Automi e Linguaggi (M. Cesati)

Facoltà di Ingegneria, Università degli Studi di Roma Tor Vergata

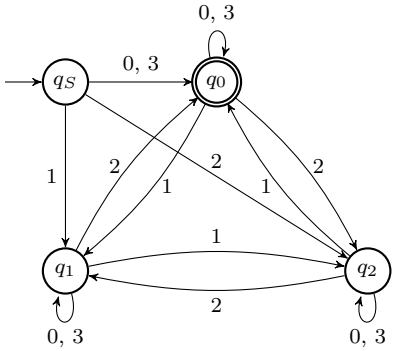
Compito scritto del 2 febbraio 2022

Esercizio 1 [5] Determinare un automa a stati finiti deterministico (DFA) per il linguaggio contenente tutti e soli i numeri in base 4 multipli di 3.

Soluzione: L'automa a stati finiti richiesto può essere costruito considerando che l'unica informazione necessaria da memorizzare è il valore modulo 3 del numero rappresentato dalle cifre lette man mano. In altri termini, se v è il valore del numero rappresentato dalle cifre in base 4 lette fino ad un certo punto, e la successiva cifra letta è $b \in \{0, \ldots, 3\}$, allora il nuovo valore rappresentato dalle cifre lette sarà $v' = v \times 4 + b$; tuttavia, è sufficiente memorizzare negli stati dell'automa soltanto il resto della divisione per tre, in quanto $v' \bmod 3 = ((v \bmod 3) \times 4 + b) \bmod 3$. Si ha dunque la seguente tabella:

$v \bmod 3$	$b = 0$	$b = 1$	$b = 2$	$b = 3$
0	0	1	2	0
1	1	2	0	1
2	2	0	1	2

È ora immediato derivare dalla tabella il seguente DFA:



Lo stato iniziale q_S ha il solo scopo di rifiutare la stringa vuota ε.

Esercizio 2 [6] Si consideri l'insieme $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ e la seguente grammatica G con variabile iniziale S:

$$S \quad \longrightarrow \quad \mathtt{a}S\mathtt{b} \quad | \quad \mathtt{b}S \quad | \quad S\mathtt{a} \quad | \quad \varepsilon$$

Quale è l'insieme di stringhe $L(G)$ generate da G? Giustificare la risposta con una dimostrazione.

Soluzione: Il linguaggio generato dalla grammatica include tutte le stringhe sull'alfabeto di terminali Σ, ossia $L(G) = \Sigma^*$. La dimostrazione è per induzione. Come base dell'induzione, consideriamo che la stringa vuota ε è generata dalla grammatica grazie alla regola $S \longrightarrow \varepsilon$. Supponiamo dunque come ipotesi induttiva che la grammatica sia in grado di generare tutte le stringhe in Σ^* di lunghezza inferiore a $n \geq 0$, e consideriamo una stringa x di lunghezza n. Possiamo distinguere i seguenti tre casi:

1. il primo carattere di x è '$\mathtt{b}$': Si consideri una derivazione da S in cui la prima regola applicata è $S \longrightarrow \mathtt{b}\,S$: la stringa ottenuta è pertanto $\mathtt{b}\,S$. Poiché $x = \mathtt{b}\,y$ ove $y \in \Sigma^*$ e $|y| = n - 1$, per l'ipotesi induttiva esiste una derivazione da S che genera la stringa terminale y. Perciò la derivazione costituita dalla regola che genera il primo carattere di x seguita dalle regole che generano y costituiscono una derivazione della stringa x. Pertanto $x \in L(G)$.

2. l'ultimo carattere di x è '$\mathtt{a}$': Si consideri una derivazione da S in cui la prima regola applicata è $S \longrightarrow S\,\mathtt{a}$: la stringa ottenuta è pertanto $S\,\mathtt{a}$. Poiché $x = y\,\mathtt{a}$ ove $y \in \Sigma^*$ e $|y| = n - 1$, per l'ipotesi induttiva esiste una derivazione da S che genera la stringa terminale y. Perciò la derivazione costituita dalla regola che genera l'ultimo carattere di x seguita dalle regole che generano y costituiscono una derivazione della stringa x. Pertanto $x \in L(G)$.

3. il primo carattere di x è '$\mathtt{a}$' e l'ultimo carattere di x è '$\mathtt{b}$': Si consideri una derivazione da S in cui la prima regola applicata è $S \longrightarrow \mathtt{a}\,S\,\mathtt{b}$: la stringa ottenuta è pertanto $\mathtt{a}\,S\,\mathtt{b}$. Poiché $x = \mathtt{a}\,y\,\mathtt{b}$ ove $y \in \Sigma^*$ e $|y| = n - 2$, per l'ipotesi induttiva esiste una derivazione da S che genera la stringa terminale y. Perciò la derivazione costituita dalla regola che genera il primo e l'ultimo carattere di x seguita dalle regole che generano y costituiscono una derivazione della stringa x. Pertanto $x \in L(G)$.

In tutti i tre casi si conclude che $x \in L(G)$. Pertanto resta dimostrato che $x \in L(G)$ per ogni possibile stringa di Σ^*, e quindi $L(G) = \Sigma^*$.

Esercizio 3 [8] Sia $B = \{w\overline{w} \,|\, w \in \{0,1\}^*\}$, ove $\overline{w}$ è la stringa ottenuta da w complementando ogni bit. Dimostrare che B non è un CFL.

Soluzione: Per assurdo, supponiamo che B sia CFL e che quindi valga per esso il "Pumping lemma". Sia dunque p la "pumping length" per B. In questo esercizio la scelta della stringa

che contraddice le conclusioni del lemma deve essere fatta con attenzione. Ad esempio, la stringa $0^p1^p1^p0^p$ non andrebbe bene, perché può essere pompata ponendo $u = 0^{p-1}$, $v = 0$, $x = \epsilon$, $y = 1$, $z = 1^{2p-1}0^p$: infatti, $uv^ixy^iz = 0^{p-1}0^i1^i1^{2p-1}0^p = 0^{p+i-1}1^p1^{p+i-1}0^p$.

Consideriamo invece la stringa $s = 0^p1^p01^p0^p1$: evidentemente $s \in B$ (in quanto $\overline{0^p1^p0} = 1^p0^p1$), quindi deve essere possibile determinare una suddivisione $s = uvxyz$ tale che per ogni $i \geq 0$, $uv^ixy^iz \in B$, $|vy| > 0$ e $|vxy| \leq p$. Distinguiamo i seguenti casi:

1. Se vy non contiene esattamente lo stesso numero di zero ed uno, pompando verso il basso o verso l'alto si otterrebbe una stringa con una eccedenza di zero od uno, che dunque non potrebbe far parte di B. Ciò implica anche che $|vy| > 1$.

2. Se la sottostringa vxy fosse interamente nella prima metà di s, allora pompando verso il basso si otterrebbe una stringa uxz in cui uno dei bit della seconda occorrenza di 1^p in s finirebbe nell'ultima posizione della prima metà di uxz (in quanto $|vxy| \leq p$ e $|vy| > 1$). Pertanto uxz non può essere in B perché il bit in ultima posizione della prima metà deve essere 0.

3. Analogamente, la sottostringa vxy non può essere interamente nella seconda metà di s, altrimenti pompando verso il basso uno dei bit della prima occorrenza di 1^p in s finirebbe nell'ultima posizione della prima metà di uxz, e ciò significa che tale stringa non potrebbe far parte di B perché essa termina con 1.

4. Pertanto, vxy deve contenere sia un bit della prima metà di s che un bit della seconda metà. Poichè però vy deve contenere un ugual numero di zero ed uno, l'unica possibilità è che vy sia o la stringa 01 oppure la stringa 10, ove lo 0 è quello in ultima posizione della prima metà di s. Pompando verso il basso si ottiene una stringa in cui nell'ultima posizione della prima metà vi è un 1, dunque tale stringa non può far parte di B.

Poiché non è possibile suddividere la stringa s in accordo al pumping lemma, concludiamo che il pumping lemma non può essere applicato, e pertanto che l'ipotesi che B fosse un CFL è falsa.

Esercizio 4 [6] Uno stato interno r di una macchina di Turing M è detto *inutile* se non esiste alcuna stringa di input che possa portare la macchina M ad assumere lo stato interno r. Dimostrare che il linguaggio contenente le codifiche di tutte le macchine di Turing con uno o più stati inutili è indecidibile.

Soluzione: Cominciamo con l'osservazione evidente che la proprietà di avere o meno uno stato inutile appartiene alla particolare macchina di Turing, e non è propria del linguaggio riconosciuto dalla macchina. Ad esempio, consideriamo una TM che non ha stati inutili, e modifichiamo la sua descrizione in modo da aggiungere uno stato che non è mai utilizzato in alcuna transizione. Ovviamente la macchina modificata riconosce lo stesso linguaggio della

macchina originale ma possiede uno stato inutile. Dunque non possiamo dimostrare quanto richiesto utilizzando il Teorema di Rice.

Per dimostrare l'indecidibilità del linguaggio contenente le codifiche di TM aventi almeno uno stato inutile costruiamo una riduzione dal problema di accettazione delle TM $\mathcal{A}_{\text{TM}}$. Come al solito dunque consideriamo una istanza $\langle M, w\rangle$ di $\mathcal{A}_{\text{TM}}$ e mostriamo come costruire una TM N che ha uno stato inutile se e solo se $M(w)$ accetta. La difficoltà di questa costruzione, però, risiede nella macchina M stessa: infatti, dovendo simulare M, N include nei suoi stati interni anche gli stati interni di M. Se M avesse stati inutili, la macchina N potrebbe avere stati inutili anche se in effetti $M(w)$ non accettasse. Dobbiamo quindi fare attenzione a costruire N in modo da rendere ogni stato di M "utile".

Siano Σ e Γ rispettivamente gli alfabeti di input e di nastro della macchina M; la TM N utilizzerà $\Sigma' = \Sigma \cup \{\#\}$ e $\Gamma' = \Gamma \cup \{\#\}$, ove '#' è un nuovo simbolo non utilizzato in M. Il simbolo '#' serve ad implementare un ciclo in cui si entra in ogni stato interno di M: in pratica viene aggiunta una nuova transizione per ogni stato interno di M in cui leggendo '#' si passa in uno stato interno di N che continua il ciclo.

Nella descrizione di N seguente si deve assumere che non vengono introdotti stati interni propri di N inutili a meno che questo che non sia affermato esplicitamente. In particolare, lo stato interno r di N è utilizzato solo quando esplicitamente menzionato. Inoltre si tenga presente che gli stati di accettazione e rifiuto di M non sono stati finali per N.

N = "On input x, where $x \in \Sigma'^*$:
1. If $x =$ '#':
    2. Execute a loop and enter in every internal state of M
    3. Halt
4. Run M on input w
5. If $M(w)$ accepts, enter in internal state r
6. Halt"

Supponiamo per assurdo che il problema di decidere se una TM ha stati interni inutili sia decidibile, e sia dunque R un decisore per tale problema. Consideriamo allora la seguente TM D:

D = "On input $\langle M, w\rangle$, where M is a TM and $w \in \Sigma^*$:
1. Build from $\langle M, w\rangle$ the encoding of the TM N
2. Run R on $\langle N\rangle$
3. If $R(\langle N\rangle)$ accepts, then reject
4. Otherwise, if $R(\langle N\rangle)$ rejects, then accept"

Se $R(\langle N\rangle)$ accetta, allora N ha almeno uno stato interno inutile. Per costruzione di N questo stato può essere unicamente r. Ciò significa che $M(w)$ non accetta. Se invece $R(\langle N\rangle)$ rifiuta,

allora ogni stato interno di N è utile, quindi anche lo stato interno di r deve essere visitato per un certo input x. In effetti l'input x di N viene ignorato se è diverso da '#'. L'unica possibilità è che $M(w)$ accetta e quindi r sia utilizzato nel passo 5 di N. Poiché D complementa la decisione di R, D è in effetti un decisore per $\mathcal{A}_{\text{TM}}$, ma questo è in contraddizione con il fatto che $\mathcal{A}_{\text{TM}}$ è indecidibile. Pertanto resta dimostrato che il problema di decidere se una TM ha stati interni inutili è indecidibile.

Esercizio 5 [6] Dimostrare che se P=NP allora ogni linguaggio in P diverso da $\emptyset$ e da Σ^* è NP-completo.

Soluzione: La dimostrazione è in effetti molto semplice considerando che una riduzione polinomiale è basata su una macchina di Turing deterministica che ha la capacità di risolvere direttamente i problemi in P, e quindi in NP, poiché assumiamo che P=NP.

Formalmente, consideriamo un qualunque linguaggio $A \in$ P diverso dagli insiemi banali, ossia dall'insieme vuoto $\emptyset$ e dall'insieme contenente tutte le stringhe Σ^*, e dimostriamo che A è NP-completo. Innanzi tutto dobbiamo provare che $A \in$ NP, ma questo è immediato perché per ipotesi $A \in$ P e P=NP. Mostriamo adesso che A è NP-hard. Sia dunque $B \in$ NP. Poiché P=NP, $B \in$ P, dunque esiste una DTM M che decide B. Trasformiamo M in un'altra DTM N che, per ogni istanza x, simula l'esecuzione di $M(x)$. Se $M(x)$ accetta, N si ferma lasciando sul nastro la codifica di un elemento $I_y \in A$ (esiste certamente perché $A \neq \emptyset$). Se invece $M(x)$ rifiuta, N si ferma lasciando sul nastro la codifica di un elemento $I_n \notin A$ (esiste certamente perché $A \neq \Sigma^*$). La DTM N esegue in tempo polinomiale e calcola una istanza-sì di A per ogni istanza-sì di B, ed una istanza-no di A per ogni istanza-no di B. Dunque N costituisce una riduzione polinomiale da B ad A. Poiché B è un generico problema in NP, A è NP-hard. La conclusione è che A è NP-completo.

Esercizio 6 [9] Si consideri il linguaggio costituito dalle codifiche delle formule booleane che hanno almeno due assegnazioni di verità che soddisfano la formula. Dimostrare che tale linguaggio è NP-completo.

Soluzione: Questo problema è generalmente chiamato Double SAT, ed è una generalizzazione molto semplice del problema SAT.

Si dimostra facilmente che Double SAT è in NP. Infatti, data una qualunque istanza $\langle\Phi\rangle$ che codifica una formula booleana, un certificato per l'esistenza di una soluzione è costituito da due liste di valori di verità. Il verificatore controlla che ciascuna lista contenga esattamente un valore (vero o falso) per ciascuna variabile di Φ, e che entrambe le assegnazioni di verità soddisfino la formula Φ.

Per dimostrare che Double SAT è NP-hard consideriamo la seguente riduzione dal problema SAT: considerata una generica istanza $\langle\Phi\rangle$ di SAT, sia Φ' la formula booleana ottenuta da

Φ aggiungendo una nuova variabile v e ponendo

$$\Phi' = \Phi \wedge (v \vee \overline{v}).$$

È immediato verificare che se la formula Φ è soddisfacibile allora esistono almeno due assegnazioni di verità che soddisfano Φ': la assegnazione che soddisfa Φ estesa con $v = T$ e la stessa assegnazione estesa con $v = F$. Viceversa, se la formula Φ non è soddisfacibile, allora nemmeno Φ' può essere soddisfacibile, perché la variabile v non appare nella formula Φ e quindi non può contribuire a rendere quella parte della formula Φ' soddisfacibile.

Da tutto ciò si può concludere che DOUBLE SAT è NP-completo.