

Automi

Appunti di Automi

Alessandro Dori

Università degli Studi di Roma "La Sapienza"
Facoltà di Informatica

12 dicembre 2024

ATTENZIONE!

Tutti i teoremi/esempi/dimostrazioni sono richiesti all'orale
per un voto 18-24, ad eccezione di quelli segnalati con (24-
30) o (30L). Quindi pure se non c'è la parentesi (orale) è
comunque richiesto all'orale. STUDIATE!

Indice

1 Linguaggi regolari	4
1.1 Automi finiti	4
1.1.1 Definizione formale di computazione	4
1.1.2 Le operazioni regolari	5
1.1.3 Definizione 1.23	5
1.1.4 Teorema 1.25	5
1.1.5 Teorema 1.26	5
1.2 Non determinismo	6
1.2.1 Definizione formale di automa finito non deterministico	6
Definizione 1.37	7
1.2.2 Equivalenza tra gli NFA e i DFA	7
Teorema 1.39(orale)	7
Corollario 1.40	8
Chiusura rispetto alle operazioni regolari	9
Teorema 1.45(orale)	9
Teorema 1.47(orale)	10
Teorema 1.49(orale)	11
1.3 Espressioni regolari	11
1.3.1 Definizione formale di espressioni regolari	12
1.3.2 Equivalenza con gli automi finiti	13
Teorema 1.54	14
Lemma 1.55(orale)	14
Lemma 1.60(orale 24-30)	15
Definizione 1.64(orale 24-30)	17
1.3.3 Linguaggi non regolari	19
Il pumping lemma	20
Teorema 1.70(orale)	20
2 Linguaggi context-free	23
2.1 Grammatiche context-free	23
Definizione formale di grammatica context-free (CFG):	23
Forma normale di Chomsky	25
Teorema 2.9(orale 24-30)	25
2.2 Automi a pila	26
Definizione formale di automa a pila	27
Equivalenza con le grammatiche context-free	29
Teorema 2.20	30
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)	42
Corollario 3.15	42
Macchine di Turing non deterministiche	42
Teorema 3.16 orale(24-30)	43
Corollario 3.18	44
Corollario	44
Enumeratori	44
Teorema 3.21 (orale)	45
3.3 La definizione di algoritmo	45
I problemi di Hilbert	45

Terminologia per la descrizione di macchine di Tuning	46
4 Decidibilità	48
4.1 Linguaggi Decidibili	48
4.1.1 Problemi decidibili relativi a linguaggi regolari	48
Teorema 4.1(orale)	48
Teorema 4.2(orale)	49
Teorema 4.3(orale)	49
Teorema 4.4(orale)	49
Teorema 4.5(orale)	50
4.1.2 Problemi decidibili relativi a linguaggi context-free	50
Teorema 4.7(orale)	50
Teorema 4.8(orale)	51
Teorema 4.9	52
4.2 Indecidibilità	52
Teorema 4.11(orale 24-30)	53
4.2.1 Il metodo della diagonalizzazione	53
Teorema 4.17	54
Corollario 4.18(orale 24-30)	55
4.2.2 Un linguaggio indecidibile	55
4.2.3 Un linguaggio non Turing-riconoscibile	57
Teorema 4.22(orale)	57
5 Riducibilità	58
5.1 Problemi indecidibili dalla teoria dei linguaggi	58
Teorema 5.1(orale)	58
Teorema 5.2(orale)	59
Teorema 5.3	60
Teorema 5.4(orale)	61
5.2 Riducibilità mediante funzione	61
5.2.1 Funzioni calcolabili	62
Teorema 5.22	63
Teorema 5.28	64
Teorema 5.30	64
6	65
7 Complessità di tempo	65
7.1 Misure di complessità	65
7.1.1 Notazione O-grande ed o-piccola	66
7.1.2 Analisi degli algoritmi	67
7.1.3 Relazioni di complessità tra modelli	69
Teorema 7.8 (orale 24-30)	70
7.1.4 Definizione 7.9	71
7.1.5 Teorema 7.11 (orale 24-30)	71
7.2 La classe P	71
7.2.1 Tempo polinomiale	72
7.2.2 Definizione 7.12	72
7.2.3 Esempi di problemi in P	73
Teorema 7.14	73
Teorema 7.15	74
Teorema 7.16	75
7.3 La classe NP	76
Definizione 7.18	77
Definizione 7.19	77
Teorema 7.20(orale)	78
Definizione 7.21	79
Corollario 7.22	79
7.3.1 Esempi di problemi in NP	79

Teorema 7.24	79
Teorema 7.25	80
7.4 NP-COMPLETEZZA	81
Teorema 7.27	82
7.4.1 Riducibilità in tempo polinomiale	82
Definizione 7.28	82
Definizione 7.29	82
Teorema 7.31 (orale)	83
Teorema 7.32 (orale)	83
7.4.2 Definizione di NP-completezza	84
Definizione 7.34	84
Teorema 7.35	84
Teorema 7.36	85
7.4.3 Il teorema di Cook e Levin (e mo so cazzo)	85
Corollario 7.42	89
7.5 Ulteriori problemi NP-completi	90
Teorema 34.10(24-30)	90
7.5.1 CLIQUE	91
Teorema 34.11(24-30)	92
7.5.2 VERTEX-COVER	93
Teorema 34.12(orale)	94
7.6 Approssimazione	95
7.6.1 Vertex Cover	96
Teorema 35.1(orale)	98
8 Complessità di Spazio	98
8.1 Teorema di Savitch (e mo so cazzo pt.2)	99
Teorema 8.5(30L)	99
8.2 LA CLASSE PSPACE	100
Definizione 8.6	101

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.

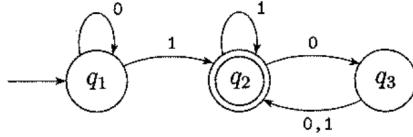


Figura 1: Esempio di automa finito

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 $A \cdot B$.

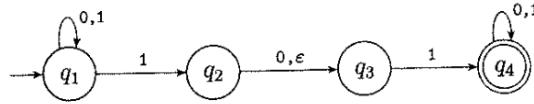
- M_1 riconosce A
- M_2 riconosce B
- M deve riconoscere $M_1 \circ M_2$, ma M non sa dove dividere il suo input.

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 .

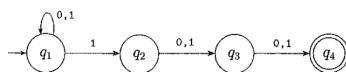
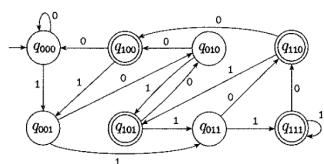


FIGURA 1.31
L'NFA N_2 che riconosce A

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_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 $\{\cdot\}$ di simboli input che consiste di un solo simbolo. Un alfabeto contenente solo un simbolo è chiamato un **alfabeto unario**.

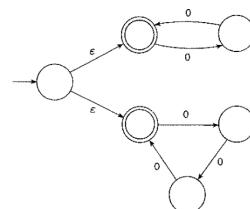


FIGURA 1.34
L'NFA N_3

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, e 000000, 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 .

- $P(Q)$ = insieme potenza di Q .

- Per ogni alfabeto Σ , scriviamo Σ_ε per denotare $\Sigma \cup \{\varepsilon\}$.

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

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

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. $\delta : Q \times \Sigma_\varepsilon \rightarrow P(Q)$ 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(oramai)

Teorema 1.39 (oramai)

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

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

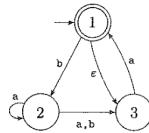


FIGURA 1.42
l'NFA N_4

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 la funzione 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 sia 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 su a e 1 su a volta va in 3 con un ε arco. Lo stato $\{3\}$ va in \emptyset sull'input b .

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

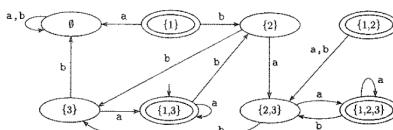


FIGURA 1.43
Un DFA D che è equivalente all'NFA N_4

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.

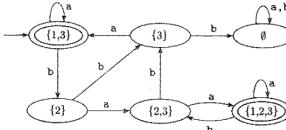


FIGURA 1.44
Il DFA D dopo aver tolto gli stati non necessari

Figura 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.

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.

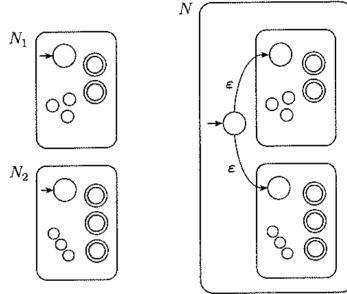


Figura 3: Esempio di NFA che riconosce l'unione di due linguaggi regolari

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 .

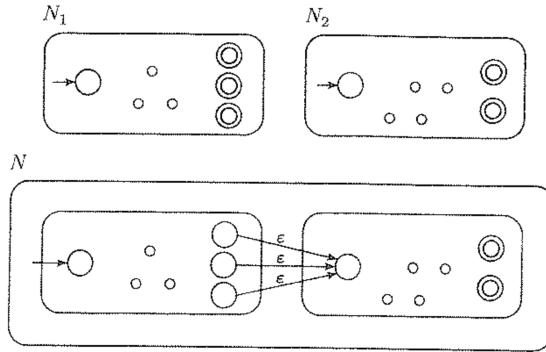


Figura 4: Esempio di concatenazione di due linguaggi regolari

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(orale)

Teorema 1.49 (orale)

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 la 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.

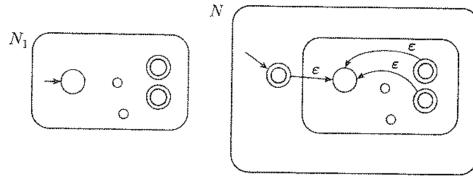


Figura 5: Esempio di operazione star su un linguaggio regolare

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. \emptyset ,
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 $\{\epsilon\}$, rispettivamente. Nel punto 3, l'espressione regolare \emptyset 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 \emptyset . L'espressione ϵ rappresenta il linguaggio che contiene una sola stringa - ossia, la stringa vuota - mentre \emptyset 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. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{la lunghezza di } w \text{ è un multiplo di } 3\}$.
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.

⁵La **lunghezza** di una stringa è il numero di simboli che essa contiene.

Figura 6: Esempio di espressione regolare

$R \cup \emptyset = R$. 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 \emptyset 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 $L(R \cup \varepsilon) = \{0, \varepsilon\}$.
- $R \circ \emptyset$ può non essere uguale a R .
- Per esempio, se $R = 0$, allora $L(R) = \{0\}$ ma $L(R \circ \emptyset) = \emptyset$.

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(oramale)

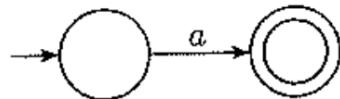
Lemma 1.55 (oramale)

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.

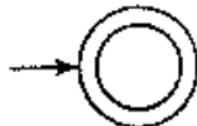
1. $R = a$ per qualche $a \in \Sigma$. Allora $L(R) = \{a\}$ e il seguente NFA riconosce $L(R)$.



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.

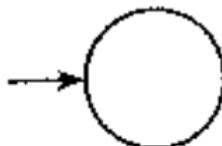
Formalmente, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, dove descriviamo δ dicendo che $\delta(q_1, a) = \{q_2\}$ e $\delta(r, b) = \emptyset$ per $r \neq q_1$ o $b \neq a$.

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. $R = \emptyset$. Allora $L(R) = \emptyset$, e il seguente NFA riconosce $L(R)$.



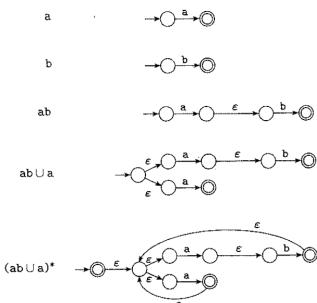
Formalmente, $N = (\{q\}, \sigma, \delta, q, 0)$, dove $\delta(r, b) = \emptyset$ per ogni r e b .

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.

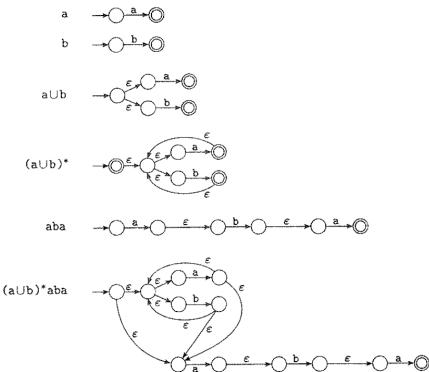
ESEMPIO 1.56

Trasformiamo l'espressione regolare $(ab \cup a^*)^*$ in un NFA in una sequenza di passi. Costruiamo l'automa dalla 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 minore 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 DFA. Alcuni dei passi meno importanti non sono mostrati.



Lemma 1.60(oreale 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.

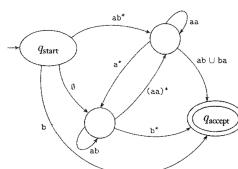


Figura 7: Esempio di GNFA

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 \emptyset tra stati che non hanno archi. Questo ultimo passo non cambierebbe il linguaggio riconosciuto perché una transizione etichettata con \emptyset 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.

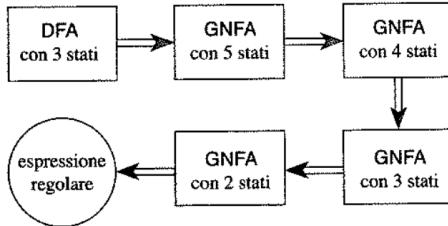


Figura 8: Trasformazione di un GNFA in un'espressione regolare

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.

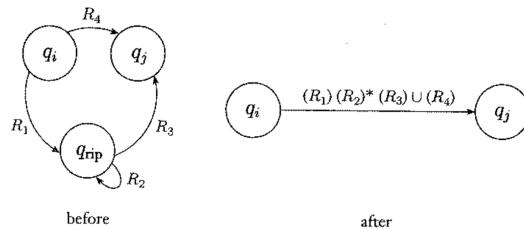


Figura 9: Esempio di trasformazione di un GNFA

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:

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

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 $R_1 = b$, $R_2 = a \cup b$, $R_3 = \epsilon$ ed $R_4 = 0$. Pertanto, la nuova etichetta sull'arco da 1 ad a è $(b)(a \cup b)^*(\epsilon) \cup 0$. 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.

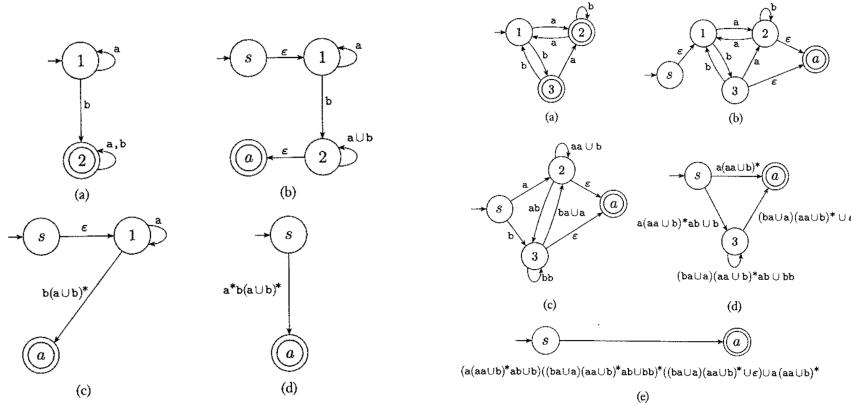


Figura 10: Trasformazione di un DFA in un'espressione regolare

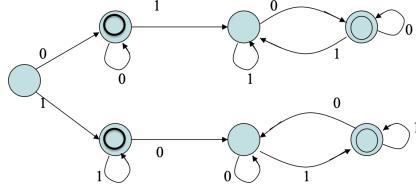
1.3.3 Linguaggi non regolari

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

Consideriamo il linguaggio $B = \{0^n 1^n | n \geq 0\} = \{\epsilon, 01, 0011, 000111, \dots\}$. 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:

[Link a chatgpt che lo spiega meglio.](#)

PUMPING LEMMA:

A 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..!)

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

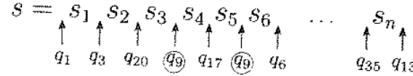
EQUIVALENTEMENTE:

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

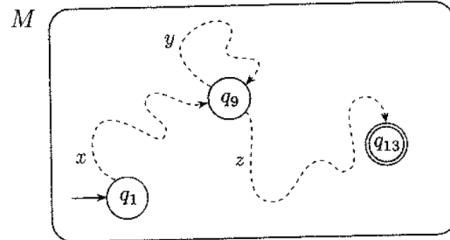
Utilizzo pratico (provare che A non è regolare):

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 $n + 1$. Poiché 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 perché 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 $xy^i z$ per ogni $i > 0$. Nel caso $i = 0$, $xy^i z = xz$, anch'essa accettata. Questo prova la condizione 1. Per verificare la condizione 2, vediamo che $|y| > 0$, poiché 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 attraversati 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 . Poiché 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 $z = s_l\dots s_n$. Poiché 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 .

ESEMPIO 1.73

Sia B il linguaggio $\{0^n1^n\mid n \geq 0\}$. Usiamo il pumping lemma per provare che B non è regolare. Assumiamo al contrario che B 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 B ed s ha lunghezza maggiore di p , il pumping lemma assicura che s può essere divisa in tre parti, $s = xyz$, tali che xyz è una stringa vuota o y uguale alla stringa 0^p1^p , allora xy^iz ha sempre lo stesso numero di simboli uguali a 0 e simboli uguali a 1 e quindi è privo di 0 .

Quindi s non può essere iterata. Ora proviamo la contraddizione desiderata.

Supponiamo che s in questo esempio richiede più attenzione che nell'Esempio 1.73. Se invece avessimo scelto $s = (01)^p$, avremmo avuto dei problemi perché abbiamo bisogno di una stringa che non può essere iterata e questa stringa può essere iterata, perfino prendendo in considerazione la condizione 3. Ricercate a vedere come iterarla! Un modo per farlo è porre $x = \epsilon$, $y = 01$ e $z = (01)^{p-1}$. Allora $xy^iz \in C$ per ogni valore di i . Se non riesci a trovare una stringa che non può essere iterata al primo tentativo, non disperare. Provate un'altra.

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 $xyyz$ può avere lo stesso numero di simboli

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. Ne successivi due esempi, alcune scelte per s non funzionano quindi è richiesta un'attenzione supplementare.

ESEMPIO 1.74

Sia $C = \{0^i1^i\mid i \geq 0\}$. Usiamo il pumping lemma per provare che C non è regolare. 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^iz è in C . Ci piacerebbe che $xy^iz = 0^i1^i$ per ogni $i \geq 0$, ma questo risultato è impossibile. Se prendiamo $x = \epsilon$, $y = 0^p$ e $z = 1^p$, allora $xy^iz = 0^i1^i$ per ogni $i \geq 0$. Quindi s non è possibile iterare.

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^iz è in C . Ci piacerebbe che $xy^iz = 0^i1^i$ per ogni $i \geq 0$, ma questo risultato è impossibile. Se prendiamo $x = 0^p$, $y = 1^p$ e $z = \epsilon$, allora $xy^iz = 0^i1^i$ per ogni $i \geq 0$. Quindi s non è possibile iterare.

Quindi 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 si può essere diviso rende più facile mostrare che la stringa $s = 0^p1^p$ che abbiamo scelto non può essere iterata. Se $|xy| \leq p$, allora y deve contenere soli simboli uguali a 0. Quindi s non può essere iterata. Quindi ci fermiamo la contraddizione desiderata.

Sarebbe utile la stringa s in questo esempio richiede più attenzione che nell'Esempio 1.73. Se invece avessimo scelto $s = (01)^p$, avremmo avuto dei problemi perché abbiamo bisogno di una stringa che non può essere iterata e questa stringa può essere iterata, perfino prendendo in considerazione la condizione 3. Ricercate a vedere come iterarla! Un modo per farlo è porre $x = \epsilon$, $y = 01$ e $z = (01)^{p-1}$. Allora $xy^iz \in C$ per ogni valore di i . Se non riesci a trovare una stringa che non può essere iterata al primo tentativo, non disperare. Provate un'altra.

Un'ultima alternativa per provare che C non è regolare segue dal fatto che sappiamo che B non è regolare. Perciò questo linguaggio 0^*1^* è regolare e la classe dei linguaggi regolari è chiusa rispetto all'intersezione, come provammo nella

Quindi, la lunghezza di xy^iz 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 $xy^iz \notin C$ e concludeiamo che C non è regolare.

ESEMPIO 1.77

Talvolta "cancellare" ("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 per E dato dal pumping lemma. Sia $s = 0^{p+1}1^p$. 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 $xyyz$ 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 $xy^iz \in E$ anche per $i = 0$, quindi consideriamo la stringa $xy^0z = xz$. Eliminare la stringa y non diminuisce 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.

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 1.75

Sia $F = \{ww\mid w \in \{0,1\}^*\}$. 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 $0^p1^p0^p$. 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 ma volta cruciale perché senza di essa potremmo iterare s se poniamo x e z uguali alla stringa vuota. Con la condizione 3 la prova segue poiché y deve consistere solo di simboli uguali a 0, quindi $xy^p \notin F$.

Osserviamo che abbiamo scelto $s = 0^p1^p0^p$ 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 $D = \{1^{n^2}\mid n \geq 0\}$. 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 . 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^iz è in D . Come negli esempi precedenti, mostriamo che questo risultato è impossibile. Farlo in questo caso richiede una piccola riflessione sulla successione dei quadrati perfetti:

$$0, 1, 4, 9, 16, 25, 36, 49, \dots$$

Notate il divario crescente tra gli elementi successivi di questa sequenza. Elementi grandi di questa sequenza non possono essere vicini l'uno all'altro.

Ora consideriamo le due stringhe xy^2z e xy^3z . 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, $|xy| \leq p$ e quindi $|y| \leq p$. Abbiamo $|xy^2z| = p^2$ e allora $|xy^3z| \leq p^2+p$. Ma $p^2+p < p^2+2p+1 = (p+1)^2$. Indice la condizione 2 implica che y non è la stringa vuota e perciò $|xy^2z| > p^2$.

Figura 11: Esempi 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** uvw , e lo denotiamo con $uAv \Rightarrow uvw$. 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

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

Questa grammatica genera stringhe come **abab**, **aaabbb**, 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 $G_4 = (V, \Sigma, R, \{\text{EXPR}\})$.

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.

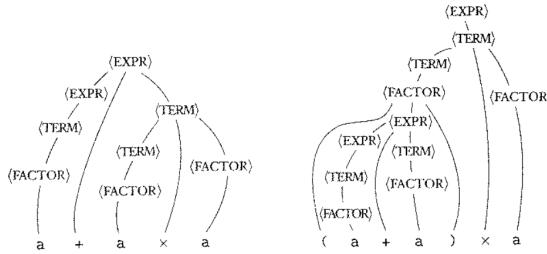


FIGURA 2.5
Alberi sintattici per le stringhe **a+axa** e **(a+a)xa**

Figura 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

$$S_1 \rightarrow 0S_11 \mid \epsilon$$

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

$$S_2 \rightarrow 1S_20 \mid \epsilon$$

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

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

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 $R_i \rightarrow \epsilon$ 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 $a+a*a$ ambiguumamente. La figura seguente mostra i due diversi alberi sintattici.

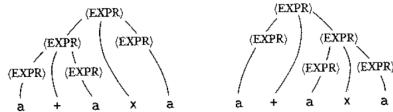


Figura 13: Esempio di grammatica context-free ambigua

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. $S \rightarrow \epsilon$, 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 $U_i \rightarrow u_i$.

[Link a chatgpt che lo spiega meglio.](#)

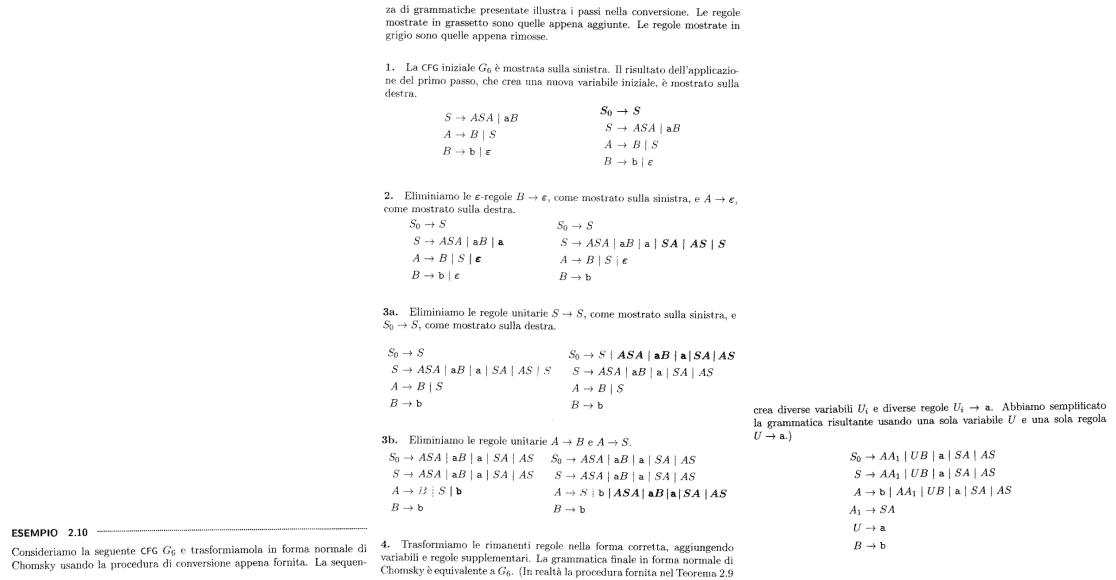


Figura 14: Esempio regole Teorema 2.9

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.

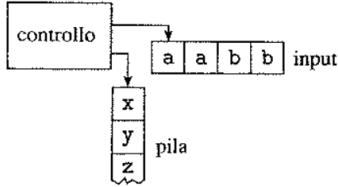


Figura 15: Rappresentazione schematica di un automa a pila

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 deterministici riconoscono alcuni linguaggi che nessun automa a pila deterministico può riconoscere. Ricordiamo che gli automi finiti deterministici e non deterministici riconoscono la stessa classe di linguaggi, quindi la situazione per gli automi a pila è diversa. Ci concentreremo sugli automi a pila non deterministici 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	$\{(q_2, \$)\}$
q_1				
q_2	$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$		
q_3		$\{(q_3, \varepsilon)\}$		$\{(q_4, \varepsilon)\}$
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 uno 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.

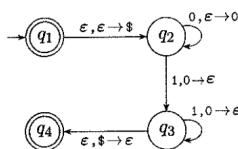


Figura 16: Esempio di automa a pila

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.

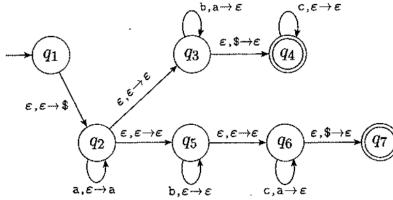


FIGURA 2.17

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.

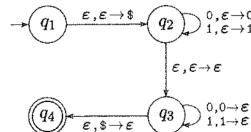


FIGURA 2.19

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

Figura 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 .

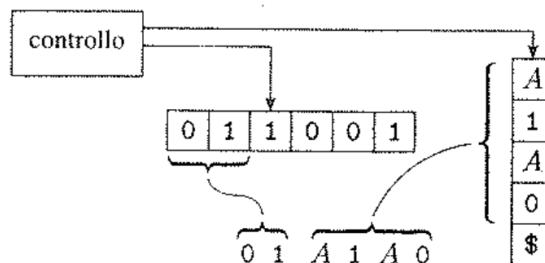


FIGURA 2.22

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.

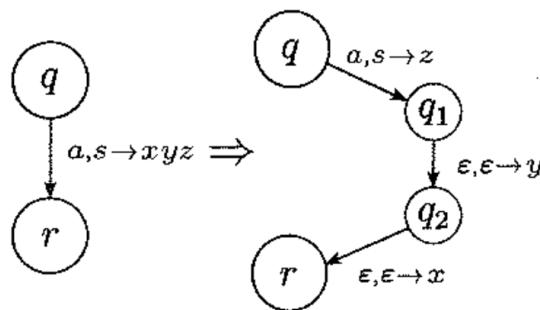
DIMOSTRAZIONE.

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 un 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} .

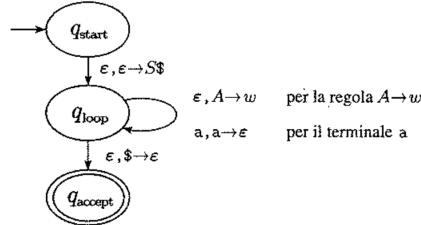
La funzione di transizione è definita come segue. Cominciamo inizializzando la pila inserendo i simboli $\$S$, realizzando così il passo 1 nella descrizione informale: $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, \$S)\}$. Poi aggiungiamo le transizioni per il ciclo principale del passo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

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

La funzione di transizione è mostrata nel diagramma seguente.

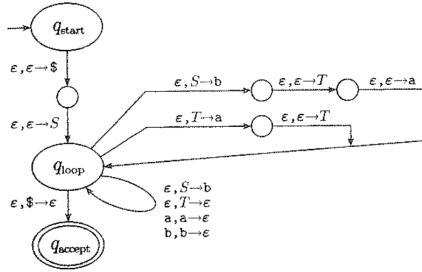


FIGURA 2.26
Diagramma di stato di P_1

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(orale 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 elimina 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.

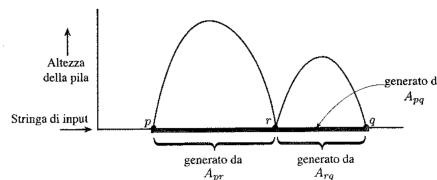


FIGURA 2.28
La computazione del PDA corrispondente alla regola $A_{pq} \rightarrow A_{pr}A_{rq}$

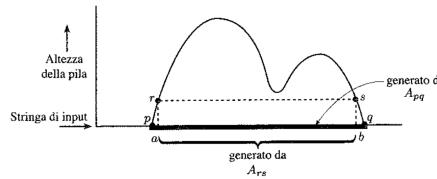


FIGURA 2.29
La computazione del PDA corrispondente alla regola $A_{pq} \rightarrow aA_{rs}b$

Figura 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 $A_{pp} \rightarrow \varepsilon$. Ovviamente, 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 $x = ayb$. Poichè $A_{rs} \Rightarrow^* y$, in k passi, l'ipotesi induttiva ci dice che P può andare da r con la pila vuota a s con la pila vuota. Poichè $A_{pq} \rightarrow aA_{rs}b$ 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 $x = yz$. Poichè $A_{pr} \Rightarrow^* y$ 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 pila vuota 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 $x = ayb$. 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 $A_{pq} \Rightarrow^* x$.

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 contendono 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.

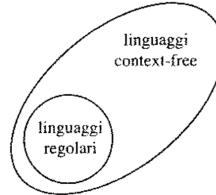


FIGURA 2.33
Relazione tra i linguaggi regolari e i linguaggi context-free

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

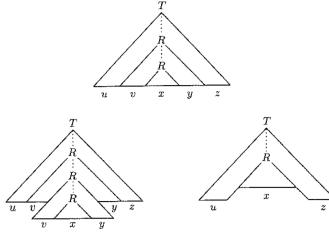
Se A è un linguaggio context-free, allora esiste un numero p (lunghezza del pumping) tale che, se s è una qualsiasi stringa in A di lunghezza almeno p , allora s può essere divisa in cinque parti $s = uvxyz$ che soddisfano le condizioni

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 perchè 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. Poichè 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 perchè 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 abbiano 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 $b^{|V|+1} = p$.

[Link a chatpt che lo spiega meglio.](#)

Scritto più precisamente:

PUMPING LEMMA:

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

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:

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

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. Scegliamo 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 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 usata 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 $0^i 1^j 0^i 1^j$, dove $i < j$ 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 $uv^2xy^2z = uzz$. Questa stringa contiene lo stesso numero di a che ha s , ma contiene meno b o meno c . Perciò 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 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^2xy^2z 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.

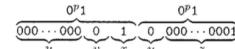
2. Quando v o y contengono più di un tipo di simbolo, uv^2xy^2z non conterrà i simboli nell'ordine corretto. Perciò essa non può essere un elemento di C e giungiamo a una contraddizione.

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 sceglieremo la stringa s è meno ovvio. Una possibilità è la stringa $0^p 1^p 0^p 1^p$. 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.



Proviamo con un altro candidato per s . Intuitivamente, la stringa $0^p 1^p 0^p 1^p$ 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 $s = 0^p 1^p 0^p 1^p$ 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.

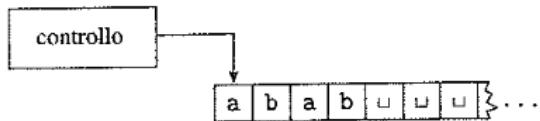


Figura 19: Schema di una macchina di Turing

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.

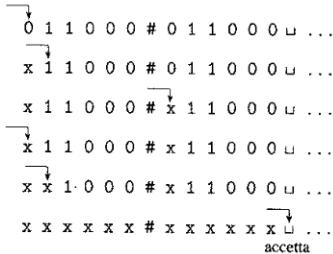
Introduciamo una macchina di Turing M_1 per testare l'appartenenza del linguaggio

$B = \{w\#w \mid w \in \{0, 1\}^*\}$. 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.

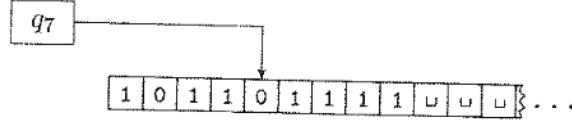


Figura 20: Esempio di macchina di Turing

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

$uaq_i bv$ 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 perchè si ferma in uno stato di rifiuto, sia perchè 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** perchè 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 forniremo regole precise per fare 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 bene sempre 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 $A = \{0^n \mid n \geq 0\}$, il linguaggio formato da tutte le stringhe di 0 la cui lunghezza è una potenza di 2.

$M_2 = \langle \text{su input } w:$

1. Muove la testina da sinistra a destra sul nastro, cancellando ogni secondo 0.
2. Se al passo 1, il nastro contieneva un solo 0, accetta.
3. Se al passo 1, il nastro contieneva più di un singolo 0 ed il loro numero era dispari, rifiuta.
4. Ripeta 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 $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$,
- $\Gamma = \{\text{x}, \text{z}\}$.
- 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} .

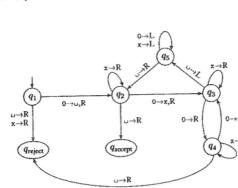


FIGURA 3.8
Diagramma di stato per la macchina di Turing M_2

In questo diagramma di stato, nella transizione da q_1 a q_2 compare l'etichetta $0 \rightarrow 0, R$. Questa etichetta indica che quando siamo nello stato q_1 e la testina è sulla posizione 0, la macchina si porta nello stato q_2 , scrive 0 e sposta la testina a destra. In altre parole, $\delta(q_1, 0) = (q_2, 0, R)$. Per chiarezza utilizziamo l'abbreviazione $0 \rightarrow R$ per indicare la transizione da q_3 a q_4 , il che significa che la macchina si sposta verso destra durante la lettura di 0 quando si trova nello stato q_3 ma non modifica il nastro, così $\delta(q_3, 0) = (q_4, 0, R)$.

Questa macchina inizia scrivendo allo stato iniziale tutto il nastro di sinistra in 0. Per determinare normalmente l'estrema sinistra del nastro nel passo 1, dovremmo utilizzare un simbolo più indicativo, quale $\#$, come delimitatore di inizio nastro qui utilizziamo un blank per rispettare l'affidato 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

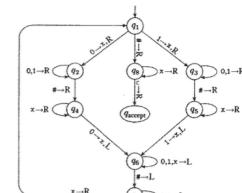
macchina su input 0000. La configurazione iniziale è q_10000 . La sequenza delle configurazioni della macchina è la seguente; leggete le colonne verso il basso e da sinistra a destra.

q_10000	$\text{x}q_20000$	$\text{z}q_30000$
$\text{x}q_20000$	$\text{z}q_30000$	q_40000
$\text{z}q_30000$	q_40000	q_50000
q_40000	q_50000	q_{accept}
q_50000	q_{accept}	q_{reject}

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 $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \dots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0, 1, \#\}$ e $\Gamma = \{0, 1, \#, x, z\}$,
- 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, barrandone 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 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_2 , 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 più 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

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

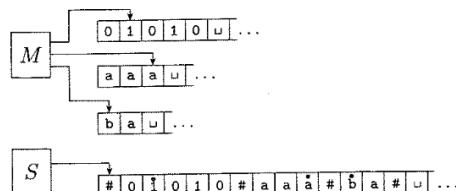
significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli da a_1 a a_k , allora la macchina va nello stato q_j , scrive i simboli da b_1 a b_k sui nastri, e muove le testine a sinistra o a destra o le lascia ferme. Le macchine di Turing multinastro sembrano più potenti 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

$$\# \dot{w}_1 w_2 \dots w_n \# \dot{\square} \# \dot{\square} \# \dots \#.$$
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 lecita 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.

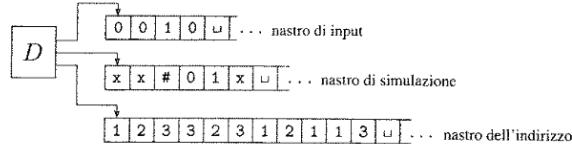
Teorema 3.16 orale(24-30)

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 per determinare quale scelta fare tra quelle consentite dalla funzione di transizione di N . Se non rimangono più simboli 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.

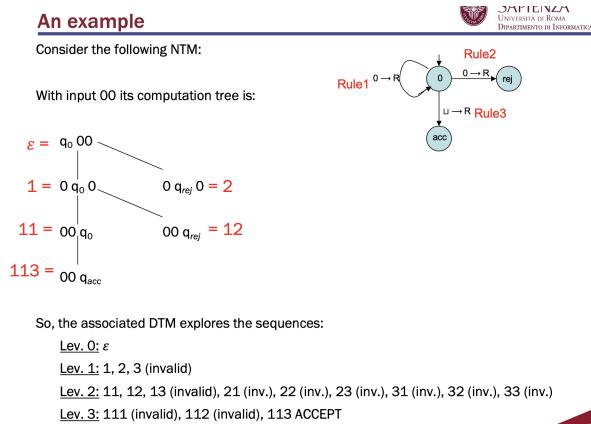


Figura 21: Esempio di computazione

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.

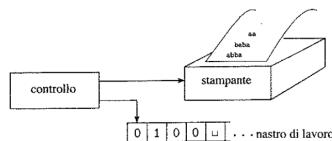


Figura 22: Schema di un enumeratore

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**, perché 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 algoritmamente. 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". È 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.

<i>Nozione intuitiva di algoritmo</i>	equivalente a	<i>Algoritmi mediante Macchina di Turing</i>
---	---------------	--

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 decisori. 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 Turing

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 marcatis. 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 ne avrete bisogno, se non espressamente 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.

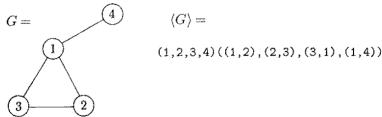


FIGURA 3.24
Un grafo G e la sua codifica $\langle G \rangle$

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 imposta 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 degli 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(orale) 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. Poiché 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, perché 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 perchè $L(C) = \emptyset$ 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 nè 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.”

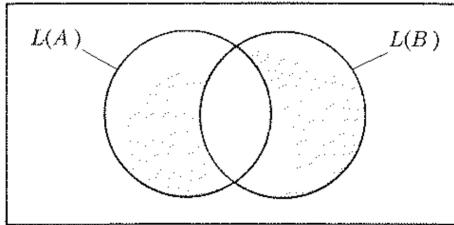


Figura 23: Differenza simmetrica di due linguaggi

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(oramai) 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 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.

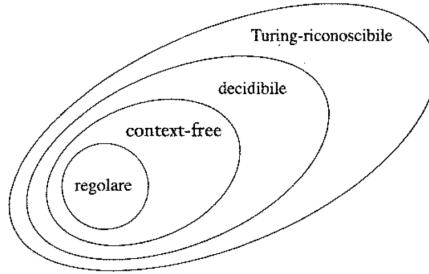


Figura 24: Relazioni tra le classi di linguaggi

4.2 Indecidibilità

In questa sezione dimostriamo uno dei teoremi più importanti della teoria della computazione: esiste un problema specifico che è algoritmicamente irrisolvibile.

Ora affronteremo il nostro primo teorema che stabilisce l'indecidibilità di uno specifico linguaggio: il problema di determinare se una macchina di Turing accetta una determinata stringa in input. Chiamiamo

tal linguaggio A_{TM} per analogia con A_{DFA} e A_{CFG} . Tuttavia, mentre A_{DFA} e A_{CFG} sono decidibili, A_{TM} non lo è. Sia

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM e } M \text{ accetta la stringa } w\}.$$

Teorema 4.11(orale 24-30)

A_{TM} è indecidibile.

Prima di vedere la dimostrazione, osserviamo che A_{TM} è Turing-riconoscibile. Quindi questo teorema mostra che i riconoscitori sono più potenti dei decisori. Richiedere che una TM si ferma su ogni input limita le tipologie dei linguaggi che possono essere riconosciuti. La seguente macchina di Turing U riconosce A_{TM} .

U = "Su input $\langle M, w \rangle$, dove M è una TM e w è una stringa:

1. Simula M su input w .
2. Se durante la computazione M entra nello stato di accettazione, accetta; se M entra nello stato di rifiuto, rifiuta."

Si noti che questa macchina cicla su input $\langle M, w \rangle$ se M cicla su w , questo è il motivo per cui non decide A_{TM} . Se l'algoritmo avesse modo di determinare che M non si ferma su w , sarebbe in tal caso in grado di rifiutare w . Come dimostreremo, un algoritmo non ha modo di determinarlo.

La macchina di Turing U è, di per sé, interessante. È un esempio della prima macchina universale di Turing introdotta nel 1936. Questa macchina è chiamata universale perché è in grado di simulare qualsiasi altra macchina fi Turing a partire dalla descrizione di tale macchina. La macchina di Turing universale ha avuto un ruolo importante nello stimolare lo sviluppo di computer a programma memorizzato.

4.2.1 Il metodo della diagonalizzazione

Definizione 4.12

Supponiamo di avere gli insiemi A e B ed una funzione f da A in B . Diciamo che f è **iniettiva**, se essa non mappa mai due elementi diversi in uno stesso punto - cioè se $f(a) \neq f(b)$ per ogni $a, b \in A$ con $a \neq b$. Diciamo che f è **suriettiva**, se tocca ogni elemento di B - cioè se per ogni $b \in B$ esiste un $a \in A$ tale che $f(a) = b$. Diciamo che A e B hanno la **stessa cardinalità** se esiste una funzione iniettiva e suriettiva $f : A \rightarrow B$. Una funzione che è sia iniettiva che suriettiva è detta **biettiva**. In una funzione biettiva ogni elemento di A viene mappato in un unico elemento di B e per ogni elemento di B esiste un unico elemento di A che viene mappato in esso. Una biezione è un modo semplice per accoppiare elementi di A con elementi di B .

ESEMPIO 4.13

Sia \mathcal{N} l'insieme dei numeri naturali $\{1, 2, 3, \dots\}$ e sia \mathcal{E} l'insieme dei numeri naturali pari $\{2, 4, 6, \dots\}$. Utilizzando la definizione di cardinalità di Cantor possiamo vedere che \mathcal{N} e \mathcal{E} hanno la stessa cardinalità. La funzione f che crea una corrispondenza tra \mathcal{N} e \mathcal{E} è semplicemente $f(n) = 2n$. Possiamo visualizzare f più facilmente con l'aiuto di una tabella.

n	$f(n)$
1	2
2	4
3	6
:	:

Naturalmente, questo esempio sembra strano. Intuitivamente, \mathcal{E} sembra più piccolo di \mathcal{N} in quanto \mathcal{E} è un sottoinsieme proprio di \mathcal{N} . Ma è possibile accoppiare ogni elemento di \mathcal{N} con un elemento di \mathcal{E} , quindi dichiariamo che questi due insiemi hanno la stessa cardinalità.

Figura 25: Biezione tra due insiemi

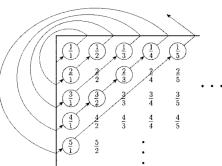
Definizione 4.14

Un insieme A è **numerabile** se è finito o ha la stessa cardinalità di \mathbb{N} .

ESEMPIO 4.15

Passiamo ora ad un esempio ancora più strano. Se consideriamo l'insieme $\mathbb{Q} = \{\frac{m}{n} | m, n \in \mathbb{N}\}$ dei numeri razionali positivi, \mathbb{Q} sembra essere molto più grande di \mathbb{N} . Tuttavia, questi due insiemi hanno la stessa cardinalità, secondo la nostra definizione. Possiamo una biezione con \mathbb{N} ? Evidentemente che $\mathbb{Q} \neq \mathbb{N}$. Un modo semplice per farlo è quello di disporre tutti gli elementi di \mathbb{Q} . Per farlo accoppiiamo il primo elemento della lista con il numero 1 di \mathbb{N} , il secondo elemento della lista con il numero 2 di \mathbb{N} , e così via. Dobbiamo garantire che ogni elemento di \mathbb{Q} sia presente una sola volta nella lista.

Per ottenere questa lista creiamo una matrice infinita contenente tutti i numeri razionali positivi, come mostrato nella Figura 4.16. La riga i -esima contiene tutti i numeri con numerator e i e la colonna j -esima ha tutti i numeri con denominatore j . In tal modo il numero $\frac{i}{j}$ occupa la i -esima riga e la j -esima colonna. Ora trasformiamo questa matrice in una lista. Un modo (errato) di fare potrebbe essere quello di cominciare la lista con tutti gli elementi della prima riga, ma questo non è un appiglio perché la prima riga è infinita, quindi non raggiungeremmo mai la seconda riga. Elencheremo invece gli elementi delle diagonali, sovrapposte al diagramma, a partire da un angolo. La prima diagonale include il singolo elemento $\frac{1}{1}$, e la seconda diagonale include i due elementi $\frac{1}{2}$ e $\frac{2}{1}$. Così i primi tre elementi della lista sono $\frac{1}{1}$, $\frac{2}{1}$ e $\frac{1}{2}$. Nella terza diagonale sorge una complicazione. Essa contiene $\frac{1}{3}$, $\frac{2}{2}$, e $\frac{3}{1}$. Se li aggiungessimo così come sono alla lista, ripeteremmo $\frac{1}{2} = \frac{2}{2}$. Risolviamo il problema omettendo un elemento quando esso dà origine ad una ripetizione. Così aggiungiamo solo i due nuovi elementi $\frac{2}{2}$ e $\frac{3}{1}$. Continuando in questo modo si ottiene una lista di tutti gli elementi di \mathbb{Q} .



Dopo aver visto la biezione da \mathbb{N} a \mathbb{Q} si potrebbe pensare di dimostrare che presi qualsiasi coppia di insiemi finiti, questi hanno la stessa dimensione. Dopo tutto dovete solo mostrare una biezione, e questo esempio mostra che esistono biezioni sorprendenti. Tuttavia, per alcuni insiemi infiniti non c'è alcuna biezione con \mathbb{N} . Questi insiemi sono semplicemente troppo grandi.

Un tale insieme è detto ***non numerabile***.

L'insieme dei numeri reali è un esempio di insieme non numerabile.

Teorema 4.17

\mathbb{R} è non numerabile.

DIMOSTRAZIONE. Per dimostrare che \mathbb{R} è non numerabile, mostriamo che non esiste una biezione tra \mathbb{N} e \mathbb{R} . La dimostrazione è per assurdo. Supponiamo quindi che esista una biezione f tra \mathbb{N} e \mathbb{R} . Il nostro compito è quello di dimostrare che f non funziona come dovrebbe. Per essere una biezione, f deve mappare ogni intero positivo in un numero reale. Tuttavia troveremo un numero x in \mathbb{R} che non è mappato da f , il che rappresenta la contraddizione cercata. Un metodo per trovare questo numero x è quello di costruirlo. Scegliamo ogni cifra di x in modo da renderlo differente da ogni numero reale accoppiato con un elemento di \mathbb{N} . Alla fine saremo sicuri che x risulta diverso da ogni numero reale accoppiato con un elemento di \mathbb{N} . Possiamo illustrare questa idea con un esempio. Supponiamo che la biezione f esista. Prendiamo $f(1) = 3.14159\dots$, $f(2) = 55.55555\dots$, $f(3) = \dots$, e così via, tanto per dare alcuni valori per f . Allora f accoppia il numero 1 con 3.14159..., il numero 2 con 55.55555..., e così via. La tabella seguente mostra alcuni valori di un'ipotetica biezione tra \mathbb{N} e \mathbb{R} .

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
\vdots	\vdots

Costruiamo il numero x desiderato, dandone la rappresentazione decimale. Si tratta di un numero compreso tra 0 e 1, quindi tutte le sue cifre significative sono cifre frazionarie che seguono la virgola decimale. Il nostro obiettivo è quello di garantire che $x \neq f(n)$ per ogni $n \in \mathbb{N}$. Per garantire $x \neq f(1)$, scegliamo la cifra di x come una qualsiasi cifra diversa dalla prima cifra decimale 1 di $f(1) = 3.14159\dots$. Arbitrariamente, scegliamola pari a 4. Per assicurarci che $x \neq f(2)$, scegliamo la seconda cifra decimale di x diversa dalla seconda cifra decimale 5 di $f(2) = 55.55555\dots$. Arbitrariamente, scegliamola pari a 6. La terza cifra frazionaria di $f(3) = 0.12345\dots$ è 3, quindi scegliamo la terza cifra frazionaria di x pari a 4. Continuando in questo modo lungo la diagonale della tavola per f , otteniamo tutte le cifre di x , come mostrato nella tabella seguente. Sappiamo che x non è $f(n)$ per ogni n perché differisce da $f(n)$ nella n -esima cifra frazionaria. (Un piccolo problema nasce dal fatto che alcuni numeri, come 0.1999... e

0.2000..., sono uguali, anche se le loro rappresentazioni decimali sono diverse. Evitiamo questo problema semplicemente non selezionando mai le cifre 9 o 0 per x).

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
:	:

$x = 0.4641 \dots$

Il teorema precedente ha un'importante applicazione nella teoria della computazione. Esso dimostra che alcuni linguaggi non sono decidibili e neppure Turing-riconoscibili, per la ragione che l'insieme dei linguaggi è non numerabile mentre l'insieme di tutte le macchine di Turing è numerabile. Poichè ogni macchina di Turing è in grado di riconoscere un solo linguaggio e ci sono più linguaggi che macchine di Turing, alcuni linguaggi non sono riconosciuti da una qualche macchina di Turing. Tali linguaggi non sono Turing-riconoscibili, come enunciato nel seguente corollario.

Corollario 4.18(oramai 24-30) Alcuni linguaggi non sono Turing-riconoscibili.

DIMOSTRAZIONE. Per dimostrare che l'insieme di tutte le macchine di Turing è numerabile dobbiamo prima osservare che l'insieme di tutte le stringhe Σ^* è numerabile, per ogni alfabeto Σ . Avendo solo un numero finito di stringhe di ogni lunghezza, possiamo formare una lista di Σ^* scrivendo tutte le stringhe di lunghezza 0, lunghezza 1, lunghezza 2, e così via. L'insieme di tutte le macchine di Turing è numerabile perchè ogni macchina di Turing M può essere codificata con una stringa $\langle M \rangle$. Se ci limitiamo a tralasciare quelle stringhe che non sono la codifica di una macchina di Turing, possiamo ottenere una lista di tutte le macchine di Turing. Per mostrare che l'insieme di tutti i linguaggi è non numerabile, dobbiamo prima osservare che l'insieme delle sequenze binarie indeterminate è non-numerabile. Una sequenza binaria infinita è una sequenza senza fine di 0 e 1. Sia \mathcal{B} l'insieme di tutte le sequenze binarie infinite. Possiamo dimostrare che \mathcal{B} è non numerabile utilizzando una dimostrazione mediante diagonalizzazione simile a quella che abbiamo utilizzato nel [Teorema 4.17](#) per mostrare la non numerabilità di \mathbb{R} . Sia \mathcal{L} l'insieme di tutti i linguaggi sull'alfabeto Σ . Mostriamo che \mathcal{L} è non numerabile dando una corrispondenza con \mathcal{B} , dimostrando così che i due insiemi hanno la stessa cardinalità. Sia $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Ogni linguaggio $A \in \mathcal{L}$ ha un'unica sequenza in \mathcal{B} . Il bit i -esimo della sequenza è un 1 se $s_i \in A$ e 0 altrimenti; questa è chiamata la **sequenza caratteristica** di A . Per esempio, se A fosse il linguaggio di tutte le stringhe che iniziano con uno 0 sull'alfabeto binario, la sua sequenza caratteristica X_A sarebbe

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \dots\};$$

$$A = \{0, 00, 01, 000, 001, \dots\};$$

$$X_A = 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ \dots .$$

La funzione $f : \mathcal{L} \rightarrow \mathcal{B}$, dove $f(A)$ è uguale alla sequenza caratteristica di A , è sia iniettiva che suriettiva, quindi è una biezione. Pertanto, essendo \mathcal{B} non numerabile, anche \mathcal{L} è non numerabile.

Abbiamo così dimostrato che l'insieme di tutti i linguaggi non può essere messo in corrispondenza biunivoca con l'insieme di tutte le macchine di Turing. Concludiamo quindi che alcuni linguaggi non sono riconosciuti da alcuna macchina di Turing.

4.2.2 Un linguaggio indecidibile

Siamo ora pronti a dimostrare il [Teorema 4.11](#), l'indecidibilità del linguaggio

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM e } M \text{ accetta la stringa } w\}.$$

DIMOSTRAZIONE. Assumiamo che A_{TM} è decidibile, per poi ottenere una contraddizione. Supponiamo che H sia un decisore per A_{TM} . Sull'input $\langle M, w \rangle$, dove M è una TM e w è una stringa, H si

ferma ed accetta se M accetta w . Inoltre, H si ferma e rifiuta se M non accetta w . In altri termini, assumiamo che H è una TM, dove

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

Ora costruiamo una nuova macchina di Turing D avente H come sottoprocedura. Questa nuova TM chiama H per determinare cosa fa M quando l'input è la sua stessa descrizione $\langle M \rangle$. Una volta che D ha determinato questa informazione, essa fa il contrario. Cioè, rifiuta se M accetta ed accetta se M non accetta. Diamo ora una descrizione di D .

D = "Sull'input $\langle M \rangle$, dove M è una TM:

1. Esegue la TM H su input $\langle M, \langle M \rangle \rangle$.
2. Dà in output l'opposto di ciò che H dà in output. Cioè, se H accetta, rifiuta; se H rifiuta, accetta."

Non lasciatevi confondere dall'idea di attivare una macchina sulla sua stessa descrizione! Ciò è simile all'esecuzione di un programma che chiama se stesso in input, qualcosa che si fa a volte nella pratica. Ad esempio, un compilatore è un programma che traduce altri programmi. Un compilatore per il linguaggio Python può essere esso stesso scritto in Python, quindi eseguire tale programma su se stesso avrebbe senso. Ricapitolando,

$$D(\langle M \rangle) = \begin{cases} \text{accetta} & \text{se } M \text{ non accetta } \langle M \rangle, \\ \text{rifiuta} & \text{se } M \text{ accetta } \langle M \rangle. \end{cases}$$

Cosa succede quando eseguiamo D con la sua stessa descrizione $\langle D \rangle$ in input? In tal caso, otteniamo

$$D(\langle D \rangle) = \begin{cases} \text{accetta} & \text{se } D \text{ non accetta } \langle D \rangle, \\ \text{rifiuta} & \text{se } D \text{ accetta } \langle D \rangle. \end{cases}$$

Indipendentemente da ciò che D fa, essa è costretta a dare il contrario, il che è ovviamente una contraddizione. Quindi, né la TM D nè la TM H possono esistere.

Rivediamo i passi di questa prova. Supponiamo che la TM H decida A_{TM} . Quindi usiamo H per costruire una TM D che prende in input $\langle M \rangle$, tale che D accetta M esattamente quando M non accetta il suo input $\langle M \rangle$. Infine, eseguiamo D su se stessa. Quindi, le macchine eseguono le seguenti azioni, dove l'ultima riga fornisce la contraddizione.

- H accetta $\langle M, w \rangle$ esattamente quando M accetta w .
- D rifiuta $\langle M \rangle$ esattamente quando M accetta $\langle M \rangle$.
- D rifiuta $\langle D \rangle$ esattamente quando D accetta $\langle D \rangle$.

Dove si usa la diagonalizzazione nella dimostrazione del [Teorema 4.11](#)? Essa diviene evidente quando si esaminano le tavole del comportamento delle TM H e D . In queste tavole indicizziamo le righe con tutte le TM M_1, M_2, M_3, \dots e le colonne con tutte le descrizioni $\langle M_1 \rangle, \langle M_2 \rangle, \langle M_3 \rangle, \dots$ delle TM. Le entrate dicono se la macchina di una determinata riga accetta l'input della colonna corrispondente. L'entrata accetta se la macchina accetta l'input, ma è vuota, se rifiuta o entra in loop su quell'input. Nella seguente figura abbiamo creato alcune voci per illustrare l'idea.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	accetta		accetta		
M_2	accetta	accetta	accetta	accetta	
M_3					
M_4	accetta	accetta			
\vdots			\vdots		

Figura 26: L'entrata i, j accetta se M_i accetta $\langle M_j \rangle$

Nella figura seguente, abbiamo aggiunto D alla Figura 4.20. Secondo la nostra assunzione, H è una TM così come D . Quindi deve comparire nella lista delle TM. Si noti che D calcola il contrario degli elementi sulla diagonale. La contraddizione si ottiene in corrispondenza del punto interrogativo, dove l'entrata dovrebbe essere l'opposto di se stessa.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots	$\langle D \rangle$	\dots
M_1	accetta	rifiuta	accetta	rifiuta		accetta	
M_2	accetta	accetta	accetta	accetta	...	accetta	...
M_3	rifiuta	rifiuta	rifiuta	rifiuta		rifiuta	
M_4	accetta	accetta	rifiuta	rifiuta		accetta	
\vdots							
D		rifiuta	rifiuta	accetta		?	
\vdots							

Figura 27: La contraddizione nella dimostrazione del Teorema 4.11

4.2.3 Un linguaggio non Turing-riconoscibile

Nella sezione precedente abbiamo dimostrato che un linguaggio - precisamente A_{TM} - non è decidibile. Ora mostriamo un linguaggio che non è neppure Turing-riconoscibile. Si noti che A_{TM} non basta allo copo, perché abbiamo dimostrato che A_{TM} è Turing-riconoscibile. Il teorema seguente mostra che, se un linguaggio ed il suo complemento sono entrambi Turing-riconoscibili, allora il linguaggio è decidibile. Quindi, per qualsiasi linguaggio non decidibile, o esso non è Turing-riconoscibile oppure il suo complemento non è Turing-riconoscibile. Ricordiamo che il complemento di un linguaggio è il linguaggio costituito da tutte le stringhe che non sono nel linguaggio. Diciamo che un linguaggio è **co-Turing-riconoscibile** se esso è il complemento di un linguaggio Turing-riconoscibile.

Teorema 4.22(oramai)

Un linguaggio è decidibile se e solo se è Turing-riconoscibile e co-Turing-riconoscibile.

In altre parole, un linguaggio è decidibile esattamente quando sia esso che il suo complemento sono Turing-riconoscibili.

DIMOSTRAZIONE. Abbiamo due direzioni da dimostrare. In primo luogo, se A è decifabile, possiamo facilmente vedere che sia A che il suo complemento \bar{A} sono Turing-riconoscibili. Qualsiasi linguaggio decidibile è Turing-riconoscibile, e il complemento di un linguaggio decidibile è anch'esso decidibile. Per la direzione inversa, se entrambi A e \bar{A} sono Turing-riconoscibili, indichiamo con M_1 il riconoscitore per A e con M_2 il riconoscitore per \bar{A} . La seguente macchina di Turing M è un decisore per A .

$M =$ "Su input w :

1. Esegue sia M_1 che M_2 su w in parallelo.
2. Se M_1 accetta, accetta; se M_2 accetta, rifiuta."

L'esecuzione di due macchine in parallelo significa che M ha due nastri, uno per simulare M_1 e l'altro per simulare M_2 . In questo caso M alterna la simulazione di un passo di M_1 con un passo di M_2 e continua così finché una delle due macchine accetta. Ora dimostriamo che M decide A . Ogni stringa w è in A o in \bar{A} . Pertanto una tra M_1 e M_2 accetta w . Poiché M si ferma ogni volta che M_1 accetta oppure M_2 accetta, allora M si ferma sempre quindi è un decisore. Inoltre, accetta tutte le stringhe in A e rifiuta tutte le stringhe che non sono in A . Quindi M è un decisore per A , e pertanto A è decidibile.

Corollario 4.23

Corollario 4.23

Il linguaggio A_{TM} non è Turing-riconoscibile.

DIMOSTRAZIONE. Sappiamo che A_{TM} è Turing-riconoscibile. Se anche il suo complemento $\overline{A_{TM}}$ fosse Turing-riconoscibile, allora A_{TM} sarebbe decidibile, il che è falso. Il [Teorema 4.11](#) ci dice che A_{TM} non è decidibile, quindi $\overline{A_{TM}}$ non è Turing-riconoscibile.

5 Riducibilità

In questo capitolo esamineremo vari altri problemi irrisolvibili. Nel far ciò introdurremo il metodo principale per dimostrare che alcuni problemi sono computazionalmente irrisolvibili. Tale metodo si chiama **riducibilità**.

Una **riduzione** è un modo di convertire un problema in un altro problema in modo tale che una soluzione al secondo problema può essere usata per risolvere il primo problema. La riducibilità coinvolge sempre due problemi, che noi chiamiamo A e B . Se A si riduce a B , possiamo usare una soluzione per B per risolvere A . Notate che la riducibilità non dice nulla circa la soluzione dei problemi A o B individualmente, ma soltanto qualcosa circa la risolubilità di A quando abbiamo una soluzione per B .

La riducibilità svolge un ruolo importante nella classificazione dei problemi in base alla decidibilità e, successivamente, anche nella teoria della complessità.

Quando A è riducibile a B , trovare la soluzione di A non può essere più difficile di risolvere B perché una soluzione per B offre una soluzione ad A . In termini di teoria della computabilità, se A è riducibile a B e B è decidibile, allora A è decidibile. Equivalentemente, se A è indecidibile e riducibile a B , B è indecidibile. Questa ultima parte è la chiave per dimostrare che alcuni problemi sono indecidibili.

In breve, il nostro metodo per dimostrare che un problema è indecidibile sarà quello di mostrare che qualche altro problema già noto per essere indecidibile si riduce ad esso.

5.1 Problemi indecidibili dalla teoria dei linguaggi

Abbiamo già stabilito l'indecidibilità di A_{TM} , il problema di determinare se una macchina di Turing accetta un dato input. Consideriamo ora un problema affine, $HALT_{TM}$, il problema di determinare se una macchina di Turing si ferma (accettando o rifiutando) su un dato input. Questo problema è generalmente noto come il **problema della fermata**. Usiamo l'indecidibilità di A_{TM} per dimostrare l'indecidibilità del problema della fermata riducendo A_{TM} a $HALT_{TM}$. Sia

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM e si ferma su input } w\}.$$

Teorema 5.1(oramai)

$HALT_{TM}$ è indecidibile.

IDEA. Questa dimostrazione è per assurdo. Assumiamo che $HALT_{TM}$ sia decidibile e usiamo questa assunzione per dimostrare che A_{TM} è decidibile, contraddicendo il [Teorema 4.11](#). L'idea chiave è quella di mostrare che A_{TM} è riducibile a $HALT_{TM}$.

Supponiamo di avere una TM R che decide $HALT_{TM}$. Usiamo quindi R per costruire una TM S che decide A_{TM} . Per intuire come possiamo costruire S , immaginate di essere S . Il vostro compito è decidere A_{TM} . Ricevete un input $\langle M, w \rangle$. Dovete dare in output *accetta* se M accetta w e *rifiuta* se M non accetta w . Provate a simulare M su w . Se accetta o rifiuta, fate lo stesso. Ma potreste non essere in grado di determinare se M è entrata in un ciclo, in questo caso la simulazione non terminerà. Questo non va bene perché voi siete un decisore e non vi è permesso ciclare. Quindi questa idea non funziona. Utilizziamo, invece, l'ipotesi che avete una TM R che decide $HALT_{TM}$. Con R , potete verificare se M si ferma su w . Se R indica che M non si ferma su w , rifiutate perché $\langle M, w \rangle \notin A_{TM}$. Tuttavia, se R indica che M si ferma su w , potete fare la simulazione senza alcun pericolo di ciclare. Così, se la TM R esistesse, potremmo decidere A_{TM} , ma sappiamo che A_{TM} è indecidibile. In virtù di questa contraddizione possiamo concludere che R non esiste. Pertanto $HALT_{TM}$ è indecidibile.

DIMOSTRAZIONE. Assumiamo al fine di ottenere una contraddizione che la TM R decide $HALT_{TM}$. Costruiamo la TM S per decidere A_{TM} , che opera come segue.

$S =$ "Su input $\langle M, w \rangle$, una codifica di una TM M ed una stringa w :

1. Esegue la TM R su input $\langle M, w \rangle$.
2. Se R rifiuta, rifiuta.
3. Se R accetta, simula M su w finché non si ferma.
4. Se M ha accettato, *accetta*; se M ha rifiutato, *rifiuta*."

Chiaramente, se R decide $HALT_{TM}$, allora S deve decidere A_{TM} . Poiché A_{TM} è indecidibile, $HALT_{TM}$ deve essere indecidibile.

Il [Teorema 5.1](#) illustra il nostro metodo per dimostrare che un problema è indecidibile. Questo metodo è comune alla maggior parte delle dimostrazioni di indecidibilità, tranne che per l'indecidibilità di A_{TM} stesso, che viene dimostrata direttamente utilizzando il metodo della diagonalizzazione.

Presentiamo ora vari altri teoremi e le loro dimostrazioni come ulteriori esempi dell'uso della riducibilità per dimostrare l'indecidibilità. Sia

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

Teorema 5.2(orale)

E_{TM} è indecidibile.

IDEA. Seguiamo lo schema adottato nel [Teorema 5.1](#). Assumiamo che E_{TM} è decidibile e mostriamo che A_{TM} è decidibile - una contraddizione. Sia R una TM che decide E_{TM} . Utilizziamo R per costruire la TM S che decide A_{TM} . Come funzionerà S quando riceve in input $\langle M, w \rangle$?

Un'idea è che S esegue R su input $\langle M \rangle$ e vede se R accetta. Se lo fa, sappiamo che $L(M) = \emptyset$ e quindi M non accetta w . Ma, se R rifiuta $\langle M \rangle$, tutto quello che sappiamo è che $L(M) \neq \emptyset$ e di conseguenza, che M accetta qualche stringa - ma ancora non sappiamo se M accetta la stringa w in particolare. Quindi abbiamo bisogno di utilizzare un'idea diversa. Invece di eseguire R su $\langle M \rangle$, eseguiamo R su una modifica di $\langle M \rangle$. Modifichiamo $\langle M \rangle$ così da garantire che M rifiuta tutte le stringhe tranne w , ma su input w , funziona come al solito. A questo punto, utilizziamo R per determinare se la macchina modificata riconosce il linguaggio vuoto. L'unica stringa che adesso la macchina può accettare è w , per cui il suo linguaggio sarà non vuoto se accetta w . Se R accetta quando riceve in input la descrizione della macchina modificata, sappiamo che la macchina modificata non accetta nulla e che M non accetta w .

DIMOSTRAZIONE. Descriviamo la macchina modificata già descritta nell'idea di dimostrazione usando la nostra notazione standard. Chiamiamola M_1 .

$M_1 =$ "Su input x :

1. Se $x \neq w$, rifiuta.
2. Se $x = w$, esegui M su w e *accetta* se M accetta."

Questa macchina ha la stringa w come parte della sua descrizione. Essa verifica se $x = w$ nel modo ovvio, attraverso la scansione dell'input e confrontandolo carattere per carattere con w per determinare se coincidono. Mettendo insieme tutto questo, assumiamo che la TM R decide E_{TM} e costruiamo la TM S che decide A_{TM} come segue.

$S =$ "Su input $\langle M, w \rangle$, una codifica di una TM M ed una stringa w :

1. Usa la descrizione di M e w per costruire la TM M_1 descritta sopra.
2. Esegue la TM R su input $\langle M_1 \rangle$.

3. Se R accetta, *rifiuta*; se R rifiuta, *accetta*."

Notate che S deve essere effettivamente in grado di calcolare una descrizione di M_1 da una descrizione di M e w . Può farlo, perché ha bisogno solo di aggiungere ad M alcuni stati in più che svolgono il test $x = w$. Se R fosse un decisore per E_{TM} , allora S sarebbe un decisore per A_{TM} . Un decisore per A_{TM} non può esistere, quindi E_{TM} non può essere deciso.

Un altro interessante problema computazionale che concerne le macchine di Turing è quello di determinare se una data macchina di Turing riconosce un linguaggio che può essere riconosciuto anche da un modello di calcolo più semplice. Ad esempio, sia $REGULAR_{TM}$ il problema di determinare se una macchina di Turing ha un automa finito equivalente. Questo problema equivale a determinarre se la macchina di Turing riconosce un linguaggio regolare. Sia

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ è una TM e } L(M) \text{ è un linguaggio regolare}\}.$$

Teorema 5.3

$REGULAR_{TM}$ è indecidibile.

IDEA. Come al solito per i teoremi sull'indecidibilità, questa dimostrazione consiste in una riduzione da A_{TM} . Assumiamo che $REGULAR_{TM}$ sia deciso da una RM R e utilizziamo questa assunzione per costruire una TM S che decide A_{TM} . Questa volta risulta meno ovvio come utilizzare la capacità di R di assistere S nel suo compito. Tuttavia siamo in grado di farlo. L'idea è che S prenda il suo input $\langle M, w \rangle$ e modifichi M in modo che la risultante TM riconosca un linguaggio regolare se e solo se M accetta w . Chiamiamo M_2 la macchina così modificata. Progettiamo M_2 in modo che riconosca il linguaggio non regolare $\{0^n 1^n \mid n \geq 0\}$ se M non accetta w , e riconosca il linguaggio regolare Σ^* se M accetta w . Dobbiamo specificare come S può costruire una tale M_2 da M e w . Qui, M_2 accetta automaticamente tutte le stringhe in $\{0^n 1^n \mid n \geq 0\}$. Inoltre, se M accetta w , M_2 accetta tutte le altre stringhe.

Notate che la TM M_2 non è costruita con lo scopo di essere eseguita su qualche input - un malinteso comune. Costruiamo M_2 al solo scopo di dare in input la sua descrizione al decisore per $REGULAR_{TM}$ che abbiamo assunto esistere. Quando tale decisore dà la sua risposta, possiamo usarla per rispondere se M accetta w o meno. Quindi possiamo decidere A_{TM} , una contraddizione.

DIMOSTRAZIONE. Definiamo R come una TM che decide $REGULAR_{TM}$ e costruiamo una TM S che decide A_{TM} . Allora S funziona come segue.

$S =$ "Su input $\langle M, w \rangle$, una codifica di una TM M ed una stringa w :

1. Costruisce la TM M_2 come segue.
 $M_2 =$ "Sull'input x :
 - (a) Se x ha la forma $0^n 1^n$ per qualche $n \geq 0$, *accetta*.
 - (b) Se x non ha tale forma, esegui M su w e *accetta* se M accetta."
2. Esegue R su input $\langle M_2 \rangle$.
3. Se R accetta, *accetta*; se R rifiuta, *rifiuta*."

Con prove simili si può dimostrare che i problemi di testare se il linguaggio di una macchina di Turing è un linguaggio context-free, un linguaggio decidibile o anche un linguaggio dinito sono indecidibili. Infatti, un risultato generale, chiamato teorema di Rice,

Rice's Theorem

Thm.: Let P be a language consisting of TM descriptions such that

1. P is nontrivial (i.e., it contains some, but not all, TM descriptions); and
2. P is defined by some property of the TM's language.

Then, P is undecidable.

Proof

By contradiction, let R_P be a decider for P ; we now show how to reduce A_{TM} to P .

Let T_0 be a TM that always rejects, so $L(T_0) = \emptyset$.

W.l.o.g., assume that $(T_0) \notin P$ (if not, proceed with \bar{P} instead of P , since a decider for P yields one for \bar{P}).

Because P is not trivial, there exists a TM T with $(T) \in P$. Given M and w , consider the TM $S_{T,M,w}$:

On input x :

- (i) Simulate M on w
- (ii) If it halts and rejects, reject
- (iii) If it accepts, simulate T on x . If it accepts, accept

If M accepts w ,

$$\text{the language of } S_{T,M,w} \text{ is the same as } T\text{'s: } L(S_{T,M,w}) = \begin{cases} L(T) & \text{if } M \text{ accepts } w \\ L(T_0) & \text{otherwise.} \end{cases}$$

We now show how to decide A_{TM} by using R_P 's ability to distinguish between T_0 and T :

On input $\langle M, w \rangle$:

- Run R_P with input $(S_{T,M,w})$. If it accepts, accept; otherwise reject

Therefore, M accepts w iff $(S_{T,M,w}) \in P$. Q.E.D.

Figura 28: Teorema di Rice

afferma che determinare una *qualsiasi proprietà* dei linguaggi riconosciuti da macchine di Turing è indecidibile.

Finora, la nostra strategia per dimostrare l'indecidibilità di linguaggi prevede una riduzione da A_{TM} . A volte la riduzione da qualche altro linguaggio indecidibile è più conveniente per dimostrare che certi linguaggi sono indecidibili. Il teorema 5.4 mostra che verificare l'equivalenza di due macchine di Turing è un problema indecidibile. Potremmo provarlo con una riduzione da A_{TM} , ma abbiamo l'occasione di dare un esempio di una prova di incedibilità mediante una riduzione da E_{TM} . Sia

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ e } M_2 \text{ sono TM e } L(M_1) = L(M_2)\}.$$

Teorema 5.4(orale)

EQ_{TM} è indecidibile.

IDEA. Mostriamo che se EQ_{TM} fosse decidibile, allora E_{TM} sarebbe decidibile eseguendo una riduzione da E_{TM} a EQ_{TM} . L'idea è semplice. E_{TM} è il problema di determinare se il linguaggio di una TM è vuoto. EQ_{TM} è il problema di determinare se i linguaggi delle due TM sono uguali. Se uno di questi linguaggi è \emptyset , ci troviamo con il problema di determinare se il linguaggio dell'altra macchina è vuoto, cioè il problema E_{TM} . Quindi, in un certo senso, il problema E_{TM} è un caso speciale del problema EQ_{TM} dove una delle macchine è fissata per riconoscere il linguaggio vuoto. Questa idea rende la riduzione più facile.

DIMOSTRAZIONE. Consideriamo una TM R che decide EQ_{TM} e costruiamo una TM S che decide E_{TM} come segue.

$S =$ "Su input $\langle M \rangle$, dove M è una TM:

1. Esegue R su input $\langle M, M_1 \rangle$, dove M_1 è una TM che rifiuta ogni input.
2. Se R accetta, *accetta*; se R rifiuta, *rifiuta*."

Se R decide EQ_{TM} , allora S decide E_{TM} . Ma il Teorema 5.2 ci dice che E_{TM} è indecidibile, quindi EQ_{TM} deve essere indecidibile.

5.2 Riducibilità mediante funzione

Abbiamo mostrato come utilizzare la tecnica della riducibilità per dimostrare che alcuni problemi sono indecidibili. In questa sezione formalizziamo il concetto di riducibilità. Questo ci permette di usare la riducibilità in modo più raffinato, come per esempio per dimostrare che alcuni linguaggi non sono Turing-riconoscibili e per applicazioni in teoria della complessità. La nozione di ridurre un problema ad un altro può essere definita formalmente in vari modi. La scelta di quale usare dipende dall'applicazione. La nostra scelta ricade su un tipo semplice di riducibilità chiamato *riducibilità mediante funzione*¹.

¹E' chiamata riducibilità multi-a-uno in altri libri di testo

Informalmente, essere in grado di ridurre il problema A al problema B utilizzando una riduzione mediante funzione significa che esiste una funzione calcolabile che trasforma istanze del problema A in istanze del problema B . Se abbiamo una tale funzione, detta *riduzione*, siamo in grado di risolvere A risolvendo istanze di B . Il motivo risiede nel fatto che una qualsiasi istanza di A può essere risolta utilizzando prima la riduzione per convertirla in un'istanza di B e poi risolvere tale istanza di B . Nel breve daremo una definizione precisa di riducibilità mediante funzione.

5.2.1 Funzioni calcolabili

Una macchina di Turing calcola una funzione iniziando con l'input della funzione sul nastro e terminando con l'output della funzione sul nastro.

Definizione 5.17

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

ESEMPIO 5.18

Tutte le operazioni aritmetiche usuali su numeri interi sono funzioni calcolabili. Per esempio, possiamo progettare una macchina che prende in input

$\langle m, n \rangle$ e restituisce $m + n$, la somma di m ed n . Non diamo dettagli qui, li lasciamo come esercizio.

ESEMPIO 5.19

Le funzioni calcolabili possono essere trasformazioni di descrizioni di macchine. Per esempio, una funzione calcolabile f prende in input w e restituisce la descrizione di una macchina di Turing $\langle M' \rangle$ se $w = \langle M \rangle$ è la codifica di una macchina di Turing M . La macchina M' è una macchina che riconosce lo stesso linguaggio di M , ma non tenta mai di muovere la testina oltre il limite sinistro del suo nastro. La funzione f realizza questo compito con l'aggiunta di vari stati nella descrizione di M . La funzione restituisce ϵ se w non è una codifica legale di una macchina di Turing.

Definizione formale di riducibilità mediante funzione

Definiamo ora la riducibilità mediante funzione. Come al solito, rappresentiamo problemi computazionali mediante linguaggi.

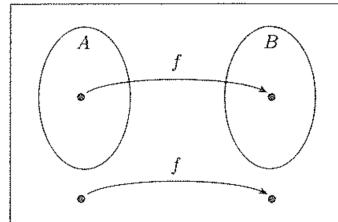
Definizione 5.120

Un linguaggio A si dice **riducibile mediante funzione** a un linguaggio B , scritto $A \leq_m B$, se esiste una funzione calcolabile $f : \Sigma^* \rightarrow \Sigma^*$ tale che per ogni w ,

$$w \in A \text{ sse } f(w) \in B.$$

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

La figura seguente illustra la riducibilità mediante funzione.



Una riduzione mediante funzione da A a B fornisce un modo per convertire problemi di appartenenza ad A in problemi di appartenenza a B . Per verificare se $w \in A$, usiamo la riduzione f per mappare w in

$f(w)$ e verifichiamo se $f(w) \in B$ o meno. Il termine *riduzione mediante funzione* deriva dalla funzione che viene utilizzata per la riduzione.

Se un problema è riducibile mediante funzione ad un secondo problema già precedentemente risolto, possiamo ottenere da questo una soluzione al problema originale.

Teorema 5.22

Sia $A \leq_m B$ e B è decidibile, allora A è decidibile.

DIMOSTRAZIONE. Siano M il decisore per B ed f la riduzione da A a B . Descriviamo un decisore N per A come segue.

$N =$ "Su input w :

1. Computa $f(w)$.
2. Esegue M su input $f(w)$ e restituisce lo stesso output di M ."

Ovviamente, se $w \in A$, allora $f(w) \in B$ perché f è una riduzione da A a B . Quindi M accetta $f(w)$ ogni volta che $w \in A$. Quindi, N svolge il compito desiderato.

Il seguente corollario al [Teorema 5.22](#) rappresenta il nostro strumento principale nelle dimostrazioni di indecidibilità.

Corollario 5.23

Se $A \leq_m B$ e A non è decidibile, allora B non è decidibile.

Ora rivediamo alcune delle nostre prove precedenti che hanno utilizzato il metodo della riducibilità per ottenere esempi di riduzione mediante funzione.

ESEMPIO 5.24

Nel Teorema 5.1 abbiamo usato una riduzione da A_{TM} per dimostrare che HALT_{TM} è indecidibile. Questa riduzione ha mostrato come un decisore per HALT_{TM} potrebbe essere utilizzato per ottenere un decisore per A_{TM} . Possiamo fornire una riduzione mediante funzione da A_{TM} a HALT_{TM} come segue. Per farlo, dobbiamo fornire una funzione calcolabile f che prende in input (M, w) e restituisce in output (M', w') , dove

$$(M, w) \in A_{\text{TM}} \text{ se e solo se } (M', w') \in \text{HALT}_{\text{TM}}.$$

$L(M_1)$ è non vuoto, quindi f è una riduzione mediante funzione da A_{TM} a $\overline{A_{\text{TM}}}$. Essa ancora dimostra che $\overline{A_{\text{TM}}}$ è indecidibile in quanto la decidibilità non è influenzata dalla complementazione, ma non fornisce una riduzione mediante funzione da A_{TM} a $\overline{A_{\text{TM}}}$. Infatti, una tale riduzione non esiste, come vi viene chiesto di dimostrare nell'Esercizio 5.5.

La sensitività delle riduzioni mediante funzione alla complementazione è importante nell'uso della riducibilità per dimostrare la non-riconoscibilità di alcuni linguaggi. Possiamo anche utilizzare la riducibilità mediante funzione per dimostrare che alcuni problemi non sono Turing-riconoscibili. Il seguente teorema è analogo al Teorema 5.22.

La seguente macchina F calcola una riduzione f .

$F =$ "Su input (M, w) :

1. Costruisce la seguente macchina M' .
 $M' =$ "Su input x :
 1. Esegue M su x .
 2. Se M accetta, *accetta*.
 3. Se M rifiuta, *cicla*."
2. Output (M', w) ."

Qui sorge un problema minore che riguarda le stringhe in input che non sono della forma corretta. Se la TM F determina che il suo input non è della forma corretta, in accordo a quanto specificato dalla linea dell'input "Su input (M, w) " e, quindi, che l'input non è un A_{TM} , la TM F da un output una stringa nulla in HALT_{TM} . Potremmo usare una simile strategia in HALT_{TM} , ma generalmente dobbiamo usare una macchina di Turing che prende una riduzione da A a B , assumendo che gli input che non sono della forma corretta sono mappati in stringhe al di fuori di B .

ESEMPIO 5.25

La dimostrazione dell'indecidibilità del Problema della Corrispondenza di Post nel Teorema 5.15 contiene due riduzioni mediante funzione. Prima, dimostra che $A_{\text{TM}} \leq_m MPCP$ e poi mostra che $MPCP \leq_m PCP$. In entrambi i casi siamo in grado di ottenere facilmente l'effettiva funzione di riduzione e mostrare che si tratta di una riduzione mediante funzione. Come mostra l'Esercizio 5.6, la riducibilità mediante funzione è transitiva, per cui queste due riduzioni insieme implicano $A_{\text{TM}} \leq_m PCP$.

ESEMPIO 5.26

Una riduzione mediante funzione da E_{TM} a EQ_{TM} si trova nella dimostrazione del Teorema 5.4. In questo caso la riduzione f mappa l'input (M) nell'output (M, M_1) , dove M_1 è la macchina che rifiuta ogni input.

ESEMPIO 5.27

La dimostrazione del Teorema 5.2, mostrando che E_{TM} è indecidibile, illustra la differenza tra la nozione formale di riducibilità mediante funzione che abbiamo studiato in precedenza e la nozione informale di riducibilità che abbiamo usato in precedenza in questo capitolo. La dimostrazione prova che E_{TM} è indecidibile riducendo A_{TM} ad esso. Vediamo se siamo in grado di riceverne questa riduzione come una riduzione mediante funzione.

Dalla riduzione originale possiamo facilmente costruire una funzione f che prende in input (M, w) e produce un output (M_1) , dove M_1 è la macchina di Turing descritta in quella dimostrazione. Ma M accetta w se

Teorema 5.28

Teorema 5.28

Se $A \leq_m B$ e B è Turing-riconoscibile, allora A è Turing-riconoscibile.

Corollario 5.29

Corollario 5.29

Se $A \leq_m B$ e A non è Turing-riconoscibile, allora B non è Turing-riconoscibile.

In una tipica applicazione di questo corollario, supponiamo che A sia $\overline{A_{TM}}$, il complemento di A_{TM} . Dal corollario Corollario 4.23, sappiamo che non è Turing-riconoscibile. La definizione di riduzione mediante funzione implica che $A \leq_m B$ ha lo stesso significato di $\overline{A} \leq_m \overline{B}$. Per dimostrare che B non è riconoscibile possiamo mostrare che $A_{TM} \leq_m B$. Possiamo anche usare la riducibilità mediante funzione per dimostrare che alcuni problemi non sono né Turing-riconoscibili né co-Turing-riconoscibili, come nel seguente teorema.

Teorema 5.30

EQ_{TM} non è né Turing-riconoscibile né co-Turing-riconoscibile.

DIMOSTRAZIONE. Come primo passo, dimostriamo che EQ_{TM} non è Turing-riconoscibile. Lo facciamo dimostrando che A_{TM} è riducibile a $\overline{EQ_{TM}}$. La funzione di riduzione f opera come segue.

$F =$ "Su input $\langle M, w \rangle$, dove M è una TM e w una stringa:

1. Costruisce le seguenti due macchine, M_1 e M_2 .

$M_1 =$ "Su ogni input:

- (a) *Rifiuta.*"

$M_2 =$ "Su ogni input:

- (a) Esegue M su w . Se accetta, *accetta.*"

2. Restituisce $\langle M_1, M_2 \rangle$."

Qui, M_1 non accetta alcuna stringa. Se M accetta w , M_2 accetta qualsiasi stringa, quindi le due macchine non sono equivalenti. Viceversa, se M non accetta w , M_2 accetta solo la stringa vuota, quindi le due macchine sono equivalenti. Perciò f riduce A_{TM} a $\overline{EQ_{TM}}$, come desiderato.

Per dimostrare che $\overline{EQ_{TM}}$ non è Turing-riconoscibile diamo una riduzione da A_{TM} al complemento di $\overline{EQ_{TM}}$ cioè, EQ_{TM} . In questo modo dimostriamo che $A_{TM} \leq_m EQ_{TM}$. La seguente TM G calcola la funzione di riduzione g .

$G =$ "Su input $\langle M, w \rangle$, dove M è una TM e w una stringa:

1. Costruisce le seguenti due macchine, M_1 e M_2 .

$M_1 =$ "Su ogni input:

- (a) *Accetta.*"

M_2 = "Su ogni input:

- (a) Esegue M su w . Se accetta, *accetta*."
2. Restituisce $\langle M_1, M_2 \rangle$."

La sola differenza tra f e g si trova nella macchina M_1 . In f , la macchina M_1 rifiuta sempre, mentre in g accetta sempre. Sia in f che in g , M accetta w sse M_2 accetta ogni input. In g , M accetta w sse M_1 ed M_2 sono equivalenti. Questo è il motivo per cui g è una riduzione da A_{TM} a EQ_{TM} .

6

Serve per mantenere coerenza con il numero delle dimostrazioni da sapere indicate nel pdf (7.1, ecc...).

7 Complessità di tempo

Anche quando un problema è decidibile, e quindi in linea di principio computazionalmente risolvibile, può non essere risolvibile in pratica se la soluzione richiede una quantità eccessiva di tempo o di memoria.

Il nostro obiettivo in questo capitolo è presentare le basi della teoria della complessità di tempo.

7.1 Misure di complessità

Iniziamo con un esempio. Consideriamo il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$. Ovviamente, A è decidibile. Di quanto tempo necessita una macchina di Turing a nastro singolo per decidere A ?

Esaminiamo la seguente TM M_1 per A . Diamo la descrizione della macchina di Turing a basso livello, includendo l'esatto movimento della testina sul nastro in modo che possiamo contare il numero di passi che M_1 effettua quando lavora.

M_1 = "Su input w :

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete se il nastro contiene almeno un 0 e un 1:
3. Scandisce il nastro, cancellando uno 0 e un 1.
4. Se rimane almeno uno 0 dopo che ogni simbolo 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni simbolo 0 è stato cancellato, *rifiuta*. Altrimenti, se non rimanono né simboli 0 né simboli 1, *accetta*."

Analizzeremo l'algoritmo della TM M_1 che decide A per determinare la quantità di tempo che impiega. Come prima cosa, introduciamo la terminologia e la notazione necessaria. Il numero di passi che utilizza un algoritmo su un particolare input può dipendere da diversi parametri. Per semplicità si calcola il tempo di esecuzione di un algoritmo semplicemente in funzione della lunghezza della stringa che rappresenta l'input e non si considerano eventuali altri parametri. Nell'analisi del **caso peggiore**, che noi considereremo, si valuta il tempo di esecuzione massimo tra tutti gli input di una determinata lunghezza. Nell'analisi del **caso medio**, si considera la media dei tempi di esecuzione di tutti gli input di una determinata lunghezza.

Definizione 7.1

Definizione 7.1

Sia M una macchina di Turing deterministica che si ferma su tutti gli input. Il **tempo di esecuzione** o la **complessità di tempo** di M è la funzione $f : \mathbb{N} \rightarrow \mathbb{N}$, dove $f(n)$ è il massimo numero di passi che M impiega su un input di lunghezza n . Se $f(n)$ è il tempo di esecuzione di M , diciamo che M ha tempo di esecuzione $f(n)$ e che M è una macchina di Turing di tempo $f(n)$. Abitualmente usiamo n per rappresentare la lunghezza dell'input.

7.1.1 Notazione O-grande ed o-piccola

Poichè il tempo esatto di esecuzione di un algoritmo è spesso un'espressione complessa, solitamente ci limitiamo ad ottenerne una stima. L'utile metodo di stima, detto **analisi asintotica**, permette di valutare il tempo di esecuzione dell'algoritmo quando viene eseguito su grandi input. Lo facciamo prendendo in considerazione solo il termine di ordine maggiore dell'espressione del tempo di esecuzione dell'algoritmo, trascurando sia il coefficiente di tale termine, che tutti i termini di ordine inferiore, perchè il termine di ordine più alto domina gli altri termini quando l'input è grande.

Ad esempio, la funzione $f(n) = 6n^3 + 2n^2 + 20n + 45$ ha quattro termini ed il termine di ordine maggiore è $6n^3$. Trascurando il coefficiente 6, diciamo che f è asintoticamente al più n^3 . La **notazione asintotica** o notazione **O-grande** per descrivere questo rapporto è $f(n) = O(n^3)$.

Formalizziamo tale nozione nella seguente definizione. Sia \mathbb{R}^+ l'insieme dei numeri reali non negativi.

Definizione 7.2

Definizione 7.2

Siano f e g funzioni da $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Si dice che $f(n) = O(g(n))$ se esistono interi positivi c e n_0 tali che per ogni $n \geq n_0$,

$$f(n) \leq cg(n).$$

Quando $f(n) = O(g(n))$, diciamo che $g(n)$ è un **limite superiore** per $f(n)$, o più precisamente, che $g(n)$ è un limite superiore asintotico per $f(n)$, per sottolineare che stiamo ignorando le costanti.

Intuitivamente, $f(n) = O(g(n))$ significa che f è minore o uguale a g se trascuriamo differenze fino ad un fattore costante. Potete pensare ad O come rappresentazione implicita di una costante. In pratica la maggior parte delle funzioni f che potete incontrare hanno un termine di ordine più alto h , chiaramente individuabile. In tal caso si scrive $f(n) = O(g(n))$, dove g è h senza il suo coefficiente.

ESEMPIO 7.3

Sia $f_1(n)$ la funzione $5n^3 + 2n^2 + 22n + 6$. Quindi, selezionando il termine di ordine maggiore $5n^3$ e trascurando il suo coefficiente 5, si ottiene $f_1(n) = O(n^3)$. Verifichiamo che questo risultato soddisfa la definizione formale. Lo facciamo ponendo c pari a 6 e n_0 a 10. Quindi $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ per ogni $n \geq 10$. Inoltre, $f_1(n) = O(n^4)$ perché n^4 è più grande di n^3 e quindi è un limite superiore asintotico per f_1 . Tuttavia, $f_1(n)$ non è $O(n^2)$. Indipendentemente dai valori che assegniamo a c e n_0 , la definizione non viene soddisfatta in questo caso.

ESEMPIO 7.4

L' O -grande interagisce con i logaritmi in una maniera particolare. Di solito quando si usano i logaritmi dobbiamo specificarne la base, come per $x =$

$\log_2 n$. La base 2 qui indica che questa uguaglianza equivale all'uguaglianza $2^x = n$. Cambiando il valore della base b cambia il valore di $\log_b n$ di un fattore costante, dovuto all'uguaglianza $\log_b n = \log_2 n / \log_2 b$. Quindi, quando scriviamo $f(n) = O(\log n)$, non è più necessario specificare la base perché stiamo comunque eliminando i fattori costanti. Sia $f_2(n)$ la funzione $3n \log_2 n + 5n \log_2 \log_2 n + 2$. In questo caso abbiamo $f_2(n) = O(n \log n)$ perché $\log n$ domina $\log \log n$.

La notazione O -grande appare anche in espressioni aritmetiche, come l'espressione $f(n) = O(n^2) + O(n)$. In tal caso ciascuna occorrenza del simbolo O rappresenta un diverso fattore costante nascosto. Poiché il termine $O(n^2)$ domina il termine $O(n)$, l'espressione è equivalente a $f(n) = O(n^2)$. Quando il simbolo O si presenta all'esponente, come nell'espressione $f(n) = 2^{O(n)}$, vale la stessa idea. Questa espressione rappresenta un limite superiore per 2^n per qualche costante c . A volte si trova l'espressione $f(n) = 2^{O(\log n)}$. Utilizzando l'uguaglianza $n = 2^{\log_2 n}$ e quindi $n^c = 2^{c \log_2 n}$, otteniamo che $2^{O(\log n)}$ rappresenta un limite superiore per n^c per qualche c . L'espressione $n^{O(1)}$ rappresenta la stessa limitazione in maniera diversa, in quanto l'espressione $O(1)$ rappresenta un valore che non è mai maggiore di una costante fissata.

Spesso si ottengono dei limiti della forma n^c per c maggiore di 0. Tali limiti sono chiamati **limiti polinomiali**. Limiti del tipo $2^{O(n^\delta)}$ sono chiamati **limiti esponenziali** quando δ è un numero reale maggiore di 0.

La notazione O -grande ha una controparte chiamata **notazione o-piccolo**. La notazione O -grande dice che una funzione è asintoticamente *non più grande* di un'altra. Per dire che una funzione è asintoticamente *più piccola* di un'altra, usiamo la notazione *o-piccolo*. La differenza tra le notazioni O -grande e o-piccolo è analoga alla differenza tra \leq e $<$.

Definizione 7.5

Siano f e g funzioni da $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Diciamo che $f(n) = o(g(n))$ se

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

In altri termini, $f(n) = o(g(n))$ significa che per ogni numero reale $c > 0$, esiste un intero positivo n_0 tale che per ogni $n \geq n_0$,

$$f(n) < cg(n).$$

ESEMPIO 7.6

Le seguenti uguaglianze sono semplici da verificare.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

Tuttavia, $f(n)$ non è mai $o(f(n))$.

7.1.2 Analisi degli algoritmi

Analizziamo l'algoritmo M_1 dato per il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$. Riscriviamo qui l'algoritmo per comodità.

$M_1 =$ "Su input w :

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete se il nastro contiene almeno un 0 e un 1:
3. Scandisce il nastro, cancellando uno 0 e un 1.
4. Se rimane almeno uno 0 dopo che ogni simbolo 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni simbolo 0 è stato cancellato, *rifiuta*. Altrimenti, se non rimanono né simboli 0 né simboli 1, *accetta*."

Per analizzare M_1 , consideriamo ciascuna delle sue quattro fasi separatamente. Nella fase 1, la macchina scansiona il nastro per verificare che l'input è del tipo $0^* 1^*$. Tale operazione di scansione usa n passi. Come abbiamo accennato in precedenza, di solito utilizziamo n per rappresentare la lunghezza dell'input. Per riposizionare la testina all'estremità sinistra del nastro utilizza ulteriori n passi. Per cui il totale

di passi utilizzati in questa fase è $2n$ passi. Nella notazione O-grande diciamo che questa fase usa $O(n)$ passi. Si noti che non abbiamo fatto menzione del riposizionamento della testina del nastro nella descrizione della macchina. L'utilizzo della notazione asintotica ci permette di omettere quei dettagli della descrizione della macchina che influenzano il tempo di esecuzione al più di un fattore costante.

Nelle fasi 2 e 3, la macchina esegue ripetutamente la scansione del nastro e cancella uno 0 e un 1 ad ogni scansione. Ogni scansione utilizza $O(n)$ passi. Poichè ogni scansione elimina due simboli, possono verificarsi al più $n/2$ scansioni. Così il tempo totale impiegato dalle fasi 2 e 3 è di $(n/2)O(n) = O(n^2)$ passi.

Nella fase 4 la macchina fa una singola scansione per decidere se accettare o rifiutare. Il tempo impiegato in questa fase è $O(n)$ passi.

Quindi il tempo totale di M_1 su un input di lunghezza n è $O(n) + O(n^2) + O(n)$, ovvero $O(n^2)$. In altre parole, il tempo di esecuzione è $O(n^2)$, il che completa l'analisi del tempo di esecuzione di questa macchina.

Vogliamo ora stabilire una notazione per classificare i linguaggi in base alla loro necessità di tempo.

Definizione 7.7

Definizione 7.7

Sia $t : \mathbb{N} \rightarrow \mathbb{R}^+$ una funzione. La **classe di complessità di tempo** $\text{TIME}(t(n))$, è definita come l'insieme di tutti i linguaggi che sono decisi da una macchina di Turing in tempo $O(t(n))$.

Ricordiamo il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$. L'analisi precedente mostra che $A \in \text{TIME}(n^2)$ perché A è deciso dalla macchina di Turing M_1 in tempo $O(n^2)$.

Esiste una macchina di turing che decide A in modo asintoticamente più veloce?

In altri termini, risulta A in $\text{TIME}(t(n))$ per $t(n) = o(n^2)$? Possiamo migliorare il tempo di esecuzione, cancellando due simboli 0 e due simboli 1 ad ogni scansione invece di uno solamente, perché questo riduce il numero di scansioni della metà. Ma ciò migliora il tempo di esecuzione solo per un fattore 2 e non influenza il tempo di esecuzione asintotico. La seguente macchina M_2 , utilizza un metodo differente per decidere A asintoticamente più velocemente. Essa mostra che $A \in \text{TIME}(n \log n)$.

$M_2 =$ "Su input w :

1. Scandisce il nastro e *rifiuta* se trova uno 0 a destra di un 1.
2. Ripete se il nastro contiene almeno un 0 e un 1:
3. Scandisce il nastro, controllando se il numero totale di simboli 0 e di 1 rimasti è pari o dispari. Se è dispari, *rifiuta*.
4. Scandisce nuovamente il nastro, cancellando prima ogni secondo 0 a partire dal primo 0, poi cancellando ogni secondo 1 a partire dal primo 1.
5. Se nessuno 0 e nessun 1 rimangono sul nastro, *accetta*. Altrimenti, *rifiuta*."

Prima di analizzare M_2 , cerchiamo di verificare se effettivamente decide A . In ogni scansione eseguita nella fase 4, il numero totale di 0 rimanenti è ridotto della metà e ogni eventuale resto viene scartato. Quindi, se abbiamo iniziato con 13 simboli 0, dopo una prima esecuzione della fase 4 rimangono solo 6 simboli 0. Dopo le successive esecuzioni di questa fase ne restano 3, 1 e infine 0. Questa fase ha lo stesso effetto sul numero di simboli 1. Esaminiamo ora la parità del numero di simboli 0 e 1 ad ogni esecuzione della fase 3. Si consideri ancora l'inizio con 13 simboli 0 e 13 simboli 1. La prima esecuzione della fase 3 trova un numero dispari di simboli 0 (perchè 13 è un numero dispari) ed un numero dispari di simboli 1. Nelle successive esecuzioni si ha un numero pari (6), poi un numero dispari (3), e un numero dispari (1). Non eseguiamo questa fase su 0 simboli 0 o 0 simboli 1 in accordo alla condizione specificata nella fase

2. Per la sequenza di parità trovata (dispari, pari, dispari, dispari), se sostituiamo pari con 0 e dispari con 1 e poi invertiamo la sequenza, otteniamo 1101, la rappresentazione binaria di 13, ossia il numero di simboli 0 e 1 iniziale. La sequenza delle parità fornisce sempre l'inverso della rappresentazione binaria.

Quando la fase 3 controlla che il numero totale di 0 e 1 rimanenti è pari, in realtà sta controllando la coerenza della parità del numero di 0 con la parità del numero di 1. Se le parità corrispondono, le rappresentazioni binarie dei numeri di 0 e di 1 corrispondono, e quindi i due numeri sono uguali.

Per analizzare il tempo di esecuzione di M_2 , per prima cosa osserviamo che ogni fase impiega un tempo $O(n)$. Determiniamo poi il numero di volte in cui ognuna viene eseguita. Le fasi 1 e 5 vengono eseguite una volta, impiegando un tempo totale di $O(n)$. La fase 4 scarta almeno metà dei simboli 0 e 1 ogni volta che viene eseguita, quindi si verificano al massimo $1 + \log_2 n$ iterazioni del ciclo prima di averli cancellati tutti. Così il tempo totale delle fasi 2,3, e 4 è $(1 + \log_2 n)O(n)$, o $O(n \log n)$. Il tempo di esecuzione di M_2 è $O(n) + O(n \log n) = O(n \log n)$.

In precedenza abbiamo dimostrato che $A \in TIME(n^2)$, ma ora abbiamo ottenuto un limite migliore - precisamente, $A \in TIME(n \log n)$. Questo risultato non può essere ulteriormente migliorato su macchine di Turing a singolo nastro. Infatti, ogni linguaggio che può essere deciso in tempo $o(n \log n)$ su una macchina di Turing a nastro singolo è regolare.

Possiamo decidere il linguaggio A in tempo $O(n)$ (chiamato anche tempo lineare) se la macchina di Turing ha un secondo nastro. La seguente TM M_3 a due nastri decide A in tempo lineare. La macchina M_3 opera diversamente dalle macchine precedenti per A . Semplicemente copia tutti i simboli 0 sul suo secondo nastro e poi li accoppia con gli 1.

M_3 = "Su input w :

1. Scandisce il nastro 1 e *rifiuta* se trova uno 0 a destra di un 1.
2. Scandisce i simboli 0 sul nastro 1 fino al primo 1. Contemporaneamente, copia ogni 0 sul nastro 2.
3. Scandisce i simboli 1 sul nastro 1 fino alla fine dell'input.
Per ogni 1 letto sul nastro 1, cancella uno 0 sul nastro 2.
Se ogni 0 è stato cancellato prima di aver letto tutti gli 1, *rifiuta*.
4. Se tutti gli 0 sono stati cancellati *accetta*. Se rimane qualche 0, *rifiuta*."

Questa macchina è semplice da analizzare. Ciascuna delle quattro fasi utilizza $O(n)$ passi, in modo che il tempo di esecuzione complessivo risulta $O(n)$ e quindi lineare. Si noti che questo tempo di esecuzione è il migliore possibile perché n passi sono necessari semplicemente per leggere l'input.

Riassumiamo quello che abbiamo dimostrato circa la complessità temporale di A , la quantità di tempo necessaria per decidere A . Abbiamo progettato una TM M_1 a nastro singolo che decide A in tempo $O(n^2)$ ed una TM M_2 a nastro singolo che decide A in tempo $O(n \log n)$. Poi abbiamo progettato una TM M_3 a due nastri che decide A in tempo $O(n)$. Quindi la complessità di tempo di A è $O(n \log n)$ su una TM a nastro singolo e $O(n)$ su una TM a due nastri. Notate che la complessità di A dipende dal modello di calcolo scelto.

Questa discussione evidenzia una differenza importante tra la teoria della complessità e la teoria della computabilità. Nella teoria della computabilità, la tesi di Church-Turing implica che tutti i modelli computazionali sono equivalenti - ossia che decidono la stessa classe di linguaggi. Nella teoria della complessità, la scelta del modello influenza sulla complessità di tempo dei linguaggi. Nella teoria della complessità, classifichiamo i problemi computazionali secondo la loro complessità di tempo. Ma con quale modalità misuriamo il tempo? Uno stesso linguaggio può avere differenti necessità di tempo in modelli diversi. Fortunatamente, i requisiti di tempo non differiscono molto per i modelli deterministici usuali. Quindi, se il nostro sistema di classificazione non è molto sensibile a differenze relativamente piccole nella complessità, la scelta dello specifico modello deterministico non è importante.

7.1.3 Relazioni di complessità tra modelli

Qui esaminiamo come la scelta del modello di calcolo può influenzare la complessità di tempo dei linguaggi. Consideriamo tre modelli: macchine di Turing a singolo nastro, macchine di Turing a due nastri

e macchine di Turing non deterministiche.

Teorema 7.8 (orale 24-30)

Teorema 7.8

Sia $t(n)$ una funzione, tale che $t(n) \geq n$. Ogni macchina di Turing multinastro di tempo $t(n)$ ammette una macchina di Turing equivalente a nastro singolo di tempo $O(t^2(n))$.

IDEA.

L'idea alla base della dimostrazione di questo teorema è molto semplice. Ricordiamo che nel [Teorema 3.13](#) abbiamo dimostrato che ogni macchina di Turing multinastro può essere simulata da una macchina di Turing a nastro singolo. Ora analizziamo la simulazione per determinare la quantità di tempo supplementare richiesta. Mostriamo che possiamo simulare ogni passo della macchina multinastro con $O(t(n))$ passi della macchina a nastro singolo.

Per cui il tempo totale impiegato è $O(t^2(n))$.

DIMOSTRAZIONE.

Sia M una TM a k -nastri avente tempo di esecuzione $t(n)$. Costruiamo una TM S a singolo nastro che ha tempo di esecuzione $O(t^2(n))$. La macchina S opera simulando M , come descritto nel [Teorema 3.13](#). Nel rivedere tale simulazione, ricordiamo che S utilizza il suo unico nastro per rappresentare il contenuto di tutti i k nastri di M . I nastri sono memorizzati consecutivamente, con le posizioni delle testine di M masicate nelle celle appropriate. Inizialmente, S mette il nastro nel formato che rappresenta tutti i nastri di M e poi simula i passi di M . Per simulare un passo, S scorre tutte le informazioni memorizzate sul suo nastro per determinare i simboli presenti sotto le testine di M . Poi S esegue un'altra scansione del suo nastro per aggiornare il contenuto dei nastri e le posizioni delle testine. Se una testina di M si sposta a destra su una parte non letta del nastro, S deve aumentare la quantità di spazio allocato per questo nastro. Lo fa spostando una parte del suo nastro di una cella a destra.

Ora analizziamo questa simulazione. Per ogni passo di M , la macchina S fa due passi sulla parte attiva del suo nastro. Il primo ottiene le informazioni necessarie per determinare la prossima mossa e il secondo la esegue. La lunghezza della parte attiva del nastro di S determina il tempo che S impiega per eseguire la scansione, quindi dobbiamo determinare un limite superiore per questa lunghezza. Per farlo prendiamo la somma delle lunghezze parti attive dei k nastri di M . Ciascuna di queste parti attive ha lunghezza al più $t(n)$, poiché M utilizza $t(n)$ celle del nastro in $t(n)$ passi, se la testina si sposta verso destra ad ogni passo e anche meno se vi sono spostamenti di qualche testina a sinistra. Quindi una scansione della parte di nastro attiva di S impiega $O(t(n))$ passi. Per simulare ciascuna delle fasi di M , S esegue due scansioni ed eventualmente fino a k spostamenti a destra. Ognuno impiega un tempo $O(t(n))$, per cui il tempo totale per S per simulare un passo di M è $O(t(n))$. Ora possiamo limitare il tempo totale impiegato dalla simulazione. La fase iniziale, in cui S mette il nastro nel formato corretto, usa $O(t(n))$ passi, per cui questa parte della simulazione utilizza $t(n) \times O(t(n)) = O(t^2(n))$ passi. Quindi l'intera simulazione di M utilizza $O(n) + O(t^2(n))$ passi. Abbiamo assunto che $t(n) \geq n$ (un'ipotesi ragionevole perché M non potrebbe nemmeno leggere l'intero input in meno tempo). Pertanto il tempo di esecuzione di S è $O(n) + O(t^2(n)) = O(t^2(n))$ e la dimostrazione è completa.

Ora vedremo un teorema analogo nel caso di macchine di Turing non deterministiche a nastro singolo. Mostriremo che ogni linguaggio decidibile su tale macchina è anche decidibile su una macchina di Turing deterministica a nastro singolo che richiede molto più tempo. Prima di farlo, dobbiamo definire il tempo di esecuzione di una macchina di Turing non deterministica. Ricordiamo che una macchina di Turing non deterministica è un decisore se tutte le sue computazioni si fermano su tutti gli input.

7.1.4 Definizione 7.9

Definizione 7.9

Sia N una macchina di Turing non deterministica che sia anche un decisore. Il **tempo di esecuzione** di N è la funzione $f : \mathbb{N} \rightarrow \mathbb{N}$, tale che $f(n)$ è il massimo numero di passi che N usa per ognuna delle computazioni su ogni input di lunghezza n , come mostrato nella figura seguente.

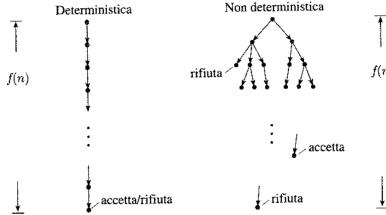


Figura 29: Misurazione del tempo nei casi deterministico e non deterministico.

La definizione del tempo di esecuzione di una macchina di Turing non deterministica non è destinata a corrispondere ad un qualche dispositivo informatico reale. Piuttosto, si tratta di un'utile definizione matematica che aiuta a caratterizzare la complessità di una classe importante di problemi computazionali, come dimostreremo a breve.

7.1.5 Teorema 7.11 (orale 24-30)

Teorema 7.11

Sia $t(n)$ una funzione, tale che $t(n) \geq n$. Ogni macchina di Turing non deterministica a singolo nastro avente tempo di esecuzione $t(n)$ ammette una macchina di Turing deterministica a singolo nastro equivalente di tempo $2^{O(t(n))}$.

DIMOSTRAZIONE.

Sia N una TM non deterministica avente tempo di esecuzione $t(n)$. Costruiamo una TM deterministica D che simula N , come nel [Teorema 3.16](#), effettuando una ricerca sull'albero delle computazioni di N . Ora analizziamo tale simulazione. Su un input di lunghezza n , ogni ramificazione dell'albero delle computazioni di N ha lunghezza al più $t(n)$. Ogni nodo dell'albero può avere al più b figli, dove b è il massimo numero di scelte possibili in accordo alla funzione di transizione di N . Così il numero totale di foglie nell'albero è al massimo $b^{t(n)}$. La simulazione procede esplorando l'albero prima in ampiezza. In altre parole, si visitano tutti i nodi a profondità d prima di passare ad uno qualsiasi dei nodi a profondità $d+1$. L'algoritmo riportato nella dimostrazione del [Teorema 3.16](#) inizia inefficientemente dalla radice e si sposta in basso verso un nodo ogni volta che visita il nodo stesso. Tuttavia l'eliminazione di tale inefficienza non altera l'enunciato del Teorema, quindi la lasciamo in questa forma. Il numero totale di nodi dell'albero è inferiore al doppio del numero di foglie, quindi è limitato da $O(b^{t(n)})$. Il tempo per partire dalla radice e raggiungere un nodo è $O(t(n))$. Pertanto il tempo di esecuzione di D è $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Come descritto nel [Teorema 3.16](#), la TM D ha tre nastri. Convertirla in una TM nastro singolo al più fa sì che si elevi al quadrato il tempo di esecuzione, per il [Teorema 7.8](#). Quindi il tempo di esecuzione del simulatore a nastro singolo è $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$ ed il teorema è dimostrato.

7.2 La classe P

I Teoremi [7.8](#) e [7.11](#) illustrano una distinzione importante. Da una parte, abbiamo dimostrato una differenza al più quadratico o polinomiale tra le complessità di tempo dei problemi misurati su macchine di Turing deterministiche a singolo nastro e multinastro. D'altra parte, abbiamo mostrato una differenza

al più esponenziale tra la complessità temporale di problemi su macchine di Turing deterministiche e non deterministiche.

7.2.1 Tempo polinomiale

Diamo un'occhiata al motivo per cui abbiamo scelto di fare questa separazione tra polinomi ed esponenziali, piuttosto che tra le altre classi di funzioni.

Algoritmi aventi tempo polinomiale sono abbastanza veloci per molti scopi, ma algoritmi aventi tempo esponenziale sono raramente utili. Algoritmi aventi tempo esponenziale si presentano in genere quando risolviamo problemi mediante una ricerca esaustiva nello spazio delle soluzioni, denominata **ricerca mediante forza bruta**. A volte, la ricerca mediante forza bruta può essere evitata attraverso una comprensione più approfondita del problema, che può suggerire un algoritmo polinomiale di maggiore utilità.

Tutti i modelli computazionali deterministici ragionevoli sono **polinomialmente equivalenti**. Cioè, uno di essi può simularne un altro con aumento solo polinomiale del tempo di esecuzione. Quando diciamo che tutti i modelli deterministici ragionevoli sono polinomialmente equivalenti, non cerchiamo di definire il termine *ragionevole*.

Da qui in poi ci concentreremo sugli aspetti della teoria della complessità temporale che non sono influenzati da differenze polinomiali del tempo di esecuzione. Ignorando tali differenze, possiamo sviluppare la teoria in un modo che non dipenda dalla scelta di un particolare modello di computazione. Ricordiamo che il nostro obiettivo è presentare le proprietà fondamentali della *computazione*, piuttosto che le proprietà delle macchine di Turing o di un qualsiasi altro modello particolare.

La nostra decisione di non tener conto delle differenze polinomiali non significa che consideriamo tali differenze non importanti. Al contrario, certamente consideriamo importante la differenza tra il tempo n ed il tempo n^3 . Ma alcune questioni, come ad esempio l'essere polinomiale o non polinomiale del problema della fattorizzazione, non dipendono da differenze di tipo polinomiale e sono ugualmente importanti. Abbiamo semplicemente scelto di concentrarci su questo tipo di questioni.

Veniamo ora ad una definizione importante nella teoria della complessità.

7.2.2 Definizione 7.12

Definizione 7.12

P è la classe di linguaggi che sono decidibili in tempo polinomiale su una macchina di Turing deterministica a singolo nastro. In altre parole,

$$P = \bigcup_k TIME(n^k).$$

La classe P ha un ruolo centrale nella nostra teoria ed è importante perché

1. P è invariante per tutti i modelli di calcolo che sono polinomialmente equivalenti ad una macchina di Turing deterministica a nastro singolo, e
2. P corrisponde approssimativamente alla classe dei problemi che sono realisticamente risolvibili su un computer.

Il punto 1 indica che P è una classe matematicamente robusta. Non è influenzata dai particolari del modello di computazione che stiamo usando.

Il punto 2 indica che P è importante da un punto di vista pratico. Quando un problema è in P, abbiamo un metodo per risolverlo che viene eseguito in tempo n^k per una costante k . Se questo tempo di esecuzione risulta pratico dipende da k e dall'applicazione. Tuttavia, fissare la soglia di risolubilità in pratica al tempo polinomiale si è dimostrato utile.

7.2.3 Esempi di problemi in P

Quando analizziamo un algoritmo per dimostrare che esso viene eseguito in tempo polinomiale, abbiamo bisogno di fare due cose. In primo luogo, dobbiamo dare un limite superiore polinomiale (in genere nella notazione O-grande) sul numero di fasi che l'algoritmo esegue quando viene eseguito su un input di lunghezza n . Poi, dobbiamo esaminare le singole fasi nella descrizione dell'algoritmo per essere sicuri che ognuna può essere implementata in tempo polinomiale su un modello deterministico ragionevole. Utilizziamo le fasi nel descrivere l'algoritmo per rendere questa seconda parte dell'analisi più semplice da eseguire.

Quando entrambi i compiti sono stati completati, possiamo concludere che l'algoritmo ha un tempo di esecuzione polinomiale perché abbiamo dimostrato che viene eseguito per un numero polinomiale di fasi, ciascuna delle quali può essere eseguita in tempo polinomiale, e la composizione di polinomi è ancora un polinomio.

Un punto che richiede attenzione è il metodo di codifica usato per i problemi. Continuiamo ad usare la notazione mediante parentesi angolari $\langle \cdot \rangle$ per indicare una codifica mediante una stringa ragionevole di uno o più oggetti, senza specificare un particolare metodo di codifica.

Molti dei problemi computazionali che incontrerete in questo capitolo contengono codifiche di grafi. Una codifica ragionevole di un grafo consiste in un elenco dei suoi nodi ed archi. Un altro è la **matrice di adiacenza**, dove l'elemento (i, j) -esimo è 1 se c'è un arco dal nodo i al nodo j e 0 se non c'è. Quando analizziamo algoritmi su grafi, il tempo di esecuzione può essere calcolato in termini del numero di nodi invece della dimensione della rappresentazione del grafo. In rappresentazioni ragionevoli di grafi, la dimensione della rappresentazione è un polinomio nel numero di nodi. Così, se analizziamo un algoritmo e mostriamo che il suo tempo di esecuzione è polinomiale (o esponenziale) nel numero di nodi, sappiamo che esso è anche polinomiale (o esponenziale) nella dimensione dell'input. Il primo problema riguarda grafi orientati. Un grafo orientato G contiene nodi s e t , come mostrato nella figura seguente.

Il problema *PATH* consiste nel determinare, se esiste, un cammino diretto s a t . Sia

$$PATH = \{\langle G, s, t \rangle \mid G \text{ è un grafo orientato che contiene un cammino diretto da } s \text{ a } t\}.$$

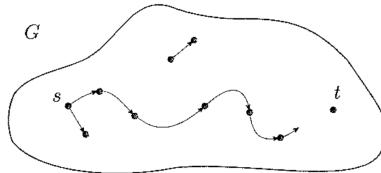


Figura 30: Il problema *PATH*: esiste un cammino da s a t ?

Teorema 7.14

$PATH \in P$.

IDEA. Dimostriamo questo teorema presentando un algoritmo di tempo polinomiale che decide *PATH*. Prima di descrivere l'algoritmo, notiamo che un algoritmo di forza bruta per questo problema non è abbastanza veloce. Un algoritmo di forza bruta per *PATH* procede esaminando tutti i potenziali cammini in G e determinando se uno di essi è un cammino diretto da s a t . Un potenziale cammino è una sequenza di nodi in G avente lunghezza al più m , dove m è il numero di nodi in G . (Se esiste un qualche cammino diretto da s a t , ne deve esistere necessariamente almeno uno avente lunghezza al massimo m , perché non è mai necessario ripetere un nodo.) Tuttavia il numero di tali percorsi potenziali è approssimativamente m^m , che è esponenziale nel numero di nodi in G . Pertanto questo algoritmo di forza bruta utilizza tempo esponenziale.

Per ottenere un algoritmo polinomiale per *PATH*, dobbiamo fare qualcosa che evita la forza bruta. Un modo è quello di utilizzare un metodo visita dei grafi quale la visita in ampiezza. In questo caso, contrassegniamo successivamente tutti nodi in G che sono raggiungibili da s mediante cammini diretti di lunghezza 1, poi 2, poi 3, fino ad m . Possiamo facilmente limitare il tempo di esecuzione di questa

strategia mediante un polinomio.

DIMOSTRAZIONE. Un algoritmo in tempo polinomiale M per $PATH$ procede come segue. $M =$ "Su input $\langle G, s, t \rangle$, dove G è un grafo orientato e s e t sono nodi di G :

1. Marca il nodo s .
2. Ripete il seguente passo fino a quando nessun nuovo nodo viene marcato:
3. Scandiona tutti gli archi di G . Se trova un arco (a, b) che va da un nodo a marcato ad un nodo b non marcato, marca il nodo b .
4. Se il nodo t è marcato, *accetta*; altrimenti, *rifiuta*."

Ora analizziamo questo algoritmo per dimostrare che lavora in tempo polinomiale. Ovviamente, le fasi 1 e 4 sono eseguite una sola volta. La fase 3 è eseguita al più m volte perché ogni volta, tranne l'ultima, marca un nodo aggiuntivo in G . Quindi il numero totale di fasi utilizzate è al massimo $1+1+m$, che da un polinomio nella dimensione di G . Le fasi 1 e 4 di M sono facilmente implementate in tempo polinomiale su un qualsiasi modello deterministico ragionevole. La fase 3 richiede una scansione dell'input ed un test che verifichi se certi nodi sono marcati o meno, ed è anch'essa facilmente implementabile in tempo polinomiale. Quindi M è un algoritmo di tempo polinomiale per $PATH$.

Consideriamo ora un altro esempio di algoritmo avente tempo polinomiale. Diciamo che due numeri sono relativamente primi se 1 è il più grande numero intero che li divide entrambi. Per esempio, 10 e 21 sono relativamente primi, anche se nessuno dei due è esso stesso un numero primo, mentre 10 e 2 non sono relativamente primi perché entrambi sono divisibili per 2. Sia $RELPRIME$ il problema di verificare se due numeri sono relativamente primi. Quindi

$$RELPRIME = \{\langle a, b \rangle \mid a \text{ e } b \text{ sono relativamente primi}\}.$$

Teorema 7.15

IDEA. Un algoritmo che risolve questo problema effettua una ricerca tra tutti i divisori di entrambi i numeri e accetta se nessuno è maggiore di 1. Tuttavia, la grandezza di un numero rappresentato in binario, o in qualsiasi altra notazione in base k per $k \geq 2$, è esponenziale nella lunghezza della sua rappresentazione. Quindi questo algoritmo di forza bruta ricerca attraverso un numero esponenziale di potenziali divisori e ha un tempo di esecuzione esponenziale.

Risolviamo invece questo problema con un antico procedimento numerico, chiamato *algoritmo di Euclide*, per il calcolo del massimo comun divisore. Il **massimo comun divisore** dei numeri naturali x e y , scritto $\gcd(x, y)$, è il più grande numero intero che divide entrambi x e y . Ovviamente, x e y sono relativamente primi se $\gcd(x, y) = 1$. Nella dimostrazione indicheremo l'algoritmo di Euclide come l'algoritmo E . Esso utilizza la funzione mod, per cui $x \bmod y$ è il resto della divisione intera di x per y .

DIMOSTRAZIONE. L'algoritmo euclideo E è il seguente.

$E =$ "Su input $\langle x, y \rangle$, dove x e y sono numeri naturali in binario:

1. Ripete finché $y = 0$:
2. Pone $x \leftarrow x \bmod y$.
3. Scambia x e y .
4. Output x ."

L'algoritmo R risolve $RELPRIME$, usando E come sottoprocedura.
 $R =$ "Su input $\langle x, y \rangle$, dove x e y sono numeri naturali in binario:

1. Esegue E su $\langle x, y \rangle$.

- Se il risultato è 1, *accetta*; altrimenti, *rifiuta*.”

Chiaramente, se E lavora correttamente in tempo polinomiale, così fa R e quindi abbiamo solo bisogno di analizzare E per valutarne il tempo e la correttezza. Per analizzare la complessità di tempo di E , per prima cosa mostriamo che ogni esecuzione della fase 2 (tranne eventualmente la prima), taglia il valore di x di almeno la metà. Dopo l'esecuzione della la fase 2, risulta $x < y$ a causa della natura della funzione mod. Dopo la fase 3, risulta $x > y$ perché i due valori sono stati scambiati. Così, quando la fase 2 viene successivamente eseguita, $x > y$. Se $x/2 \geq y$, allora $xmody < y \leq x/2$ e x diminuisce di almeno la metà. Se $x/2 < y$, allora $xmody = x - y < x/2$ e x diminuisce di almeno la metà.

I valori di x e y sono scambiati ogni volta che viene eseguita la fase 3, così ognuno dei valori originali di z e y è ridotto di almeno la metà ad ogni iterazione del ciclo. Quindi, il numero massimo di volte in cui le fasi 2 e 3 sono eseguite risulta pari al minore tra $2\log_2 x$ e $2\log_2 y$. Questi logaritmi sono proporzionali alle lunghezze delle rappresentazioni, il che dà il numero di fasi eseguite pari a $O(n)$. Ogni fase di E utilizza solo tempo polinomiale, quindi il tempo totale di esecuzione è polinomiale.

L'ultimo esempio di algoritmo polinomiale mostra che ogni linguaggio context-free è decidibile in tempo polinomiale.

Teorema 7.16

Ogni linguaggio context-free è un elemento di P .

IDEA. Nel [Teorema 4.9](#) abbiamo dimostrato che ogni CFL è decidibile. Per fare ciò abbiamo dato un algoritmo che lo decide. Se questo algoritmo venisse eseguito in tempo polinomiale, questo teorema seguirebbe come corollario. Ricapitoliamo l'algoritmo e vediamo se funziona abbastanza velocemente. Sia L un CFL generato da una CFG G in forma normale di Chomsky. Qualsiasi derivazione di una stringa w ha $2n - 1$ passi, dove n è la lunghezza di w perché G è in forma normale di Chomsky. Il decisore per L funziona provando tutte le possibili derivazioni con $2n - 1$ passi quando il suo input è una stringa di lunghezza n . Se uno di questi è una derivazione di w , il decisore accetta, altrimenti, rifiuta.

Una rapida analisi di questo algoritmo dimostra che non viene eseguito in tempo polinomiale. Il numero di derivazioni con k passi può essere esponenziale in k , per cui questo algoritmo può richiedere tempo esponenziale.

Per ottenere un algoritmo di tempo polinomiale introduciamo una potente tecnica chiamata **programmazione dinamica**. Memorizziamo la soluzione ad ogni sottoproblema in modo da doverlo risolvere solo una volta. Lo facciamo creando una tabella di tutti i sottoproblemi e inserendo le loro soluzioni sistematicamente appena le troviamo. Nel caso in oggetto, consideriamo i sottoproblemi consistenti nel determinare se ogni variabile in G genera ciascuna sottostringa di w . L'algoritmo inserisce la soluzione a questo sottoproblema in una tabella $n \times n$. Per $i \leq j$, la voce (i, j) -esima della tabella contiene la collezione di variabili che generano la sottostringa $w_1w_{i+1}...W_j$. Per $i > j$, le voci della tabella non sono utilizzate. L'algoritmo riempie le voci della tabella per ogni sottostringa di w . In primo luogo si riempiono le voci corrispondenti alle sottostringhe di lunghezza 1, poi quelle di lunghezza 2 e così via. Si utilizzano le voci per le lunghezze minori per determinare quelle per le lunghezze superiori.

Per esempio, si supponga che l'algoritmo abbia già stabilito quali variabili generano tutte le sottostringhe di lunghezza fino a k . Per determinare se una variabile A genera una particolare sottostringa di lunghezza $k + 1$, l'algoritmo suddivide la sottostringa in due parti non vuote in tutti i k modi possibili. Per ogni suddivisione, l'algoritmo esamina ogni regola $A \rightarrow BC$ per determinare se B genera la prima parte e C genera la seconda parte, utilizzando le voci della tabella precedentemente calcolate. Se entrambe B e C generano le rispettive parti, A genera la sottostringa e quindi viene aggiunta la voce corrispondente nella tabella. L'algoritmo inizia il processo con le stringhe di lunghezza 1 esaminando la tabella per le regole $A \rightarrow b$.

DIMOSTRAZIONE. Il seguente algoritmo D implementa l'idea della dimostrazione. Sia G una CFG in forma normale di Chomsky che genera il CFL L . Si supponga che S sia la variabile iniziale. (Ricordiamo che la stringa vuota viene gestita in modo speciale nella grammatica in forma normale di Chomsky. L'algoritmo gestisce il caso particolare in cui $w = \epsilon$ nella fase 1.) I commenti appaiono tra doppie

parentesi quadre.

D = "Su input $w = w_1w_2\dots w_n$:

1. Per $w = \varepsilon$, se $S \rightarrow \varepsilon$ è una regola di G , accetta; altrimenti, rifiuta.
[caso $w = \varepsilon$]
2. Per $i = 1$ a n : [esamina ogni sottostringa di lunghezza 1]
3. Per ogni variabile A :
4. Testa se $A \rightarrow b$ è una regola, dove $b = w_i$.
5. Se sì, inserisci A in $table(i, i)$.
6. Per $l = 2$ a n : [l è la lunghezza della sottostringa]
7. Per $i = 1$ a $n - l + 1$: [i è la posizione iniziale della sottostringa]
8. Sia $j = i + l - 1$. [j è la posizione finale della sottostringa]
9. Per $k = i$ a $j - 1$: [k è la posizione di suddivisione]
10. Per ogni regola $A \rightarrow BC$:
11. Se B è in $table(i, k)$ e C è in $table(k + 1, j)$, inserisci A in $table(i, j)$.
12. Se S è in $table(1, n)$, accetta; altrimenti, rifiuta."

Ora analizziamo D . Ogni fase è facilmente implementata in modo da essere eseguita in tempo polinomiale. Le fasi 4 e 5 sono eseguite al più nv volte, dove v è il numero di variabili in G ed è una costante fissa indipendente da n ; quindi queste fasi sono eseguite $O(n)$ volte. La fase 6 è eseguita al più n volte. Ogni volta che la fase 6 viene eseguita, la fase 7 viene eseguita al più n volte. Ogni volta che la fase 7 viene eseguita, le fasi 8 e 9 sono eseguite al più r volte. Ogni volta che la fase 9 viene eseguita, la fase 10 è eseguita r volte, dove r è il numero di regole di G ed è un'altra costante. Quindi la fase 11, il ciclo interno dell'algoritmo, viene eseguito $O(n^3)$ volte. La somma totale mostra che D esegue $O(n^3)$ fasi.

7.3 La classe NP

Come abbiamo osservato nella precedente, per molti problemi possiamo evitare la ricerca mediante forza bruta ed ottenere una soluzione polinomiale. Tuttavia, i tentativi di evitare la forza bruta nel caso di altri problemi, tra cui molti utili ed interessanti, non hanno avuto successo e non sono noti algoritmi polinomiali per risolvere tali problemi. Come mai i tentativi di trovare algoritmi polinomiali per questi problemi non hanno avuto successo? Non conosciamo la risposta a questa importante domanda. Forse questi problemi ammettono algoritmi in tempo polinomiale che si basano su principi non ancora noti. O forse alcuni di questi problemi semplicemente non possono essere risolti in tempo polinomiale. Possono essere intrinsecamente difficili. Una scoperta notevole riguardo a questa domanda dimostra che le complessità di molti problemi sono legate tra di loro. Un algoritmo polinomiale per un certo problema può essere utilizzato per risolvere un'intera classe di problemi. Per capire questo fenomeno, cominciamo con un esempio. Un cammino Hamiltoniano in un grafo orientato G è un cammino orientato che attraversa ogni nodo del grafo esattamente una volta. Consideriamo il problema di verificare se un grafo orientato contiene un cammino Hamiltoniano che collega due nodi specificati, come mostrato nella seguente figura. Sia

$$HAMPATH = \{ \langle G, s, t \rangle \mid G \text{ è un grafo orientato che contiene un cammino Hamiltoniano da } s \text{ a } t \}.$$

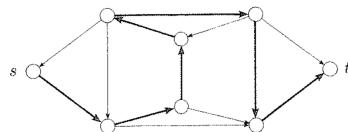


Figura 31: Il problema $HAMPATH$: esiste un cammino Hamiltoniano da s a t ?

Possiamo facilmente ottenere un algoritmo avente tempo esponenziale per il problema *HAMPAT* modificando l'algoritmo di forza bruta per *PATH* dato nel [Teorema 7.14](#). Abbiamo bisogno solo di aggiungere un controllo per verificare che il cammino potenziale è Hamiltoniano. Nessuno sa se *HAMPATH* è risolvibile in tempo polinomiale. Il problema *HAMPATH* ha una caratteristica chiamata verificabilità polinomiale che è importante per capire la sua complessità. Anche se non conosciamo una maniera veloce (ad esempio, in tempo polinomiale) per determinare se un grafo contiene un cammino Hamiltoniano, se un tale cammino è stato scoperto in qualche modo (magari utilizzando un algoritmo di tempo esponenziale), si potrebbe facilmente convincere qualcun altro della sua esistenza semplicemente esibendolo. In altre parole, verificare l'esistenza di un cammino Hamiltoniano può essere molto più facile che determinare la sua esistenza.

Un altro problema polinomialmente verificabile è l'essere composto. Ricordiamo che un numero naturale è **composto** se è il prodotto di due numeri interi maggiori di 1 (cioè, un numero composto è un numero che non è primo). Sia

$$COMPOSITES = \{x \mid x = pq \text{ per gli interi } p, q > 1\}.$$

Alcuni problemi potrebbero non essere polinomialmente verificabili. Per esempio, consideriamo $\overline{\text{HAMPATH}}$, il complemento del problema *HAMPATH*. Anche se riuscissimo a determinare (in qualche modo) che un grafo non ha un cammino Hamiltoniano, non conosciamo alcun modo per verificarne l'inesistenza senza utilizzare lo stesso algoritmo avente tempo esponenziale usato per la determinazione originaria. Una definizione formale è la seguente.

Definizione 7.18

Definizione 7.18

Un **verificatore** per un linguaggio A è un algoritmo V , dove

$$A = \{w \mid V \text{ accetta } \langle w, c \rangle \text{ per qualche stringa } c\}.$$

Misuriamo il tempo di un verificatore solo in termini della lunghezza di w , quindi un **verificatore in tempo polinomiale** viene eseguito in tempo polinomiale nella lunghezza di w . Un linguaggio A è **polynomialmente verificabile** se ammette un verificatore in tempo polinomiale.

Un verificatore usa ulteriori informazioni, rappresentate dal simbolo c nella [Definizione 7.18](#), per verificare che una stringa w è un elemento di A . Questa informazione è chiamata **certificato**, o **prova**, di appartenenza ad A . Osservate che per i verificatori polinomiali, il certificato ha lunghezza polinomiale (nella lunghezza di w) perché questo è tutto ciò cui il verificatore può accedere a causa del suo limite sul tempo. Proviamo ad applicare questa definizione per i linguaggi *HAMPATH* e *COMPOSITES*. Per il problema *HAMPATH*, un certificato per una stringa $\langle G, s, t \rangle \in \text{HAMPATH}$ è un cammino Hamiltoniano da s a t in G . Per il problema *COMPOSITES*, un certificato per il numero x è uno dei suoi divisori. In entrambi i casi, il verificatore, avendo il certificato, può verificare in tempo polinomiale che l'input è il linguaggio.

Definizione 7.19

Definizione 7.19

NP è la classe dei linguaggi che ammettono un verificatore in tempo polinomiale.

La classe NP è importante perché contiene molti problemi di interesse pratico. Dalla discussione precedente, sia *HAMPATH* che *COMPOSITES* sono elementi di NP. Come abbiamo accennato, *COMPOSITES* è anche elemento di P, che è un sottoinsieme di NP; ma dimostrare questo risultato

più forte è più difficile. Il termine NP proviene da ***tempo polinomiale non deterministico*** e deriva da una caratterizzazione alternativa che utilizza macchine di Turing non deterministiche di tempo polinomiale. I problemi in NP sono a volte chiamati problemi NP.

Quella che segue è una macchina di Turing non deterministica che decide il problema *HAMPATH* in tempo polinomiale non deterministico. Ricordiamo che nella [Definizione 7.9](#), abbiamo definito il tempo di una macchina non deterministica come il tempo usato dalla computazione corrispondente alla ramificazione più lunga.

N_1 = "Su input $\langle G, s, t \rangle$, dove G è un grafo orientato e s e t sono nodi di G :

1. Scrive una lista di m numeri, p_1, \dots, p_m , dove m è il numero di nodi in G . Ogni numero nella lista è scelto in modo non deterministico tra 1 e m .
2. Controlla se vi sono ripetizioni nella lista. Se ne trova una, *rifiuta*.
3. Controlla se $s = p_1$ e $t = p_m$. Se una delle due è falsa, *rifiuta*.
4. Per ogni i tra 1 e $m - 1$, controlla se (p_i, p_{i+1}) è un arco di G . Se non lo è *rifiuta*. Altrimenti tutti i test sono stati superati, quindi *accetta*."

Per analizzare questo algoritmo e verificare che viene eseguito in tempo polinomiale non deterministico, esaminiamo ogni sua fase. Nella fase 1, la selezione non deterministica richiede chiaramente tempo polinomiale. Nelle fasi 2 e 3, ogni parte è un semplice controllo, quindi entrambe richiedono tempo polinomiale. Quindi, questo algoritmo viene eseguito in tempo polinomiale non deterministico.

Teorema 7.20(orale)

Teorema 7.20

Un linguaggio è in NP sse esso viene deciso in tempo polinomiale da una macchina di Turing non deterministica.

IDEA. Mostriamo come convertire un verificatore di tempo polinomiale in una NTM equivalente e viceversa. La NTM simula il verificatore per indovinare il certificato. Il verificatore simula la NTM utilizzando il ramo di computazione accettante come certificato.

DIMOSTRAZIONE. Per la parte diretta di questo teorema, assumiamo $A \in \text{NP}$ e mostriamo che A è deciso da una NTM N avente tempo polinomiale. Sia V il verificatore polinomiale per A la cui esistenza è assicurata dalla definizione di NP. Si supponga che V è una TM avente tempo di esecuzione n^k e costruiamo N come segue.

N = "Su input w di lunghezza n :

1. In maniera non deterministica seleziona una stringa c di lunghezza al più n^k
2. Esegue V su input $\langle w, c \rangle$.
3. Se V accetta, *accetta*; altrimenti, *rifiuta*."

Per dimostrare la direzione inversa del teorema, supponiamo che A è deciso da una NTM N avente tempo polinomiale e costruiamo come segue un verificatore V avente tempo polinomiale.

V = "Su input $\langle w, c \rangle$, dove w e c sono stringhe:

1. Simula N su input w , trattando ogni simbolo di c come la descrizione di una scelta non deterministica da formulare ad ogni passo (come nella dimostrazione del [Teorema 3.16](#)).

2. Se questo ramo di computazione di N accetta, *accetta*; altrimenti, *rifiuta.*"

Definiamo la classe di complessità di tempo non deterministico $NTIME(t(n))$ in modo analogo alla classe di complessità di tempo deterministico $TIME(t(n))$.

Definizione 7.21

Definizione 7.21

$$NTIME(t(n)) = \{L \mid L \text{ è un linguaggio deciso da una macchina di Turing non deterministica di tempo } O(t(n))\}$$

Corollario 7.22

$$NP = \bigcup_k NTIME(n^k).$$

La classe NP è insensibile alla scelta di un modello computazionale ragionevole non deterministico in quanto tutti questi modelli sono polinomialmente equivalenti. Nel descrivere e analizzare algoritmi non deterministici aventi tempo polinomiale, seguiamo le convenzioni precedenti per gli algoritmi di tempo polinomiale deterministici. Ogni fase di un algoritmo non deterministico di tempo polinomiale deve avere una chiara implementazione in un tempo polinomiale non deterministico su un modello di calcolo non deterministico ragionevole. Analizziamo l'algoritmo per dimostrare che ogni ramo utilizza al massimo un numero polinomiale di fasi.

7.3.1 Esempi di problemi in NP

Una *clique* in un grafo non orientato è un sottografo, in cui ogni due nodi sono collegati da un arco. Una k -*clique* è una clique che contiene k nodi. La seguente illustra un grafo con una 5-clique.

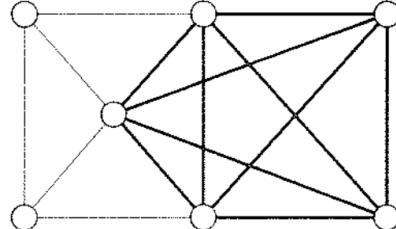


Figura 32: Un grafo con una 5-clique

Il problema della clique consiste nel determinare se un grafo contiene una clique di una dimensione specificata. Sia

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ è un grafo che contiene una } k\text{-clique}\}.$$

Teorema 7.24

$$CLIQUE \in NP.$$

IDEA. La clique è il certificato.

DIMOSTRAZIONE. Il seguente algoritmo è un verificatore V per $CLIQUE$.

V = "Su input $\langle\langle G, k \rangle, c \rangle$:

1. Controlla se c è sottografo con k nodi in G .
2. Controlla se G contiene tutti gli archi tra i nodi di c .
3. Se entrambi i controlli sono veri, *accetta*; altrimenti, *rifiuta*.”

DIMOSTRAZIONE ALTERNATIVA. Se preferite pensare a NP in termini di macchine di Turing di tempo polinomiale non deterministico, si può dimostrare questo teorema dandone una che decide *CLIQUE*. Osservate la somiglianza tra le due prove.

N = ”Su input $\langle G, k \rangle$ dove G è un grafo:

1. Seleziona non deterministicamente un sottoinsieme c di k nodi di G .
2. Controlla se G contiene tutti gli archi tra i nodi di c .
3. Se sì, *accetta*; altrimenti, *rifiuta*.”

Come altro esempio, consideriamo il problema di aritmetica sugli interi *SUBSET – SUM*. Ci viene dato un insieme di numeri x_1, \dots, x_k ed un valore obiettivo t . Vogliamo determinare se l'insieme contiene un sottoinsieme che somma a t . Quindi,

$$\text{SUBSET} - \text{SUM} = \{\langle S, t \rangle \mid S = \{x_1, \dots, x_k\}, \text{ e per qualche } \{y_1, \dots, y_l\} \subseteq \{x_1, \dots, x_k\}, \text{ si ha } \sum y_i = t\}.$$

Per esempio, $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in \text{SUBSET} - \text{SUM}$ perché $4+21=25$. Notate che $\{x_1, \dots, x_k\}$ e $\{y_1, \dots, y_l\}$ sono considerati **multinsiemi** e quindi permettono la ripetizione di elementi.

Teorema 7.25

SUBSET – SUM è in NP.

IDEA. Il sottoinsieme è il certificato.

DIMOSTRAZIONE. L'algoritmo seguente è un verificatore V per *SUBSET – SUM*.

V = ”Su input $\langle \langle S, t \rangle, c \rangle$:

1. Verifica se c è una collezione di numeri la cui somma è t .
2. Verifica se S contiene tutti i numeri in c .
3. Se entrambi i controlli sono veri, *accetta*; altrimenti, *rifiuta*.”

DIMOSTRAZIONE ALTERNATIVA. Possiamo anche dimostrare questo teorema esibendo una macchina di Turing di tempo polinomiale non deterministico per *SUBSET – SUM* come segue.

N = ”Su input $\langle S, t \rangle$:

1. Seleziona non deterministicamente un sottoinsieme c dei numeri in S .
2. Verifica se c è una collezione di numeri la cui somma è t .
3. Se sì, *accetta*; altrimenti, *rifiuta*.”

Si osservi che i complementi di questi insiemi, \overline{CLIQUE} e $\overline{SUBSET - SUM}$, non sono elementi di NP. Verificare che qualcosa *non* è presente sembra essere più difficile di verificare che è presente. Definiamo una classe di complessità separata, denominata **coNP**, che contiene i linguaggi che sono il complemento di un linguaggio in NP. Non sappiamo se coNP è diversa da NP.

La questione $P = NP$

Come abbiamo detto, NP è la classe dei linguaggi che sono risolubili in tempo polinomiale con una macchina di Turing non deterministica; o, equivalentemente, è la classe dei linguaggi per cui l'appartenenza può essere verificata in tempo polinomiale. P è la classe dei linguaggi per cui l'appartenenza può essere decisa in tempo polinomiale. Riassumiamo queste informazioni come segue, dove informalmente ci riferiamo ad un problema risolvibile in tempo polinomiale come *risolvibile "velocemente."*

P = la classe dei linguaggi per cui l'appartenenza è *decidibile* velocemente.

NP = la classe dei linguaggi per cui l'appartenenza è *verificabile* velocemente.

Abbiamo presentato esempi di linguaggi, quali *HAMPATH* e *CLIQUE*, che sono elementi di NP, ma di cui non è nota l'appartenenza a P. Il potere della verificabilità polinomiale sembra essere molto maggiore di quello della decidibilità polinomiale. Ma, anche se difficile da immaginare, P e NP potrebbero essere uguali. Non siamo in grado di dimostrare l'esistenza di un solo linguaggio in NP che non è in P.

La domanda se $P = NP$ è uno dei maggiori problemi irrisolti dell'informatica teorica e della matematica contemporanea. Se queste classi fossero uguali, qualsiasi problema polinomialmente verificabile sarebbe polinomialmente decidibile. La maggior parte dei ricercatori ritengono che le due classi non sono uguali, perché molte persone hanno investito, senza successo, enormi sforzi per trovare algoritmi aventi tempo polinomiale per problemi in NP. I ricercatori hanno anche tentato di dimostrare che le due classi sono diverse, ma ciò comporterebbe dimostrare che non esiste un algoritmo veloce che può sostituire la ricerca esaustiva. Cio è fuori dalla portata scientifica attuale. La figura seguente illustra le due possibilità.

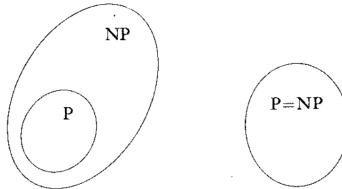


Figura 33: La questione $P = NP$

Il miglior metodo deterministico attualmente noto per decidere se un linguaggio è in NP utilizza tempo esponenziale. In altre parole, possiamo dimostrare che

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k}).$$

ma non sappiamo se NP è contenuta in una classe di complessità deterministica più piccola.

7.4 NP-COMPLETEZZA

Un progresso importante sulla questione P diverso da NP ci fu all'inizio degli anni '70 con il lavoro di Stephen Cook e Leonid Levin. Essi scoprirono vari problemi appartenenti a NP la cui complessità individuale è correlata a quella dell'intera classe. Se esistesse un algoritmo di tempo polinomiale per uno qualsiasi di essi, tutti i problemi in NP diventerebbero risolvibili in tempo polinomiale. Questi problemi vengono detti **NP-completi**.

Se un qualsiasi problema appartenente a NP richiede tempo più che polinomiale, lo stesso vale per uno NP-completo. Quindi, provare che il problema è NP-completo costituisce un'evidenza forte della sua non polinomialità.

Il primo problema NP-completo che presentiamo è il **problema della soddisfacibilità**.

Una formula booleana è un'espressione che coinvolge variabili e operazioni booleane. Per esempio,

$$\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{y})$$

è una formula booleana. Una formula booleana è soddisfacibile se qualche assegnamento di 0 e di 1 alle variabili fa sì che la formula valga 1. La formula precedente è soddisfacibile perché l'assegnamento $x = 0, y = 1$ e $z = 0$ fa sì che ϕ valga 1. Diciamo che l'assegnamento *soddisfa* ϕ . Il **problema della soddisfacibilità** consiste nel determinare se una formula booleana è soddisfacibile. Sia

$$SAT = \{\langle \phi \rangle \mid \phi \text{ è una formula booleana soddisfacibile}\}.$$

Il teorema che ora enunciamo lega la complessità del problema *SAT* alle complessità di tutti i problemi appartenenti a NP.

Teorema 7.27

$SAT \in P$ se e solo se $P = NP$.

7.4.1 Riducibilità in tempo polinomiale

Quando il problema A si riduce al problema B , una soluzione per B può essere usata per risolvere A . Ora definiamo una versione della riducibilità che tiene conto dell'efficienza della computazione. Quando il problema A è riducibile *efficientemente* al problema B , una soluzione efficiente per B può essere usata per risolvere A efficientemente.

Definizione 7.28

Definizione 7.28

Una funzione $f : \Sigma^* \rightarrow \Sigma^*$ è una **funzione calcolabile in tempo polinomiale** se esiste una macchina di Turing di tempo polinomiale M che si arresta con $f(w)$ soltanto sul suo nastro, quando ha iniziato con un qualsiasi input w .

Definizione 7.29

Definizione 7.29

Il linguaggio A è **riducibile mediante funzione in tempo polinomiale**, o semplicemente **riducibile in tempo polinomiale**, al linguaggio B , scritto $A \leq_p B$, se esiste una funzione calcolabile in tempo polinomiale $f : \Sigma^* \rightarrow \Sigma^*$ tale che per ogni w ,

$$w \in A \iff f(w) \in B.$$

La funzione f è chiamata **riduzione in tempo polinomiale** di A a B .

Come con un'ordinaria riduzione mediante funzione, una riduzione in tempo polinomiale di A a B fornisce un modo per convertire la verifica dell'appartenenza ad A nella verifica dell'appartenenza a B - ma ora la conversione viene realizzata efficientemente. Per verificare se $w \in A$, utilizziamo la riduzione f per associare w ad $f(w)$ e verificare se $f(w) \in B$.

Se un linguaggio è riducibile in tempo polinomiale ad un linguaggio per cui già si sa che possiede una soluzione di tempo polinomiale, si ottiene una soluzione di tempo polinomiale per il linguaggio originale, come prova il teorema seguente.

Teorema 7.31 (orale)

Se $A \leq_p B$ e $B \in P$, allora $A \in P$.

DIMOSTRAZIONE. Sia M l'algoritmo di tempo polinomiale che decide B ed f la riduzione di tempo polinomiale di A a B . Descriviamo un algoritmo di tempo polinomiale N che decide A come segue.

N = "Su input w :

1. Calcola $f(w)$.
2. Esegue M con input $f(w)$ e dai in output qualsiasi cosa M da in output."

Risulta $w \in A$ ogni volta che $f(w) \in B$ perché f è una riduzione di A a B . Pertanto, M accetta $f(w)$ ogni volta che $w \in A$. Inoltre, N computa in tempo polinomiale perché ciascuna delle sue due fasi viene eseguita in tempo polinomiale. Si noti che la fase 2 viene eseguita in tempo polinomiale perché la composizione di due polinomi è un polinomio.

Prima di mostrare una riduzione di tempo polinomiale, introduciamo $3SAT$, un caso speciale di problema di soddisfacibilità per cui tutte le formule sono in una forma speciale. Un **letterale** è una variabile booleana o una variabile booleana negata, come x o \bar{x} . Una **clausola** consiste in diversi letterali connessi tramite operatori \vee , come $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$. Una formula booleana è in **forma normale congiuntiva**, ed è detta una **formula cnf**, se comprende diverse clausole connesse tramite operatori \wedge , come

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6).$$

Essa è una formula **3cnf** se tutte le clausole hanno tre letterali come in

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Sia $3SAT = \{\langle \phi \rangle \mid \phi \text{ è una formula 3cnf soddisfacibile}\}$. Se un assegnamento soddisfa una formula cnf, ciascuna clausola deve contenere almeno un letterale che vale 1.

Il teorema seguente presenta una riduzione di tempo polinomiale dal problema $3SAT$ al problema $CLIQUE$.

Teorema 7.32 (orale)

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$.

IDEA. La riduzione di tempo polinomiale f che mostriamo di $3SAT$ a $CLIQUE$ converte formule in grafi. Nei grafi costruiti, clique di una dimensione specificata corrispondono ad assegnamenti soddisfacenti per la formula. Le strutture all'interno del grafo sono progettate per simulare il comportamento delle variabili e delle clausole.

DIMOSTRAZIONE. Sia ϕ una formula con k clausole complemento

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k).$$

La riduzione f genera la stringa $\langle G, k \rangle$, dove G è un grafo non orientato definito come segue.

I nodi in G sono organizzati in k gruppi di tre nodi ciascuno, detti triple, t_1, \dots, t_k . Ciascuna tripla corrisponde ad una delle clausole in ϕ , e ciascun nodo in una tripla corrisponde ad un letterale nella clausola associata. Si etichetti ciascun nodo di G con il suo letterale corrispondente in ϕ .

Gli archi di G connettono tutti meno due tipi di coppie di nodi in G . Non ci sono archi presenti tra nodi nella stessa tripla, e non ci sono archi presenti tra nodi con etichette complementari, come x_2 ed \bar{x}_2 . La seguente illustra questa costruzione quando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$.

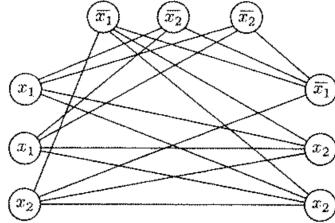


Figura 34: Un grafo costruito da una formula 3cnf

Facciamo vedere ora perché questa costruzione funziona. Dimostriamo che ϕ è soddisfacibile se e solo se G ha una clique di dimensione k .

Si supponga che ϕ abbia un assegnamento che la soddisfa. In questo assegnamento soddisfacente, almeno un letterale è vero in ogni clausola. In ciascuna tripla di G , si selezioni un nodo corrispondente al letterale vero nell'assegnamento soddisfacente. Se più di un letterale è vero in una particolare clausola, si scelga uno dei letterali veri arbitrariamente. I nodi appena selezionati formano una clique di dimensione k . Il numero di nodi selezionati è k perché ne è stato scelto uno per ognuna delle k triple. Ciascuna coppia dei nodi selezionati è collegata da un arco perché nessuna coppia rientra in una delle eccezioni descritte in precedenza. Non possono essere della stessa tripla perché ne è stato scelto soltanto uno per tripla. Non possono avere etichette complementari perché i letterali associati erano entrambi veri nell'assegnamento soddisfacente. Pertanto, G contiene una clique di dimensione k .

Si supponga che G abbia una clique di dimensione k . Nessuna coppia di nodi di una clique occorre nella stessa tripla perché i nodi nella stessa tripla non sono connessi da archi. Pertanto, ciascuna delle k triple contiene esattamente uno dei k nodi della clique. Si assegnino valori di verità alle variabili di ϕ , in modo tale che ciascun letterale che etichetta un nodo della clique risulti vero. Questa operazione è sempre possibile perché due nodi etichettati in modo complementare non sono connessi da un arco e, quindi, non possono essere entrambi nella clique. L'assegnamento così definito per le variabili soddisfa ϕ perché ciascuna tripla contiene un nodo della clique e, quindi, ciascuna clausola contiene un letterale a cui è stato assegnato VERO. Pertanto, ϕ è soddisfacibile.

I Teoremi 7.31 e 7.32 insieme mostrano che se *CLIQUE* è risolvibile in tempo polinomiale lo è anche *3SAT*.

A questo punto volgiamo l'attenzione ad una definizione che ci permette in modo simile di collegare le complessità di un'intera classe di problemi.

7.4.2 Definizione di NP-completezza

Definizione 7.34

Un linguaggio B è **NP-completo** se soddisfa due condizioni:

1. $B \in NP$.
2. Ogni linguaggio $A \in NP$ è riducibile in tempo polinomiale a B .

Teorema 7.35

Se un linguaggio B è NP-completo e $B \in P$, allora $P = NP$.

DIMOSTRAZIONE. Il teorema discende direttamente dal Teorema [Teorema 7.31](#).

Teorema 7.36

Se B è NP-completo e $B \leq_p C$ per qualche $C \in NP$, allora C è NP-completo.

DIMOSTRAZIONE. Già sappiamo che C appartiene a NP, quindi dobbiamo far vedere che ogni A appartenente a NP è riducibile in tempo polinomiale a C . Poiché B è NP-completo, ogni linguaggio appartenente a NP è riducibile in tempo polinomiale a B , e B a sua volta è riducibile in tempo polinomiale a C . Le riduzioni di tempo polinomiale possono essere composte; cioè, se A è riducibile in tempo polinomiale a B e B è riducibile in tempo polinomiale a C , allora A è riducibile in tempo polinomiale a C . Pertanto, ogni linguaggio appartenente a NP è riducibile in tempo polinomiale a C .

7.4.3 Il teorema di Cook e Levin (e mo so cazzo)

Una volta che disponiamo di un problema NP-completo, possiamo ottenerne altri attraverso riduzioni polinomiali da esso. Tuttavia, determinare il primo problema NP-completo è più difficile. Lo facciamo ora, provando che SAT è NP-completo.

Teorema 7.37 (30L)

SAT è NP-completo.

Questo teorema implica il [Teorema 7.27](#).

IDEA. Mostrare che SAT appartiene a NP è facile, e lo faremo velocemente. La parte difficile della dimostrazione è far vedere che qualsiasi linguaggio appartenente a NP è riducibile in tempo polinomiale a SAT .

Per far ciò, costruiremo una riduzione di tempo polinomiale per ciascun linguaggio A appartenente a NP a SAT . La riduzione per A prende una stringa w e produce una formula booleana ϕ che simula la macchina NP per A su input w . Se la macchina accetta, ϕ ha un assegnamento che la soddisfa che corrisponde ad una computazione accettante. Se la macchina non accetta, nessun assegnamento soddisfa ϕ . Pertanto, w appartiene ad A se e solo se ϕ è soddisfacibile.

In realtà, costruire la riduzione per farla funzionare in questo modo è un compito concettualmente semplice, sebbene debbano essere gestiti diversi dettagli. Una formula booleana può contenere le operazioni booleane AND, OR, e NOT, e queste operazioni costituiscono la base per la circuiteria usata nei calcolatori elettronici. Quindi, il fatto che possiamo progettare una formula booleana per simulare una macchina di Turing non è sorprendente.

I dettagli sono nell'implementazione di questa idea.

DIMOSTRAZIONE. Prima di tutto, mostriamo che SAT appartiene a NP. Una macchina non deterministica di tempo polinomiale può ipotizzare un assegnamento per una data formula ϕ ed accettare se l'assegnamento soddisfa ϕ .

Successivamente, prendiamo un qualsiasi linguaggio A appartenente a NP e facciamo vedere che A è riducibile in tempo polinomiale a SAT . Sia N una macchina di Turing non deterministica che decide A in tempo n^k per qualche costante k . (Per comodità in effetti assumiamo che N computi in tempo $n^k - 3$; ma soltato i lettori interessati ai dettagli dovrebbero preoccuparsi di questo punto minore.) La nozione che segue aiuta a descrivere la riduzione. Un **tableau** per N su w è una tabella $n^k \times n^k$ le cui righe sono le configurazioni di una diramazione della computazione di N su input w , come mostrato nella figura seguente.

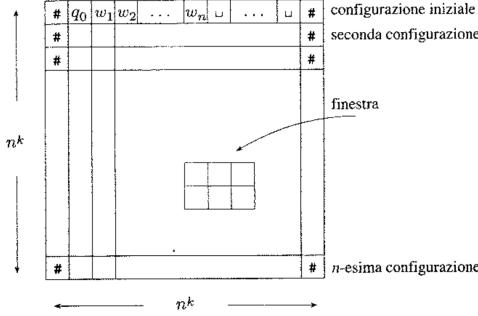


Figura 35: Un tableau per una macchina di Turing non deterministica

Per convenienza nel seguito assumiamo che ciascuna configurazione inizi e finisce con un simbolo $\#$. Pertanto, la prima e l'ultima colonna di un tableau sono tutti $\#$. La prima riga di un tableau è la configurazione iniziale di N su w , e ciascuna riga segue dalla precedente in accordo alla funzione di transizione di N . Un tableau è accettante se qualche riga del tableau è una configurazione accettante.

Ogni tableau accettante per N su w corrisponde ad una diramazione della computazione accettante di N su w . Quindi, il problema di stabilire se N accetta w è equivalente al problema di stabilire se esiste un tableau accettante per N su w .

Passiamo ora alla descrizione della riduzione di tempo polinomiale f di A a SAT . Su input w , la riduzione produce una formula ϕ . Iniziamo descrivendo le variabili di ϕ . Siano Q e Γ l'insieme degli stati e l'alfabeto del nastro di N , rispettivamente. Sia $C = Q \cup \Gamma \cup \{\#\}$. Per ciascun i e j tra 1 e n^k e per ciascun s in C , introduciamo una variabile $X_{i,j,s}$.

Ciascuna delle $(n^k)^2$ entrate di un tableau è chiamata **cella**. La cella in riga i e colonna j viene denotata con $cell[i, j]$ e contiene un simbolo di C . Rappresentiamo i contenuti delle celle con le variabili di ϕ . Se $x_{i,j,s}$ assume il valore 1, significa che $cell[i, j]$ contiene s .

Progettiamo a questo punto ϕ in modo tale che un assegnamento che soffisfa le variabili di ϕ corrisponda ad un tableau accettante per N su w . La formula ϕ è l'AND di quattro parti: $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$. Descriviamo ciascuna parte, una per volta.

Come anticipato precedentemente, porre a 1 la variabile $x_{i,j,s}$ corrisponde a posizionare il simbolo s in $cell[i, j]$. La prima cosa che dobbiamo garantire al fine di ottenere una corrispondenza tra un assegnamento ed un tableau è che l'assegnamento ponga ad 1 esattamente una variabile per ciascuna cella. La formula ϕ_{cell} garantisce questo requisito esprimendolo in termini di operazioni booleane:

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

I simboli \wedge e \vee sono usati per le operazioni AND e OR. Per esempio, l'espressione nella formula precedente

$$\bigvee_{s \in C} x_{i,j,s}$$

è un'abbreviazione per

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \dots \vee x_{i,j,s_l}.$$

dove $C = \{s_1, s_2, \dots, s_l\}$. Quindi, ϕ_{cell} è in realtà un'espressione grande che contiene un frammento per ciascuna cella nel tableau, poiché i e j variano da 1 a n^k . La prima parte del frammento stabilisce che almeno una variabile assume valore 1 nella cella corrispondente. La seconda parte di ciascun frammento stabilisce che non più di una variabile assume valore 1 (letteralmente, stabilisce che, in ogni coppia di variabili, almeno una assume valore 0) nella cella corrispondente. Questi frammenti sono collegati attraverso operazioni \wedge .

La prima parte di ϕ_{cell} all'interno delle parentesi garantisce che almeno una variabile che è associata con ciascuna cella vale 1, laddove la seconda parte garantisce che non più di una variabile vale 1 per ciascuna cella. Qualsiasi assegnamento alle variabili che soddisfa ϕ (e di conseguenza ϕ_{cell}) deve avere

esattamente una variabile ad 1 per ogni cella. Pertanto, qualsiasi assegnamento che soddisfa ϕ specifica un simbolo in ciascuna cella della tabella. Le parti $\phi_{start}, \phi_{move}$, e ϕ_{accept} garantiscono che questi simboli corrispondano realmente ad un tableau accettante come segue.

La formula ϕ_{start} , assicura che la prima riga della tabella sia la configurazione iniziale di N su w , richiedendo esplicitamente che le variabili corrispondenti siano ad 1:

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}.$$

La formula ϕ_{accept} garantisce che una configurazione accettante sia presente nel tableau. Essa assicura che q_{accept} , il simbolo per lo stato di accettazione, compaia in una delle celle del tableau, richiedendo che una delle variabili corrispondenti sia a 1:

$$\phi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}.$$

Infine, la formula ϕ_{move} garantisce che ciascuna riga del tableau corrisponda ad una configurazione che segue legittimamente dalla configurazione della riga precedente, in accordo alle regole di N . Lo fa assicurando che ogni finestra di celle 2×3 sia lecita. Diciamo che una finestra 2×3 è lecita se non viola le azioni specificate dalla funzione di transizione di N . In altre parole, una finestra è lecita se può esser presente quando una configurazione segue correttamente da un'altra.

Per esempio, siano a, b , e c elementi dell'alfabeto di nastro, e q_1 e q_2 stati di N . Si assuma che quando nello stato q_1 con la testina che legge una a , N scrive una b , resta nello stato q_1 , e muove a destra; e che quando nello stato q_1 con la testina che legge una b , N non deterministicamente

1. scrive una c , entra in q_2 e muove a sinistra, oppure
2. scrive una a , entra in q_2 e muove a destra.

In termini formali, $\delta(q_1, a) = \{(q_1, b, R)\}$ e $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$. Esempi di finestre lecite sono mostrati nella figura seguente.

(a)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>q₁</td><td>b</td></tr><tr><td>q₂</td><td>a</td><td>c</td></tr></table>	a	q ₁	b	q ₂	a	c	(b)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>q₁</td><td>b</td></tr><tr><td>a</td><td>a</td><td>q₂</td></tr></table>	a	q ₁	b	a	a	q ₂	(c)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>a</td><td>q₁</td></tr><tr><td>a</td><td>a</td><td>b</td></tr></table>	a	a	q ₁	a	a	b
a	q ₁	b																					
q ₂	a	c																					
a	q ₁	b																					
a	a	q ₂																					
a	a	q ₁																					
a	a	b																					
(d)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>#</td><td>b</td><td>a</td></tr><tr><td>#</td><td>b</td><td>a</td></tr></table>	#	b	a	#	b	a	(e)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>b</td><td>a</td></tr><tr><td>a</td><td>b</td><td>q₂</td></tr></table>	a	b	a	a	b	q ₂	(f)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>b</td><td>b</td><td>b</td></tr><tr><td>c</td><td>b</td><td>b</td></tr></table>	b	b	b	c	b	b
#	b	a																					
#	b	a																					
a	b	a																					
a	b	q ₂																					
b	b	b																					
c	b	b																					

Figura 36: Finestre lecite

Nella figura le finestre (a) e (b) sono lecite perché la funzione di transizione permette ad N di muovere nella direzione indicata. La finestra (c) è lecita perché, con q_1 presente sul lato destro della riga di sopra, non sappiamo su quale simbolo la testina sia. Detto simbolo potrebbe essere una a , e q_1 potrebbe cambiarlo in una b e muovere a destra. Una tale possibilità genererebbe questa finestra, quindi essa non viola le regole di N . La finestra (d) è ovviamente lecita perché la riga di sopra e quella di sotto sono identiche, situazione che si verifica se la testina non è adiacente alla posizione della finestra. Si noti che $\#$ in una finestra lecita può comparire a sinistra o a destra sia nella riga superiore che in quella inferiore. La finestra (e) è lecita perché lo stato q_1 potrebbe essere immediatamente a destra della riga superiore e la macchina, leggendo una b , si sarebbe mossa a sinistra nello stato q_2 che ora compare all'estremità destra della riga inferiore. Infine, la finestra (f) è lecita perché lo stato q_1 potrebbe essere stato immediatamente a sinistra della riga superiore, e la macchina potrebbe aver cambiato la b in una c ed effettuato una mossa a sinistra.

Le finestre mostrate nella figura seguente non sono lecite per la macchina N .

(a)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>b</td><td>a</td></tr><tr><td>a</td><td>a</td><td>a</td></tr></table>	a	b	a	a	a	a	(b)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>q₁</td><td>b</td></tr><tr><td>q₂</td><td>a</td><td>a</td></tr></table>	a	q ₁	b	q ₂	a	a	(c)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>b</td><td>q₁</td><td>b</td></tr><tr><td>q₂</td><td>b</td><td>q₂</td></tr></table>	b	q ₁	b	q ₂	b	q ₂
a	b	a																					
a	a	a																					
a	q ₁	b																					
q ₂	a	a																					
b	q ₁	b																					
q ₂	b	q ₂																					

Figura 37: Finestre non lecite

Nella finestra (a), il simbolo centrale nella riga superiore non può cambiare perché non c'è uno stato ad esso adiacente. La finestra (b) non è lecita perché la funzione di transizione specifica che la b viene cambiata in una c ma non in una a . La finestra (c) non è lecita perché due stati compaiono nella riga inferiore.

Fatto 7.41

Se la riga superiore del tableau è la configurazione iniziale ed ogni finestra nel tableau è lecita, ciascuna riga del tableau è una configurazione che segue legittimamente dalla precedente.

Proviamo l'asserto considerando ogni coppia di configurazioni adiacenti nel tableau, indicate come la configurazione superiore e la configurazione inferiore. Nella configurazione superiore, ogni cella che contiene un simbolo di nastro e non è adiacente ad un simbolo di stato rappresenta la cella centrale superiore in una finestra in cui la riga superiore non contiene stati. Pertanto, questo stesso simbolo deve comparire al centro in basso nella finestra. Quindi, esso è presente nella stessa posizione nella configurazione inferiore.

La finestra contenente il simbolo di stato nella cella centrale superiore garantisce che le tre posizioni corrispondenti vengano aggiornate consistentemente tramite la funzione di transizione. Pertanto, se la configurazione superiore è una configurazione lecita, altrettanto lo è la configurazione inferiore, e quella inferiore discende da quella superiore in accordo alle regole di N . Si noti che questa dimostrazione, seppur immediata, dipende in maniera cruciale dalla nostra scelta di utilizzare una finestra di dimensione 2×3 .

Torniamo ora alla costruzione di ϕ_{move} . Essa garantisce che tutte le finestre nel tableau sono lecite. Ciascuna finestra contiene sei celle, che possono essere impostate in un numero fissato di modi per produrre una finestra lecita. La formula move stabilisce che le impostazioni delle sei celle devono essere uno di questi modi, ossia

$$\phi_{move} = \bigwedge_{1 \leq i \leq n^k, 1 < j < n^k} (\text{la finestra } (i, j) \text{ è lecita}).$$

La finestra (i, j) ha $cell[i, j]$ in posizione centrale in alto. Sostituiamo il testo "la finestra (i, j) è lecita" in questa formula con la formula seguente. Denotiamo il contenuto delle sei celle di una finestra con a_1, \dots, a_6 .

$$\bigvee_{\substack{a_1, \dots, a_6 \\ \text{è una finestra lecita}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

Nel seguito analizziamo la complessità della riduzione per mostrare che essa opera in tempo polinomiale. Per far ciò, esaminiamo la dimensione di ϕ . Prima di tutto, stimiamo il numero di variabili che ha. Si ricordi che il tableau è una tabella $n^k \times n^k$, quindi contiene n^{2k} celle. Ciascuna cella ha l variabili associate ad essa, dove l è il numero di simboli in C . Poiché l dipende soltanto dalla TM N e non dalla lunghezza dell'input n , il numero totale di variabili è $O(n^{2k})$.

Stimiamo la dimensione di ciascuna delle parti di ϕ . La formula ϕ_{cell} contiene un frammento di dimensione fissata della formula per ciascuna cella del tableau, quindi la sua dimensione è $O(n^{2k})$. La formula ϕ_{start} ha un frammento per ciascuna cella nella riga superiore, quindi la sua dimensione è $O(n^k)$. Le formule ϕ_{move} e ϕ_{accept} contengono ciascuna un frammento di dimensione fissata della formula per ciascuna cella del tableau, quindi la loro dimensione è $O(n^{2k})$. Pertanto, la dimensione complessiva di ϕ è $O(n^{2k})$. Tale limite è sufficiente per i nostri scopi perché dimostra che la dimensione di ϕ è polinomiale in n . Se fosse stata più che polinomiale, la riduzione non avrebbe avuto alcuna possibilità di generarla in tempo polinomiale. (In realtà, le nostre stime sono più basse di un fattore $O(\log n)$, poiché ciascuna variabile ha indici che possono arrivare fino a n^k e, quindi, possono richiedere $O(\log n)$ simboli da scrivere nella formula, ma questo fattore addizionale non cambia la polinomialità del risultato.) Per rendersi conto che la formula può essere generata in tempo polinomiale, si noti la sua natura altamente ripetitiva. Ciascun componente della formula è composto da molti frammenti quasi identici, che differiscono

soltanto negli indici in modo molto semplice. Pertanto, possiamo costruire facilmente una riduzione che produce o in tempo polinomiale dall'input w .

Quindi, abbiamo concluso la dimostrazione del teorema di Cook e Levin, mostrando che SAT è NP-completo. Mostrare la NP-completezza di altri linguaggi generalmente non richiede una dimostrazione così lunga. Al contrario, la NP-completezza può essere provata con una riduzione di tempo polinomiale da un linguaggio che è già noto essere NP-completo. Possiamo usare SAT per questo scopo; ma usare $3SAT$, il caso speciale di SAT è solitamente più facile. Si ricordi che le formule appartenenti a $3SAT$ sono in forma normale congiuntiva (cnf) con tre letterali per clausola. Prima di tutto, dobbiamo dimostrare che $3SAT$ stesso è NP-completo. Proviamo questo asserto come corollario del [Teorema 7.37](#).

Corollario 7.42

$3SAT$ è NP-completo.

DIMOSTRAZIONE. Ovviamente $3SAT$ appartiene a NP, quindi dobbiamo provare solamente che tutti i linguaggi appartenenti a NP si riducono a $3SAT$ in tempo polinomiale. Un modo per farlo è facendo vedere che SAT si riduce in tempo polinomiale a $3SAT$. Invece, preferiamo farlo modificando la dimostrazione del [Teorema 7.37](#), in modo tale che produca direttamente una formula in forma normale congiuntiva con tre letterali per clausola.

Il [Teorema 7.37](#) produce una formula che è già quasi in forma congiuntiva normale. La formula ϕ_{cell} è un grosso AND di sottoformule, ciascuna delle quali contiene un grosso OR ed un grosso AND di diversi OR. Quindi, ϕ_{cell} è un AND di clausole e, di conseguenza, è già in forma cnf. La formula ϕ_{start} è un grosso AND di variabili. Prendendo ciascuna di queste variabili come clausole di dimensione 1, notiamo che ϕ_{start} è in forma cnf. La formula ϕ_{accept} è un grosso OR di variabili, ed è perciò una singola clausola. La formula ϕ_{move} è l'unica che non è già in forma cnf, ma possiamo facilmente convertirla in una formula che è in forma cnf come segue.

Si ricordi che ϕ_{move} è un grosso AND di sottoformule, ciascuna delle quali è un OR di diversi AND che descrive tutte le possibili finestre lecite. La legge distributiva, stabilisce che possiamo sostituire un OR di vari AND con un equivalente AND di vari OR. Fare ciò può incrementare significativamente la dimensione di ciascuna sottoformula, ma tale operazione può incrementare la dimensione complessiva di ϕ_{move} solamente di un fattore costante, poiché la dimensione di ciascuna sottoformula dipende solo da N . Il risultato è una formula che è in forma normale congiuntiva.

Ora che abbiamo scritto la formula in forma cof, la convertiamo in una con tre letterali per clausola. In ciascuna clausola che correntemente ha uno o due letterali, duplichiamo uno dei letterali, fino a quando il numero totale diventa tre. Ciascuna clausola che ha più di tre letterali la dividiamo in più clausole e aggiungiamo ulteriori variabili per preservare la soddisfacibilità o non soddisfacibilità della clausola originale. Per esempio, sostituiamo la clausola $(a_1 \vee a_2 \vee a_3 \vee a_4)$, dove ciascun a_i è un eletterale, con l'espressione di due clausole $(a_1 \vee a_2 \vee z)$ e $(\bar{z} \vee a_3 \vee a_4)$, dove z è una nuova variabile. Se qualche impostazione degli a_i soddisfa la clausola originale, possiamo trovare una impostazione di z in modo tale che le due nuove clausole siano soddisfatte e viceversa. In generale, se la clausola contiene l letterali,

$$(a_1 \vee a_2 \vee \dots \vee a_l)$$

possiamo sostituirla con le $l - 2$ clausole

$$(a_1 \vee a_2 \vee z_1) \wedge (\bar{z}_1 \vee a_3 \vee z_2) \wedge (\bar{z}_2 \vee a_4 \vee z_3) \wedge \dots \wedge (\bar{z}_{l-3} \vee a_{l-1} \vee a_l).$$

È possibile verificare facilmente che la nuova formula è soddisfacibile se e solo se la formula originale lo era, quindi la dimostrazione è completa.

7.5 Ulteriori problemi NP-completi

Per mostrare che un problema C è NPC , dobbiamo:

1. Provare che appartiene a NP ; e
2. Trovare un $B \in NPC$ tale che $B \leq_P C$.

Noi applicheremo questo procedimento per dimostrare che tre problemi sono NPC , partendo dal fatto che $SAT \in NPC$.

In particolare:

$$SAT \leq_P 3\text{-CNF-SAT} \leq_P CLIQUE \leq_P VERTEX-COVER$$

Ad oggi, i problemi NPC sono molte migliaia, alcuni anche molto lontani da SAT (originariamente mostrato essere completo per NP).

- Un **letterale** in una formula booleana è un'occorrenza di una variabile o della sua negazione.
- Una formula booleana è in **forma normale congiuntiva (CNF)** se è espressa come un AND di clausole, ognuna delle quali è un OR di uno o più letterali.
- **OSSERVAZIONE:** Un assegnamento soddisfa una formula in CNF se e solo se ogni clausola contiene almeno un letterale il cui valore è 1.
- Una formula booleana è in **3-CNF** se ogni clausola ha esattamente tre letterali distinti.

Definizione:

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

Teorema 34.10(24-30)

$3\text{-CNF-SAT} \in NPC$.

DIMOSTRAZIONE:

- $3\text{-CNF-SAT} \in NP$: il certificato è l'assegnamento; la verifica consiste nel sostituire ogni variabile con il valore assegnatole e nel calcolare il valore della formula.
- Mostriamo ora che $SAT \leq_P 3\text{-CNF-SAT}$, cioè che ogni formula booleana ϕ può essere portata in forma 3-CNF senza modificarne la soddisfacibilità.

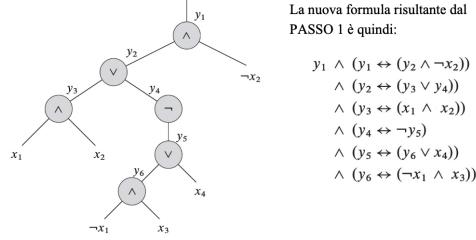
Passo 1: Trasformazione in Forma 3-CNF

1. Costruiamo un albero di parsing binario per la formula in input, con letterali come foglie e connettivi come nodi interni.
2. Se la formula contiene una clausola che è l'OR/AND di più letterali, usiamo l'associatività per parentesizzare l'espressione, in modo che ogni nodo interno dell'albero risultante abbia al più 2 figli.
3. Introduciamo una (nuova) variabile y_i per l'output di ogni nodo interno.
4. La nuova formula ϕ' è la congiunzione tra la variabile alla radice dell'albero ed una sottoformula per ogni nodo interno (che ne descrive l'operazione).
5. Nota: Ogni sottoformula ha al più 3 letterali.

Esempio:

ESEMPIO: Consideriamo la formula $((x_1 \wedge x_2) \vee \neg((\neg x_1 \wedge x_3) \vee x_4)) \wedge \neg x_2$

Il suo albero sintattico (i cui nodi sono già annotati) è:



Passo 2: Conversione in CNF

- Siccome ϕ' è già una congiunzione di sottoformule, basta convertire ognuna in CNF.
- Ogni sottoformula ha al più 3 letterali, possiamo costruire la tavola di verità per ognuna di esse (con al più 8 righe ognuna) e applicare la procedura per il calcolo della forma normale congiuntiva.

Esempio:

ESEMPIO: Consideriamo la sottoformula $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

e quindi la CNF associata è

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Passo 3: Uniformazione delle Clausole

- Per ogni clausola C in ϕ'' , trasformiamo ϕ'' in ϕ''' in cui tutte le clausole hanno esattamente 3 letterali:
 - Se C ha 3 letterali, includiamo C in ϕ''' .
 - Se $C = l_1 \vee l_2$, includiamo $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ in ϕ''' .
 - Se $C = l$, includiamo $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ in ϕ''' .

Conclusione:

- Ogni passo preserva la soddisficiabilità.
- Ogni passo può essere fatto in tempo polinomiale:
 - Ottenere ϕ' da ϕ introduce al più 1 variabile e 1 clausola per ogni connettivo in ϕ
 - Ottenere ϕ'' da ϕ' può introdurre al più 8 clausole per ogni clausola di ϕ'
 - Ottenere ϕ''' da ϕ'' introduce al più 4 clausole per ogni clausola di ϕ''
- Quindi, la dimensione di ϕ''' è polinomiale nella dimensione di ϕ .

7.5.1 CLIQUE

Definizione: Dato un grafo non diretto $G = (V, E)$, una *clique* è $V' \subseteq V$ tale che ogni coppia di vertici in V' è connessa da un arco di E .

- La dimensione di una *clique* è il numero di vertici che contiene.

- Il problema della *clique* consiste nel trovare una *clique* di dimensione massima.
- Questo problema può essere tradotto in un problema decisionale chiedendo se esiste una *clique* di una data dimensione k .

Definizione Formale:

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ è un grafo non diretto che contiene una clique di dimensione } k\}.$$

Teorema 34.11(24-30)

$CLIQUE \in NPC$.

DIMOSTRAZIONE:

- $CLIQUE \in NP$: il certificato è il sottinsieme V' ; la verifica consiste nel controllare che ogni $v \in V'$ appartenga a V , che V' abbia dimensione k , e che, per ogni $u, v \in V'$, $\{u, v\}$ appartenga a E . Ciò richiede $O(|V|^2)$ operazioni.

Mostriamo che $3-CNF-SAT \leq_P CLIQUE$, cioè che, data una formula $3-CNF \phi$, si trova (in tempo polinomiale) una coppia (G, k) tale che ϕ è soddisfacibile se e solo se G ammette una clique di dimensione k .

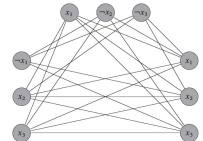
Costruzione del Grafo: Sia $\phi = C_1 \wedge \dots \wedge C_k$, dove ogni $C_r = (l_1^r \vee l_2^r \vee l_3^r)$.

Definiamo $G = (V, E)$ come

- $V = \bigcup_{r=1}^k \{v_1^r, v_2^r, v_3^r\}$ //un vertice per ogni letterale di ogni clausola
- $E = \{\{v_i^r, v_j^s\} \mid r \neq s \text{ e } l_i^r \neq \neg l_j^s\}$ //un arco tra ogni coppia di letterali che stanno in clausole diverse e non sono una la negazione dell'altro

Esempio:

ESEMPIO: consideriamo la formula $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
Il grafo ad essa associato è



Dimostrazione della corrispondenza tra ϕ e G :

1. Assumiamo che ϕ abbia un assegnamento che la soddisfa:

- Ogni clausola C_r contiene almeno un letterale l_i^r che vale 1.
- Consideriamo l'insieme formato scegliendo esattamente un tale letterale da ogni clausola.
- Questo forma un sottinsieme V' di V di dimensione k . Mostriamo che V' è una clique:
 - Per ogni coppia di vertici $v_i^r, v_j^s \in V'$, per costruzione di V' abbiamo che $r \neq s$ e che l_i^r ed l_j^s valgono entrambi 1 (quindi, i letterali non sono uno la negazione dell'altro).
 - Per costruzione di G , l'arco $\{v_i^r, v_j^s\}$ appartiene a E .

2. Assumiamo che G abbia una clique V' di dimensione k :

- Nessun arco di G connette vertici della stessa tripla, quindi V' contiene esattamente un vertice per ogni tripla.
- Assegniamo 1 ad ogni letterale l_i^r tale che $v_i^r \in V'$ (variabili che non corrispondono a vertici della clique possono assumere qualsiasi valore).

- Non assegneremo mai 1 ad un letterale e al suo negato, siccome G non ha archi tra letterali complementari.
- Ogni clausola è soddisfatta, quindi ϕ è soddisfacibile.

Conclusione: $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$. Q.E.D.

Esempio:

Torniamo all'esempio della formula

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Il grafo ad essa associato è:

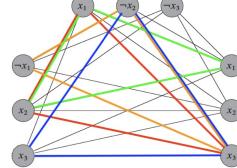


Figura 38: Esempio di clique

Ogni clique di dimensione 3 corrisponde ad un assegnamento che soddisfa la formula, e viceversa. Per esempio:

- La clique identificata dagli archi in rosso $\Rightarrow x_1 = x_2 = x_3 = 1$.
- La clique identificata dagli archi in verde $\Rightarrow x_1 = x_2 = 1$ (x_3 può avere qualsiasi valore).
- La clique identificata dagli archi in arancio $\Rightarrow x_1 = x_2 = 0$ e $x_3 = 1$.
- La clique identificata dagli archi in blu $\Rightarrow x_2 = 0$ e $x_3 = 1$ (x_1 può avere qualsiasi valore).

7.5.2 VERTEX-COVER

Definizione: Dato un grafo non diretto $G = (V, E)$, un *vertex cover* è un $V' \subseteq V$ tale che, per ogni $\{u, v\} \in E$, o $u \in V'$ o $v \in V'$ (o entrambe).

- Ogni vertice copre gli archi che lo hanno come estremo; un *vertex cover* è un insieme che copre tutti gli archi.

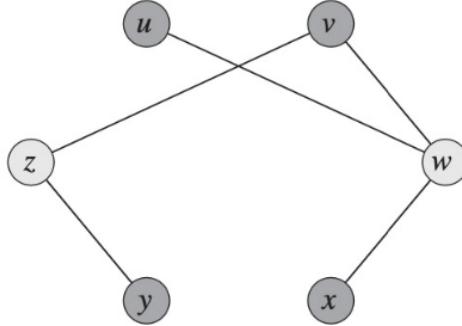


Figura 39: Esempio di vertex cover

- L'insieme $\{w, z\}$ è un *vertex-cover*.

La dimensione di un *vertex-cover* è il numero di vertici che contiene.

Problema decisionale:

- Il problema del *vertex-cover* è di trovare un *vertex cover* di dimensione minima.
- Può essere tradotto in un problema decisionale, chiedendo se un grafo ha un *vertex cover* di una certa dimensione k .

Definizione Formale:

$$\text{VERTEX-COVER} = \{\langle G, k \rangle \mid G \text{ è un grafo non diretto con un vertex cover di dimensione } k\}.$$

Teorema 34.12(oramai)
 $\text{VERTEX-COVER} \in \text{NPC}$.

DIMOSTRAZIONE:

1. $\text{VERTEX-COVER} \in \text{NP}$: il certificato è il sottinsieme V' . La verifica consiste nel controllare che ogni $v \in V'$ appartiene a V , che V' ha dimensione k , e che, per ogni $\{u, v\} \in E$, o u o v appartiene a V' . Ciò richiede $O(|V| + |E|)$ operazioni.
2. Mostriamo che $\text{CLIQUE} \leq_P \text{VERTEX-COVER}$:

- Dato un grafo $G = (V, E)$ e un intero k , definiamo una nuova coppia (G', k') tale che G ha una clique di dimensione k se e solo se G' ha un *vertex cover* di dimensione k' .
- G' è il complemento di G , con $G' = (V, \bar{E})$ dove $\bar{E} = \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$.

Dato (G, k) per CLIQUE , la coppia da considerare per VERTEX-COVER è $(\bar{G}, |V| - k)$.

1. Assumiamo che G abbia una clique V' di dimensione k :

- Mostriamo che $V \setminus V'$ è un *vertex cover* per \bar{G} .
- Sia $\{u, v\}$ un qualsiasi arco in \bar{E} . Allora:
 - Per definizione, $\{u, v\} \notin E$.
 - Essendo V' una clique, almeno uno tra u e v non appartiene a V' , poiché ogni coppia di vertici di V' è connessa da un arco di E .
 - Quindi, almeno uno tra u e v appartiene a $V \setminus V'$, come richiesto.
- Infine, $|V \setminus V'| = |V| - k$, essendo $|V'| = k$.

2. assumiamo che \bar{G} ha un *vertex cover* V' di dimensione $|V| - k$:

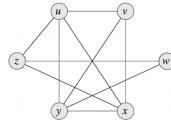
- Mostriamo che $V \setminus V'$ è una clique per G .
- Sia u, v una qualsiasi coppia di vertici distinti in $V \setminus V'$. Allora:
 - $\{u, v\}$ non può appartenere a \bar{E} , altrimenti non sarebbe coperto da V' (che invece è un *vertex cover*).
 - Quindi, per definizione, $\{u, v\} \in E$, come richiesto.
- Infine, $|V \setminus V'| = k$, essendo $|V'| = |V| - k$.

Conclusione: $\text{CLIQUE} \leq_P \text{VERTEX-COVER}$. Q.E.D.

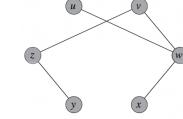
Esempio:

- Consideriamo il grafo G .
- Il suo complemento G' è rappresentato nella figura seguente

ESEMPIO: Consideriamo il grafo



Il suo complemento è il grafo



Il grafo di sinistra ha una clique di dimensione 4 (cioè $\{u,v,x,y\}$); essa corrisponde a un vertex cover di dimensione 2 nel grafo complemento (cioè $\{w,z\}$):

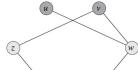
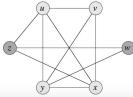


Figura 40: Esempio di vertex cover

7.6 Approssimazione

Molti problemi pratici sono *NPC*, ma troppo rilevanti per essere abbandonati solo perché non sappiamo come trovare una soluzione ottimale in tempo polinomiale.

Tre approcci per aggirare l'NP-completezza:

1. Utilizzare un algoritmo di tempo esponenziale per input sufficientemente piccoli.
2. Isolare sottocasi importanti risolvibili in tempo polinomiale.
3. Trovare soluzioni vicine all'ottimo in tempo polinomiale (sia nel caso pessimo che medio).

Un algoritmo che restituisce soluzioni prossime all'ottimo sarà chiamato **algoritmo di approssimazione**.

Approximation Ratio

Immaginiamo di avere un problema di ottimizzazione in cui ogni potenziale soluzione ha un costo positivo.

Definizione

Un algoritmo ha un *approximation ratio* di $\rho(n)$ se, per ogni input di dimensione n , il costo C della soluzione prodotta differisce dal costo C^* della soluzione ottimale per al più un fattore $\rho(n)$:

$$\text{Per un problema di massimizzazione: } 0 < C \leq C^*, \quad \rho(n) = \frac{C^*}{C}.$$

$$\text{Per un problema di minimizzazione: } 0 < C^* \leq C, \quad \rho(n) = \frac{C}{C^*}.$$

Se un algoritmo raggiunge un *approximation ratio* di $\rho(n)$, lo chiameremo un $\rho(n)$ -**algoritmo di approssimazione**.

Nota Bene:

- L'approximation ratio non è mai minore di 1.
- Un 1-algoritmo di approssimazione produce una soluzione ottimale.
- Un approximation ratio maggiore restituisce una soluzione peggiore dell'ottimale.

Per molti problemi, abbiamo algoritmi di approssimazione polinomiali con approximation ratios che sono costanti (anche piccole).

Per altri problemi, i migliori algoritmi di approssimazione polinomiali hanno approximation ratio che crescono in funzione della dimensione dell'input n .

Alcuni problemi NPC ammettono algoritmi di approssimazione polinomiali che possono raggiungere approssimazioni sempre migliori usando più tempo computazionale.

⇒ si può negoziare il tempo computazionale per la qualità dell'approssimazione.

7.6.1 Vertex Cover

Definizione: Un *vertex cover* in un grafo non diretto $G = (V, E)$ è un sottinsieme $V' \subseteq V$ tale che, per ogni $\{u, v\} \in E$, o $u \in V'$ o $v \in V'$ (o entrambe).

- La dimensione di un vertex cover è $|V'|$.
- Il problema consiste nel trovare un vertex cover di dimensione minima, detto *optimal vertex cover*.

Osservazione

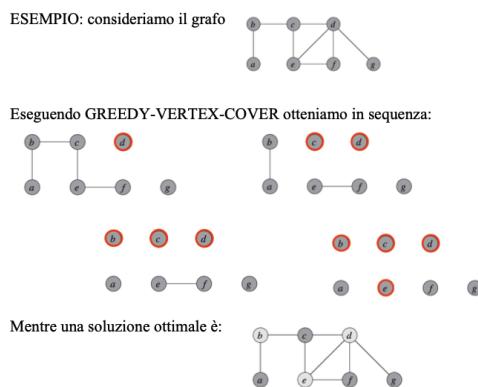
Questo problema è la versione ottimizzata di *VERTEX-COVER* ($\in NPC$), quindi non si può risolvere in tempo polinomiale a meno che $P = NP$.

IDEA: Ad ogni passo, selezioniamo il nodo che «copre» più archi.

```

GREEDY-VERTEX-COVER( $G$ )
 $C \leftarrow \emptyset$ 
while  $E \neq \emptyset$ 
    Pick a vertex  $v \in V$  of
        maximum degree in the current graph
     $C \leftarrow C \cup \{v\}$ 
     $E \leftarrow E \setminus \{e \in E : v \in e\}$ 
return  $C$ 
```

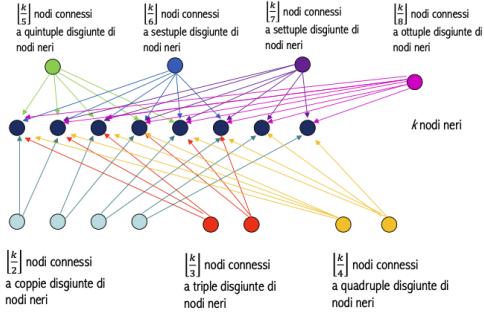
Esempio:



In questo esempio, il rapporto tra la soluzione calcolata dall'algoritmo e l'ottimo è 4/3.

⇒ ma questo deve valere per ogni grafo di input!

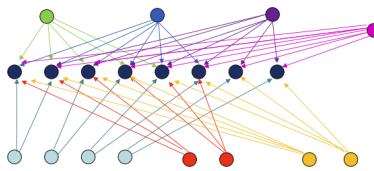
CONTROESEMPIO: consideriamo il seguente grafo:



Per applicare *GREEDY-VERTEX-COVER* a questo grafo:

1. Selezioniamo i nodi con il grado maggiore, in ordine decrescente.
2. Ripetiamo fino a coprire tutti gli archi.

1. Selezioniamo il nodo rosa (grado=8), poiché tutti gli altri nodi hanno grado al più 7
2. Poi selezioniamo il nodo viola (grado=7), poiché tutti gli altri nodi hanno grado al più 6
3. Poi il nodo blu (grado=6), poiché tutti gli altri nodi hanno grado al più 5



4. Poi il nodo verde (grado=5), poiché tutti gli altri nodi hanno grado al più 4
5. Poi i due nodi arancioni in qualsiasi ordine (grado=4)
6. Poi i nodi rossi (in qualsiasi ordine)
7. Infine i nodi azzurri (in qualsiasi ordine)



Quindi, l'algoritmo *GREEDY-VERTEX-COVER* restituisce tutti i nodi colorati, che sono

$$\sum_{i=2}^k \lfloor \frac{k}{i} \rfloor \geq k(\ln k - 2).$$

In realtà, il vertex cover ottimale sarebbe quello formato solo dai nodi neri (quindi, di dimensione k). Quindi, in questo caso l'approximation ratio è almeno $O(\log k)$ (dove k è una funzione di n).

N.B.: si può provare che questo upper bound è preciso, quindi questo è il caso pessimo.

⇒ si può fare meglio con un *approximation ratio* costante.

Esempio di un algoritmo migliorato:

Consideriamo il seguente
algoritmo:

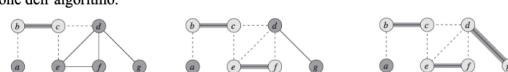
APPROX-VERTEX-COVER(G)

- 1 $C = \emptyset$
- 2 $E' = G.E$
- 3 **while** $E' \neq \emptyset$
- 4 let (u, v) be an arbitrary edge of E'
- 5 $C = C \cup \{u, v\}$
- 6 remove from E' every edge incident on either u or v
- 7 **return** C

ESEMPIO:



Applicazione dell'algoritmo:



Vertex cover restituito:



Teorema 35.1(orale)

APPROX-VERTEX-COVER è un 2-algoritmo di approssimazione polinomiale.

DIMOSTRAZIONE:

- *APPROX-VERTEX-COVER* richiede tempo $O(|V| + |E|)$:
 - Ogni nodo è aggiunto a C al massimo una volta.
 - Ogni arco è rimosso da E' al massimo una volta (o perchè selezionato o perchè cancellato).
- L'insieme C restituito è un vertex cover, siccome l'algoritmo cicla finchè ogni arco di E' è stato coperto da un vertice in C .
- *APPROX-VERTEX-COVER* restituisce un insieme la cui dimensione è al più doppia della dimensione di un vertex cover ottimo:
 - Sia A l'insieme degli archi scelti nella linea4 (dello pseudo-codice).
 - Poiché non esistono coppie in A con estremi condivisi, $|C| = 2|A|$.
 - Ogni vertex cover ottimale C^* deve includere almeno un estremo per ogni arco di A : $|A| \leq |C^*|$.
 - Quindi, $|C| = 2|A| \leq 2|C^*|$.

8 Complessità di Spazio

In questo capitolo valutiamo la complessità dei problemi computazionali in termini della quantità di spazio, o memoria, che essi richiedono. Tempo e spazio sono due dei fattori più importanti quando ricerchiamo soluzioni pratiche per numerosi problemi computazionali. La complessità di spazio condivide molte caratteristiche della complessità di tempo e rappresenta un modo ulteriore per classificare i problemi in accordo alla propria difficoltà computazionale.

Come per la complessità di tempo, abbiamo necessità di selezionare un modello per misurare lo spazio utilizzato da un algoritmo. Continuiamo a servirci del modello della macchina di Turing per la stessa ragione per cui l'abbiamo usato per misurare il tempo. Le macchine di Turing sono matematicamente semplici ed abbastanza vicine ai calcolatori reali da fornire risultati significativi.

Definizione 8.1

Sia M una macchina di Turing deterministica che si arresta su tutti gli input. La **complessità di spazio** di M è la funzione $f : \mathbb{N} \rightarrow \mathbb{N}$, dove $f(n)$ è il numero massimo di celle del nastro che M scandisce su ogni input di lunghezza n . Se la complessità di spazio di M è $f(n)$, diciamo anche che M computa in spazio $f(n)$. Se M è una macchina di Turing non deterministica dove tutte le diramazioni si arrestano su tutti gli input, definiamo la sua complessità di spazio $f(n)$ come il massimo numero di celle del nastro che M scandisce su qualsiasi diramazione della sua computazione per ogni input di lunghezza n .

Tipicamente stimiamo la complessità di spazio delle macchine di Turing attraverso la notazione asintotica.

Definizione 8.2

Sia $f : \mathbb{N} \rightarrow \mathbb{R}^+$ una funzione.

$\text{SPACE}(f(n)) = \{L \mid L \text{ è un linguaggio deciso da una macchina di Turing deterministica con spazio } O(f(n))\}$.

$\text{NSPACE}(f(n)) = \{L \mid L \text{ è un linguaggio deciso da una macchina di Turing non deterministica con spazio } O(f(n))\}$.

ESEMPIO 8.3

Nel Capitolo 7 abbiamo introdotto il problema NP-completo *SAT*. Qui mostriamo che *SAT* può essere risolto con un algoritmo che richiede una quantità di spazio lineare. Ritroviamo che *SAT* non può essere risolto con un algoritmo di tempo polinomiale, ma è meno chiaro se l'algoritmo di tempo lineare, perché *SAT* è NP-completo. Lo spazio sembra essere più potente del tempo perché lo spazio può essere riutilizzato, mentre il tempo no.

- M = “Su input $\langle \phi \rangle$, dove ϕ è una formula booleana:
1. Per ogni assegnamento di verità alle variabili x_1, \dots, x_m di ϕ :
 2. Verifica ϕ sull’assegnamento di verità.
 3. Se ϕ vale 1, accetta; altrimenti, rifiuta.”

La macchina M_1 chiaramente computa in spazio lineare perché ad ogni iterazione del ciclo si legge la stessa porzione del nastro. La macchina ha necessità di memorizzare soltanto l’assegnamento di verità corrente, e ciò può esser fatto con spazio $O(n)$. Il numero di variabili m è al più n , la lunghezza dell’input, pertanto la macchina computa in spazio $O(n)$.

ESEMPIO 8.4

In questo esempio illustriamo la complessità di spazio non deterministica di un linguaggio. Nella prossima sezione mostreremo come determinare la complessità di spazio non deterministica possa essere utile per determinare la sua complessità di spazio deterministico. Si consideri il problema di

verificare se un automa finito non deterministico accetta tutte le stringhe.

Sia

$$ALL_{\text{NFA}} = \{\langle A \rangle \mid A \text{ è un NFA e } L(A) = \Sigma^*\}.$$

Forniamo un algoritmo non deterministico lineare in spazio che decide il complemento di questo linguaggio, $\overline{ALL}_{\text{NFA}}$. L’idea che sta dietro all’algoritmo è di usare il non determinismo per individuare una stringa che viene rifiutata dall’NFA, e di usare una quantità di spazio lineare per tener traccia degli stati in cui l’NFA potrebbe trovarsi in un certo istante. Si noti che l’appartenenza di questo linguaggio a NP o a coNP non è nota.

$N =$ “Su input $\langle M \rangle$, dove M è un NFA:

1. Poni un marcatori sullo stato iniziale dell’NFA.
2. Ripeti 2 q volte, dove q è il numero di stati di M :
3. Non contiene M di scelta: seleziona un simbolo di input e cambia le posizioni dei marcatori sugli stati di M per simulare la lettura del simbolo.
4. Accetta se i passi 2 e 3 individuano una stringa che M rifiuta; cioè, se ad un certo punto nessuno dei marcatori si trova su stati finali di M . Altrimenti, rifiuta.”

Se M rifiuta stringhe, no deve rifiutare di lunghezza al più 2 q , perché in ogni stringa più lunga che viene rifiutata, le locazioni dei marcatori determinati nell’algoritmo precedente dovrebbero ripetersi. La parte della stringa tra le ripetizioni può essere rimossa per ottenere una stringa rifiutata più corta. Pertanto, N decide $\overline{ALL}_{\text{NFA}}$. (Si noti che N accetta anche input formati impropriamente.)

Il solo spazio richiesto da questo algoritmo serve per memorizzare la locazione dei marcatori e per il contatore usato nel ciclo, e ciò può esser fatto in spazio lineare. Quindi, l’algoritmo computa con complessità di spazio non deterministica $O(n)$. Di seguito proviamo un teorema che fornisce informazioni sulla complessità di spazio deterministico di ALL_{NFA} .

8.1 Teorema di Savitch (e mo so cazzo pt.2)

Il teorema di Savitch è uno dei primi risultati riguardanti la complessità di spazio. Mostra che le macchine deterministiche possono simulare le macchine non deterministiche usando una quantità di spazio sorprendentemente molto piccola. Per la complessità di tempo, tale simulazione sembra richiedere un incremento esponenziale in tempo. Per la complessità di spazio, il teorema di Savitch mostra che ogni TM non deterministica che usa spazio $f(n)$ può essere convertita in una TM deterministica che usa soltanto spazio $f^2(n)$.

Teorema 8.5(30L)

Teorema di Savitch

Per ogni funzione $f : \mathbb{N} \rightarrow \mathbb{R}^+$ dove $f(n)$, dove $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

IDEA: Dobbiamo simulare deterministicamente una NTM con complessità di spazio $f(n)$. Un approccio ingenuo consiste nel procedere tentando tutte le diramazioni seguite dalla computazione della NTM, una per una. La simulazione deve tener traccia di quale diramazione la macchina sta tentando al momento per poter passare alla successiva. Ma una diramazione che usa spazio $f(n)$ può richiedere $2^{O(f(n))}$ passi e ciascun passo potrebbe consistere in una scelta non deterministica. L’esplorazione sequenziale delle diramazioni richiederebbe la memorizzazione di tutte le scelte effettuate su una particolare diramazione per poter individuare la diramazione successiva. Pertanto, questo approccio può usare spazio $2^{O(f(n))}$, eccedendo il nostro obiettivo di usare spazio $O(f^2(n))$.

Invece, usiamo un approccio diverso, considerando il problema più generale che segue. Date due configurazioni della NTM, c_1 e c_2 , e un numero t , verifichiamo se la NTM può transire da c_1 a c_2 in al più t passi, usando soltanto spazio $f(n)$. Chiamiamo questo problema il **problema della resa**. Risolvendo il problema della resa, dove c_1 è la configurazione iniziale, c_2 è la configurazione finale, e t è il numero massimo di passi che la macchina non deterministica può effettuare, possiamo stabilire se la macchina accetta il proprio input.

A tal fine forniamo un algoritmo ricorsivo deterministico che risolve il problema della resa. L’algoritmo opera cercando una configurazione intermedia c_m : e verificando ricorsivamente se (1) c_1 può portare a c_m in al più $t/2$ passi, e (2) se c_m può portare a c_2 in al più $t/2$ passi. La riutilizzazione dello spazio per ognuno dei due test ricorsivi permette un risparmio di spazio significativo.

Questo algoritmo richiede spazio per memorizzare le chiamate ricorsive. Ciascun livello della ricorsione usa spazio $O(f(n))$ per memorizzare una configurazione. La profondità della ricorsione è $\log t$, dove t è il tempo massimo che la macchina non deterministica può usare su ogni diramazione. Risulta $t = 2^{O(f(n))}$, quindi $\log t = O(f(n))$. Pertanto la simulazione deterministica usa spazio $O(f^2(n))$.

DIMOSTRAZIONE: Sia N una NTM che decide il linguaggio A in spazio $f(n)$. Costruiamo una TM deterministica M che decide A . La macchina M utilizza la procedura $CANYIELD$, che verifica se una delle configurazioni di N può darne un'altra entro uno specificato numero di passi. Questa procedura risolve il problema della resa descritto nell'idea della prova.

Sia w una stringa di input per N . Per le configurazioni c_1 e c_2 di N , e l'intero t , $CANYIELD(c_1, c_2, t)$ da in output *accetta* se N può passare dalla configurazione c_1 alla configurazione c_2 in t o meno passi attraverso un qualche cammino non deterministico. Altrimenti, $CANYIELD$ da in output *rifiuta*. Per convenienza, assimiamo che t sia una potenza di 2.

$CANYIELD =$ "Su input c_1, c_2, t :

1. Se $t = 1$, allora verifica direttamente se $c_1 = c_2$ o se c_1 può passare a c_2 in un passo, in accordo alle regole di transizione di N . *Accetta* se uno dei due test ha successo; altrimenti, *rifiuta*.
2. Se $t > 1$, allora per ogni configurazione c_m di N usando spazio $f(n)$:
3. Esegui $CANYIELD(c_1, c_m, t/2)$.
4. Esegui $CANYIELD(c_m, c_2, t/2)$.
5. Se i passi 3 e 4 sono entrambi accettanti, allora *accetta*.
6. Se non ha ancora accettato, *rifiuta*."

Ora definiamo M per simulare N come segue. Prima modifichiamo N in modo tale che, quando accetta, cancella il proprio nastro e muove la testina sulla cella più a sinistra - entrando con ciò in una configurazione detta c_{accept} . Denotiamo con c_{start} la configurazione iniziale di N su w . Selezioniamo una costante d in modo tale che N abbia al più $2^{df(n)}$ configurazioni usando un nastro di $f(n)$ celle, dove n è la lunghezza di w . Allora sappiamo che $2^{df(n)}$ fornisce un limite superiore al tempo di esecuzione di ogni diramazione di N su w .

$M =$ "Su input w :

1. Fornisci in output il risultato di $CANYIELD(c_{start}, c_{accept}, 2^{df(n)})$.

L'algoritmo $CANYIELD$ ovviamente risolve il problema della resa e, di conseguenza, M simula correttamente N . Dobbiamo analizzarne l'esecuzione per verificare che M lavora in spazio $O(f^2(n))$.

Ogni qualvolta $CANYIELD$ invoca se stessa ricorsivamente, memorizza il numero della fase corrente e i valori di c_1, c_2 , e t in una pila, in modo tale che questi valori possano essere recuperati dalla chiamata ricorsiva. Ciascun livello della ricorsione usa perciò uno spazio aggiuntivo pari a $O(f(n))$. Inoltre, ciascun livello della ricorsione divide la taglia di t a metà. Inizialmente t è uguale a $2^{df(n)}$, e quindi la profondità della ricorsione è $O(\log 2^{df(n)})$ ovvero $O(f(n))$. Pertanto, lo spazio totale usato è $O(f^2(n))$, come asserito. Una difficoltà tecnica sorge in questo argomento perché l'algoritmo M deve conoscere il valore di $f(n)$ quando invoca $CANYIELD$. Possiamo gestire questa difficoltà modificando M in modo tale che tenti con $f(n) = 1, 2, 3, \dots$. Per ciascun valore $f(n) = i$, l'algoritmo modificato usa $CANYIELD$ per stabilire se la configurazione di accettazione è raggiungibile. In aggiunta, l'algoritmo usa $CANYIELD$ per stabilire se N usa almeno spazio $i + 1$ verificando se N può raggiungere qualcuna delle configurazioni di lunghezza $i + 1$ dalla configurazione iniziale.

Se la configurazione finale è raggiungibile, M accetta; se nessuna configurazione di lunghezza $i + 1$ è raggiungibile, M rifiuta; e altrimenti, M continua con $f(n) = i + 1$. (Avremmo potuto gestire questa difficoltà in un altro modo, assumendo che M possa calcolare $f(n)$ in spazio $O(f(n))$, ma poi avremmo dovuto aggiungere questa assunzione all'enunciato del teorema.)

8.2 LA CLASSE PSPACE

In analogia con la classe P, definiamo la classe PSPACE per la complessità di spazio.

Definizione 8.6

Definizione

PSPACE è la classe dei linguaggi che sono decidibili in spazio polinomiale con una macchina di Turing deterministica.

In altre parole,

$$PSPACE = \bigcup_k SPACE(n^k).$$

Definiamo NPSPACE, la controparte non deterministica di PSPACE, in termini delle classi NSPACE. Tuttavia, in virtù del teorema di Savitch, PSPACE = NPSPACE, perché il quadrato di un polinomio è ancora un polinomio.

Negli esempi 8.3 e 8.4, abbiamo mostrato che *SAT* è contenuto in $SPACE(n)$ e che *ALL_NFA* è contenuto in $coNSPACE(n)$ e, quindi, per il teorema di Savitch, in $SPACE(n^2)$, perché le classi di complessità di spazio deterministiche sono chiuse rispetto al complemento. Pertanto, entrambi i linguaggi sono in PSPACE.

Esaminiamo ora la relazione di PSPACE con P ed NP. Osserviamo che $P \subseteq PSPACE$, perché una macchina che esegue velocemente non può usare una quantità di spazio grande. Precisamente, per $t(n) \geq n$, qualsiasi macchina che opera in tempo $t(n)$ può usare al più spazio $t(n)$, perché una macchina può esplorare al più una nuova cella ad ogni passo della propria esecuzione. Similmente, $NP \subseteq NPSPACE$ e, pertanto, $NP \subseteq PSPACE$.

Viceversa, possiamo limitare la complessità di tempo di una macchina di Turing in termini della sua complessità di spazio. Per $f(n) \geq n$, una TM che usa spazio $f(n)$ può avere al più $2^{O(f(n))}$ configurazioni differenti, attraverso una semplice generalizzazione della priva del Lemma 5.8. Una computazione di una TM che si arresta non può ripetere una configurazione. Pertanto, una TM che usa spazio $f(n)$ deve computare in tempo $f(n)2^{O(f(n))}$, quindi $PSPACE \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$.

Riassumiamo le nostre conoscenze circa le relazioni tra le classi di complessità definite fino ad ora nella serie di inclusioni

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME.$$

Non sappiamo se una qualsiasi di queste inclusioni è in realtà un'uguaglianza. Qualcuno potrebbe trovare una simulazione simile a quella usata nel teorema di Savitch che fonde alcune di queste classi nella stessa classe. In realtà, molti ricercatori ritengono che tutte le inclusioni siano proprie. Il diagramma seguente raffigura le relazioni tra queste classi, assumendo che siano tutte differenti.

