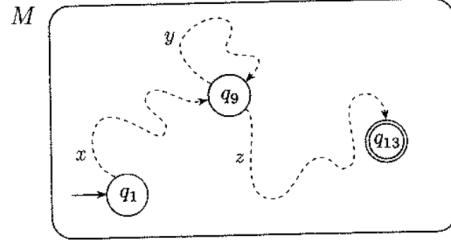
1. Scegliere p arbitrario.
2. Scegliere $s \in A$ lunga al massimo p e decomponila in tutti i possibili xyz , con $|y| > 0$ e $|xy| \leq p$.
3. Per ogni decomposizione mostrare che $\forall i \in \mathbb{N}. xy^i z \notin A$.
4. Concludere che A non è regolare.

IDEA: Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce A . Assegnamo alla lunghezza del pumping p il numero degli stati di M . Mostriamo che ogni stringa s in A di lunghezza almeno p può essere divisa nelle tre parti xyz , soddisfacenti le nostre tre condizioni. Cosa accade se nessuna stringa in A è di lunghezza almeno p ? Allora il nostro compito è perfino più facile perché il teorema diventa *banalmente* vero: ovviamente le tre condizioni valgono per tutte le stringhe di lunghezza almeno p se non vi è alcuna di tali stringhe. Se s in A ha lunghezza almeno p , consideriamo la sequenza di stati che M attraversa nella computazione con input s . Inizia con lo stato iniziale q_1 , poi va in q_3 , poi per esempio in q_{13} . Se s è in A , sappiamo che M accetta s , quindi q_{13} è uno stato accettante. Se supponiamo che n sia la lunghezza

di s , la sequenza di stati $q_1, q_3, q_{20}, \dots, q_{13}$ ha lunghezza $n + 1$. Poichè n è almeno p , sappiamo che $n + 1$ è più grande di p , il numero di stati di M . Quindi, la sequenza deve contenere uno stato che si ripete (**principio della piccionaia**). La figura seguente mostra la stringa s e la sequenza di stati che M attraversa quando elabora s . Lo stato q_9 è quello che si ripete.



Ora dividiamo s nelle tre componenti x, y e z . La componente x è la parte di s che compare prima di q_9 , la componente y è la parte tra le due occorrenze di q_9 e la componente z è la parte restante di s , che viene dopo la seconda occorrenza di q_9 . Quindi x porta M dallo stato q_1 a q_9 , y riporta M da q_9 a q_9 e z porta M da q_9 allo stato accettante q_{13} , come mostrato nella figura seguente.



Vediamo perchè questa divisione di s soddisfa le tre condizioni. Supponiamo di eseguire M sull'input $xyyz$. Sappiamo che x porta q_1 a q_9 , e poi il primo y lo riporta da q_9 a q_9 , come fa il secondo y e poi z lo porta in q_{13} . Quindi essendo q_{13} uno stato accettante, M accetta l'input $xyyz$. Il discorso è analogo per xy^iz per ogni $i > 0$. Nel caso $i = 0$, $xy^iz = xz$, anch'essa accettata. Questo prova la condizione 1. Per verificare la condizione 2, vediamo che $|y| > 0$, poichè era la parte di s tra due diverse occorrenze dello stato q_9 . Per ottenere la condizione 3, ci assicuriamo che q_9 sia la prima ripetizione nella sequenza. Per il principio della piccionaia, i primi $p + 1$ stati nella sequenza devono contenere una ripetizione. Quindi $|xy| \leq p$.

DIMOSTRAZIONE: Sia $M = (Q, \Sigma, \delta, q_1, F)$ un DFA che riconosce A e sia p il numero di stati di M . Sia $s = s_1s_2\dots s_n$ una stringa in A di lunghezza n , dove $n \geq p$. Sia r_1, \dots, r_{n+1} la sequenza di stati traversati da M mentre elabora s , quindi $r_{i+1} = \delta(r_i, s_i)$ per $1 \leq i \leq n$. Questa sequenza ha lunghezza $n + 1$, che è almeno $p + 1$. Due tra i primi $p + 1$ stati devono essere lo stesso stato, per il principio della piccionaia. Chiamiamo il primo di questi r_j e il secondo r_l . Poichè r_l si presenta tra le prime $p + 1$ posizioni in una sequenza che inizia in r_1 , abbiamo $l \leq p + 1$. Ora sia $x = s_1\dots s_{j-1}$, $y = s_j\dots s_{l-1}$ e $z = s_l\dots s_n$. Poichè x porta M da r_1 a r_j , y porta M da r_j a r_j e z porta M da r_j a r_{n+1} , che è uno stato accettante, M deve accettare xy^iz per $i \geq 0$. Sappiamo che $j \neq l$, perciò $|y| > 0$; e $l \leq p + 1$, perciò $|xy| \leq p$. Quindi tutte le condizioni del pumping lemma sono rispettate.

Per usare il pumping lemma per provare che un linguaggio B non è regolare, in primo luogo si assume che B sia regolare per ottenere una contraddizione. Poi si usa il pumping lemma per assicurare l'esistenza di una lunghezza del pumping p tale che tutte le stringhe di lunghezza maggiore o uguale a p in B possano essere iterate. In seguito, si trovi una stringa s in B che ha lunghezza maggiore o uguale a p , ma che non può essere iterata. Infine, si dimostri che s non può essere iterata considerando tutti i modi di dividere s in x, y e z (prendendo in considerazione la condizione 3 del pumping lemma se è utile) e, per ogni tale divisione, trovando un valore i tale che $xy^iz \notin B$. Questo passo finale spesso comporta il dover raggruppare i vari modi di dividere s in diversi casi e l'analizzarli individualmente. L'esistenza di s contraddirrebbe il pumping lemma se B fosse regolare. Quindi B non può essere regolare. Trovare s a volte richiede un po di ragionamento creativo. Potresti dover cercare tra diversi candidati per s prima di scoprirla uno che funzioni. Prova con elementi di B che sembrano esibire l'essenza della non regolarità di B .

Usage of the Pumping Lemma

P.L. A regular $\Rightarrow \exists p \in \mathbb{N} \forall s \in A (|s| \geq p \Rightarrow \exists x,y,z \text{ s.t. } (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \in \mathbb{N}. xy^i z \in A))$

Hence, the contrapositive of this statement is
 $\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall x,y,z (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \Rightarrow \exists i \in \mathbb{N}. xy^i z \notin A))$
 $\Rightarrow A$ is not regular

Equivalently:
 $\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall x,y,z ((s = xyz \wedge |y| > 0 \wedge |xy| \leq p) \Rightarrow \exists i \in \mathbb{N}. xy^i z \notin A))$
 $\Rightarrow A$ is not regular

Practical use (for proving that A is not regular):

- Consider a generic p
- Find a string $s \in A$ long at least p and decompose it in all possible xyz , with $|y| \leq p$ and $|y| \leq p$
- For each such decomposition, find an i such that $xy^i z \notin A$
- Then, A is not regular

nota a fondo pagina 3 (pagina 49). Ma $C \cap 0^*1^*$ è uguale a B , e noi sappiamo che B non è regolare dall'Esempio 1.73.

ESEMPIO 1.73

Sia $C = \{w\mid w \in \{0,1\}^*\}$. Mostriamo che C non è regolare, usando il pumping lemma.

Assumiamo al contrario che C sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Scelgiamo s uguale alla stringa 0^p1^p . Poiché s è un elemento di C ed s ha lunghezza maggiore di p , il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa $xy^i z$ è in C . Consideriamo tre casi per mostrare che questo risultato è impossibile.

1. La stringa y consiste solo di simboli uguali a 0. In questo caso, la stringa $xy^i z$ ha più simboli uguali a 0 che simboli uguali a 1 e quindi non è un elemento di C , violando la condizione 1 del pumping lemma. Questo caso porta a una contraddizione.
2. La stringa y consiste solo di simboli uguali a 1. Anche questo caso porta a una contraddizione.
3. La stringa y consiste sia di simboli uguali a 0 che di simboli uguali a 1. In questo caso, la stringa $xy^i z$ può avere lo stesso numero di simboli

notata a fondo pagina 3 (pagina 49). Ma $C \cap 0^*1^*$ è uguale a B , e noi sappiamo che B non è regolare dall'Esempio 1.73.

ESEMPIO 1.75

Sia $F = \{w\mid w \in \{0,1\}^*\}$. Mostriamo che F non è regolare, usando il pumping lemma.

Assumiamo al contrario che F sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia s la stringa 0^p1^p . Poiché s è un elemento di F ed s ha lunghezza maggiore di p , il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, che verificano le tre condizioni del lemma. Mostriamo che questo risultato è impossibile.

La condizione 3 è ancora una volta cruciale perché senza di essa potremmo iterare s se poniamo $x = z$ uguali alla stringa vuota. Con la condizione 3 la prova segue poiché y deve consistere di simboli uguali a 0, quindi $xy^i z \in F$.

Ora vediamo che abbiamo scelto $s = 0^p1^p$ come stringa che esibisce l'«essenza» della non regolarità di F , invece per esempio della stringa 0^p0^p . Sebbene 0^p0^p sia un elemento di F , essa non riesce a dar luogo a una contraddizione poiché può essere iterata.

ESEMPIO 1.76

Mostriamo un linguaggio non regolare unario. Sia $D = \{1^n \mid n \geq 0\}$. In altre parole, D contiene tutte le stringhe di simboli uguali a 1 la cui lunghezza è un quadrato perfetto. Usiamo il pumping lemma per provare che D non è regolare. La prova è per contraddizione.

Assumiamo al contrario che D sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia s la stringa 1^{p^2} . Poiché s è un elemento di D ed s ha lunghezza almeno p , il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa $xy^i z$ è in D . Come negli esempi precedenti, mostriamo che questo risultato è impossibile. Parlo in questo caso richiede una piccola riflessione sulla successione dei quadrati perfetti:

0, 1, 4, 9, 16, 25, 36, 49, ...

Nota il divario crescente tra gli elementi successivi di questa sequenza. Eleggono di questi numeri non possono vicini l'uno l'altro.

Ora consideriamo le due stringhe xz e $xy^2 z$. Queste stringhe differiscono l'una dall'altra per una sola ripetizione di y , e conseguentemente le loro lunghezze differiscono di una quantità uguale alla lunghezza di y . Per la condizione 3 del pumping lemma, $|x| \leq p$ e quindi $|y| \leq p$. Abbiamo $|xy| < p^2$ e allora $|xy^2 z| \leq p^2 + p$. Ma $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Inoltre la condizione 2 implica che y non è la stringa vuota e perciò $|xy^2 z| > p^2$.

uguali a 0 e simboli uguali a 1, ma essi non saranno nell'ordine corretto, con qualche 1 prima di qualche 0. Quindi non è un elemento di B , il che è una contraddizione.

Pertanto una contraddizione è inevitabile se assumiamo che B sia regolare, quindi B non è regolare. Nota che possiamo semplificare questo ragionamento applicando la condizione 3 del pumping lemma per eliminare i casi 2 e 3.

In questo esempio, trovare la stringa s era facile perché una qualsiasi stringa in B di lunghezza p o maggiore avrebbe funzionato. Nei successivi due esempi, alcune scelte per s non funzionano quindi è richiesta un'attenzione supplementare.

ESEMPIO 1.74

Sia $C = \{w\mid w$ ha lo stesso numero di simboli uguali a 0 e simboli uguali a 1 $\}$. Usiamo il pumping lemma per provare che C non è regolare. La prova è per contraddizione.

Assumiamo al contrario che C sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Come nell'Esempio 1.73, sia s la stringa 0^p1^p . Poiché s è un elemento di C che ha lunghezza maggiore di p , il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che per ogni $i \geq 0$ la stringa $xy^i z$ è in C . Ci piacerebbe mostrare che questo risultato è impossibile.

Quindi vediamo che s possa essere iterata.

Qui la condizione 3 nel pumping lemma è utile. Essa stabilisce che quando iteriamo s , essa deve essere divisa in modo che $|xy| \leq p$. Questa restrizione sul modo in cui s può essere divisa rende più facile mostrare che la stringa $s = 0^p1^p$ che abbiamo scelta non può essere iterata. Se $|xy| \leq p$, allora y deve consistere solo di simboli uguali a 0, perciò $xy^i z \notin C$. Quindi, s non può essere iterata. Questo ci fornisce un esempio chiaro di contraddizione.

Scelgono la stringa $s = 0^p1^p$. Questo esempio richiede più attenzione che nell'Esempio 1.73.

Se invece assumessimo $s = (01)^p$, avremmo avuto dei problemi perché abbiamo bisogno di una stringa che non può essere iterata e una stringa può essere iterata, però prendendo in considerazione la condizione 3. Riesci a vedere come iterarla? Un modo per farlo è porre $x = \epsilon$, $y = 01$ e $z = (01)^{p-1}$. Allora $xy^i z \in C$ per ogni valore di i . Se non riesci a trovare una stringa che non può essere iterata al primo tentativo, non disperare. Provare un'altra!

Un metodo alternativo per provare che C non è regolare segue dal fatto che sappiamo che B non è regolare. Se C fosse regolare, anche $C \cap 0^*1^*$ sarebbe regolare. Questo perché il linguaggio 0^*1^* è regolare e la classe dei linguaggi regolari è chiusa rispetto all'intersezione, come provammo nella

Quindi, la lunghezza di $xy^i z$ è compresa strettamente tra i quadrati perfetti consecutivi p^2 e $(p+1)^2$. Perciò questa lunghezza non può essere essa stessa un quadrato perfetto. Dunque arriviamo alla contraddizione che $xy^i z \notin D$ e concludevamo che D non è regolare.

ESEMPIO 1.77

Valtola "cancelare" ("pumping down") è utile quando applichiamo il pumping lemma. Usiamo il pumping lemma per mostrare che $E = \{0^i1^j \mid i > j\}$ non è regolare. La prova è per contraddizione.

Assumiamo che E sia regolare. Sia p la lunghezza del pumping data dal pumping lemma. Sia $s = 0^p1^p$. Allora s può essere divisa in xyz , dove le tre parti soddisfano le condizioni del pumping lemma. Per la condizione 3, y consiste solo di simboli uguali a 0. Esaminiamo la stringa $xy^2 z$ per vedere se essa può essere in E . Aggiungere una copia supplementare di y aumenta il numero di simboli uguali a 0. Ma E contiene tutte le stringhe in 0^*1^* che hanno più simboli uguali a 0 che simboli uguali a 1, quindi aumentando il numero di simboli uguali a 0 otterremo ancora una parola in E . Non vi è contraddizione. Dobbiamo provare qualcosa altro.

Il pumping lemma afferma che $xy^i z \in E$ anche quando $i = 0$, quindi consideriamo la stringa $xy^0 z = xz$. Eliminare la stringa y da minimizzare il numero di simboli uguali a 0 in s . Ricorda che s ha solo uno 0 in più dei simboli uguali a 1. Pertanto, xz non può avere più simboli uguali a 0 di quelli uguali a 1, di conseguenza non può essere un elemento di E . Quindi otteniamo una contraddizione.

Figure 11: Esempio di applicazione del pumping lemma

2 Linguaggi context-free

In questo capitolo presentiamo le **grammatiche context-free**, un metodo più potente per descrivere linguaggi. Queste grammatiche possono descrivere alcuni aspetti che hanno una struttura ricorsiva. I linguaggi associati alle grammatiche context-free sono chiamati **linguaggi context-free**. Introduciamo anche gli **automi a pila** (pushdown automata), una classe di macchine che riconoscono i linguaggi context-free.

2.1 Grammatiche context-free

Un esempio di grammatica context-free, che chiamiamo G_1 , è il seguente

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

Una grammatica consiste di un insieme di **regole di sostituzione**, anche chiamate **produzioni**. Ogni regola appare come una linea nella grammatica, costituita da un simbolo e una stringa separati da una freccia. Il simbolo è chiamato **variabile**. La stringa consiste di variabili e altri simboli chiamati **terminali**. I terminali sono analoghi ai simboli dell'alfabeto di input e sono spesso reappresentati da lettere minuscole, numeri o simboli speciali. Una delle variabili è chiamata **variabile iniziale**. Essa generalmente si trova sul lato sinistro della regola più in alto. Per esempio, la grammatica G_1 ha tre regole. Le variabili di G_1 sono A e B , e A la variabile iniziale. I suoi terminali sono 0, 1 e #.

Definizione formale di grammatica context-free (CFG):

Definizione 2.2

Una grammatica context-free è una quadrupla $G = (V, \Sigma, R, S)$ dove:

1. V è un insieme finito di **variabili**,
2. Σ è un insieme finito di **terminali** disgiunto da V ,
3. R è un insieme finito di **regole**, o produzioni, ciascuna delle quali è della forma $A \rightarrow \alpha$, dove A è una variabile e α è una stringa di variabili e terminali (cioè $\alpha \in (V \cup \Sigma)^*$), e
4. S è la variabile iniziale.

Se u, v, w sono stringhe di variabili e terminali e $A \rightarrow w$ è una regola della grammatica, diciamo che uAv **produce** uwv , e lo denotiamo con $uAv \Rightarrow uwv$. Diciamo che u **deriva** v , e lo denotiamo con $u \Rightarrow^* v$, se $u = v$ o se esiste una sequenza u_1, u_2, \dots, u_k , con $k \geq 0$ e

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

Il **linguaggio della grammatica** è $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$, cioè l'insieme di tutte le stringhe che possono essere derivate dalla variabile iniziale.

Esempi di grammatiche context-free

ESEMPIO 2.3

Consideriamo la grammatica $G_3 = (\{S\}, \{a, b\}, R, S)$. L'insieme delle regole, R , è

$$S \rightarrow aSb \mid SS \mid \varepsilon.$$

Questa grammatica genera stringhe come **abab**, **aaabb**, e **aabbabb**. Si può più facilmente vedere qual è questo linguaggio pensando ad **a** come una parentesi aperta “(**“** e **b** come una parentesi chiusa “**)**”. Visto in questo modo, $L(G_3)$ è il linguaggio di tutte le stringhe di parentesi correttamente annidate. Si osservi che il lato destro di una regola può essere la parola vuota ε .

ESEMPIO 2.4

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

V è $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ e Σ è $\{a, +, \times, (,)\}$. Le regole sono

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

Le due stringhe **a+axa** e **(a+a)xa** possono essere generate dalla grammatica G_4 . Gli alberi sintattici sono mostrati nella figura seguente.

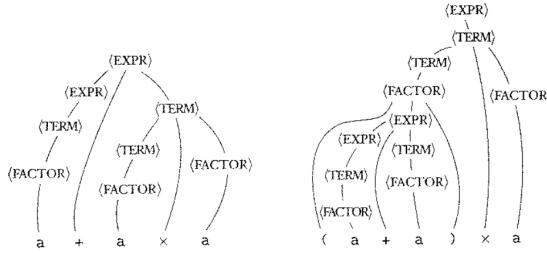


FIGURA 2.5
Alberi sintattici per le stringhe **a+axa** e **(a+a)xa**

Figure 12: Esempio di grammatica context-free

Progettare grammatiche context-free

Le tecniche seguenti sono utili, singolarmente o combinate, quando affrontiamo il problema di costruire una CFG. Innanzitutto, molti CFL (context-free language) sono l'unione di CFL più semplici. Se devi costruire una CFG per un CFL che puoi dividere in componenti più semplici, fallo e poi costruisci grammatiche separate per ciascuna componente. Queste singole grammatiche possono facilmente essere fuse in una grammatica per il linguaggio iniziale unendo le loro regole e poi aggiungendo la nuova regola $S \rightarrow S_1 \mid S_2 \mid \dots \mid S_k$, dove le variabili S_i sono le variabili iniziali per le grammatiche individuali. Per esempio, per ottenere una grammatica per il linguaggio $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$, costruiamo prima la grammatica

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

per il linguaggio $\{0^n 1^n \mid n \geq 0\}$ e poi la grammatica

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

per il linguaggio $\{1^n 0^n \mid n \geq 0\}$ e poi aggiungiamo la regola $S \rightarrow S_1 \mid S_2$ ottenendo la grammatica

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 1S_20 \mid \varepsilon. \end{aligned}$$

In secondo luogo, costruire una CFG per un linguaggio che sia regolare è facile se si può prima costruire un DFA per quel linguaggio. Si può trasformare un DFA in una CFG equivalente nel modo seguente. Si introduca una variabile R_i per ogni stato q_i del DFA. Si aggiunga la regola $R_I \rightarrow aR_j$ alla CFG se $\delta(q_i, a) = q_j$ è una transizione nel DFA. Si aggiunga la regola $R_i \rightarrow \varepsilon$ se q_i è uno stato accettante del DFA. Si assuma R_0 come variabile iniziale della grammatica, dove q_0 è lo stato iniziale della macchina. Si può verificare che la CFG risultante genera lo stesso linguaggio di quello riconosciuto dal DFA.

Ambiguità

Qualche volta una grammatica può generare la stessa stringa in più modi diversi. Una tale stringa avrà diversi alberi sintattici e quindi diversi significati. Se una grammatica genera la stessa stringa in più modi diversi, diciamo che la stringa è derivata **ambiguamente** in quella grammatica. Se una grammatica genera alcune stringhe ambiguumamente, diciamo che la grammatica è **ambigua**.

Per esempio, consideriamo la grammatica G_5 :

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

Questa grammatica genera la stringa $a+axa$ ambiguumamente. La figura seguente mostra i due diversi alberi sintattici.

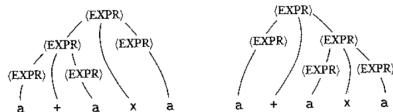


Figure 13: Esempio di grammatica context-free ambigua

Questa grammatica non tiene conto delle usuali regole di precedenza e quindi può raggruppare $+$ prima di \times o viceversa. Quindi la grammatica è ambigua. Quando diciamo che una grammatica genera ambiguumamente una stringa, intendiamo che la stringa ha due diversi alberi sintattici, non due differenti derivazioni. Due derivazioni possono differire sono nell'ordine in cui esse sostituiscono le variabili ma non nell'aloro struttura complessiva. Una derivazione di una stringa w in una grammatica G è una **derivazione a sinistra** se a ogni passo la variabile sostituita è quella che si trova più a sinistra.

Definizione 2.7

Una stringa w è derivata **ambiguamente** in una grammatica context-free G se essa ha due o più diverse derivazioni a sinistra. Una grammatica G è **ambigua** se essa genera qualche stringa ambiguumamente.

Forma normale di Chomsky

Quando si lavora con le grammatiche context-free, è spesso conveniente averle in forma semplificata. La forma di Chomsky è una delle più semplici e utili e viene utilizzata per dare algoritmi che lavorano con grammatiche context-free.

Forma Normale di Chomsky

Una grammatica context-free è in **forma normale di Chomsky** se ogni regola è della forma:

1. $A \rightarrow BC$, dove A, B, C sono variabili e né B né C sono la variabile iniziale, oppure
2. $A \rightarrow a$, dove A è una variabile e a è un terminale, oppure
3. $S \rightarrow \varepsilon$, dove S è la variabile iniziale.

Teorema 2.9(orale 24-30) (se non hai voglia di leggerti la dimostrazione, leggi le regole nell'esempio).

Teorema 2.9

Ogni linguaggio context-free è generato da una grammatica context-free in forma normale di Chomsky.

IDEA. Possiamo trasformare ogni grammatica G in forma normale di Chomsky. La trasformazione ha diversi passi nei quali le regole che violano le condizioni sono rimpiazzate con regole equivalenti. Innanzitutto, aggiungiamo una nuova variabile iniziale. Poi eliminiamo tutte le ε -regole della forma $A \rightarrow \varepsilon$. Eliminiamo anche tutte le **regole unitarie** della forma $A \rightarrow B$. In entrambi i casi, modifichiamo la grammatica in modo da essere sicuri che generi ancora lo stesso linguaggio. Infine, trasformiamo le restanti regole nella forma adeguata.

DIMOSTRAZIONE. In primo luogo, aggiungiamo una nuova variabile iniziale S_0 e la regola $S_0 \rightarrow S$, dove S era la variabile iniziale di partenza. Questo cambiamento garantisce che la variabile iniziale non compare sul lato destro di una regola. In secondo luogo, ci occupiamo di tutte le ε -regole. Eliminiamo una ε -regola, dove A non è la variabile iniziale. Poi, per ogni occorrenza di A sul lato destro di una regola, aggiungiamo una nuova regola con quell'occorrenza cancellata. In altre parole, se $R \rightarrow uAv$ è una regola in cui u e v sono stringhe di variabili e terminali, aggiungiamo la regola $R \rightarrow uv$. Facciamo così per ogni occorrenza di A , in modo che la regola $R \rightarrow uAvAw$ faccia sì che vengano aggiunte $R \rightarrow uvAw$, $R \rightarrow uAvw$, ed $R \rightarrow uvw$. Se abbiamo la regola $R \rightarrow A$, aggiungiamo $R \rightarrow \varepsilon$. Ripetiamo questi passi fino a eliminare tutte le ε -regole che non coinvolgono la variabile iniziale. Inoltre, ci occupiamo delle regole unitarie. Eliminiamo una regola unitaria $A \rightarrow B$. Poi, per ogni regola $B \rightarrow u$, aggiungiamo la regola $A \rightarrow u$ a meno che questa non sia una regola unitaria precedentemente cancellata. Come prima, u è una stringa di variabili e terminali. Ripetiamo questi passi fino a eliminare tutte le regole unitarie. Infine, trasformiamo tutte le restanti regole nella forma appropriata. Rimpiazziamo ogni regola $A \rightarrow u_1u_2\dots u_k$ dove $k \geq 3$ e ciascun u_i è una variabile o un simbolo terminale, con le regole $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, $A_2 \rightarrow u_3A_3$, ... and $A_{k-2} \rightarrow u_{k-1}U_k$. Le A_i sono nuove variabili. Rimpiazziamo ogni terminale u_i nelle precedenti regole con la nuova variabile U_i e aggiungiamo la regola $U_i \rightarrow u_i$.

ESEMPIO 2.10

Consideriamo la seguente CFG G_6 e trasformiamola in forma normale di Chomsky usando la procedura di conversione appena fornita. La sequenza di grammatiche presentate illustra i passi nella conversione. Le regole mostrate in grassetto sono quelle appena aggiunte. Le regole mostrate in grigio sono quelle appena rimosse.

- La CFG iniziale G_6 è mostrata sulla sinistra. Il risultato dell'applicazione del primo passo, che crea una nuova variabile iniziale, è mostrato sulla destra.

$$\begin{array}{ll} S \rightarrow ASA | aB & S_0 \rightarrow S \\ A \rightarrow B | S & S \rightarrow ASA | aB \\ B \rightarrow b | \epsilon & A \rightarrow B | S \\ & B \rightarrow b | \epsilon \end{array}$$

- Eliminiamo le ϵ -regole $B \rightarrow \epsilon$, come mostrato sulla sinistra, e $A \rightarrow \epsilon$, come mostrato sulla destra.

$$\begin{array}{ll} S_0 \rightarrow S & S_0 \rightarrow S \\ S \rightarrow ASA | aB | a & S \rightarrow ASA | aB | a | SA | AS | S \\ A \rightarrow B | S | \epsilon & A \rightarrow B | S | \epsilon \\ B \rightarrow b | \epsilon & B \rightarrow b \end{array}$$

- Eliminiamo le regole unitarie $S \rightarrow S$, come mostrato sulla sinistra, e $S_0 \rightarrow S$, come mostrato sulla destra.

$$\begin{array}{ll} S_0 \rightarrow S & S_0 \rightarrow S | ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS | S & S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow B | S & A \rightarrow B | S \\ B \rightarrow b & B \rightarrow b \end{array}$$

- Eliminiamo le regole unitarie $A \rightarrow B$ e $A \rightarrow S$.

$$\begin{array}{ll} S_0 \rightarrow ASA | aB | a | SA | AS & S_0 \rightarrow ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS & S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow B | S | b & A \rightarrow S | b | ASA | aB | a | SA | AS \\ B \rightarrow b & B \rightarrow b \end{array}$$

- Trasformiamo le rimanenti regole nella forma corretta, aggiungendo variabili e regole supplementari. La grammatica finale in forma normale di Chomsky è equivalente a G_6 . (In realtà la procedura fornita nel Teorema 2.9 crea diverse variabili U_i e diverse regole $U_i \rightarrow a$. Abbiamo semplificato la grammatica risultante usando una sola variabile U e una sola regola $U \rightarrow a$.)

$$\begin{array}{l} S_0 \rightarrow AA_1 | UB | a | SA | AS \\ S \rightarrow AA_1 | UB | a | SA | AS \\ A \rightarrow b | AA_1 | UB | a | SA | AS \\ A_1 \rightarrow SA \\ U \rightarrow a \\ B \rightarrow b \end{array}$$

Figure 14: Esempio regole Teorema 2.9

2.2 Automi a pila

In questa sezione introduciamo un nuovo tipo di modello computazionale chiamato **automa a pila** (pushdown automata). Questi automi sono come gli automi finiti non deterministici ma hanno una componente in più chiamata **pila** stack. La pila fornisce memoria aggiuntiva oltre alla quantità finita di essa disponibile nel controllo. La pila consente a tali automi di riconoscere alcuni linguaggi non regolari. La figura seguente è una rappresentazione schematica di un automa finito. Il controllo rappresenta gli stati e la funzione di transizione, il nastro contiene la stringa di input, e la freccia rappresenta la testina sull'input, che indica il successivo simbolo di input da leggere.

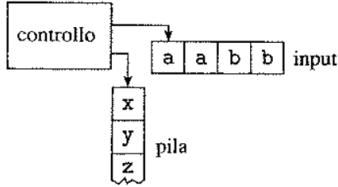


Figure 15: Rappresentazione schematica di un automa a pila

Un automa a pila (PDA) può scrivere simboli nella pila e rileggerli in seguito. In qualunque momento il simbolo sulla cima (top) della pila può essere letto e rimosso. L'operazione di scrivere un simbolo sulla pila è spesso chiamata **push**, quella di eliminare un simbolo dalla pila è spesso chiamata **pop**. In altre parole, una pila è in dispositivo di memoria "last in, first out".

Una pila è molto importante perché può mantenere una quantità non limitata di informazioni. Ricordiamo che un automa finito non può riconoscere il linguaggio $\{0^n 1^n \mid n \geq 0\}$ poiché non può memorizzare numeri molto grandi nella sua memoria finita. Un PDA è in grado di riconoscere questo linguaggio poiché può usare la sua pila per memorizzare il numero di simboli uguali a 0 che ha visto.

La seguente descrizione informale mostra come opera l'automa per questo linguaggio.

Legge i simboli di input. Scrive ciascuno 0 letto sulla pila. Non appena vede simboli uguali a 1, cancella uno 0 dalla pila per ogni 1 letto. Se la lettura dell'input termina esattamente quando la pila diventa priva di simboli uguali a 0, accetta l'input. Se la pila si svuota ma restano simboli uguali a 1 o se i simboli uguali a 1 sono finiti ma la pila contiene ancora simboli uguali a 0 o se qualche 0 appare nell'input dopo i simboli uguali a 1, rifiuta l'input.

Gli automi a pila possono essere non deterministici. Automi a pila deterministici e non deterministici non sono computazionalmente equivalenti. Gli automi a pila non deterministicamente riconoscono alcuni linguaggi che nessun automa a pila deterministico può riconoscere. Ricordiamo che gli automi finiti deterministicamente e non deterministicamente riconoscono la stessa classe di linguaggi, quindi la situazione per gli automi a pila è diversa. Ci concentreremo sugli automi a pila non deterministicamente perché questi automi sono computazionalmente equivalenti alle grammatiche context-free.

Definizione formale di automa a pila

La definizione di automa a pila è simile a quella di un automa finito, tranne che per la pila. La pila è un dispositivo che contiene simboli presi da un qualche alfabeto. La macchina può usare differenti alfabeti per il suo input e la sua pila, quindi ora specifichiamo sia un alfabeto di simboli di input Σ sia un alfabeto di pila Γ .

Nel cuore di ogni definizione di automa c'è la funzione di transizione, che descrive il suo comportamento. Ricordiamo che $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. Il dominio della funzione di transizione è $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Quindi lo stato corrente, il prossimo simbolo di input letto, e il simbolo sulla cima della pila determinano la mossa seguente di un automa a pila. L'uno o l'altro simbolo può essere ε , il che determina che la macchina si muova senza leggere un simbolo di input o senza leggere un simbolo dalla pila.

Per quanto riguarda il codominio della funzione di transizione, dobbiamo considerare cosa può fare l'automa quando è in una specifica situazione. Esso può trovarsi in un nuovo stato ed eventualmente scrivere un simbolo sulla cima della pila. La funzione δ può indicare quest'azione restituendo un elemento di Q insieme a un elemento di Γ_ε , cioè un elemento di $Q \times \Gamma_\varepsilon$. Poiché permettiamo il non determinismo in questo modello, una situazione può avere diverse mosse successive lecite. La funzione di transizione ingloba il non determinismo nel modo usuale, restituendo un insieme di elementi di $Q \times \Gamma_\varepsilon$, cioè un elemento di $P(Q \times \Gamma_\varepsilon)$. Mettendo tutto questo insieme, la nostra funzione di transizione δ assume la forma $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$.

Automi a Pila

Un automa a pila (PDA) è una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, dove Q, Σ, Γ, F sono tutti insiemi finiti, e

1. Q è l'insieme finito di stati,
2. Σ è l'alfabeto di input,
3. Γ è l'alfabeto della pila,
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale,
6. $F \subseteq Q$ è l'insieme degli stati accettanti.

Un automa a pila $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computa come segue. Accetta un input w se w può essere scritto come $w = w_1 w_2 \dots w_m$, dove ciascun $w_i \in \Sigma_\varepsilon$ ed esistono sequenze di stati $r_0, r_1, \dots, r_m \in Q$ e di stringhe $s_0, s_1, \dots, s_m \in \Gamma^*$ che soddisfano le tre condizioni seguenti. Le stringhe s_i rappresentano la sequenza del contenuto della pila che M ha su un ramo accettante della computazione.

1. $r_0 = q_0$ e $s_0 = \varepsilon$. Questa condizione significa che M inizia correttamente, nello stato iniziale e con una pila vuota.
2. Per $i = 0, \dots, m-1$, abbiamo $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$. Questa condizione afferma che M si muove correttamente in base allo stato, al simbolo di pila, e al prossimo simbolo in input.
3. $r_m \in F$. Questa condizione afferma che alla fine dell'input M si trova in uno stato accettante.

ESEMPIO 2.14

Quello che segue è la descrizione formale del PDA (pagina 116) che riconosce il linguaggio $\{0^n 1^n \mid n \geq 0\}$. Sia M_1 la sestupla $(Q, \Sigma, \Gamma, \delta, q_1, F)$, dove

$$Q = \{q_1, q_2, q_3, q_4\},$$

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

$$\Gamma = \{0, \$\},$$

$$F = \{q_1, q_4\}$$

δ è data dalla seguente tabella, nella quale le voci vuote indicano \emptyset .

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

Possiamo anche usare un diagramma di stato per descrivere un PDA, come nelle Figure 2.15, 2.17 e 2.19. Tali diagrammi sono simili ai diagrammi di stato usati per descrivere gli automi finiti, modificati per mostrare come il PDA usa la sua pila quando passa da uno stato a un altro. Scriviamo " $a, b \rightarrow c$ " per indicare che, quando la macchina sta leggendo una a dall'input, essa può sostituire il simbolo b sulla sommità della pila con una c . Ognuno dei simboli a, b e c può essere ε . Se a è ε , la macchina può effettuare questa transizione senza leggere alcun simbolo dall'input. Se b è ε , la macchina può realizzare questa transizione senza leggere ed eliminare alcun simbolo dalla pila. Se c è ε , la macchina non scrive alcun simbolo sulla pila durante questa transizione.

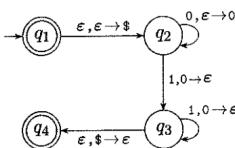


Figure 16: Esempio di automa a pila

La definizione di un PDA non contiene alcun meccanismo esplicito per permettere al PDA di controllare se la pila è vuota. Questo PDA è in grado di ottenere lo stesso effetto inserendo inizialmente un simbolo speciale $\$$ nella pila. Allora nel caso in cui veda il $\$$ di nuovo, sa che la pila è di fatto vuota. Questo PDA è in grado di ottenere lo stesso risultato poiché lo stato accettante potrebbe essere eventualmente raggiunto solo quando la macchina è alla fine dell'input. Quindi d'ora in poi assumiamo che i PDA possono controllare la fine dell'input e sappiamo che possiamo implementarlo nello stesso modo.

ESEMPIO 2.16

Questo esempio illustra un automa a pila che riconosce il linguaggio

$$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i = j \text{ oppure } i = k\}.$$

Informalmente, il PDA per questo linguaggio opera dapprima leggendo e inserendo le a nella pila. Quando le a sono terminate, la macchina le ha tutte nella pila, quindi può abbinarle con le b o con le c . Quest'operazione è un po' complicata poiché la macchina non sa in anticipo se far corrispondere le a con le b o con le c . In questo caso il non determinismo risulta utile.

Usando il suo non determinismo, il PDA può ipotizzare di far corrispondere le a con le b o con le c come mostrato in Figura 2.17. Si pensi alla macchina come se avesse due rami del suo non determinismo, uno per ogni possibile ipotesi. Se il numero di una delle due lettere b o c corrisponde al numero di a , quel ramo accetta e l'intera macchina accetta. Il Problema 2.27 chiede di mostrare che il non determinismo è necessario per riconoscere questo linguaggio con un PDA.

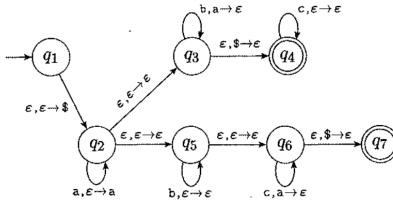


FIGURA 2.17

Diagramma di stato per il PDA M_2 che riconosce $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i = j \text{ oppure } i = k\}$

ESEMPIO 2.18

In questo esempio diamo un PDA M_3 che riconosce il linguaggio $\{ww^R \mid w \in \{0,1\}^*\}$. Ricordiamo che w^R significa w scritta al contrario. Nel seguito diamo la descrizione informale e il diagramma di stato del PDA.

Inizia inserendo nella pila i simboli letti. In qualunque momento ipotizza non deterministicamente di aver raggiunto la prima metà della stringa e

quindi cambia azione, eliminando dalla pila un simbolo per ciascun simbolo letto, se essi coincidono. Se essi coincidono sempre e la pila si svuota nello stesso momento in cui l'input è terminato, accetta; altrimenti rifiuta.

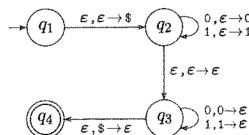


FIGURA 2.19

Diagramma di stato per il PDA M_3 che riconosce $\{ww^R \mid w \in \{0,1\}^*\}$

Figure 17: Esempio di automa a pila

Equivalenza con le grammatiche context-free

In questa sezione mostriamo che grammatiche context-free e automi a pila sono computazionalmente equivalenti. Entrambi sono in grado di descrivere la classe dei linguaggi context-free. Mostreremo come trasformare ogni grammatica context-free in automa a pila che riconosce lo stesso linguaggio e viceversa. Ricordiamo che abbiamo definito un linguaggio context-free come un linguaggio che può essere descritto con una grammatica context-free. Quindi il nostro obiettivo è il teorema seguente.

Teorema 2.20

Teorema 2.20

Un linguaggio è context-free se e solo se qualche automa a pila lo riconosce.

Lemma 2.21(orale)

Lemma 2.21 (orale)

Se un linguaggio è context-free, allora esiste un automa a pila che lo riconosce.

IDEA.

Sia A un CFL. Dalla definizione sappiamo che esiste una CFG, G , che genera A . Mostriamo come trasformare G in un PDA equivalente, che chiamiamo P . Il PDA P che ora descriviamo, opererà accettando il suo input w , se G genera tale input, determinando se esiste una derivazione per w . Ricordiamo che una derivazione è semplicemente la sequenza di sostituzioni fatte nel processo di generazione di una stringa mediante una grammatica. Ogni passo di derivazione produce una **stringa intermedia** di variabili e terminali. Noi progettiamo P in modo che possa stabilire se una serie di sostituzioni che usano le regole di G possa condurre dalla variabile iniziale a w .

Una delle difficoltà nell'esaminare se esiste una derivazione per w consiste nel capire quali sostituzioni fare. Il non determinismo di un PDA gli consente di ipotizzare la sequenza di sostituzioni giuste. In ogni passo della derivazione, una delle regole per una particolare variabile è scelta non deterministicamente e usata per sostituire quella variabile. Il PDA P inizia scrivendo la variabile iniziale sulla sua pila. Esso passa attraverso una serie di stringhe intermedie, facendo una sostituzione dietro l'altra. Infine può giungere a una stringa che contiene solo simboli terminali, il che significa che ha usato la grammatica per derivare una stringa. Allora P accetta se questa stringa è identica alla stringa che ha ricevuto in input.

Implementare questa strategia su un PDA richiede un'idea supplementare. Dobbiamo capire come il PDA immagazzina le stringhe intermedie quando passa dall'una all'altra. Usare semplicemente la pila per immagazzinare ciascuna stringa intermedia è allettante. Tuttavia non funziona del tutto, poiché il PDA ha bisogno di trovare le variabili nelle stringhe intermedie e fare sostituzioni. Il PDA può accedere solo al simbolo sulla cima della pila e questo potrebbe essere un simbolo terminale e non una variabile. Il modo per aggirare questo problema è mantenere solo parte della stringa intermedia sulla pila: i simboli che iniziano con la prima variabile nella stringa intermedia. Tutti i simboli terminali che compaiono prima della prima variabile sono subito abbinati con i simboli nella stringa di input.

La figura seguente mostra il PDA P .

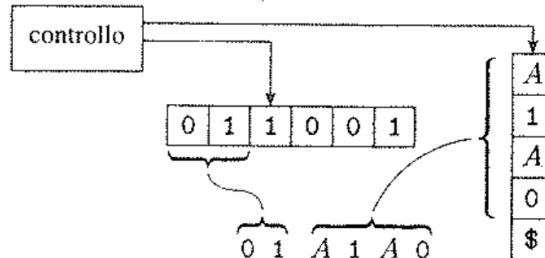


FIGURA 2.22
Come P rappresenta la stringa intermedia $01A1A0$

Quello che segue è una descrizione informale di P .

1. Inserisce il simbolo marcatore $\$$ e la variabile iniziale sulla pila.

2. Ripete i seguenti passi finché la pila non è vuota:

- Se sulla cima della pila c'è il simbolo di una variabile A , sceglie non deterministicamente una delle regole per A e sostituisce A con la stringa sul lato destro della regola.
- Se sulla cima della pila c'è un simbolo terminale a , legge il simbolo seguente dall'input e lo confronta con a . Se essi sono uguali, ripete. Se essi non sono uguali, rifiuta su questo ramo del non determinismo.
- Se sulla cima della pila c'è il simbolo $\$$, entra nello stato accettante. In questo modo accetta l'input se esso è stato completamente letto.

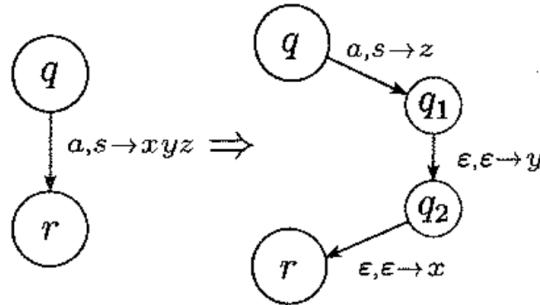
DIMOSTRAZIONE.

Ora diamo i dettagli formali della costruzione dell'automa a pila $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$. Per rendere più chiara la costruzione, usiamo una notazione abbreviata per la funzione di transizione. Questa notazione fornisce un modo per scrivere un'intera stringa sulla pila in un passo della macchina. Possiamo simulare quest'azione introducendo stati aggiuntivi per scrivere la stringa in simbolo alla volta, come realizzato nella seguente costruzione formale.

Siano q ed r stati del PDA e siano a in Σ_ε ed s in Γ_ε . Supponiamo di volere che il PDA vada da q a r quando legge a ed elimina s . Inoltre, vogliamo che esso inserisca l'intera stringa $u = u_1 \dots u_l$ sulla pila simultaneamente. Possiamo eseguire questa azione introducendo nuovi stati q_1, \dots, q_{l-1} e definendo la funzione di transizione come segue:

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

Useremo la notazione $(r, u) \in \delta(q, a, s)$ per denotare che quando q è lo stato in cui si trova l'automa, a è il prossimo simbolo di input ed s è il simbolo sulla cima della pila, il PDA può leggere a ed eliminare s , poi inserire la stringa u nella pila e passare nello stato r . La figura seguente ne mostra la realizzazione.



Gli stati di P sono $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$ dove E è l'insieme degli stati necessari per realizzare l'abbreviazione appena descritta. Lo stato iniziale è q_{start} . Lo stato accettante è q_{accept} .

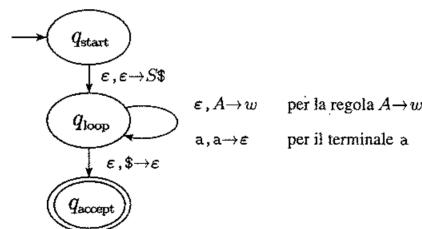
La funzione di transizione è definita come segue. Cominciamo inizializzando la pila inserendo i simboli $\$$ e S , realizzando così il passo 1 nella descrizione informale: $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, \$S)\}$. Poi aggiungiamo le transizioni per il ciclo principale del passo 2.

In primo luogo, trattiamo il caso (a) in cui la cima della pila contiene una variabile. Poniamo $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) \mid \text{dove } A \rightarrow w \text{ è una regola in } R\}$.

In secondo luogo, trattiamo il caso (b) in cui la cima della pila contiene un terminale. Poniamo $\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}$.

Infine, trattiamo il caso (c) in cui il marcitore scelto per indicare la pila vuota $\$$ è sulla cima della pila. Poniamo $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$.

Il diagramma di stato è mostrato nella figura seguente.



Questo completa la prova del Lemma 2.21.

ESEMPIO 2.25

Usiamo la procedura sviluppata nel Lemma 2.21 per costruire un PDA P_1 dalla seguente CFG G .

$$\begin{aligned} S &\rightarrow aTb \mid b \\ T &\rightarrow Ta \mid \epsilon \end{aligned}$$

La funzione di transizione è mostrata nel diagramma seguente.

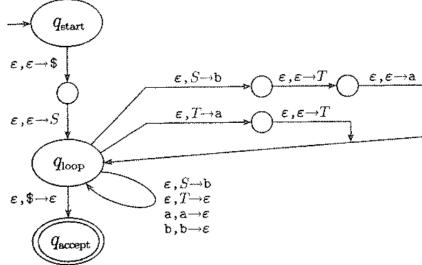


FIGURA 2.26
Diagramma di stato di P_1

Ora dimostriamo la direzione inversa del [Teorema 2.20](#). Vogliamo trasformare un PDA in una CFG. Progettiamo la grammatica per simulare l'automa. Questo compito è impegnativo perché "programmare" un automa è più facile che "programmare" una grammatica.

Lemma 2.27(oreale 24-30)
Lemma 2.27

Se un linguaggio è riconosciuto da un automa a pila, allora esso è context-free.

IDEA.

Abbiamo un PDA P e vogliamo costruire una CFG G che genera tutte le stringhe che P accetta. In altre parole, G dovrebbe generare una stringa se quella stringa fa andare il PDA dal suo stato iniziale a uno stato accettante.

Per raggiungere questo risultato, progettiamo una grammatica che fa un po' di più. Per ciascuna coppia di stati p e q in P , la grammatica avrà una variabile A_{pq} . Questa variabile genera tutte le stringhe che possano portare P da p con pila vuota a q con pila vuota. Si osservi che tali stringhe possono anche condurre P da p a q , indipendentemente dal contenuto della pila in p , lasciando la pila in q nella stessa condizione in cui era in p .

Innanzitutto, semplifichiamo il nostro compito modificando leggermente P per munirlo delle seguenti tre casistiche.

1. Ha un unico stato accettabile q_{accept} .
2. Svuota la sua pila prima di accettare.
3. Ciascuna transizione inserisce un simbolo sulla pila (effettua push) o ne elimina uno dalla pila (effettua pop), ma non fa entrambe le azioni contemporaneamente.

Dare a P le caratteristiche 1 e 2 è facile. Per munirlo della caratteristica 3, sostituiamo ciascuna transizione che contemporaneamente elimina ed inserisce simboli con una sequenza di due transizioni che attraversa un nuovo stato, e sostituiamo ogni transizione che non elimmina né inserisce simboli con una sequenza di due transizioni che inserisce e poi elimina un simbolo arbitrario della pila.

Per progettare G in modo che A_{pq} generi tutte le stringhe che portano P da p con pila vuota a q con pila vuota, dobbiamo capire come P agisce su queste stringhe. Per ognuna di tali stringhe x , la prima mossa di P su x deve essere un push, poiché ogni mossa è un push o un pop e P non può eliminare da una pila vuota. Analogamente, l'ultima mossa su x deve essere un pop perché alla fine la pila deve essere vuota.

Durante la computazione di P su x , si presentano due eventualità. O il simbolo eliminato alla fine è il simbolo che era stato inserito all'inizio, oppure no. Nel primo caso, la pila potrebbe essere vuota solo all'inizio e alla fine della computazione di P su x . Altrimenti, il simbolo inizialmente inserito deve essere stato eliminato in qualche punto prima della fine di x e quindi la pila si svuota in questo punto.

Simuliamo la prima possibilità con la regola $A_{pq} \rightarrow aA_{rs}b$ dove a è l'input letto nella prima mossa, b è l'input letto nell'ultima mossa, r è lo stato che segue p ed s è lo stato che precede q . Simuliamo la seconda possibilità con la regola $A_{pq} \rightarrow A_{pr}A_{rq}$, dove r è lo stato in cui la pila diventa vuota.

DIMOSTRAZIONE.

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept})$ e costruiamo G . L'insieme delle variabili di G è $\{A_{pq} \mid p, q \in Q\}$. La variabile iniziale è $A_{q_0, q_{accept}}$. Ora descriviamo le regole di G nei seguenti tre punti.

1. Per ogni $p, q, r, s \in Q$, $u \in \Gamma$ e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε) , pon la regola $A_{pq} \rightarrow aA_{rs}b$ in G .
2. Per ogni $p, q, r \in Q$, pon la regola $A_{pq} \rightarrow A_{pr}A_{rq}$ in G .
3. Infine, per ogni $p \in Q$, pon la regola $A_{pp} \rightarrow \varepsilon$ in G .

Si può intuire questa costruzione dalle figure seguenti.

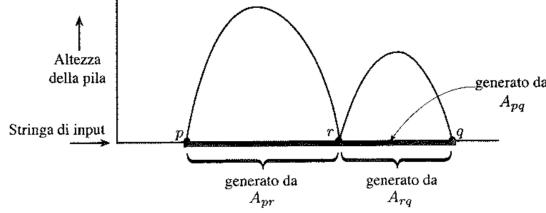


FIGURA 2.28
La computazione del PDA corrispondente alla regola $A_{pq} \rightarrow A_{pr}A_{rq}$

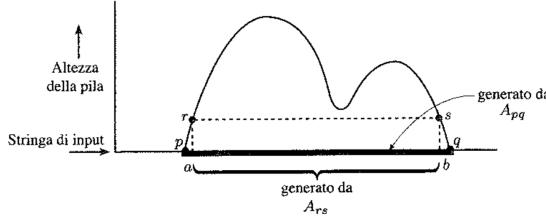


FIGURA 2.29
La computazione del PDA corrispondente alla regola $A_{pq} \rightarrow aA_{rs}b$

Figure 18: Esempio di grammatica context-free

Ora proviamo che questa costruzione funziona dimostrando che A_{pq} genera x se e solo se x porta P da p con la pila vuota a q con la pila vuota. Consideriamo ogni direzione del "se e solo se" come un enunciato separato.

Fatto 2.30

Se A_{pq} genera x , allora x può portare P da p con la pila vuota a q con la pila vuota.

Dimostriamo questo enunciato per induzione sul numero dei passi nella derivazione di x da A_{pq} .

Base. La derivazione è in un solo passo.

Una derivazione in un solo passo deve usare una regola il cui lato destro non contenga variabili. Le uniche regole in G con nessuna variabile sul lato destro sono $A_{pp} \rightarrow \varepsilon$. Ovviamamente, l'input ε porta P da p con la pila vuota a p con la pila vuota quindi la base è dimostrata.

Passo induttivo. Assumiamo che l'enunciato sia vero per le derivazioni di lunghezza al più k , dove $k \geq 1$, e proviamo che esso è vero per le derivazioni di lunghezza $k + 1$.

Supponiamo che $A_{pq} \Rightarrow^* x$ in $k + 1$ passi. Il primo passo in questa derivazione è $A_{pq} \Rightarrow aA_{rs}b$ oppure $A_{pq} \Rightarrow A_{pr}A_{rq}$. Trattiamo i due casi separatamente.

Nel primo caso, consideriamo la parte y di x che A_{rs} genera, quindi $x = ayb$. Poiché $A_{rs} \Rightarrow^* y$, in k passi, l'ipotesi induttiva ci dice che P può andare da r con la pila vuota a s con la pila vuota. Poiché $A_{pq} \rightarrow aA_{rs}b$ è una regola di G , $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε) , per qualche simbolo

di pila u . Quindi, se P inizia in p con la pila vuota, dopo aver letto a può andare nello stato r e inserire u in cima alla pila. Pertanto, x può portare P da p con la pila vuota a q con la pila vuota.

Nel secondo caso, consideriamo le parti y e z di x che A_{pr} e A_{rq} generano, rispettivamente, quindi $x = yz$. Poiché $A_{pr} \Rightarrow^* y$ e $A_{rq} \Rightarrow^* z$, in al più k passi, l'ipotesi induttiva ci dice che y può portare P da p a r e z può portare P da r a q , con la pila vuota all'inizio e alla fine della computazione. Quindi x può portare P da p con la pila vuota a q con la pila vuota. Questo completa la prova del passo induttivo.

Fatto 2.31

Se x può portare P da p con la pila vuota a q con la pila vuota, allora A_{pq} genera x .

Dimostriamo questo enunciato per induzione sul numero di passi che P impiega per andare da p con la pila vuota a q con la pila vuota sull'input x .

Base. La computazione è in 0 passi. Se una computazione è in 0 passi, essa inizia e termina nello stesso stato diciamo, p . Quindi dobbiamo mostrare che $A_{pp} \Rightarrow^* x$. P non può leggere alcun carattere in 0 passi, quindi $x = \varepsilon$. Per costruzione, G ha la regola $A_{pp} \rightarrow \varepsilon$, perciò la base è dimostrata.

Passo induttivo. Assumiamo l'enunciato vero per le computazioni di lunghezza al più k , dove $k \geq 0$, e dimostriamo che è vero per le computazioni di lunghezza $k + 1$. Supponiamo che P abbia una computazione dove x porta da p a q con pile vuote in $k + 1$ passi. O la pila è vuota solo all'inizio e alla fine di questa computazione oppure essa si svuota anche altrove. Nel primo caso, il simbolo che è stato inserito nella prima mossa deve essere lo stesso simbolo che è stato rimosso nell'ultima mossa. Chiamiamo u questo simbolo.

Sia a il simbolo di input letto nella prima mossa, b il simbolo di input letto nell'ultima mossa, r lo stato dopo la prima mossa ed s lo stato prima dell'ultima mossa. Allora $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε) e quindi la regola $A_{pq} \rightarrow aA_{rs}b$ è in G . Sia y la parte di x senza a e b , quindi $x = ayz$. L'input y può portare P da r a s senza toccare il simbolo u che è sulla cima della pila e quindi P può andare da r con una pila vuota a s con una pila vuota sull'input y . Abbiamo eliminato il primo e l'ultimo passo dei $k + 1$ passi nella computazione iniziale su x , perciò la computazione su y ha $(k + 1) - 2 = k - 1$ passi. Quindi l'ipotesi induttiva ci dice che $A_{rs} \Rightarrow^* y$. Allora $A_{pr} \Rightarrow^* x$.

Nel secondo caso, sia r uno stato in cui la pila si svuota oltre che all'inizio o alla fine della computazione su x . Allora le parti della computazione da p a r e da r a q contengono al più k passi. Sia y l'input letto nella prima parte e sia z l'input letto nella seconda parte. L'ipotesi induttiva ci dice che $A_{pr} \Rightarrow^* y$ e $A_{rq} \Rightarrow^* z$. Poiché la regola $A_{pq} \rightarrow A_{pr}A_{rq}$ è in G , $A_{pq} \Rightarrow^* x$ e la dimostrazione è completa.

Questo completa la prova del Lemma 2.27 e del Teorema 2.20.

Abbiamo appena dimostrato che gli automi a pila riconoscono la classe dei linguaggi context-free. Questa dimostrazione ci consente di stabilire una relazione tra i linguaggi regolari e i linguaggi context-free. Poiché ogni linguaggio regolare è riconosciuto da un automa finito e ogni automa finito è automaticamente un automa a pila che semplicemente ignora la sua pila, ora sappiamo che ogni linguaggio regolare è anche context-free.

Corollario 2.32

Ogni linguaggio regolare è anche context-free.

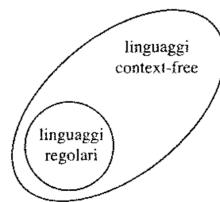


FIGURA 2.33
Relazione tra i linguaggi regolari e i linguaggi context-free

2.3 Linguaggi non context-free

In questa sezione presentiamo una tecnica per dimostrare che alcuni linguaggi non sono context-free. Presentiamo un pumping lemma per i linguaggi context-free. Esso stabilisce che per ogni linguaggio context-free esiste un particolare valore chiamato la **lunghezza del pumping** tale che tutte le stringhe più lunghe nel linguaggio possono essere "iterate". Questa volta il significato di iterazione è un po' più complesso. Significa che la stringa può essere divisa in cinque parti in modo tale che la seconda e la quarta parte possono insieme essere replicate un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

Il pumping lemma per i linguaggi context-free Teorema 2.34 orale(24-30)

Il pumping lemma per i linguaggi context-free

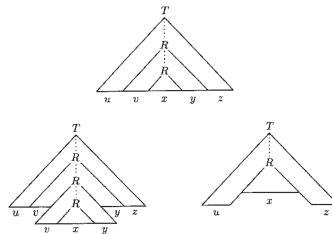
Se A è un linguaggio context-free, allora esiste un numero p (lunghezza del pumping) tale che, se s è una qualsiasi stringa in A di lunghezza almeno p , allora s può essere divisa in cinque parti $s = uvxyz$ che soddisfano le condizioni

1. per ogni $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$ e
3. $|vxy| \leq p$.

Quando s è divisa in $uvxyz$, la condizione 2 afferma che v oppure y non è la stringa vuota. Altrimenti il teorema sarebbe banalmente vero. La condizione 3 afferma che le parti v, x, y insieme hanno lunghezza al più p . Questa condizione tecnica a volte è utile per dimostrare che alcuni linguaggi non sono context-free.

IDEA. Sia A un CFL e sia G una CFG che lo genera. Dobbiamo mostrare che ogni stringa sufficientemente lunga s in A può essere iterata e restare in A . L'idea dietro questa strategia è semplice.

Sia s una stringa molto più lunga in A . (Chiariremo in seguito cosa intendiamo con "molto lunga"). Poichè s è in A , essa è derivabile da G e quindi ha un albero sintattico. L'albero sintattico per s deve essere molto alto perché s è molto lunga. Cioè l'albero sintattico deve contenere un cammino lungo dalla variabile alla radice dell'albero a uno dei simboli terminali su una foglia. Per il principio della piccionaia, qualche simbolo di variabile R si deve ripetere in questo cammino lungo. Come mostra la figura seguente, questa ripetizione ci permette di sostituire il sottoalbero sotto la seconda occorrenza di R con il sottoalbero sotto la prima occorrenza di R e ottenere ancora un albero sintattico consentito. Pertanto, possiamo dividere s in cinque parti $uvxyz$, come indicato nella figura, e possiamo replicare il secondo e quarto pezzo e ottenere una stringa ancora nel linguaggio. In altre parole, $uv^i xy^i z$ è ancora in A per ogni $i \geq 0$.



DIMOSTRAZIONE. Sia G una CFG per il CFL A . Sia b il massimo numero di simboli nel lato destro di una regola (assumiamo che sia almeno 2). Sappiamo che, in ogni albero sintattico costruito usando questa grammatica, un nodo non può avere più di b figli. Quindi, se l'altezza dell'albero sintattico è al più h , la lunghezza della stringa generata è al più b^h . Viceversa, se una stringa generata ha lunghezza maggiore o uguale a b^{h+1} , ciascuno dei suoi alberi sintattici deve avere un'altezza maggiore o uguale a $h + 1$.

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

Per vedere come iterare una tale stringa s , sia τ uno dei suoi alberi sintattici. Se s ha diversi alberi sintattici, scegliamo un albero sintattico τ che abbia il più piccolo numero di nodi. Sappiamo che τ deve avere altezza maggiore o uguale a $|V| + 1$, quindi il suo cammino più lungo dalla radice a una foglia ha lunghezza almeno $|V| + 1$. Questo cammino ha almeno $|V| + 2$ nodi; uno etichettato da un terminale, gli altri etichettati da variabili. Quindi questo cammino ha almeno $|V| + 1$ variabili. Poiché G ha solo $|V|$ variabili, qualche variabile R è presente più di una volta su questo cammino. Per un utilizzo successivo, scegliamo R in modo che sia una variabile che si ripete tra le $|V| + 1$ variabili più in basso su questo cammino.

Dividiamo s in $uvxyz$. Ogni occorrenza di R ha un sottoalbero sotto essa che genera una parte della stringa s . L'occorrenza più in alto di R ha un sottoalbero più grande e genera vxy , mentre l'occorrenza più in basso genera solo x con un sottoalbero più piccolo. Entrambi questi sottoalberi sono generati dalla stessa variabile, quindi possiamo sostituire l'uno con l'altro e ottenere ancora un albero sintattico corretto. Sostituire ripetutamente il più piccolo con il più grande fornisce gli alberi sintattici per le stringhe $uv^i xy^i z$ per ogni $i > 1$. Sostituire il più grande con il più piccolo genera la stringa uxz . Questo dimostra la condizione 1 del lemma.

Ora veniamo alle condizioni 2 e 3.

Per ottenere la condizione 2, dobbiamo essere sicuri che v e y non sono entrambe ε . Se lo fossero, l'albero sintattico ottenuto sostituendo il più piccolo sottoalbero al più grande avrebbe meno nodi di τ e genererebbe ancora s . Questo non è possibile perché abbiamo scelto τ in modo che sia un albero sintattico per s con il più piccolo numero di nodi. Questa è la ragione per aver selezionato così τ .

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

Scritto più precisamente:

PUMPING LEMMA:

$$A \text{ C.F.} \Rightarrow \exists p \in \mathbb{N} \forall s \in A (|s| \geq p \Rightarrow \exists u, v, x, y, z \ t.c(s = uvxyz \wedge |vy| > 0 \wedge |vxy| \leq p \wedge \forall i \in \mathbb{N}. uv^i xy^i z \in A))$$

CONTRAPPOSTA:

$$\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall u, v, x, y, z (s \neq uvxyz \vee |vy| = 0 \vee |vxy| \geq p \vee \exists i \in \mathbb{N}. uv^i xy^i z \notin A)) \Rightarrow A \text{ is not C.F.}$$

EQUIVALENTEMENTE:

$$\forall p \in \mathbb{N} \exists s \in A (|s| \geq p \wedge \forall u, v, x, y, z ((s = uvxyz \wedge |vy| > 0 \vee |vxy| \leq p) \Rightarrow \exists i \in \mathbb{N}. uv^i xy^i z \notin A)) \Rightarrow A \text{ is not C.F.}$$

ESEMPIO 2.36

Usiamo il pumping lemma per mostrare che il linguaggio $B = \{a^n b^n c^n | n \geq 0\}$ non è context-free.

Assumiamo che B sia un CFL e giungiamo a una contraddizione. Sia p la lunghezza del pumping per B la cui esistenza è garantita dal pumping lemma. Scelgiamo la stringa $s = a^p b^p c^p$. Ovviamente s è un elemento di B e di lunghezza almeno p . Il pumping lemma afferma che s può essere iterata, ma noi mostriamo che non può esserlo. In altre parole, mostriamo che, non importa come dividiamo s in $uvxyz$, una delle tre condizioni del lemma è violata.

In primo luogo, la condizione 2 stabilisce che v o y non è vuota. Allora consideriamo due casi, a seconda che le sottostringhe v e y contengano più di un tipo di simbolo dell'alfabeto o no.

1. Quando entrambi v e y contengono solo un tipo di simbolo dell'alfabeto, v non contiene entrambi i simboli a e b o entrambi i simboli b e c e lo stesso vale per y . In questo caso, la stringa uv^2xy^2z non può contenere lo stesso numero di a , b e c . Quindi, essa non può essere un elemento di B . Questo viola la condizione 1 del lemma e allora abbiamo una contraddizione.
2. Quando v o y contengono più di un tipo di simbolo, uv^2xy^2z può contenere un ugual numero dei tre simboli dell'alfabeto ma non nell'ordine corretto. Perciò essa non può essere un elemento di B e si verifica un assurdo.

Uno di questi casi deve verificarsi. Poiché entrambi i casi conducono a un assurdo, la contraddizione è inevitabile. Quindi proviamo l'assunzione che B sia un CFL deve essere falsa. Pertanto abbiamo provato che B non è un CFL.

ESEMPIO 2.37

Sia $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$. Usiamo il pumping lemma per mostrare che C non è un CFL. Questo linguaggio è simile al linguaggio B nell'Esempio 2.36, ma provare che esso non è context-free è un po' più complicato.

Assumiamo che C sia un CFL e otteniamo una contraddizione. Sia p la lunghezza del pumping fornita dal pumping lemma. Usiamo la stringa $s = a^p b^p c^p$ che abbiamo usato prima, ma questa volta dobbiamo eliminare ("pump down") oltre a iterare ("pump up"). Sia $s = uvxyz$ e consideriamo nuovamente i due casi presenti nell'Esempio 2.36.

1. Quando v e y contengono solo un tipo di simbolo dell'alfabeto, v non contiene entrambi i simboli a e b o entrambi i simboli b e c e lo stesso vale per y . Nota che il ragionamento usato in precedenza nel caso 1 non è più utilizzabile. Il motivo è che C contiene stringhe con numeri

cui s può essere divisa. Essa afferma che noi possiamo iterare s dividendo $s = uvxyz$, dove $|vxy| \leq p$.

Innanziutto, mostriamo che la sottostringa vzy deve stare a cavallo del punto centrale di s . Altrimenti, se la sottostringa è presente solo nella prima metà di s , la stringa uv^2xy^2z sposta un 1 nella prima posizione della seconda metà e quindi essa non può essere della forma ww . Analogamente, se vzy è presente nella seconda metà di s , la stringa uv^2xy^2z sposta uno 0 nell'ultima posizione della prima metà e quindi essa non può essere della forma ww .

Ma se la sottostringa vzy è a cavallo del punto centrale di s , la stringa uzz ha la forma $0^p 1^0 0^j 1^p$, dove i e j non possono essere entrambi p . Questa stringa non è della forma ww . Quindi s non può essere iterata e D non è un CFL.

diversi di a , b , e c se la sequenza di tali numeri è non decrescente. Dobbiamo analizzare la situazione con più attenzione per mostrare che s non può essere iterata. Osserviamo che, poiché v e y contengono solo un tipo di simbolo dell'alfabeto, uno dei simboli a , b o c non è presente in v o y . Suddividiamo ulteriormente questo caso in tre sottocasi, a seconda di quale simbolo non sia presente.

a. *Il simbolo a non è presente.* Allora proviamo a eliminare ("pumping down") ottenendo la stringa $uv^0xy^0z = uzz$. Questa stringa contiene lo stesso numero di a che ha s , ma contiene meno b o c e quindi essa non è un elemento di C e abbiamo una contraddizione.

b. *Il simbolo b non è presente.* Allora dei simboli uguali ad a o a^c devono apparire in v o y poiché esse non possono essere entrambe la stringa vuota. Se sono presenti delle a , la stringa uv^2xy^2z contiene più a che b , quindi essa non è in C . Se sono presenti delle c , la stringa uv^0xy^0z contiene più b che c , quindi essa non è in C . In ogni caso abbiamo una contraddizione.

c. *Il simbolo c non è presente.* Allora la stringa uv^2xy^2z contiene più a o più b che c , quindi essa non è in C , e abbiamo un assurdo.

2. Quando v o y contengono più di un tipo di simbolo, uv^2xy^2z non contrerà i simboli nell'ordine corretto. Perciò essa non può essere un elemento di C e giungiamo a una contraddizione.

Pertanto abbiamo mostrato che s non può essere iterata in contrasto con il pumping lemma e che C non è context-free.

ESEMPIO 2.38

Sia $D = \{ww | w \in \{0,1\}^*\}$. Usiamo il pumping lemma per mostrare che D non è un CFL. Assumiamo che D sia un CFL e otteniamo una contraddizione. Sia p la lunghezza del pumping fornita dal pumping lemma.

Questa volta scegliere la stringa s è meno ovvio. Una possibilità è la stringa $0^p 1^0 p^p$. È un elemento di D e ha lunghezza maggiore di p , quindi sembra essere un buon candidato. Ma questa stringa si può iterare, dividendola come segue, perciò non è adeguata per il nostro scopo.

$$\overbrace{000 \dots 000}^{\text{0}^p} \underbrace{0}_u \quad \underbrace{1}_v \quad \underbrace{0}_x \quad \underbrace{000 \dots 000}_{\text{0}^p} \underbrace{1^p}_z$$

Proviamo con un altro candidato per s . Intuitivamente, la stringa $0^p 1^0 p^p 1^p$ sembra catturare maggiormente l'"essenza" del linguaggio D di quanto facesse il precedente candidato. In effetti, possiamo mostrare che questa stringa funziona nel modo seguente.

Mostriamo che la stringa $s = 0^p 1^0 p^p 1^p$ non può essere iterata. Questa volta usiamo la condizione 3 del pumping lemma per limitare il modo in

3 La Tesi di Church-Turing

3.1 Macchine di Turing

Introduciamo ora un modello molto più potente, proposto inizialmente da Alan Turing nel 1936, chiamato **macchina di Turing**. Simile ad un automa finito, ma con una memoria illimitata e senza restrizioni, una macchina di Turing è un modello molto più preciso di un computer. Una macchina di Turing utilizza un nastro infinito come propria memoria illimitata. Ha una testina che è in grado di leggere e scrivere simboli ed è libera di muoversi lungo il nastro. Inizialmente il nastro contiene solo la stringa di input e tutto il resto è vuoto. Se la macchina deve memorizzare un'informazione, la può scrivere sul nastro. Gli output "accetta" e "rifiuta" sono ottenuti occupando appositi stati di accettazione e rifiuto. Se non raggiunge uno stato di accettazione o di rifiuto, la macchina andrà avanti per sempre, senza mai fermarsi.

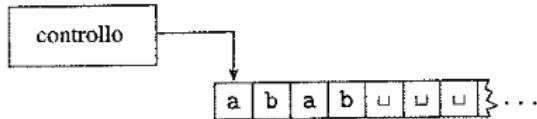


Figure 19: Schema di una macchina di Turing

L'elenco seguente riassume le differenze tra automi finiti e macchine di Turing.

1. Una macchina di Turing può sia scrivere che leggere sul nastro.
2. La testina di lettura-scrittura può muoversi sia verso sinistra che verso destra.
3. Il nastro è infinito.
4. Gli stati speciali di accettazione e rifiuto hanno effetto immediato.

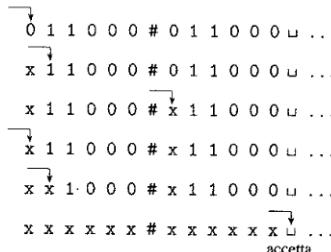
Introduciamo una macchina di Turing M_1 per testare l'appartenenza del linguaggio

$B = \{w\#w \mid w \in \{0,1\}^*\}$. Vogliamo che M_1 accetti se l'input è un elemento di B e che rifiuti altrimenti. Il vostro obiettivo è quello di determinare se l'input comprende due stringhe identiche separate da un simbolo $\#$. L'input è troppo lungo per voi per ricordarlo tutto, ma vi è permesso di muovervi avanti e indietro lungo l'input e porre dei contrassegni. La strategia ovvia è quella di muoversi a zig-zag in maniera simmetrica attorno al simbolo $\#$ e stabilire se gli elementi di ugual indice su i due lati corrispondono. Per tenere traccia dei simboli già controllati, M_1 barra ogni simbolo già esaminato. Se ha barrato tutti i simboli, significa che tutti i simboli corrispondono ed M_1 va in uno stato di accettazione. Se si accorge di una mancata corrispondenza, M_1 entra in uno stato di rifiuto. In pratica l'algoritmo di M_1 è il seguente:

M_1 = "Sulla stringa di input w :

1. Si muove a zig-zag lungo il nastro, raggiungendo posizioni corrispondenti su i due lati del simbolo $\#$, per controllare se queste posizioni contengono lo stesso simbolo. In caso negativo, o nel caso in cui non si trovi il simbolo $\#$, rifiuta. Barra gli elementi già controllati.
2. Quando tutti gli elementi a sinistra del simbolo $\#$ sono stati barrati, verifica la presenza di eventuali simboli rimanenti a destra di $\#$. Se rimane qualche elemento, allora rifiuta, altrimenti accetta.

La figura seguente contiene varie istantanee non consecutive del nastro di M_1 una volta avviata sull'input 011000#011000.



Definizione formale di macchina di Turing

Il cuore della definizione di una macchina di Turing è la funzione di transizione δ perchè essa ci dice come la macchina effettua un passo.

Per una macchina di Turing prende la forma $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. Cioè, quando la macchina occupa un certo stato q e la testina punta alla casella del nastro contenente un simbolo a , e se $\delta(q, a) = (r, b, L)$, la macchina scrive il simbolo b al posto di a , e passa nello stato r . La terza componente è L oppure R e indica se la testina si muove a sinistra o a destra dopo la scrittura.

Definizione 3.3

Una **macchina di Turing** è una 7-upla, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, dove Q, Σ, Γ sono insiemi finiti e

1. Q è l'insieme finito di stati,
2. Σ è l'alfabeto di input, non contiene il simbolo di **blank** \sqcup ,
3. Γ è l'alfabeto della nastro, contiene Σ e il simbolo di blank \sqcup ,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale,
6. $q_{accept} \in Q$ è lo stato di accettazione,
7. $q_{reject} \in Q$ è lo stato di rifiuto, $q_{reject} \neq q_{accept}$.

Una macchina di Turing computa nel seguente modo. Inizialmente riceve il suo input $w = w_1 w_2 \dots w_n \in \Sigma^*$ sulle n celle più a sinistra del nastro, mentre il resto del nastro è vuoto (simboli blank). La testina parte dalla posizione più a sinistra del nastro. Si noti che Σ non contiene il simbolo di blank \sqcup , in tal modo il primo simbolo blank che compare sul nastro segna la fine dell'input. Se M tenta di spostare la testina a sinistra quando si trova all'estremità sinistra del nastro, allora la testina rimane ferma. La computazione continua fino a quando la macchina raggiunge uno stato di accettazione o di rifiuto. Se nessuno dei due stati è raggiunto, la macchina continua a computare all'infinito. Durante la computazione di una macchina di Turing, si verificano cambiamenti dello stato corrente, del contenuto corrente del nastro, e della posizione corrente della testina. Un'impostazione di questi tre elementi è chiamata **configurazione** della macchina di Turing. Una configurazione è rappresentata come uqv , dove u è la stringa di simboli a sinistra della testina, q è lo stato corrente, e v è la stringa di simboli a destra della testina. La configurazione iniziale è $q_0 w$, dove w è l'input. Dopo l'ultimo simbolo di w , il nastro contiene solo simboli blank.

Per esempio, $1011q_701111$ indica che la testina è in stato q_7 , il nastro contiene 1011 a sinistra della testina, e 1111 a destra della testina.

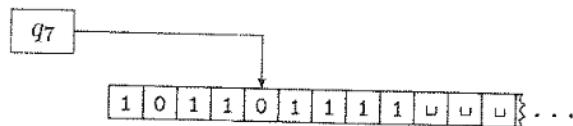


Figure 20: Esempio di macchina di Turing

Supponiamo di avere $a, b, c \in \Gamma$ così come u, v in Γ^* e gli stati q_i e q_j . In tal caso $uaq_i bv$ e $uq_j acv$ sono due configurazioni. diciamo che

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

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

Casi particolari si verificano quando la testina è alla fine del nastro. Per l'estremità sinistra, la configurazione $q_i bv$ produce $q_j cv$ se $\delta(q_i, b) = (q_j, c, L)$.

Una macchina di Turing M **accetta** l'input w se esiste una sequenza di configurazioni C_1, C_2, \dots, C_k tale che:

1. C_1 è la configurazione iniziale di M su w ,
2. per ogni i , M va dalla configurazione C_i alla configurazione C_{i+1} ,
3. C_k è una configurazione accettante.

L'insieme di stringhe che M accetta rappresenta il **linguaggio di M**, o il **linguaggio riconosciuto da M**, denotato con $L(M)$.

Definizione 3.5

Un linguaggio è **Turing-riconoscibile** se esiste una macchina di Turing che lo riconosce.

Quando attiviamo una macchina di Turing su un input, essa può accettare, rifiutare o andare avanti all'infinito. Una macchina di Turing può non accettare sia perché si ferma in uno stato di rifiuto, sia perché non si ferma affatto. A volte distinguere una macchina che è entrata in un ciclo infinito da una che sta ancora computando è difficile. Per questo motivo preferiamo macchine di Turing che si fermano su ogni input; tali macchine non ciclano mai. Una macchina è detta **decisore** perché prende in ogni caso una decisione. Diciamo che un decisore **decide** un certo linguaggio se riconosce tale linguaggio.

Definizione 3.6

Un linguaggio è **Turing-decidibile** se esiste una macchina di Turing che lo decide.

Esempi di macchine di Turing

Come abbiamo fatto nel caso degli automati finiti e a pila, possiamo descrivere formalmente una particolare macchina di Turing specificando ciascuna delle sue sette parti. Tuttavia, scendere a quel livello di dettaglio può essere scomodo, tranne che per macchine di Turing molto piccole. Di conseguenza, non spareremo molte risorse per dare tali descrizioni. Per lo più daremo solo descrizioni ad alto livello perché sono abbastanza precise per i nostri scopi e sono molto più semplici da comprendere. Tuttavia, è importante ricordare che ogni descrizione ad alto livello è solo un'abbreviazione della sua controparte formale. Con pazienza e attenzione potremo fornire una descrizione formale completa di ognuna delle macchine di Turing in questo libro. Per aiutarvi a stabilire il nesso tra le descrizioni formali e le descrizioni ad alto livello, nei prossimi due esempi mostreremo i diagrammi di stato.

ESEMPIO 3.7

In questo esempio descriviamo una macchina di Turing (TM) M_2 che decide $A = \{0^{2^n} \mid n \geq 0\}$, il linguaggio formato da tutte le stringhe di 0 la cui lunghezza è una potenza di 2.

M_2 = "su input w:

1. Muove la testina da sinistra a destra sul nastro, cancellando ogni secondo 0.
2. Se al passo 1, il nastro conteneva un solo 0, **accetta**.
3. Se al passo 1, il nastro conteneva più di un singolo 0 ed il loro numero era dispari, **rifiuta**.
4. Riporta la testina all'estremità sinistra del nastro.
5. Va al passo 1."

Ogni iterazione del passo 1 dimezza il numero di 0. Quando la macchina si muove lungo il nastro nel passo 1, ricorda se il numero di 0 è pari o dispari. Se tale numero è dispari e maggiore di 1, il numero di 0 iniziale in input non poteva essere una potenza di 2. Pertanto la macchina, su questa istanza, rifiuta. Tuttavia, se il numero di 0 è 1, il numero iniziale doveva essere una

potenza di 2. Quindi, in questo caso la macchina accetta. Ora diamo la descrizione formale di $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$,
- $\Gamma = \{\text{0}, \text{x}, \text{R}\}$.
- Descriviamo δ mediante un diagramma di stato (cfr. Figura 3.8).
- Gli stati iniziale, di accettazione e di rifiuto sono rispettivamente q_1 , q_{accept} e q_{reject} .

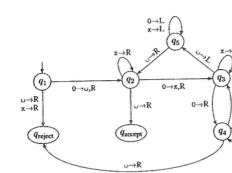


FIGURA 3.8
Diagramma di stato per la macchina di Turing M_2

In questo diagramma di stato, nella transizione da q_1 a q_2 compare l'etichetta $0 \rightarrow \text{R}$. Questa etichetta indica che quando siamo nello stato q_1 e la testina legge 0, la macchina si porta nello stato q_2 , scrive x e sposta la testina a destra. In altre parole, $\delta(q_1, 0) = (q_2, \text{x}, \text{R})$. Per chiarezza utilizziamo l'abbreviazione $0 \rightarrow \text{R}$ per indicare la transizione da q_1 a q_2 . Il simbolo x significa che la macchina non cancella nulla durante la lettura di 0 quando si trova nello stato q_2 ma non modifica il nastro, così $\delta(q_2, 0) = (q_2, \text{R}, \text{R})$.

Questa macchina inizia scrivendo un simbolo blank sullo 0 più a sinistra del nastro in modo da poter identificare l'estrema sinistra del nastro nel passo 4. Mentre normalmente utilizzeremmo un simbolo più indicativo, quale * , come delimitatore di inizio nastro qui utilizziamo un blank per rispettare l'alfabeto del nastro e quindi mantenere piccolo il diagramma di stato. L'esempio 3.11 fornisce un altro metodo per determinare la fine del nastro a sinistra. In seguito diamo un esempio di esecuzione di questa

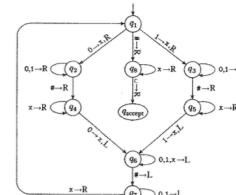
macchina su input 0000. La configurazione iniziale è q_10000 . La sequenza delle configurazioni della macchina è la seguente; leggete le colonne verso il basso e da sinistra a destra.

0.0000	00000	000000
q_10000	q_1x000	q_1xx00
	q_20000	q_2x00
		q_2xx0
	q_2000	q_2x0
		q_2xx
	q_200	q_2x
		q_2x0
	q_20	q_2x0
		q_2x00
	q_2000	q_2x000
		q_2x0000

ESEMPIO 3.9

Quella che segue è la descrizione formale di $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, la macchina di Turing che abbiamo descritto in maniera informale (cfr. pagina 175) per decidere il linguaggio $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \dots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0, 1, \#, \text{x}, \text{R}\}$ e $\Gamma = \{0, 1, \#, \text{x}, \text{R}\}$.
- Descriviamo δ mediante il diagramma di stato (cfr. la figura seguente).
- Gli stati iniziale, di accettazione e di rifiuto sono rispettivamente q_1 , q_{accept} e q_{reject} .



Nella Figura 3.10, che illustra il diagramma di stato della TM M_1 , compare l'etichetta $0,1\rightarrow R$ sulla transizione che va da q_3 a se stesso. Tale etichetta significa che la macchina rimane in q_3 ed effettua una mossa a destra quando legge uno 0 oppure un 1 nello stato q_3 . Non cambia il simbolo sul nastro.

La fase 1 è attuata mediante gli stati da q_1 a q_7 , e la fase 2 dagli stati rimanenti. Per semplificare la figura, non mostrano lo stato di rifiuto e le transizioni che vanno nello stato di rifiuto. Tali transizioni avvengono implicitamente ogni volta che uno stato è privo di una transizione di uscita per un determinato simbolo. Quindi, poiché nello stato q_5 non è presente nessuna freccia uscente con il simbolo $\#$, se un $\#$ è presente sotto la testina quando la macchina si trova nello stato q_5 , la macchina va nello stato di rifiuto. Per completezza, aggiungiamo che la testina si muove a destra in ciascuna di queste transizioni, nello stato di rifiuto.

ESEMPIO 3.11

In questo esempio, una TM M_3 esegue operazioni di aritmetica elementare. Essa decide il linguaggio $C = \{a^i b^j c^k \mid i \times j = k \text{ e } i, j, k \geq 1\}$.

M_3 = "Sulla stringa di input w:

1. Muove la testina da sinistra a destra sul nastro per determinare se è un elemento di $a^+ b^+ c^+$ e *rifiuta* se non lo è.
2. Riporta la testina all'estremità sinistra del nastro.
3. Barra una a e muove la testina a destra fino a trovare una b. Fa da spola tra le b e le c, barrando una di ognuna fino al termine delle b. Se tutte le c sono state barrate e rimane qualche b, *rifiuta*.
4. Ripristina le b barrate e ripete la fase 3 se rimane qualche a da barrare. Se tutte le a sono state barrate, controlla se anche tutte le c sono state barrate. Se sì, *accetta*; altrimenti, *rifiuta*."

Esaminiamo più da vicino le quattro fasi di M_3 . Nella fase 1, la macchina funziona come un automa finito. Nessuna scrittura è necessaria in quanto la testina si sposta da sinistra a destra, utilizzando gli stati al fine di determinare se l'input è nella forma corretta. La fase 2 sembra altrettanto semplice, ma contiene una sottigliezza. Come fa la TM a trovare l'estremità sinistra del nastro? Trovare l'estremità destra dell'input è facile perché esso termina con un simbolo blank. Ma, inizialmente l'estremità sinistra non ha alcun simbolo che permette di individuarla. Una tecnica che permette alla macchina di trovare l'estremità sinistra del nastro consiste nel marcire in qualche modo il simbolo più a sinistra quando la macchina parte con la testina su tale simbolo. A questo punto la macchina può eseguire una scansione a sinistra finché non trova il simbolo marcato, quando vuole riportare

segno più a destra, allora vuol dire che tutte le stringhe sono state confrontate, quindi *accetta*.
5. Va alla fase 3."

Questa macchina illustra la tecnica di marcatura dei simboli del nastro. Nella fase 2, la macchina pone un segno al di sopra di un simbolo, $\#$ in questo caso. Nell'effettiva implementazione, la macchina ha due simboli differenti, $\#$ e $\tilde{\#}$, nell'alfabeto del proprio nastro. Dire che la macchina mette un segno sopra un $\#$ significa che la macchina scrive il simbolo $\tilde{\#}$ in quella locazione. Rimuovere il segno vuol dire che la macchina scrive il simbolo $\#$ senza il puntino sopra. In generale, potremmo voler marcire vari simboli del nastro. Per fare ciò si può semplicemente includere le versioni col puntino di tutti questi simboli nell'alfabeto del nastro.

Concludiamo dagli esempi precedenti che i linguaggi descritti A, B, C ed E sono decidibili. Tutti i linguaggi decidibili sono Turing-riconoscibili, quindi questi linguaggi sono anche Turing-riconoscibili. Dimostrare che un linguaggio è Turing-riconoscibile, ma non decidibile è più difficile, lo faremo nel Capitolo 4.

la testina fino all'estremità sinistra del nastro. L'esempio 3.7 illustra questa tecnica; si utilizza un simbolo blank per marcare l'estremità sinistra del nastro.

Un metodo più sottile per trovare l'estremità sinistra del nastro sfrutta il modo in cui abbiamo definito il modello di macchina di Turing. Ricordiamo che, se la macchina tenta di muovere la testina oltre l'estremità sinistra del nastro, rimane ferma in quella stessa posizione. Possiamo usare questa caratteristica per creare un rilevatore dell'estremità sinistra. Per rilevare se la testina è situata sul lato estremo sinistro, la macchina può scrivere un simbolo speciale sulla posizione corrente, mentre memorizza nel controllo il simbolo che ha sostituito. Pù quindi tentare di spostare la testina a sinistra. Se è ancora sopra il simbolo speciale, il movimento verso sinistra non è avvenuto e quindi la testina deve essere per forza all'estremità sinistra. Se invece si ritrova su un simbolo diverso, alcuni simboli stanno a sinistra di quella posizione sul nastro. Prima di andare oltre, la macchina deve assicurarsi di risostituire il simbolo modificato con quello originale. Le fasi 3 e 4 hanno implementazioni semplici e utilizzano ciascuna stati diversi.

ESEMPIO 3.12

Qui, una TM M_4 sta risolvendo quello che viene chiamato il problema degli elementi distinti. Ha in input un elenco di stringhe su $\{0,1\}$ separate da simboli $\#$ ed il suo compito è accettare se tutte le stringhe sono diverse. Il linguaggio è

$$E = \{\#x_1\#x_2\#\dots\#x_l \mid \text{ogni } x_i \in \{0,1\}^* \text{ e } x_i \neq x_j \text{ per ogni } i \neq j\}.$$

La macchina M_4 funziona confrontando x_1 con x_2 mediante x_1 , poi confrontando x_2 con x_3 mediante x_1 , e così via. Le seguenti sono una descrizione informale della TM M_4 che decide questo linguaggio.

M_4 = "Sulla stringa di input w:

1. Mette un segno sul simbolo del nastro più a sinistra. Se questo simbolo era un blank, *accetta*. Se il simbolo era un $\#$, continua con la fase successiva. Altrimenti, *rifiuta*.
2. Scorre a destra fino al successivo $\#$ e vi mette sopra un secondo segno. Se nessun $\#$ viene trovato prima di un simbolo blank, allora era presente solo x_1 , quindi *accetta*.
3. Procede a zig-zag confrontando le due stringhe a destra dei simboli $\#$ segnati. Se sono uguali, *rifiuta*.
4. Sposta il punto a destra dei due segni sul successivo simbolo $\#$ a destra. Se non viene trovato nessun simbolo $\#$ prima di un simbolo blank, sposta il segno più a sinistra sul successivo $\#$ alla sua destra e sposta il segno più a destra sul successivo $\#$. Questa volta, se un simbolo $\#$ non è disponibile dopo il

3.2 Varianti delle Macchine di Turing

Macchine di Turing multinastro

Una macchina di Turing multinastro è una macchina di Turing con più nastri, ognuno con la propria testina. Ogni testina può muoversi indipendentemente dalle altre. Inizialmente l'input su trova sul nastro 1, mentre gli altri nastri sono vuoti. La funzione di transizione per una macchina di Turing multinastro è simile a quella di una macchina di Turing, ma con una piccola differenza. La funzione di transizione per una macchina di Turing multinastro è della forma

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

, dove k è il numero di nastri. L'espressione

$$\delta(q_i, a_1 a_2 \dots a_k) = (q_j, b_1 b_2 \dots b_k, L, R, \dots, L)$$

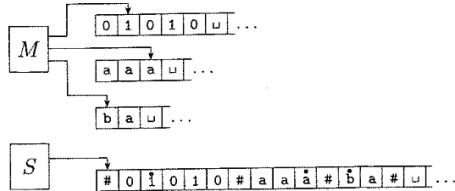
significa che, se la macchina si trova nello stato $q - i$ e le testine da 1 a k leggono i simboli da a_1 a a_k , allora la macchina va nello stato q_j , scrive i simboli da b_1 a b_k sui nastri, e muove le testine a sinistra o a destra o le lascia ferme. Le macchine di Turing multinastro sembrano più potenti delle macchine di Turing ordinarie, ma possiamo dimostrare che sono equivalenti in potenza di calcolo. Ricordiamo che due macchine sono equivalenti se riconoscono lo stesso linguaggio.

Teorema 3.13 orale(24-30)

Teorema 3.13 (orale 24-30)

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

DIMOSTRAZIONE. Mostriamo come convertire una macchina di Turing multinastro M in una TM S equivalente a nastro singolo. L'idea chiave è quella di mostrare come simulare M con S . Supponiamo che M abbia k nastri. Allora S simula l'effetto di k nastri memorizzando le loro informazioni sul singolo nastro. Essa utilizza il nuovo simbolo $\#$ per separare le informazioni dei nastri. Oltre al contenuto dei nastri, S deve tenere traccia delle posizioni delle testine. Lo fa scrivendo un simbolo con un punto sopra per contrassegnare la posizione in cui si troverebbe la testina di un nastro. Pensate a questi come nastri e testine "virtuali". Come in precedenza, i simboli del nastro "puntati" sono semplicemente simboli nuovi che sono stati aggiunti all'alfabeto del nastro.



$S =$ "Sulla stringa di input w :

1. Inizialmente S mette il suo nastro nel formato che rappresenta tutti i k nastri di M . Il nastro formattato contiene

$$\# \dot{w}_1 w_2 \dots w_n \# \downarrow \# \downarrow \# \dots \#.$$
2. Per simulare ogni singola mossa, S scansiona il suo nastro dal primo $\#$, che segna l'estremità sinistra, al $(k+1)$ mo $\#$, che segna l'estremità destra, per determinare i simboli puntati dalle testine virtuali. Successivamente S fa un secondo passaggio per aggiornare i nastri in accordo alla funzione di transizione M .
3. Se in qualsiasi momento S sposta una delle testine virtuali a destra su un $\#$, questa azione significa che M ha spostato la testina corrispondente sulla parte di nastro vuota non letta in precedenza. Quindi S scrive un simbolo blank in questa cella del nastro e sposta il contenuto del nastro, da questa cella fino al simbolo $\#$ più a destra, di una unità a destra. Poi prosegue la simulazione come prima.

Corollario 3.15

Corollario 3.15

Un linguaggio è Turing-riconoscibile se e solo se è riconosciuto da una macchina di Turing multinastro.

Macchine di Turing non deterministiche

Una macchina di Turing non deterministica è una macchina di Turing che può avere più di una mossa legata in una certa situazione. La funzione di transizione di una macchina di Turing non deterministica è della forma

$$\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\}).$$

La computazione di una macchina di Turing non deterministica è un albero i cui rami corrispondono alle diverse scelte possibili previste per la macchina. Se qualche ramo della computazione porta allo stato di accettazione, allora la macchina accetta l'input. Ora mostriamo che il non determinismo non influisce sulla potenza di calcolo del modello macchina di Turing.

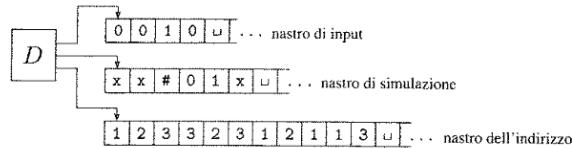
Teorema 3.16 orale(24-30)

Teorema 3.16 orale(24-30)

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

IDEA. Possiamo simulare qualsiasi TM non deterministica N con una TM deterministica D . L'idea di base consiste nel provare tutte le possibili scelte che può fare N durante la computazione non deterministica. Se D trova lo stato di accettazione su uno qualsiasi di questi rami, allora D accetta. Altrimenti la simulazione di D non terminerà. Guardiamo alla computazione di N su un input w come ad un albero. Ogni ramo dell'albero rappresenta una scelta non deterministica. Ogni nodo dell'albero è una configurazione di N . La radice dell'albero è la configurazione iniziale di N su w . La TM D esplora questo albero alla ricerca di una configurazione di accettazione. Eseguire accuratamente questa ricerca è cruciale perché D potrebbe non visitare l'intero albero. Un'idea è quella di esplorare l'albero con una DFS(depth-first search). Se D esplorasse in questo modo, essa potrebbe rimanere per sempre ad esplorare un eventuale cammino infinito e non trovare una configurazione accettante in qualche altro cammino. Quindi progettiamo D in modo da esplorare l'albero utilizzando una BFS(breadth-first search). Questa strategia esplora tutti i cammini che terminano alla stessa profondità prima di andare ad esplorare ogni cammino che termina alla profondità successiva. Questo metodo garantisce che D visita ogni node dell'albero finché non incontra una configurazione di accettazione.

DIMOSTRAZIONE. La TM D deterministica che simula ha tre nastri. Dal [Teorema 3.13](#), sappiamo che possiamo simulare una TM multinastro con una TM a nastro singolo. La macchina D utilizza i suoi tre nastri in maniera particolare, come illustrato nella figura seguente. Il nastro 1 contiene sempre la stringa di input e non viene mai modificato. Il nastro 2 mantiene una copia del nastro di N corrispondente a qualche diramazione della sua computazione non deterministica. Il nastro 3 tiene traccia della posizione di D nell'albero delle computazioni di N .



Consideriamo in primo luogo la rappresentazione dei dati sul nastro 3. Ogni nodo dell'albero può avere al massimo b figli, dove b è la dimensione del più grande insieme di scelte possibili date dalla funzione di transizione di N . Ad ogni nodo della struttura assegniamo un indirizzo che è una stringa sull'alfabeto $\Gamma_b = \{1, 2, \dots, b\}$. Assegniamo l'indirizzo 231 al nodo a cui si arriva partendo dalla radice, spostandosi al suo secondo figlio, spostandosi ancora da tale nodo al suo terzo figlio ed infine spostandosi al primo figlio di quest'ultimo nodo. Ogni simbolo della stringa ci dice quale deve essere la scelta successiva, durante la simulazione di un passo in una ramificazione di una computazione non deterministica di N . A volte il simbolo può non corrispondere ad una scelta se sono disponibili troppe poche scelte per configurazione. In tal caso l'indirizzo non è valido e non corrisponde ad alcun nodo. Il nastro 3 contiene una stringa su Γ_b . Essa rappresenta la ramificazione della computazione di N dalla radice al nodo indirizzato da tale stringa, a meno che l'indirizzo non sia valido. La stringa vuota è l'indirizzo della radice dell'albero. Siamo ora pronti a descrivere D .

1. Inizialmente il nastro 1 contiene l'input w e i nastri 2 e 3 sono vuoti.
2. Copia il nastro 1 sul nastro 2 ed inizializza la stringa sul nastro 3 a ϵ .
3. Utilizza il nastro 2 per simulare N con input w su una ramificazione della sua computazione non deterministica. Prima di ogni passo di N , consulta il simbolo successivo sul nastro 3 o se questa scelta deterministica non è valida, interromper questo cammino andando alla fase 4. Va alla fase 4 anche quando si verifica una configurazione di rifiuto. Se incontra una configurazione di accettazione, allora accetta l'input.
4. Sostituisce la stringa sul nastro 3 con la stringa successiva rispetto all'ordine sulle stringhe. Simula la ramificazione successiva della computazione di N andando al passo 2.

Corollario 3.18

Corollario 3.18

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

Possiamo modificare la dimostrazione del teorema precedente in modo che se N si ferma sempre su tutte le ramificazioni durante la computazione, D si ferma sempre. Chiameremo una macchina non deterministica **decisore** se tutte le ramificazioni si fermano su ogni input.

Corollario

Corollario

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

Enumeratori

Come accennato in precedenza, alcune persone usano il termine *linguaggio ricorsivamente enumerabile* per indicare un linguaggio Turing-riconoscibile. Questo termine deriva da una variante di macchina di Turing chiamata **enumeratore**. Definito in modo informale, un enumeratore è una macchina di Turing con una stampante collegata. Ogni volta che la macchina di Turing vuole aggiungere una stringa alla lista, invia la stringa alla stampante. La figura seguente fornisce uno schema di questo modello.

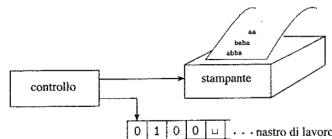


Figure 21: Schema di un enumeratore

Un enumeratore E inizia con un nastro di input vuoto. Se l'enumeratore non si ferma, esso può stampare un elenco infinito di stringhe. Il linguaggio enumerato da E è la collezione di tutte le stringhe che esso stampa. Inoltre, E potrebbe generare le stringhe del linguaggio in qualsiasi ordine, anche con ripetizioni.

Teorema 3.21 (orale)

Teorema 3.21 orale

Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo enumera.

DIMOSTRAZIONE. Come prima cosa, dimostriamo che se abbiamo un enumeratore E che enumera un linguaggio A , allora esiste una TM M che riconosce A . La TM M funziona come segue.

M = "Sulla stringa di input w :

1. Esegue E . Ogni volta che E genera una stringa, la confronta con w .
2. Se w appare nell'output di E , accetta".

Chiaramente, M accetta quelle stringhe che compaiono sulla lista di E' .

Ora occupiamoci dell'altra direzione. Se la TM M riconosce un linguaggio A , possiamo costruire il seguente enumeratore E per A . Sia s_1, s_2, s_3, \dots l'elenco di tutte possibili stringhe in Σ^* .

E = "Ignora l'input.

1. Ripete i seguenti passi per $i = 1, 2, 3, \dots$.
2. Esegue M per i passi su ogni input, s_1, s_2, \dots, s_i .
3. Se qualche computazione accetta, stampa la corrispondente s_j ."

Se M accetta una particolare stringa s , alla fine s apparirà nella lista generata da E . In realtà essa apparirà nella lista un numero infinito di volte, perché M ritorna all'inizio su ogni stringa per ogni ripetizione del passo 1. Questa procedura fa sì che M lavori in parallelo su tutte le possibili stringhe di input.

3.3 La definizione di algoritmo

Parlando in maniera informale, un algoritmo è un insieme di istruzioni semplici per l'esecuzione di un certo compito. Anche se gli algoritmi hanno avuto una lunga storia nel campo della matematica, la nozione di algoritmo non è stata formalizzata con precisione fino al ventesimo secolo. Prima di allora, i matematici avevano una nozione intuitiva di algoritmo, e invocavano questo concetto quando li usavano o li descrivevano.

La seguente storia racconta come una definizione precisa di algoritmo sia stata cruciale nel caso di un importante problema matematico.

I problemi di Hilbert

Nel 1900, il matematico David Hilbert presentò una lista di 23 problemi matematici che avrebbero dovuto guidare la ricerca matematica del ventesimo secolo. Il decimo problema sulla sua lista riguardava gli algoritmi.

Prima di descrivere il problema, introduciamo brevemente i polinomi. Un **polinomio** è una somma di termini, dove ogni **termine** è il prodotto di alcune variabili ad una costante, chiamata **coefficiente**. Per esempio

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

è un termine con coefficiente 6, e

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

è un polinomio con quattro termini sulle variabili x, y, z .

In questa discussione, consideriamo come coefficienti solo numeri interi. Una **radice** di un polinomio è un'assegnazione di valori alle sue variabili per cui il valore del polinomio è 0. Questo polinomio ha radice $x = 5, y = 3, z = 0$. Questa è una **radice intera**, perchè a tutte le variabili sono assegnati valori interi.

Il decimo problema di Hilbert consiste nell'ideare un algoritmo per verificare se un polinomio abbia o meno una radice intera. Hilbert non usò il termine "algoritmo", piuttosto "un processo in base al quale esso può essere determinato da un numero finito di operazioni". Nel modo in cui Hilbert ha formulato questo problema, ha chiesto esplicitamente che fosse "progettato" un algoritmo. In tal modo, egli diede per scontato l'esistenza di un tale algoritmo. Come ora sappiamo, non esiste un algoritmo per questo problema; si tratta di un problema non risolvibile algoritmicamente. Per poter provare che un algoritmo non esiste è necessario avere una definizione di algoritmo chiara. Per poter avere progressi sul decimo problema si è dovuto attendere una tale definizione. La definizione è arrivata nel 1936 da parte di Alonzo Church ed Alan Turing.

Church usò un sistema di notazione detto λ -calcolo per definire gli algoritmi. Turing lo fece con le sue "macchine". E' stato dimostrato che queste due definizioni sono equivalenti. Questa connessione tra la nozione informale di algoritmo e la relativa definizione precisa è ora chiamata: la **Tesi di Church-Turing**.

La Tesi di Church-Turing fornisce la definizione di algoritmo necessaria per risolvere il decimo problema di Hilbert. Nel 1970 Yuri Matijasevic, un matematico sovietico, dimostrò che il decimo problema di Hilbert è irrisolvibile.

Nozione intuitiva di algoritmo	equivalente a	Algoritmi mediante Macchina di Turing
-----------------------------------	---------------	--

Riscriviamo il decimo problema di Hilbert nella nostra terminologia. In tal modo introduciamo alcuni temi che analizzeremo nei prossimi capitoli. Sia

$$D = \{p \mid p \text{ è un polinomio con una radice intera}\}.$$

Il decimo problema di Hilbert chiede in sostanza se l'insieme D è decidibile. La risposta è negativa. Di contro possiamo dimostrare che D è Turing-riconoscibile. Prima di fare ciò, consideriamo un problema più semplice. Si tratta di un analogo del decimo problema di Hilbert per i polinomi che hanno un'unica variabile, come per esempio $4x^3 - 2x^2 + x - 7$. Sia

$$D_1 = \{p \mid p \text{ è un polinomio su } x \text{ avente una radice intera}\}.$$

Ecco una TM M_1 che riconosce D_1 :

M_1 = "Sulla stringa di input $\langle p \rangle$: dove p è un polinomio sulla variabile x .

1. Valuta p con x posta successivamente ai valori $0, 1, -1, 2, -2, 3, -3, \dots$. Se in qualsiasi momento la valutazione del polinomio è 0, accetta.”

Se p ha una radice intera, M_1 ad un certo punto la trova e accetta. Se p non ha una radice intera, M_1 sarà in esecuzione per sempre. Nel caso di più variabili, possiamo presentare una TM M simile che riconosce D . Ora M considera tutte le possibili impostazioni delle variabili con valori interi. Sia M_1 che M sono riconoscitori, ma non decisor. Possiamo convertire M_1 in un decisore per D_1 , perché possiamo calcolare dei limiti all'intervallo di valori in cui possono trovarsi le radici di un polinomio a singola variabile e limitare la ricerca a questo intervallo di valori.

Terminologia per la descrizione di macchine di Tuning

Continuiamo a parlare di macchine di Turing, ma il centro reale del nostro interesse saranno gli algoritmi. Cioè, le macchine di Turing servono soltanto come modello preciso per la definizione di algoritmo. Stabiliamo ora un formato e la notazione che utilizzeremo per descrivere le macchine di Turing. L'input di una macchina di Turing è sempre una stringa. Se volessimo fornire in input un oggetto diverso da una stringa, dovremmo prima rappresentare tale oggetto come una stringa. Le stringhe possono rappresentare facilmente polinomi, grafi, grammatiche, automi e qualsiasi combinazione di questi oggetti. Una macchina di Turing può essere programmata per decodificarno la rappresentazione in modo che possa essere interpretata nel modo corretto. La nostra notazione per la codifica di un oggetto O nella sua rappresentazione sotto forma di stringa è $\langle O \rangle$. Se abbiamo vari oggetti O_1, O_2, \dots, O_k , la loro codifica congiunta è $\langle O_1, O_2, \dots, O_k \rangle$.

Nel nostro formato, descriviamo gli algoritmi delle macchine di Turing con un segmento di testo tra parentesi. Dividiamo l'algoritmo in fasi, ciascuna fase di solito consiste di più passi di calcolo della macchina di Turing. Indichiamo la struttura a blocchi dell'algoritmo con un'indentazione ulteriore. La prima riga dell'algoritmo descrive l'input alla macchina. Se la descrizione dell'input è semplicemente w , l'input è considerato una stringa. Se la descrizione dell'input è la codifica di un oggetto del tipo $\langle A \rangle$, la macchina di Turing dapprima implicitamente controlla se l'input codifica correttamente un oggetto della forma desiderata, rifiutando in caso contrario.

ESEMPIO 3.23

Sia A il linguaggio costituito da tutte le stringhe che rappresentano grafi non orientati connessi. Ricordiamo che un grafo si dice **connesso** se ogni nodo può essere raggiunto da ogni altro nodo spostandosi lungo gli archi del grafo. Scriviamo

$$A = \{ \langle G \rangle \mid G \text{ è un grafo connesso non orientato} \}.$$

La seguente è una descrizione ad alto livello di una TM M che decide A .

- $M =$ “Su input $\langle G \rangle$, la codifica di un grafo G :
1. Seleziona il primo nodo di G e lo marca.
 2. Ripete la fase seguente fino a quando non vengono più marcati nuovi nodi:
 3. per ogni nodo in G , marcalo se esso è connesso con un arco ad un nodo già marcato.
 4. Esamina tutti i nodi di G per determinare se sono tutti marcati. Se lo sono, *accetta*, altrimenti, *rifiuta*.”

Come ulteriore esercizio, esaminiamo alcuni dettagli implementativi della macchina di Turing M . Di solito nel seguito non forniremo questo livello di dettaglio e neppure avrete bisogno, se non esplicitamente richiesto in un esercizio. In primo luogo, dobbiamo capire come $\langle G \rangle$ codifica il grafo G come una stringa. Consideriamo una codifica che consiste in una lista dei nodi di G seguita da un elenco degli archi di G . Ogni nodo è un numero decimale ed ogni arco è denotato dalla coppia di numeri decimali che rappresentano i nodi ai due estremi dell'arco. La figura seguente mostra un grafo e la sua codifica.

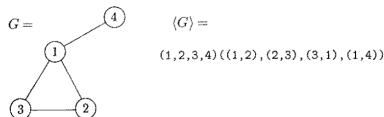


FIGURA 3.24
Un grafo G e la sua codifica $\langle G \rangle$

Quando M riceve l'input $\langle G \rangle$, verifica innanzitutto che l'input sia effettivamente la codifica di un grafo. Per fare ciò, M scandisce il nastro per assicurarsi che ci siano due liste e che siano nella forma corretta. Il primo elenco deve essere un elenco di numeri decimali distinti ed il secondo dovrebbe essere un elenco di coppie di numeri decimali. Successivamente, M controlla varie cose. In primo luogo, l'elenco dei nodi non deve contenere ripetizioni; in secondo luogo, ogni nodo che figura nell'elenco degli archi deve apparire anche nella lista dei nodi. Per il primo, si può utilizzare la procedura descritta nell'Esempio 3.12 per la TM M_4 che controlla la diversità degli elementi. Un metodo simile funziona per il secondo controllo. Se l'input supera questi controlli, allora esso è la codifica di qualche grafo G . Questa verifica completa il controllo dell'input, ed M va alla fase 1. Per la

fase 1, M segna il primo nodo con un punto sulla cifra più a sinistra. Per la fase 2, M scorre la lista dei nodi per trovare un nodo n_1 non puntato (cioè marcato con il punto) e lo contrassegna in modo diverso – diciamo, sottolineando il primo simbolo. Poi M scorre nuovamente la lista per trovare un altro nodo n_2 puntato e sottolineato anche questo. Ora M scorre la lista degli archi. Per ogni arco, M controlla se i due nodi sottolineati n_1 e n_2 sono quelli che compaiono in tale arco. Se lo sono, M punta n_1 , rimuove la sottolineatura e riprende dall'inizio della fase 2. Altrimenti, se non lo sono, M controlla l'arco successivo sulla lista. Se non ci sono più archi, $\{n_1, n_2\}$ non è un arco di G . Quindi M muove la sottolineatura da n_2 al successivo nodo puntato e chiama ora questo nuovo nodo n_2 . Ripete la procedura descritta in questo paragrafo per controllare, come prima, se la nuova coppia $\{n_1, n_2\}$ è un arco. Se non ci sono più nodi puntati, n_1 non è collegato a nessun nodo puntato. Quindi M impone la sottolineatura in modo che n_1 risulti il successivo nodo non puntato ed n_2 sia il primo nodo puntato e ripete la procedura descritta in questo paragrafo. Se non ci sono più nodi senza punto, M non è stata in grado di trovare nuovi nodi da marcare, quindi passa alla fase 4. Durante la fase 4, M scorre l'elenco dei nodi per determinare se sono tutti puntati. Se lo sono, entra nello stato di accettazione, altrimenti entra nello stato di rifiuto. Questo completa la descrizione della TM M .

4 Decidibilità

In questo capitolo iniziamo ad investigare la potenza degli algoritmi nella risoluzione di problemi. Dimostreremo che alcuni problemi possono essere risolti in maniera algoritmica ed altri no.

4.1 Linguaggi Decidibili

In questa sezione ci concentriamo su linguaggi che riguardano gli automi delle grammatiche. Ad esempio, presentiamo un algoritmo che verifica se una stringa è o meno un elemento di un linguaggio context-free(CFL). Questi linguaggi sono interessanti per vari motivi. Il primo è che alcuni problemi di questo tipo sono correlati ad applicazioni. Questo problema di verificare se una CFG genera una stringa è correlato al problema del riconoscimento e compilazione dei programmi in un linguaggio di programmazione. Il secondo motivo è che altri problemi relativi ad automi e grammatiche non sono decidibili mediante algoritmi.

4.1.1 Problemi decidibili relativi a linguaggi regolari

Iniziamo con alcuni problemi computazionali relativi ad automi finiti. Forniamo alcuni algoritmi per testare se un automa finito accetta o meno una stringa, se il linguaggio di un automa finito è vuoto e se due automi finiti sono equivalenti. Si noti che abbiamo scelto di rappresentare i problemi computazionali mediante linguaggi. Ad esempio, il **problema dell'accettazione** per DFA consiste nel testare se un particolare automa finito deterministico accetta una data stringa, può essere espresso come un linguaggio, A_{DFA} . Questo linguaggio contiene le codifiche di tutti i DFA con le stringhe che essi accettano. Definiamo

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

Il problema di verificare se un DFA B accetta l'input w coincide con il problema di verificare se $\langle B, w \rangle$ è un elemento del linguaggio A_{DFA} . Mostrare che il linguaggio è decidibile equivale a mostrare che il problema computazionale è decidibile. Nel seguente teorema si mostra che A_{DFA} è decidibile.

Teorema 4.1(oramai) A_{DFA} è un linguaggio decidibile.

IDEA.

Abbiamo semplicemente bisogno di presentare una TM M che decide A_{DFA} .
 $M = "Su input \langle B, w \rangle, dove B è un DFA e w è una stringa:$

1. Simula B su input w .
2. Se la simulazione termina in uno stato di accettazione, accetta. Se non termina in uno stato di accettazione rifiuta."

DIMOSTRAZIONE.

In primo luogo, esaminiamo l'input $\langle B, w \rangle$. Si tratta di una rappresentazione di un DFA B e di una stringa w . Una rappresentazione ragionevole di B è semplicemente una lista delle sue cinque componenti: $Q, \Sigma, \delta, q_0, F$. Quando M riceve il suo input, per prima cosa verifica se esso rappresenta correttamente un DFA B ed una stringa w . In caso contrario, M rifiuta. Poi M effettua direttamente la simulazione. Tiene traccia dello stato corrente di B e della posizione corrente di B nell'input w scrivendo queste informazioni sul suo nastro. Inizialmente lo stato corrente di B è q_0 e la posizione corrente di B nell'input w è la prima posizione più a sinistra di w . Gli stati e le posizioni vengono aggiornati in base alla funzione di transizione specificata δ . Quando M termina l'elaborazione dell'ultimo simbolo di w , M accetta l'input se B è in uno stato di accettazione; M rifiuta l'input se B non è in uno stato di accettazione.

Possiamo dimostrare un teorema simile per automi a stati finiti non deterministici. Dato

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

Mostriamo che A_{NFA} è decidibile.

Teorema 4.2(orale) A_{NFA} è un linguaggio decidibile.

DIMOSTRAZIONE.

Presentiamo una TM N che decide A_{NFA} . Potremmo progettare N in modo che operi come M , simulando un NFA invece di un DFA. Invece, procederemo in modo diverso per illustrare una nuova idea: N usa M come una sottoprocedura. Poichè M è progettata per funzionare con un DFA, N prima converte l'NFA che riceve come input in un DFA e poi lo passa ad M .

N = "Sulla stringa di input $\langle B, w \rangle$, dove B è un NFA e w è una stringa:

1. Converte l'NFA B in un DFA equivalente C , usando la procedura di conversione data nel [Teorema 1.39](#).
2. Esegue la TM M del [Teorema 4.1](#) su input $\langle C, w \rangle$.
3. Se M accetta, accetta; altrimenti rifiuta."

L'esecuzione della TM M nella fase 2 significa incorporare M nella progettazione di N come sottoprocedura.

In modo analogo, possiamo determinare se un'espressione regolare genera una data stringa. Consideriamo

$$A_{REX} = \{ \langle R, w \rangle \mid R \text{ è un'espressione regolare che genera la stringa di input } w \}.$$

Teorema 4.3(orale) A_{REX} è un linguaggio decidibile.

DIMOSTRAZIONE.

La seguente TM P decide A_{REX} .

P = "Sulla stringa di input $\langle R, w \rangle$, dove R è un'espressione regolare e w è una stringa:

1. Converte l'espressione regolare R in un NFA A equivalente, mediante la procedura di conversione data nel [Teorema 1.54](#).
2. Esegue la TM N su input $\langle A, w \rangle$.
3. Se N accetta, accetta; altrimenti rifiuta."

I Teoremi 4.1, 4.2 e 4.3 ci dicono che, ai fini della decidibilità, è equivalente presentare alla macchina di Turing un DFA, un NFA, oppure un'espressione regolare, perchè la macchina è in grado di convertire una codifica nell'altra.

Ora affrontiamo un diverso tipo di problema concernente gli automi a stati finiti: il **test del vuoto** per il linguaggio di un automa finito. Nei precedenti tre teoremi dovevamo decidere se un automa finito accettasse una particolare stringa. Nella prossima dimostrazione, dobbiamo determinare se un automa finito accetta una qualche stringa. Consideriamo

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

Teorema 4.4(orale) E_{DFA} è un linguaggio decidibile.

DIMOSTRAZIONE.

Il DFA accetta almeno una stringa se e solo se dallo stato iniziale può raggiungere uno stato di accettazione percorrendo il verso delle frecce del DFA. Per verificare questa condizione possiamo progettare un TM T che utilizza un algoritmo di marcatura analogo a quello utilizzato nell'[Esempio 3.23](#).

T = "Su input $\langle A \rangle$, dove A è un DFA:

1. Marca lo stato iniziale di A .
2. Ripete finchè nessuno stato nuovo viene marcato:
3. Marca qualsiasi stato che ha una transizione proveniente da uno stato già marcato.
4. Se nessuno stato di accettazione è stato marcato, accetta; altrimenti rifiuta."

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

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

Teorema 4.5(orale) EQ_{DFA} è un linguaggio decidibile.

DIMOSTRAZIONE.

Per dimostrare questo teorema usiamo il [Teorema 4.4](#). Costruiamo un nuovo DFA C a partire da A e B , dove C accetta solo quelle stringhe che sono accettate da A o da B , ma non da entrambi. In particolare, se A e B riconoscono lo stesso linguaggio, C non accetterà nulla. Il linguaggio di C è

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

Questa espressione è a volte denotata come **differenza simmetrica** di $L(A)$ e $L(B)$ ed è illustrata nella figura seguente. Qui $\overline{L(A)}$ denota il complemento di $L(A)$. La differenza simmetrica è utile perché $L(C) = \emptyset$ se e solo se $L(A) = L(B)$. Siamo in grado di costruire C da A e B con le costruzioni fatte per dimostrare che la classe dei linguaggi regolari risulta chiusa rispetto alle operazioni di complemento, unione e intersezione. Queste costruzioni sono algoritmi che nì possono essere eseguiti da macchine di Turing. Una volta che abbiamo costruito C possiamo usare il [Teorema 4.4](#) per determinare se $L(C) = \emptyset$ o meno. Se è vuoto, $L(A)$ e $L(B)$ devono essere uguali.

$F = "Su$ input $\langle A, B \rangle$, dove A e B sono DFA:

1. Costruisce il DFA C come descritto.
2. Esegue la TM T del [Teorema 4.4](#) su input $\langle C \rangle$.
3. Se T accetta, accetta; altrimenti rifiuta."

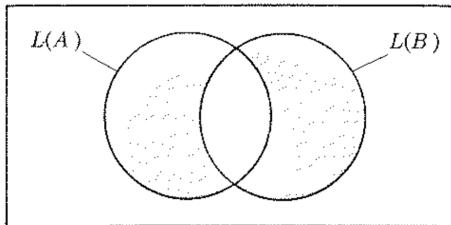


Figure 22: Differenza simmetrica di due linguaggi

4.1.2 Problemi decidibili relativi a linguaggi context-free

Descriviamo ora algoritmi per determinare se una CFG genera una particolare stringa e per determinare se il linguaggio di una CFG è vuoto.

Consideriamo

$$A_{CFG} = \{\langle G, w \rangle \mid G \text{ è una CFG che genera la stringa di input } w\}.$$

Teorema 4.7(orale) A_{CFG} è un linguaggio decidibile.

IDEA.

Per la CFG G e la stringa w vogliamo determinare se G genera w . Un'idea è quella di utilizzare G per passare attraverso tutte le derivazioni per determinare se ne esiste una di w . Quest'idea non funziona, poichè bisognerebbe provare infinite derivazioni. Se GG non genera w , questo algoritmo potrebbe non fermarsi. Tale idea dà una macchina di Turing che è un riconoscitore, non un decisore, per A_{CFG} . Per rendere questa macchina di Turing un decisore occorre assicurare che l'algoritmo provi solo un numero finito di derivazioni. Precedentemente abbiamo dimostrato che, se G fosse in forma normale di Chomsky, qualsiasi derivazione di w avrebbe $2n - 1$ passi, dove n è la lunghezza di w . In tal caso sarebbe sufficiente controllare solo derivazioni di $2n - 1$ passi per determinare se G genera w . Tali derivazioni sono in numero finito. Possiamo convertire G in forma normale di Chomsky utilizzando la procedura data nel [Teorema](#).

DIMOSTRAZIONE.

Diamo la TM S per A_{CFG}
 $S =$ "Su input $\langle G, w \rangle$, dove G è una CFG e w è una stringa:

1. Converti la CFG G in forma normale di Chomsky.
2. Lista tutte le derivazioni di $2n - 1$ passi, dove n è la lunghezza di w ; tranne se $n = 0$, in tal caso lista tutte le derivazioni di un passo.
3. Se una di tali derivazioni genera w , accetta; altrimenti rifiuta."

Il problema di determinare se una CFG genera una particolare stringa è correlato al problema della compilazione dei linguaggi di programmazione. Ricordiamo che abbiamo dato procedure per la conversione in entrambi i sensi tra CFG e PDA nel [Teorema 2.20](#). Quindi tutto quello che diciamo circa la decidibilità dei problemi concernenti le CFG vale anche per i PDA. Torniamo al problema dei test del vuoto per il linguaggio di una CFG. Come abbiamo fatto per i DFA, possiamo dimostrare che il problema di determinare se una CFG genera almeno una strinfa è decidibile. Sia

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

Teorema 4.8(orale) E_{CFG} è un linguaggio decidibile.

IDEA.

Per progettare un algoritmo per questo problema, potremmo tentare di utilizzare la TM S del [Teorema 4.7](#). Tale teorema afferma che siamo in grado di verificare se una CFG genera una particolare stringa w . Per determinare se $L(G) = \emptyset$, l'algoritmo potrebbe tentare di passare attraverso tutte le possibili w , una per una. Però esistono infinite w da testare, quindi questo metodo potrebbe far sì che non termini mai. Abbiamo bisogno di trovare un approccio differente. Per determinare se il linguaggio di una grammatica è vuoto, abbiamo bisogno di testare se la variabile iniziale può generare una stringa di terminali. L'algoritmo fa questo risolvendo un problema più generale. Determina *per ogni variabile* se essa è in grado di generare una stringa di terminali. Quando l'algoritmo ha determinato che una variabile può generare qualche stringa di terminali, l'algoritmo tiene traccia di queste informazioni marcando tale variabile. Dapprima, l'algoritmo marca tutti i simboli terminali della grammatica. Poi, scandisce tutte le regole della grammatica. Se trova una regola che consente a qualche variabile di essere sostituita da una stringa di simboli, che sono già tutti marcati, l'algoritmo sa che anche questa variabile può essere marcata. L'algoritmo continua in questo modo finché non può più marcare ulteriori variabili. La TM R implementa questo algoritmo.

DIMOSTRAZIONE.

$R =$ "Su input $\langle G \rangle$, dove G è una CFG:

1. Marca tutti i simboli terminali di G .
2. Ripeti finchè nessuna nuova variabile viene marcata:
3. Marca una qualsiasi variabile A tale che G ha una regola $A \rightarrow U_1U_2\dots U_k$, dove ogni U_i è già stato marcato.
4. Se la variabile iniziale non è segnata, accetta; altrimenti rifiuta."

Consideriamo ora il problema di determinare se due grammatiche context-free generano lo stesso linguaggio. Sia

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

Il [Teorema 4.5](#) fornisce un algoritmo che decide l'analogo linguaggio EQ_{DFA} per i DFA. Abbiamo utilizzato la procedura di decisione per E_{DFA} per dimostrare che EQ_{DFA} è decidibile. Ma qualcosa non va con questa idea! La classe dei linguaggi context-free non è chiusa rispetto al complemento o intersezione. Infatti, EQ_{CFG} non è decidibile. Ora mostriamo che i linguaggi context-free possono essere decisi dalle macchine di Turing.

Teorema 4.9 Ogni linguaggio context-free è decidibile.

IDEA.

Sia A un CFL. Il nostro obiettivo è mostrare che A è decidibile. Una (cattiva) idea è quella di convertire un PDA per A direttamente in una TM. Ciò non è difficile da realizzare, perché simulare una pila con il nastro di una TM è semplice. Il PDA per A può essere non deterministico, ma ciò sembra andare bene, perché siamo in grado di convertirlo in una TM non deterministica e sappiamo che qualsiasi TM non deterministica può essere convertita in una TM deterministica equivalente. C'è tuttavia una difficoltà. Alcuni rami della computazione del PDA possono andare avanti per sempre, leggendo e scrivendo la pila senza mai arrestarsi. La TM che effettua la simulazione avrebbe quindi alcuni cammini che non terminano mai, quindi la TM non sarebbe un decisore.

E' necessaria una diversa soluzione. Dimostriamo il teorema con la TM S che abbiamo progettato nel [Teorema 4.7](#) per decidere A_{CFG} .

DIMOSTRAZIONE.

Sia G una CFG per A , progettiamo una TM M_G che decide A . Costruiamo una copia di G in M_G . Essa funziona come segue.

M_G = "Su input w :

1. Esegue la TM S del [Teorema 4.7](#) su input $\langle G, w \rangle$.
2. Se S accetta, accetta; altrimenti rifiuta."

Il Teorema 4.9 fornisce il collegamento finale nelle relazioni tra le quattro principali classi di linguaggi che abbiamo descritto fin'ora: regolari, context-free, decidibili e Turing-riconoscibili.

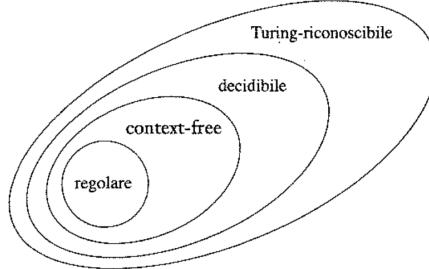


Figure 23: Relazioni tra le classi di linguaggi

4.2 Indecidibilità