Automi e Linguaggi Formali
Esame del 12 Giugno 2018 – Parte II

Tempo a disposizione: 1 h 30 min

1. La domanda riguarda la dimostrazione che per ogni PDA P che accetta per stack vuoto, esiste una grammatica G_P che genera lo stesso linguaggio che P riconosce. Ora, immaginate che P abbia solo 2 stati (p e q) e che abbia la seguente transizione: $\delta(q, a, X) = \{(q, YZ)\}$. Mostrare le produzioni che G_P possiede in corrispondenza di questa transizione.

Le produzioni sono:

$$[qXq] \to a[qYq][qZq] \mid a[qYp][pZq]$$
$$[qXp] \to a[qYq][qZp] \mid a[qYp][pZp]$$

2. Data la seguente CFG: $S \to ASB \mid \epsilon \quad A \to aAS \mid a \quad B \to SbS \mid A \mid bb$, descrivere come si eliminano da essa le $\epsilon-$produzioni, ottenendo una grammatica che genera $L(S) - \{\epsilon\}$.

La sola variabile che può generare ϵ è S, quindi la nuova grammatica ha le seguenti produzioni:

$$S \to ASB \mid AB \quad A \to aAS \mid aA \mid a \quad B \to SbS \mid bS \mid Sb \mid b \mid A \mid bb$$

3. Il Teorema di Rice dimostra che tutte le proprietà sui linguaggi RE (cioè riconosciuti dalle Macchine di Turing) sono indecidibili. Consideriamo per esempio la seguente proprietà P_{CF}: il linguaggio è Context Free. Spiegare come viene definito il corrispondente linguaggio $L_{P_{CF}}$ che è indecidibile per il Teorema di Rice.

$L_{P_{CF}}$={ M | M è una TM che accetta un linguaggio CF}

4. Si chiede di descrivere la Forma Normale di Chomsky, di descrivere l'enunciato del pumping Lemma e di spiegare (meglio che potete) come si arriva a dimostrare il pumping Lemma partendo dalla Forma Normale.

La forma normale di Chomsky prevede solo 2 tipi di produzioni: $A \to a \mid BC$. In classe abbiamo dimostrato che ogni albero di derivazione di una grammatica in CNF è tale che se n è l'altezza dell'albero allora il prodotto dell'albero ha lunghezza minore o uguale di 2^{n-1}. Per cui se m è il numero di variabili della grammatica, allora un qualsiasi albero di derivazione con prodotto 2^m ha altezza almeno $m + 1$ e quindi sul suo cammino più lungo almeno una variabile ripete. Questo fatto consente di dimostrare che per ogni linguaggio CF esiste un valore $n \geq 2^m$ tale che ogni stringa z del linguaggio lunga almeno n puo essere scomposta in 5 parti, $z = uvwxy$, con $|vwx| \leq n$, e vx non vuoto e inoltre, per ogni $i \geq 0$, anche uv^iwx^iy appartiene al linguaggio. Questo viene chiamato il pumping lemma dei CFL e serve a dimostrare che certi linguaggi non sono CF.

5. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Un *circuito quasi Hamiltoniano* in un grafo G è un ciclo che attraversa esattamente una volta tutti i vertici del grafo *tranne uno*. Il *problema del circuito quasi Hamiltoniano* è il problema di stabilire se un grafo contiene un circuito quasi Hamiltoniano.

(a) Dimostrare che il problema è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
Sia n il numero di vertici del grafo che è istanza di QHC. Un certificato per QHC è una sequenza ordinata di n-1 vertici distinti. Occorre tempo polinomiale per verificare se ogni nodo è collegato al successivo e l'ultimo al primo.

(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema del circuito quasi Hamiltoniano come sottoprocedura.
Dato un qualsiasi grafo G che è un'istanza di HC, costruiamo in tempo costante un nuovo grafo G' che è un'istanza di QHC, aggiungendo a G un vertice isolato (senza archi). Chiaramente G' ha un QHC sse G ha un HC.

(c) Il problema del circuito quasi Hamiltoniano è un problema NP-completo?
- ☐ No, è un problema in P
- ☐ No, è un problema NP ma non NP-completo
- ☐ Si

Nei precedenti punti abbiamo dimostrato che QHC è NP e che è NP-hard. Quindi è NP-completo.

Automi e Linguaggi Formali
Esame del 5 Luglio 2018 – Parte II

Tempo a disposizione: 1 h 30 min

1. Dare la definizione dei linguaggi L_e e L_{ne}. Spiegare come si dimostra che L_{ne} non è ricorsivo. Si usa una riduzione.

$L_e = \{M \mid L(M) = \emptyset\}, L_{ne} = \{M \mid L(M) \neq \emptyset\}$. Si dimostra che L_{ne} non è ricorsivo, usando una riduzione da L_u a L_{ne}. Un'istanza di L_u è costituita da una coppia (M, w) e la coppia appartiene a L_u se $w \in L(M)$. Da (M, w) costruiamo un'istanza M' di L_{ne} (infatti le istanze di L_{ne} sono macchine di Turing) come segue: per un qualsiasi input x, M' lo sostituisce con w e poi simula M con input w. Se M accetta, allora M' accetta x (quindi $M' \in L_{ne}$), se M non accett,a M' fa lo stesso (quindi $L(M') = \emptyset$). E' facile capire che $(M, w) \in L_u$ sse $M' \in L_{ne}$. Visto che L_u è indecidibile (RE, ma non ricorsivo), la riduzione appena descritta mostra che anche L_{ne} è indecidibile e quindi non ricorsivo.

2. Quali proprietà dei linguaggi RE sono dette triviali?

Una proprietà P sugli RE è triviale (o banale) se $L_P = \emptyset$ oppure $L_P = \{L \mid L \in RE\}$. Insomma P è banale se non è soddisfatta da alcun linguaggio RE oppure se è soddisfatta da tutti i linguaggi RE.

3. Il linguaggio $L = \{a^k b^{2k} c^{3k} \mid k \geq 0\}$ è CF o non CF? Nel primo caso fornire una CFG che genera L (o un PDA che lo riconosce). Nel secondo caso dimostrare che L non è CF.

L non è CF e lo si dimostra usando il pumping lemma dei CFL. Se L fosse CF, allora ci sarebbe un $n > 0$ tale che ogni stringa $w \in L$ con $|w| \geq n$, avrebbe la struttura $w = uvxyz$ con $|vxy| \leq n$ e $vy \neq \epsilon$ e inoltre tutte le stringhe, $uv^i xy^i z$, con $i \geq 0$ sarebbero in L. Consideriamo la stringa $w = a^n b^{2n} c^{3n} \in L$, ovviamente $|w| > n$ e quindi $w = uvxyz$ con le proprietà ricordate prima. Ora, la parte centrale di w, vxy può consistere di soli a, b o c, oppure di a e b o di b e c, ma in nessun caso di tutti e 3 i simboli a, b e c. Quindi nelle stringhe $uv^i xy^i z$ con $i > 0$ è impossibile che il numero dei 3 simboli continui a soddisfare la condizione richiesta per essere in L. In particolare, l'unico simbolo oppure i 2 simboli che vengono "pompati" aumentando il valore di i, cresceranno, mentre il simbolo, o i 2 simboli non "pompati" resteranno inalterati. Per cui ci sono i tali che $uv^i xy^i z \notin L$ e quindi L non è CF.

4. Descrivere un PDA che accetta per pila vuota e che riconosca il seguente linguaggio $L = \{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$. E' possibile costruire il PDA passando prima per una CFG che genera L. In questo caso è richiesta una dimostrazione o almeno una spiegazione convincente del fatto che la CFG generi veramente L.

Costruiamo direttamente il PDA P richiesto , senza passare per la CFG che genera L. P ha gli stati q_a e q_b e le seguenti transizioni:

$\delta(q_a, a, Z) = \{(q_a, aZ), (q_a, aaZ)\}, \delta(q_a, a, a) = \{(q_a, aa), (q_a, aaa)\}, \delta(q_a, b, a) = \{(q_b, \epsilon)\},$

$\delta(q_a, \epsilon, Z) = \{(q_a, \epsilon)\},$

$\delta(q_b, b, a) = \{(q_b, \epsilon)\}, \delta(q_b, \epsilon, Z) = \{(q_b, \epsilon)\}$

L'idea è semplice: quando si vede un a in input, esso può contare come 1 solo a oppure come 2 a. Questi 1/2 a sono inseriti sullo stack. Quando iniziano i b dell'input, per ogni b si fa il pop di una a. Se l'input è in L, c'è una sequenza di scelte di 1/2 a inseriti nello stack che fa coincidere il numero di a messi sullo stack con il numero di b della seconda parte dell'input. Per questa scelta, dopo aver considerato l'intera stringa, lo stack conterrà Z e l'ultima transizione con q_b lo svuota bloccando il calcolo di P. C'è anche la transizione $\delta(q_a, \epsilon, Z) = \{(q_a, \epsilon)\}$ che svuota lo stack e serve ad accettare la parola vuota.

La costruzione era più semplice passando per una CFG che genera L. Una tale grammatica potrebbe essere la seguente: $S \to aSb \mid aSbb \mid \epsilon$. Dalla grammatica si produce il PDA seguendo la costruzione vista nel corso. Che la grammatica data generi strighe in L è semplice da vedere, visto che ogni produzione genera 1 a e, corrispondentemente, o 1 o 2 b. E' anche facile convincersi che la grammatica produce tutto L, infatti per ogni stringa di L è semplice trovare una derivazione della grammatica che la genera. Sia $a^n b^{n+k} \in L$, con $0 <= k <= n$. Allora una derivazione che deriva questa stringa parte da S e applica k volte la produzione $S \to aSbb$, dopo di che applica $n - k$ volte la produzione $S \to aSb$ e termina con $S \to \epsilon$. La prima parte della derivazione produce $S \Rightarrow^* a^k S b^{2k}$ e la seconda parte aggiunge $n - k$ a e b: $S \Rightarrow^* a^k a^{n-k} S b^{n-k} b^{2k} \Rightarrow a^k a^{n-k} b^{n-k} b^{2k} = a^n b^{n+k}$. Ovviamente la grammatica è molto ambigua, ma questo non ha alcuna importanza per lo scopo per cui si intende usarla.

5. "Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Il problema kCOLOR è il problema di trovare una colorazione di un grafo non orientato usando k colori diversi.

(a) Dimostrare che il problema 4COLOR (colorare un grafo con 4 colori) è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

(b) Mostrare come si può risolvere il problema 3COLOR (colorare un grafo con 4 colori) usando 4COLOR come sottoprocedura.

(c) Per quali valori di k il problema kCOLOR è NP-completo?

- ☐ Per nessun valore: kCOLOR è un problema in P
- ☐ Per tutti i $k \geq 3$
- ☐ Per tutti i valori di k

Le risposte seguono:

a) se i vertici del grafo sono numerati da 1 a n, allora una sequenza di lunghezza n dei 4 colori disponibili, dove il colore in posizione i della sequenza è associato al vertice i, è un certificato. Per verificare che la colorazione corrispondente al certificato ha risposta SI, basta verificare che il vertice i abbia colore diverso da tutti i vertici a cui è collegato e questa operazione è lineare nella taglia del grafo , visto che basta esaminare ogni arco del grafo 2 volte: una per ciascuno dei 2 vertici collegati dall'arco.

b) Si riduce 3COLOR a 4COLOR come segue: dato un qualsiasi grafo G, istanza di 3COLOR, si aggiunge a G un vertice collegato a tutti i vertici di G. Il grafo G' così ottenuto è un'istanza di 4COLOR e infatti G è colorabile con 3 colori sse G' lo è con 4 colori. Per cui 4COLOR è almeno altrettanto intrattabile di 3COLOR.

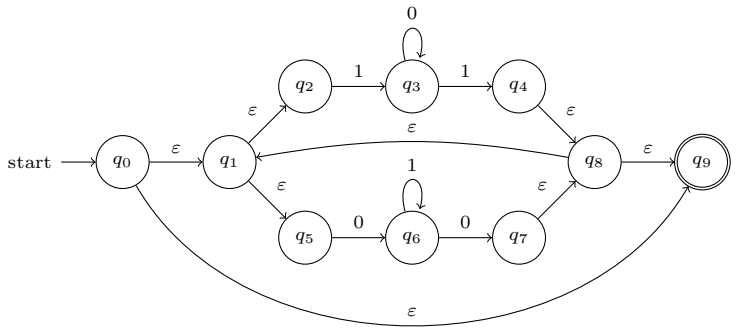
c) I problemi kCOLOR con k≥ 3 sono NP-completi.

Automi e Linguaggi Formali
Esame del 3 Settembre 2018

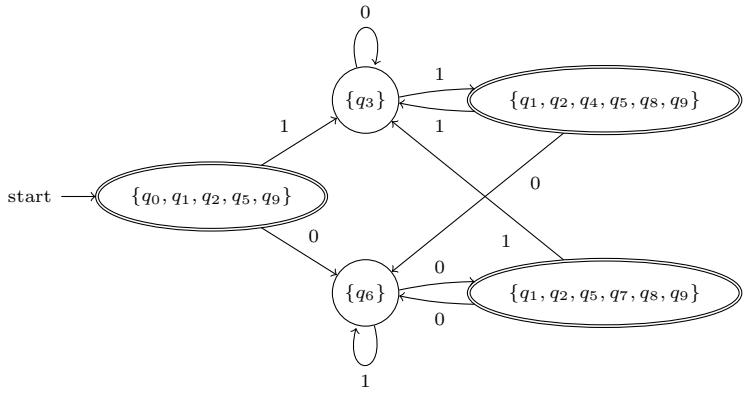
Tempo a disposizione: 2 ore

1. (a) Convertire l'espressione regolare $(\mathbf{10^*1 + 01^*0})^*$ in un ε-NFA (si può usare l'algoritmo visto a lezione oppure creare direttamente l'automa). **Importante:** l'automa deve essere nondeterministico e deve sfruttare le ε-transizioni per riconoscere il linguaggio.

L'esercizio ha molte possibili soluzioni, una delle quali è la seguente:



(b) Trasformare l'ε-NFA ottenuto al punto precedente in un DFA.



2. Considerate il linguaggio $L = \left\{0^{2n}1^m0^n : n, m \geq 0\right\}$. Questo linguaggio è regolare? Dimostrare formalmente la risposta.

Il linguaggio non è regolare. Supponiamo per assurdo che lo sia:

- *sia h la lunghezza data dal Pumping Lemma;*
- *consideriamo la parola $w = 0^{2h}10^h$, che appartiene ad L ed è di lunghezza maggiore di h;*
- *sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq h$;*
- *poiché $|xy| \leq h$, allora xy è completamente contenuta nel prefisso 0^{2h} di w, e quindi sia x che y sono composte solo da 0. Inoltre, siccome $y \neq \varepsilon$, possiamo dire che $y = 0^p$ per qualche valore $p > 0$. Allora la parola xy^2z è nella forma $0^{2h+p}10^h$, e quindi non appartiene al linguaggio perché il numero di 0 nella prima parte della parola non è uguale al doppio del numero di 0 nella seconda parte della parola.*

Abbiamo trovato un assurdo quindi L_1 non può essere regolare.

3. Descrivere un automa a pila che accetti per stack vuoto il linguaggio che consiste di stringhe composte di a e b tali che, se n è il numero degli a, allora il numero dei b è tra n e $2n$. Il linguaggio contiene la stringa vuota. Si osservi che nelle stringhe del linguaggio, gli a e i b possono essere mescolati (cioè non necessariamente della forma a^nb^m).

Il PDA P che riconosce L per stack vuoto ha 2 stati q_1 e q_2 e le seguenti transizioni:

$\delta(q_1, a, Z) = \{(q_1, aZ), (q_1, aaZ)\}, \delta(q_1, a, a) = \{(q_1, aa), (q_1, aaa)\}, \delta(q_1, a, b) = \{(q_1, \epsilon), (q_2, \epsilon)\},$

$\delta(q_1, \epsilon, Z) = \{(q_1, \epsilon)\}$,

$\delta(q_1, b, a) = \{(q_1, \epsilon)\}, \delta(q_1, b, Z) = \{(q_1, bZ)\}, \delta(q_1, b, b) = \{(q_1, bb)\}$

$\delta(q_2, \epsilon, b) = \{(q_1, \epsilon)\}, \delta(q_2, \epsilon, Z) = \{(q_1, aZ)\}$

Lo stato q_2 serve quando si consuma l'input a si decide che deve contare doppio e la cima della pila contiene b, allora la transizione da q_1 a q_2 consuma il primo b della pila e lo stato q_2 vede quello che c'era sotto quel b. Se c'era un altro b, lo toglie, mentre se c'era Z, allora aggiunge a allo stack. In entrambi i casi torna in q_1.

4. Dare la definizione precisa del linguaggio L_{ne} e successivamente dimostrare che esso è un linguaggio RE. La prova richiede di esibire una TM che riconosca in tempo finito tutte le stringhe del linguaggio L_{ne}.

$L_{ne} = \{M|\ L(M) \neq \emptyset\}$. Per dimostrare che L_{ne} è RE è necessario definire una TM M che lo riconosce. Gli input di L_{ne} sono tutte le stringhe binarie che vengono interpretate come TM. La TM M sarà come segue: data in input una stringa binaria che rappresenta la TM M', M genera nondeterministicamente tutte le possibili stringhe binarie e per ogni tale stringa w, simula M'' su w (per farlo usa la TM universale), se M' accetta qualcuna delle w prodotte, allora M accetta M', infatti, in questo caso, M' appartiene a L_{ne}. Il punto centrale è la generazione nondeterministica di tutte le possibili stringhe binarie w. Ogni w è finita, e quindi verrà prodotta in un tempo finito e se esiste w che è accettata da M', M accetterà M' in un tempo finito. Se invece M' non accetta alcuna w e quindi $M' \notin L_{ne}$, allora M continuerà il suo calcolo per sempre e questo è compatibile con il fatto che L_{ne} sia RE.

5. Stabilire se il seguente linguaggio $L = \{a^n b^m c^k \mid n = k, n \geq 0, m \geq 0, k \geq 0\}$ è CF oppure no. Se pensate che sia CF, esibite una CFG che lo generi oppure un PDA che lo riconosca (spiegando perché questo è vero). Se pensate che non sia CF, date un argomento che dimostri che effettivamente non lo sia.

L è CF. La seguente grammatica CF genera L:

$S \to aSc \mid B$

$B \to bB \mid \epsilon$

6. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Considerate il seguente problema, che chiameremo HAM375: dato un grafo G con n vertici, trovare un ciclo che attraversa esattamente una volta $n - 375$ vertici del grafo (ossia tutti i vertici di G *tranne* 375).

(a) Dimostrare che il problema HAM375 è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
Sia n il numero di vertici del grafo che è istanza di HAM375. Un certificato per HAM375 è una sequenza ordinata di $n - 375$ vertici distinti. Occorre tempo polinomiale per verificare se ogni nodo è collegato al successivo e l'ultimo al primo.

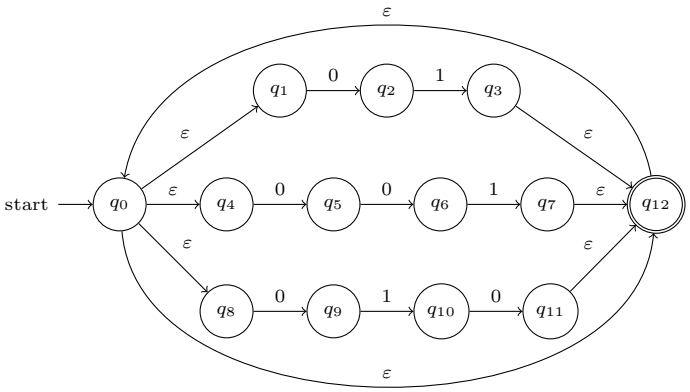
(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema HAM375 come sottoprocedura.
Dato un qualsiasi grafo G che è un'istanza del problema del circuito Hamiltoniano, costruiamo in tempo polinomiale un nuovo grafo G' che è un'istanza di HAM375, aggiungendo a G 375 vertici isolati (senza archi). Chiaramente G' ha un ciclo che attraversa esattamente una volta $n - 375$ vertici del grafo se e solo se G ha un ciclo Hamiltoniano.

Automi e Linguaggi Formali
Esame del 20 Settembre 2018

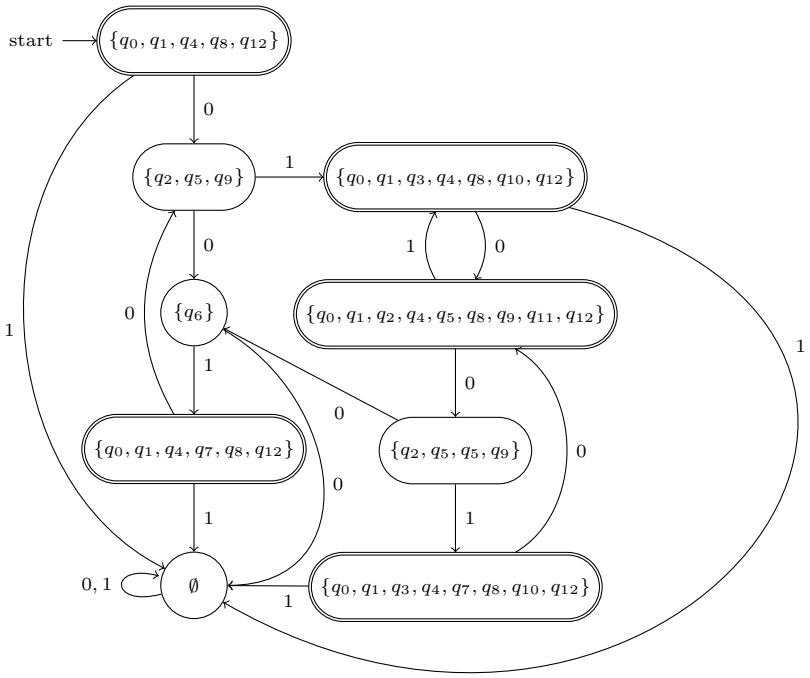
Tempo a disposizione: 2 h 30 m

1. (a) Convertire l'espressione regolare $(\mathbf{01} + \mathbf{001} + \mathbf{010})^*$ in un ε-NFA (si può usare l'algoritmo visto a lezione oppure creare direttamente l'automa). **Importante:** l'automa deve essere nondeterministico e deve sfruttare le ε-transizioni per riconoscere il linguaggio.

L'esercizio ha molte possibili soluzioni, una delle quali è la seguente:



(b) Trasformare l'ε-NFA ottenuto al punto precedente in un DFA.



2. Considerate il linguaggio $L = \{www \mid w \in \{a, b\}^*\}$. Questo linguaggio è regolare? Dimostrare formalmente la risposta.

Il linguaggio non è regolare. Supponiamo per assurdo che lo sia:

- *sia h la lunghezza data dal Pumping Lemma;*
- *consideriamo la parola $v = a^h b a^h b a^h b$, che appartiene ad L ed è di lunghezza maggiore di h;*
- *sia $v = xyz$ una suddivisione di v tale che $y \neq \varepsilon$ e $|xy| \leq h$;*
- *poiché $|xy| \leq h$, allora xy è completamente contenuta nel prefisso a^h di v, e quindi sia x che y sono composte solo da a. Inoltre, siccome $y \neq \varepsilon$, possiamo dire che $y = a^p$ per qualche valore $p > 0$. Allora la parola xy^2z è nella forma $a^{2h+p} b a^h b a^h b$, e quindi non appartiene al linguaggio perché non è possibile suddividerla in tre sottostringhe uguali tra di loro.*

Abbiamo trovato un assurdo quindi L non può essere regolare.

3. (a) Se i sta per la keyword if ed e sta per la keyword else, allora si chiede di definire una CFG capace di generare tutte le stringhe che rappresentano comandi if e if-else annidati e concatenati, come per esempio iii, iiee, iie, ieieiie e anche iieeiiee.

Una CFG che genera il linguaggio richiesto è $S \rightarrow iS \mid iSeS \mid \epsilon$. E' abbasanza facile convincersi che è capace di generare qualsiasi sequenza di if-else annidati.

(b) Riuscite a spiegare qual'è la regola che i compilatori dei linguaggi di programmazione usano per associare ogni else ad un if nelle stringhe del punto precedente?

La regola è che ogni "else" viene assocoato al prim "if" libero alla sua sinistra. Quindi la stringa iie è interpretata come i (ie).

(c) La vostra grammatica del punto (a) è ambigua o no? Argomentate la risposta. In caso pensiate che sia ambigua, è possibile trovare un'altra CFG che generi lo stesso linguaggio e che non sia ambigua? Argomentate la risposta e se pensate che sia possibile, allora date una tale CFG non ambigua

La grammatica è ambigua. Ci sono 2 derivazioni left-most per generare iie:

$S \Rightarrow^{lm} iS \Rightarrow^{lm} iiSeS$ e dopo i 2 S scompaiono in ϵ

e la seconda derivazione è:

$S \Rightarrow^{lm} iSeS \Rightarrow^{lm} iiSeS$ e di nuono i due S diventano ϵ.

La grammatica può essere resa non ambigua come segue: $S \rightarrow iS \mid iS'eS \mid \epsilon$ e $S' \rightarrow iS'eS'|\epsilon$.

4. Descrivere un PDA che accetta per pila vuota ed è capace di riconoscere il linguaggio $L = \{(ab)^n(ca)^n \mid n \geq 1\}$. Il vostro è un automa deterministico o nondeterministico? Spiegare la risposta.

Il PDA ha 2 stati q1 e q2 e le seguenti transizioni:

$\delta(q1, a, Z0) = \{((q1, aZ0)\}, \delta(q1, b, a) = \{((q1, ba)\}, \delta(q1, a, b) = \{((q1, ab)\}, \delta(q1, c, b) = \{((q2, \epsilon)\},$

$\delta(q2, c, b) = \{((q2, \epsilon)\}, \delta(q2, a, a) = \{((q2, \epsilon)\}, \delta(q2, \epsilon, Z0) = \{((q2, \epsilon)\}.$

Il PDA è chiaramente deterministico. L'ultima transizione, svuota la pila e, se l'input è consumato, accetta.

5. Dare la definizione del linguaggio L_U e spiegare in dettaglio come si dimostra che L_U è un linguaggio RE e non ricorsivo.

$L_U = \{(M, w) \mid w \in L(M)\}$. Esiste una macchian di Turing capace di accettare L_U. Questa macchina di Turing ha diversi nastri e prende in input una qualsiasi coppia (M, w), scrive M su uno dei nastri, poi scrive w su un altro e $q0 = 0$ sul terzo e simula il calcolo di M su w cercando le mosse sul primo nastro. Insomma sfrutta l'idea della macchina universale, che, avendo una qualsiasi macchina di Turing su un nastro, è capace di simulare il suo calcolo. Da questo argomento segue che L_U è RE. Per dimostrare che L_U non è ricorsivo, usiamo il fatto che, se lo fosse, allora anche $\hat{L_U}$ lo sarebbe e, visto che potremmo ridurre L_d a $\hat{L_U}$, avremmo un assurdo. Per capire la riduzione è sufficiente capire che $\hat{L_U} = \{(M, w) \mid w \notin L(M)\}$. L_d ha come istanza una qualsiasi macchina di Turing M. Da una tale istanza di L_d, la corrispondente istanza di $\hat{L_U}$ è semplicemente (M, M). Ovviamente se $\hat{L_U}$ fosse ricorsivo, sarebbe possibile determinare se $M \notin L(M)$ e quindi sarebbe possibile decidere L_d. Assurdo.

6. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-completo.

Considerate il seguente problema, che chiameremo SubsetSum: dato un insieme di numeri interi S ed un valore obiettivo t, stabilire se esiste un sottoinsieme $S' \subseteq S$ tale che la somma dei numeri in S' è uguale a t.

(a) Dimostrare che il problema SubsetSum è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

Il certificato per SubsetSum è dato dal sottoinsieme S'. Occorre verificare che ogni elemento di S' appartenga anche ad S e che la somma dei numeri contenuti in S' sia uguale a t.

(b) Mostrare come si può risolvere il problema SetPartitioning usando il problema SubsetSum come sottoprocedura.

Dato un qualsiasi insieme di numeri interi S che è un'istanza di SetPartitioning, costruiamo in tempo polinomiale un'istanza di SubsetSum. L'insieme di numeri interi di input rimane S, mentre il valore obiettivo t è uguale alla metà della somma degli elementi contenuti in S.

In questo modo, se S' è una soluzione di SubsetSum*, allora la somma dei valori contenuti nell'insieme $S - S'$ sarà pari alla metà della somma degli elementi di S. Quindi ho diviso S in sue sottoinsiemi $S_1 = S'$ e $S_2 = S - S'$ tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2.*
Viceversa, se S_1 e S_2 sono una soluzione di SetPartitioning*, allora la somma dei numeri contenuti in S_1 sarà uguale alla somma dei numeri in S_2, e quindi uguale alla metà della somma dei numeri contenuti in S. Quindi sia S_1 che S_2 sono soluzione di* SubsetSum *per il valore obiettivo t specificato sopra.*

Automi e Linguaggi Formali
Esame del 19 Giugno 2019 – Parte II

Tempo a disposizione: 1 h 30 min

1. Descrivere la relazione tra i linguaggi liberi da contesto, quelli riconosciuti da DPDA che accettano per stato finale e quelli riconosciuti da DPDA che accettano per pila vuota. Spiegare anche la relazione tra i linguaggi non ambigui e quelli riconosciuti dai DPDA per entrambe le modalità di accettazione.

I DPDA accettano linguaggi diversi a seconda del modo di accettazione. I DPDA che accettano per stato finale accettano tutti i linguaggi regolari e una parte dei linguaggi CF. Per esempio il linguaggio dei palindromi pari costituito dalle stringhe ww r non può essere accettato in modo deterministico perché è necessario "indovinare" la metà dell'input. I DPDA che accettano per stack vuoto sono meno espressivi in quanto riconoscono i linguaggi accettati dai DPDA per stato finale che soddisfano la proprietà del prefisso. Un linguaggio L ha la proprietà del prefisso quando non esistono 2 stringhe diverse di L tali che una sia un prefisso dell'altra. E' possibile dimostrare che i linguaggi accettati da DPDA che accettano per stack vuoto sono non inerentemente ambigui. E' possibile dimostrare che anche i linguaggi riconosciuti dai DPDA che accettano per stato finale sono non inerentemente ambigui. Questo risultato lo si ottiene con un "trucco" che permette di dare la proprietà del prefisso a tutti i linguaggi accettati dai DPDA che accettano per stato finale. Il trucco è di aggiungere un simbolo speciale (che si assume non appaia nelle stringhe del linguaggio originale) alla fine di ciascuna stringa del linguaggio. Questo garantisce che il nuovo linguaggio abbia la proprietà del prefisso e che quindi possa essere riconosciuto da un DPDA che accetta per stack vuoto.

2. Dato l'automa a pila $P = (\{q\}, \{a, b\}, \{a, Z\}, \delta, q, Z, \{q\})$ dove δ è come segue:
$\delta(q, a, Z) = \{(q, aZ)\}, \delta(q, a, a) = \{(q, aa)\}, \delta(q, b, a) = \{(q, \epsilon)\}$.
P accetta per stato finale (q).

- Descrivere precisamente il linguaggio riconosciuto da P.
- Trasformare P in un PDA P' che accetta per pila vuota lo stesso linguaggio accettato da P (per stato finale).
- Rispetto al determinismo-nondeterminismo, ci sono differenze tra P e P'? Spiegate i motivi di uguaglianza-differenza.

L(P) consiste delle stringhe w tali che ogni prefisso di w abbia un numero di a maggiore o uguale al numero dei b. ϵ è in L(P). Per ottenere P' basta aggiungere uno stato e le transizioni $\delta(q, \epsilon, ?) = \{(p, \epsilon)\}$ e $\delta(p, \epsilon, ?) = \{(p, \epsilon)\}$, dove ? sta per qualsiasi simbolo dello stack. Lo stato p serve a svuotare la pila senza consumare input. In questo modo P' accetta un input w per stack vuoto sse P accetta w per stato finale. P è deterministico, mentre P' è non deterministico. Non c'è modo di avere un DPDA che accetti L(P) per stack vuoto visto che L(P) non ha la proprietà del prefisso.

3. Dare la definizione del linguaggio L_U e spiegare in dettaglio come si dimostra che L_U è un linguaggio RE e non ricorsivo.

$L_u = \{(M, w) | w \in L(M)\}$. Per dimostrare che è RE si deve produrre una TM che riconosca L_u. Questa TM è la TM Universale che è indicata con U. U è capace di simulare il calcolo di una qualsiasi TM M su un qualsiasi input w. Ovviamente sia la TM M che w sono stringhe binarie. U si basa sul fatto che le TM si possono codificare in un modo semplice e tale che U possa riconsocere e simulare le transizioni di una TM qualsiasi. U è come un computer capace si eseguire tutti i programmi che noi gli diamo. Per dimostrare che L_u non è ricorsivo, ragioniamo per assurdo. Se L_u fosse ricorsivo, sarebbe ricorsivo anche comp(L_u) e con un algoritmo per comp(L_u) potremmo facilmente costruire un algoritmo per L_d. Infatti un'istanza di L_d è una TM M, basterà trasformare M in (M,M) per avere un'istanza di comp(L_u) e l'algoritmo che abbiamo ipotizzato risponderebbe SI/NO a (M,M) e le risposte SI individuano le TM che sono in L_d. Questo è un assurdo visto che L_d è non RE e quindi L_u non è ricorsivo.

4. Data la seguente CFG: $S \to ASB \mid \epsilon,\ A \to aBAS \mid a \mid \epsilon,\ B \to SbAb \mid AB \mid b$, descrivere, possibilmente seguendo l'algoritmo visto in classe, come si eliminano da essa le $\epsilon-$produzioni, ottenendo una grammatica che genera $L(S) \setminus \{\epsilon\}$.

Le variabili che generano ϵ sono S ed A. Dobbiamo quindi eliminare le $\epsilon-$produzioni e per le produzioni che contengono S e/o A nella parte destra dobbiamo produrre tante produzioni quante sono le combinazioni di presenza/assenza di queste 2 variabili. Quindi se consideriamo la produzione $B \to SbAb$, avremo le produzioni, $B \to SbAb \mid bAb \mid Sbb \mid bb$.

5. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-completo.

Considerate la seguente variante del problema, che chiameremo QuasiPartitioning: dato un insieme di numeri interi S, stabilire se può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 *meno 1*.

(a) Dimostrare che il problema QuasiPartitioning è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

Il certificato è dato da una coppia di insiemi di numeri interi S_1, S_2. Per verificarlo occorre controllare che rispetti le seguenti condizioni:

- i due insiemi $S1, S_2$ devono essere una partizione dell'insieme S;
- la somma degli elementi in S_1 deve essere uguale alla somma degli elementi in S_2 meno 1.

Entrambe le condizioni si possono verificare in tempo polinomiale.

(b) Dimostrare che il problema QuasiPartitioning è NP-hard, mostrando come si può risolvere il problema SetPartitioning usando il problema QuasiPartitioning come sottoprocedura.

Dimostrare che QuasiPartitioning è NP-hard, usando SetPartitioning come problema di riferimento richiede diversi passaggi:

1. Descrivere un algoritmo per risolvere SetPartitioning usando QuasiPartitioning come subroutine. Questo algoritmo avrà la seguente forma: data un'istanza di SetPartitioning, trasformala in un'istanza di QuasiPartitioning, quindi chiama l'algoritmo magico black-box per QuasiPartitioning.
2. Dimostrare che la riduzione è corretta. Ciò richiede sempre due passaggi separati, che di solito hanno la seguente forma:
   - Dimostrare che l'algoritmo trasforma istanze "buone" di SetPartitioning in istanze "buone" di QuasiPartitioning.
   - Dimostrare che se la trasformazione produce un'istanza "buona" di QuasiPartitioning, allora era partita da un'istanza "buona" di SetPartitioning.
3. Mostrare che la riduzione funziona in tempo polinomiale, a meno della chiamata (o delle chiamate) all'algoritmo magico black-box per QuasiPartitioning. (Questo di solito è banale.)

Una istanza di SetPartitioning è data da un insieme S di numeri interi da suddividere in due. Una istanza di QuasiPartitioning è data anch'essa da un insieme di numeri interi S'. Quindi la riduzione deve trasformare un insieme di numeri S input di SetPartitioning in un altro insieme di numeri S' che diventerà l'input per la black-box che risolve QuasiPartitioning.

Come primo tentativo usiamo una riduzione che crea S' aggiungendo un nuovo elemento a S di valore 1, e proviamo a dimostrare che la riduzione è corretta:

$\Rightarrow$ sia S un'istanza buona di SetPartitioning. Allora è possibile partizionare S in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Se aggiungiamo il nuovo valore 1 ad S_1 otteniamo una soluzione per QuasiPartitioning, e abbiamo dimostrato che S' è una istanza buona di QuasiPartitioning.

$\Leftarrow$ sia S' un'istanza buona di QuasiPartitioning. Allora è possibile partizionare S' in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 meno 1. Controlliamo quale dei due sottoinsiemi contiene il nuovo elemento 1 aggiunto dalla riduzione:
   - se $1 \in S_1$, allora se tolgo 1 da S_1 la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2. Abbiamo trovato una soluzione per SetPartitioning con input S.
   - se $1 \in S_2$, allora se tolgo 1 da S_2 quello che succede è che la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2 meno 2. In questo caso abbiamo un

problema perché quello che otteniamo *non è una soluzione di* SetPartitioning*!*

Quindi il primo tentativo di riduzione non funziona: ci sono dei casi in cui istanze cattive di SetPartitioning diventano istanze buone di QuasiPartitioning: per esempio, l'insieme $S = \{2, 4\}$, che non ha soluzione per SetPartitioning, diventa $S' = \{2, 4, 1\}$ dopo l'aggiunta dell'1, che ha soluzione per QuasiPartitioning: basta dividerlo in $S_1 = \{4\}$ e $S_2 = \{2, 1\}$.

Dobbiamo quindi trovare un modo per "forzare" l'elemento 1 aggiuntivo ad appartenere ad S_1 nella soluzione di QuasiPartitioning. Per far questo basta modificare la riduzione in modo che S' contenga tutti gli elementi di S *moltiplicati per* 3, oltre all'1 aggiuntivo. Formalmente:

$$S' = \{3x \mid x \in S\} \cup \{1\}.$$

Proviamo a dimostrare che la nuova riduzione è corretta:

- $\Rightarrow$ sia S un'istanza buona di SetPartitioning. Allora è possibile partizionare S in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Se moltiplichiamo per 3 gli elementi di S_1 ed S_2, ed aggiungiamo il nuovo valore 1 ad S_1 otteniamo una soluzione per QuasiPartitioning, e abbiamo dimostrato che S' è una istanza buona di QuasiPartitioning.
- $\Leftarrow$ sia S' un'istanza buona di QuasiPartitioning. Allora è possibile partizionare S' in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 meno 1. Vediamo adesso che, a differenza della riduzione precedente, non è possibile che $1 \in S_2$: se così fosse, allora se tolgo 1 da S_2 quello che succede è che la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2 meno 2. Tuttavia, gli elementi che stanno in S_1 ed S_2 sono tutti quanti multipli di 3 (tranne l'1 aggiuntivo). Non è possibile che due insiemi che contengono solo multipli di 3 abbiano differenza 2. Quindi l'1 aggiuntivo non può appartenere a S_2 e deve appartenere per forza a S_1. Come visto prima, se tolgo 1 da S_1 la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2. Abbiamo trovato una soluzione per SetPartitioning con input S.

In questo caso la riduzione è corretta. Per completare la dimostrazione basta osservare che per costruire S' dobbiamo moltiplicare per 3 gli n elementi di S ed aggiungere un nuovo elemento. Tutte operazioni che si fanno in tempo polinomiale.

Automi e Linguaggi Formali
Esame del 3 Luglio 2019 – Parte II

Tempo a disposizione: 1 h 30 min

1. Si consideri la seguente grammatica CF, G:

$S \to aBb\ ,\ B \to aBb \mid BB \mid \epsilon$

i) Descrivere un DPDA P che riconosce per stack vuoto il linguaggio L(G). Spiegare perché il vostro P fa quanto richiesto. In particolare, assicuratevi che P sia deterministico.

ii) Descrivere il calcolo di P con input uguale a "abab"e spiegare se vi sembra corretto oppure no, spiegando la vostra posizione.

i) Il DPDA che riconosce $L(S)$ è come segue:

$$\delta(q_0, a, Z_0) = \{((q_1, aZ_0)\}, \qquad \delta(q_1, a, a) = \{(q_1, aa)\}$$
$$\delta(q_1, b, a) = \{(q_1, \epsilon)\}, \qquad \delta(q_1, \epsilon, Z_0) = \{(q_1, \epsilon)\}$$

ii) consumando il primo a andrebbe in q_1 con a sullo stack, poi consumerebbe il b facendo il pop dell'a sullo stack. Quindi sullo stack avremmo Z_0 per cui la sola mossa possibile sarebbe quella con ϵ che svuota lo stack. Visto che la stringa non sarebbe finita e che il DPDA si blocca non appena lo stack si svuota, "abab" verrebbe rifiutata il che è giusto visto che non è in $L(S)$. Infatti $L(S)$ è un linguaggio con la proprietà del prefisso.

2. Il linguaggio $L = \{a^n w w^r b^n | n \geq 0 \text{ e } w \in \{a, b\}^*\}$ è CF, ma fingete di non saperlo ed applicate ad esso il Pumping Lemma per cercare di dimostare che L non sia CF. Cercate di individuare in quale passaggio la prova fallisce e perché.

Sia m la costante del PL e consideriamo una qualsiasi z di L di lunghezza maggiore di m. Il PL ci dice che $z = uvtxy$ con $|vtx| \leq m$ e $vx \neq \epsilon$ e tale che per ogni $k \geq 0$, $uv^k tx^k y \in L$. Se in z la parte centrale ww^r fosse piccola rispetto ad m, sarebbe facile considerare che $t = ww^r$ e $v = a^q$ e $x = b^q$ per cui anche pompando v e x a piacimento si resterebbe in L. Consideriamo quindi il caso in cui ww^r sia di lunghezza maggiore o uguale di m. Anche in questo caso abbiamo una soluzione che ci fa restare in L: $t = \epsilon$, mentre v è un suffisso di w di lunghezza q e x è un prefisso di w^r della stessa lunghezza. Allora vx sarebbe un palindromo e $v^q x^q$ continuerebbe ad essere un palindromo, rimanendo in L.

3. La domanda riguarda le variabili inutili di una CFG e come eliminarle:

i) Definire le variabili inutili di una grammatica CF.

ii) Spiegare in generale come si possono eliminare tutte le variabili inutili di una CFG.

iii) Applicare la procedura descritta nel punto precedente alla seguente CFG ottenendo una nuova grammatica equivalente a quella precedente e senza variabili inutili:
$S \to\ AB \mid CA\ ,\ A \to\ a,\ B \to\ BC \mid AB \mid DB,\ C \to\ aB \mid b\ ,\ D \to\ aDA \mid a$

i) Si devono eliminare le variabili non generatrici e poi quelle non raggiungibili. Una variabile X è generatrice se $X \Rightarrow^* w$ con w composto da soli terminali. Si osservi che w può essere anche ϵ. Una variabile X è raggiungibile se $S \Rightarrow^* \alpha X \beta$, dove S è il simbolo iniziale della grammatica.

ii) Per calcolare le variabili generatrici, si parte con un insieme Q che contiene tutti i simboli terminali, poi si aggiungono a Q le variabili X tali che ci sia una produzione $X \to \alpha$ con $\alpha \in Q*$ (si osservi che α può anche essere vuota). Si continua ad aggiungere variabili a Q fino a quando è possibile. Per calcolare i simboli raggiungibili, Si parte con $P = \{S\}$ e poi si aggiungono a P i simboli nella parte destra α di produzioni $X \to \alpha$ tali che $X \in P$. Si continua fino a che si aggiungono variabili a P.

- Nella grammatica data B non è generatrice, quindi eliminando dalla grammatica le produzioni con B, otteniamo, $S \to\ CA\ ,\ A \to\ a,\ C \to b\ ,\ D \to\ aDA \mid a$
  Poi è facile vedere che D non è raggiungibile e quindi alla fine resta: $S \to\ CA\ ,\ A \to\ a,\ C \to\ b$

4. La domanda riguarda le macchine di Turing (TM):

i) Descrivere come si rappresenta una TM con una sequenza di 0 e 1.

ii) Spiegare perché la rappresentazione binaria delle TM è importante per dimostrare che esistono linguaggi non RE.

iii) Dare un esempio di linguaggio non RE e dimostrare che effettivamente è non RE.

i) Una macchina di Turing viene codificata con una sequenza di 0 e 1 che rappresenta le transizioni della TM. Una transizione: $\delta(q_i, X_k) = (q_j, X_t, L)$ è rappresentata da $0^i10^k10^j10^t10$. Quindi ogni simbolo è rappresentato da una opportuna sequenza di 0. Gli 1 servono per separare gli 0. Lo stato iniziale è sempre q_1 e lo stato finale q_2. C'è un solo stato finale. Il simbolo di nastra X_1 rappresenta lo 0, X_2 rappresenta 1 e X_3 è il blank. Gli altri simboli $X_4, X_5, ...$ sono simboli utili per la TM rappresentata. Left/Right sono rappresentati con 0/00. Le diverse transizioni sono separate da coppie di 1, quindi la rappresentazione di una TM è una stringa $C_1 11 C_2 11 ... 11 C_k$, dove ciascun C_l rappresenta una transizione come quella appena descritta.

ii) Che le TM siano stringhe binarie, permette di avere TM che ricevono TM come input. Seguendo questa idea è possibile definire una tabella infinita che ha stringhe binarie crescenti $w_1, w_2, ...$ nelle righe e nelle colonne. Si tratta delle stesse stringhe, ma nelle righe sono interpretate come TM e nelle colonne come input. In ogni entrata M[i,j] della tabella metteremo 1 se la TM w_i accetta la stringa w_j e 0 se non l'accetta. Su questa tabella useremo la tecnica detta della diagonalizzazione per dimostrare che esistono linguaggi che non sono RE, cioè che non sono riconosciuti da alcuna TM.

iii) Il linguaggio $L_d = \{w_i \mid w_i \textit{ non accetta } w_i \textit{ come input}\}$. Dimostriamo che L_d non è RE. Ragioniamo per assurdo e assumiamo che invece sia RE. Quindi esiste una TM, diciamo w_j, che accetta L_d. Adesso osserviamo come si comporta w_j con se stessa come input. Ovviamente può accettare w_j oppure no. Supponiamo che w_j accetti se stessa. Allora, visto che la TM w_j accetta L_d, significa che $w_j \in L_d$, ma se $w_j \in L_d$ allora, per definizione di L_d, w_j non accetta w_j. Quindi abbiamo una contraddizione. Supponiamo ora che w_j non accetti w_j, ma allora w_j dovrebbe essere in L_d, ma non lo è visto che w_j accetta L_d. Quindi abbiamo di nuovo una contraddizione. Il che dimostra che non esiste TM che accetta L_d.

5. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-completo.

Considerate la seguente variante del problema, chiamata 3WayPartitioning: stabilire se un insieme di numeri interi S può essere suddiviso in tre sottoinsiemi disgiunti S_1, S_2 e S_3 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 che è uguale alla somma dei numeri in S_3.

(a) Dimostrare che il problema 3WayPartitioning è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

(b) Dimostrare che il problema 3WayPartitioning è NP-hard, mostrando come si può risolvere il problema SetPartitioning usando il problema 3WayPartitioning come sottoprocedura.

(a) Il certificato è dato da tre insiemi di insiemi di numeri interi S_1, S_2, S_3. Per verificarlo occorre controllare che rispetti le seguenti condizioni:

- i tre insiemi devono essere una partizione dell'insieme S;
- la somma degli elementi in S_1 deve essere uguale alla somma degli elementi in S_2 che deve essere uguale alla somma dei numeri in S_3.

Entrambe le condizioni si possono verificare in tempo polinomiale.

(b) Dimostrare che 3WayPartitioning è NP-hard, usando SetPartitioning come problema di riferimento richiede diversi passaggi:

1. Descrivere un algoritmo per risolvere SetPartitioning usando 3WayPartitioning come subroutine. Questo algoritmo avrà la seguente forma: data un'istanza di SetPartitioning, trasformala in un'istanza di 3WayPartitioning, quindi chiama l'algoritmo magico black-box per 3WayPartitioning.
2. Dimostrare che la riduzione è corretta. Ciò richiede sempre due passaggi separati, che di solito hanno la seguente forma:
   - Dimostrare che l'algoritmo trasforma istanze "buone" di SetPartitioning in istanze "buone" di 3WayPartitioning.

- Dimostrare che se la trasformazione produce un'istanza "buona" di 3WayPartitioning, allora era partita da un'istanza "buona" di SetPartitioning.

3. Mostrare che la riduzione funziona in tempo polinomiale, a meno della chiamata (o delle chiamate) all'algoritmo magico black-box per 3WayPartitioning. (Questo di solito è banale.)

Una istanza di SetPartitioning è data da un insieme S di numeri interi da suddividere in due. Una istanza di 3WayPartitioning è data anch'essa da un insieme di numeri interi S'. Quindi la riduzione deve trasformare un insieme di numeri S input di SetPartitioning in un altro insieme di numeri S' che diventerà l'input per la black-box che risolve 3WayPartitioning.

Se $t = \sum_{x \in S} x$ è la somma dei numeri che appartengono ad S, la riduzione che crea S' aggiungendo un nuovo elemento a S di valore $t/2$ (metà della somma dei numeri che appartengono ad S). Dimostriamo che è corretta:

$\Rightarrow$ sia S un'istanza buona di SetPartitioning. Allora è possibile partizionare S in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. In particolare, sia S_1 che S_2 sommano a $t/2$. Se aggiungiamo il nuovo valore $t/2$ ad S_1 otteniamo una soluzione per 3WayPartitioning, in cui i tre insiemi sono S_1, S_2 e $S_3 = \{t/2\}$, e abbiamo dimostrato che S' è una istanza buona di 3WayPartitioning.

$\Leftarrow$ sia S' un'istanza buona di 3WayPartitioning. Allora è possibile partizionare S' in tre sottoinsiemi S_1, S_2 ed S_3 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 che è uguale alla somma dei numeri in S_3. Per come è definito S', abbiamo che ognuno dei tre insiemi somma a $t/2$. Di conseguenza uno dei tre insiemi, che possiamo chiamare S_3, contiene solamente il nuovo elemento $t/2$, mentre gli altri due, che chiamiamo S_1 e S_2, formano una partizione degli elementi dell'insieme S di partenza. Di conseguenza, abbiamo trovato una soluzione per SetPartitioning con input S.

Per completare la dimostrazione basta osservare che per costruire S' dobbiamo aggiungere un nuovo elemento ad S, operazione che si riesce a fare in tempo polinomiale.

Automi e Linguaggi Formali
Esame del 19 settembre 2019 – Parte II

Tempo a disposizione: 1 h 30 min

1. Dare la definizione dei linguaggi L_e e L_{ne}. Spiegare come si dimostra che L_{ne} non è ricorsivo. Si usa una riduzione.

$$L_e = \{M|L(M) = \emptyset\}, L_{ne} = \{M|L(N) \neq \emptyset\}$$

Per dimostrare che L_{ne} è non ricorsivo, si riduce L_u ad esso. La costruzione funziona nel modo seguente: prendiamo un'istanza qualsiasi di L_u, essa è (M, w), $(M, w) \in L_u$ sse M accetta w. L'istanza di L_{ne} corrispondente a (M, w) è una TM M' tale che, per ogni input x, simula M su w e accetta x solo se M accetta w. Ovviamente $M' \in L_{ne}$ sse $(M, w) \in L_u$. Da questo segue che L_{ne} è almeno tanto indecidibile quanto L_u e quindi non ricorsivo.

2. Si consideri le seguenti proprietà dei linguaggi RE:

i) il linguaggio è accettato da una machina di Turing;

ii) il linguaggio è accettato da una macchina di Turing i cui calcoli sono sempre finiti.

Per ciascuna proprietà, date la definizione del linguaggio corrispondente e spiegate se la proprietà è decidibile o no.

La proprietà (i) è soddisfatta da tutti i linguaggi RE, visto che coincide con la definizione di RE. Per cui è una proprietà banale e quindi il Teorema di Rice non si applica ad essa. Il corrispondente linguaggio consiste di tutte le stringhe in $\{0, 1\}^*$ dato che ogni tale stringa rappresenta una TM eventualmente una TM che non accetta nulla. La proprietà (ii) coincide con i linguaggi ricorsivi che sono un sottoinsieme stretto dei linguaggi RE, quindi non è una proprietà banale per cui il Teorema di Rice ci dice che è indecidibile. Il linguaggio corrispondente a (ii) è $\{M|L(M)$ è ricorsivo $\}$.

3. Il linguaggio $L = \{a^k b^{2k} c^{3k} \mid k \geq 0\}$ è CF o non CF? Nel primo caso fornire una CFG che generi L (o un PDA che lo riconosca). Nel secondo caso dimostrare che L non è CF.

L non è CF e useremo il Pumping Lemma (PL) per dimostrarlo. Supponiamo che lo sia. Quindi il PL ci dice che c'è un $n > 0$ tale che per ogni stringa z in L e tale che $|z| \geq n$,esiste una divisione $z = uvwxy$, tale che $|vwx| \leq n$, $|vx| \neq \epsilon$ e per ogni $i \geq 0, uv^i wx^i y \in L$. Per il linguaggio L dell'esercizio, troviamo una contraddizione prendendo $z = a^n b^{2n} c^{3n}$. Ovviamente $|z| > n$, e quindi esiste una scomposizione $z = uvwxy$ tale che $|uwx| \leq n$. Per questo motivo uwx potrà al massimo contenere 2 simboli diversi, cioè a e b oppure b e c, ma è impossibile che contenga a, b e c. Per cui, pompando u e x aumenteremo il numero di 2 simboli lasciando inalterato il terzo. Per cui otterremo stringhe non in L. A maggior ragione questo succede se uwx contiene solo a o solo b o solo c.

4. Descrivere un PDA che accetta per pila vuota ed è capace di riconoscere il linguaggio $L = \{(ab)^n (ca)^n \mid n \geq 0\}$.

Il vostro è un automa deterministico o nondeterministico? Spiegare la risposta.

Il PDA deve essere nondeterministico perché deve accettare anche ϵ. La sua funzione di transizione è questa:

$$\delta(q, a, Z) = \{(q, aZ)\}, \delta(q, b, a) = \{(q, ba)\}, \delta(q, a, b) = \{(q, ab)\}, \delta(q, \epsilon, Z) = \{(q, \epsilon)\},$$

$$\delta(q, c, b) = \{(p, \epsilon)\}, \delta(p, a, a) = \{(q, \epsilon)\}, \delta(p, c, b) = \{(p, \epsilon)\}, \delta(p, \epsilon, Z) = \{(p, \epsilon)\}$$

E' non deterministico a causa di:

$$\delta(q, a, Z) = \{(q, aZ)\} \text{ e } \delta(q, \epsilon, Z) = \{(q, \epsilon)\}$$

5. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Un *circuito* 1/3-*Hamiltoniano* in un grafo G è un ciclo che attraversa esattamente una volta *un terzo* dei vertici del grafo. Il *problema del circuito* 1/3-*Hamiltoniano* è il problema di stabilire se un grafo contiene un circuito 1/3-Hamiltoniano.

(a) Dimostrare che il problema del circuito 1/3-Hamiltoniano è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema del circuito 1/3-Hamiltoniano come sottoprocedura.

Per la parte (a) un certificato è una sequenza di n/3 nodi. E' facile (lineare) verificare se un tale sequenza di nodi forma un circuito hamiltoniano o no. Per la parte (b), mappiamo una qualsiasi istanza del problema del circuito Hamiltoniamo che è formata da un grafo G in un'istanza del 1/3-*circuito Hamiltoniano, composta da 3 copie di G completamente disgiunte tra loro. E' facile vedere che se il grafo con 3 copie di G ha un circuito Hamiltoniano su 1/3 dei nodi, lo ha su una delle 3 componenti e quindi su G. Se invece il grafo con 3 copie di G non ha un circuito Hamiltoniano su 1/3 dei nodi, allora non c'è circuito Hamiltoniano su G. Quindi abbiamo ridotto il problema del circuito Hamiltoniano a quello del 1/3 Hamiltoniano, dimostrando, assieme al punto (a), che esso è NP completo.*

Automi e Linguaggi Formali
Soluzione della prima prova intermedia

1. Per risolvere l'esercizio dobbiamo dimostrare che *(a)* ogni linguaggio riconosciuto da una TM con reset a sinistra è Turing-riconoscibile e *(b)* ogni linguaggio Turing-riconoscibile è riconosciuto da una TM con reset a sinistra.

(a) Mostriamo come convertire una TM con reset a sinistra M in una TM standard S equivalente. S simula il comportamento di M nel modo seguente. Se la mossa da simulare prevede uno spostamento a destra, allora S esegue direttamente la mossa. Se la mossa prevede un $RESET$, allora S scrive il nuovo simbolo sul nastro, poi scorre il nastro a sinistra finché non trova il simbolo $\triangleright$, e riprende la simulazione dall'inizio del nastro. Per ogni stato q di M, S possiede uno stato q_{RESET} che serve per simulare il reset e riprendere la simulazione dallo stato corretto.

S = "Su input w:

1. scrive il simbolo $\triangleright$ subito prima dell'input, in modo che il nastro contenga $\triangleright w$.
2. Se la mossa da simulare è $\delta(q, a) = (r, b, R)$, allora S la esegue direttamente: scrive b sul nastro, muove la testina a destra e va nello stato r.
3. Se la mossa da simulare è $\delta(q, a) = (r, b, RESET)$, allora S esegue le seguenti operazioni: scrive b sul nastro, poi muove la testina a sinistra e va nello stato r_{RESET}. La macchina rimane nello stato r_{RESET} e continua a muovere la testina a sinistra finché non trova il simbolo $\triangleright$. A quel punto la macchina sposta la testina un'ultima volta a sinistra, poi di una cella a destra per tornare sopra al simbolo di fine nastro. La computazione riprende dallo stato r.
4. Se non sei nello stato di accettazione o di rifiuto, ripeti da 2."

(b) Mostriamo come convertire una TM standard S in una TM con reset a sinistra M equivalente. M simula il comportamento di S nel modo seguente. Se la mossa da simulare prevede uno spostamento a destra, allora M può eseguire direttamente la mossa. Se la mossa da simulare prevede uno spostamento a sinistra, allora M simula la mossa come descritto dall'algoritmo seguente. L'algoritmo usa un nuovo simbolo $\triangleleft$ per identificare la fine della porzione di nastro usata fino a quel momento, e può marcare le celle del nastro ponendo un punto al di sopra di un simbolo.

M = "Su input w:

1. Scrive il simbolo $\triangleleft$ subito dopo l'input, per marcare la fine della porzione di nastro utilizzata. Il nastro contiene $\triangleright w \triangleleft$.
2. Simula il comportamento di S. Se la mossa da simulare è $\delta(q, a) = (r, b, R)$, allora M la esegue direttamente: scrive b sul nastro, muove la testina a destra e va nello stato r. Se muovendosi a destra la macchina si sposta sulla cella che contiene $\triangleleft$, allora questo significa che S ha spostato la testina sulla parte di nastro vuota non usata in precedenza. Quindi M scrive un simbolo blank marcato su questa cella, sposta $\triangleleft$ di una cella a destra, e fa un reset a sinistra. Dopo il reset si muove a destra fino al blank marcato, e prosegue con la simulazione mossa successiva.
3. Se la mossa da simulare è $\delta(q, a) = (r, b, L)$, allora S esegue le seguenti operazioni:
   - 3.1 scrive b sul nastro, marcandolo con un punto, poi fa un reset a sinistra
   - 3.2 Se il simbolo subito dopo $\triangleright$ è già marcato, allora vuol dire che S ha spostato la testina sulla parte vuota di sinistra del nastro. Quindi M scrive un blank e sposta il contenuto del nastro di una cella a destra finché non trova il simbolo di fine nastro $\triangleleft$. Fa un reset a sinistra e prosegue con la simulazione della prossima mossa dal nuovo blank posto subito dopo l'inizio del nastro. Se il simbolo subito dopo $\triangleright$ non è marcato, lo marca, resetta a sinistra e prosegue con i passi successivi.
   - 3.3 Si muove a destra fino al primo simbolo marcato, e poi a destra di nuovo.
   - 3.4 se la cella in cui si trova è marcata, allora è la cella da cui è partita la simulazione. Toglie la marcatura e resetta. Si muove a destra finché non trova una cella marcata. Questa cella è quella immediatamente precedente la cella di partenza, e la simulazione della mossa è terminata
   - 3.5 se la cella in cui si trova non è marcata, la marca, resetta, si muove a destra finché non trova una marcatura, cancella la marcatura e riprende da 3.3.
4. Se non sei nello stato di accettazione o di rifiuto, ripeti da 2."

2. (a) Dimostriamo separatamente i due versi del se e solo se.

- Supponiamo che A sia Turing-riconoscibile. Allora esiste una Macchina di Turing M che riconosce A. Consideriamo la funzione f tale che $f(w) = \langle M, w \rangle$ per ogni stringa $w \in \Sigma^*$. Questa funzione è calcolabile ed è una funzione di riduzione da A a A_{TM}. Infatti, se $w \in A$ allora anche $\langle M, w \rangle \in A_{TM}$ perché la macchina M accetta le stringhe che appartengono ad A. Viceversa, se $w \notin A$, allora $\langle M, w \rangle \notin A_{TM}$ perché la macchina M rifiuta le stringhe che non appartengono ad A.
- Supponiamo che $A \leq_m A_{TM}$. Sappiamo che A_{TM} è un linguaggio Turing-riconoscibile. Per le proprietà delle riduzioni mediante funzione, possiamo concludere che anche A è Turing-riconoscibile.

(b) Dimostriamo separatamente i due versi del se e solo se.

- Supponiamo che A sia decidibile. Allora esiste una Macchina di Turing M che decide A. Consideriamo la funzione f definita nel modo seguente:

$$f(w) = \begin{cases} 01 & \text{se } M \text{ accetta } w \\ 10 & \text{se } M \text{ rifiuta } w \end{cases}$$

  M è un decisore e la sua computazione termina sempre. Quindi la funzione f può essere calcolata dalla seguente macchina di Turing:

  $F =$ "su input w:
  1. Esegui M su input w.
  2. Se M accetta, restituisci 01, se M rifiuta, restituisci 10."

  f è anche funzione di riduzione da A a 0^*1^*. Infatti, se $w \in A$ allora $f(w) = 01$ che appartiene al linguaggio 0^*1^*. Viceversa, se $w \notin A$, allora $f(w) = 10$ che non appartiene a 0^*1^*.
- Supponiamo che $A \leq_m 0^*1^*$. Sappiamo che 0^*1^* è un linguaggio decidibile. Per le proprietà delle riduzioni mediante funzione, possiamo concludere che anche A è decidibile.

3. Per mostrare che *LPATH* è NP-completo, dimostriamo prima che *LPATH* è in NP, e poi che è NP-Hard.

- *LPATH* è in NP. Il cammino da s a t di lunghezza maggiore o uguale a k è il certificato. Il seguente algoritmo è un verificatore per *LPATH*:

  V = "Su input $\langle\langle G, s, t, k\rangle, c\rangle$:
  1. Controlla che c sia una sequenza di vertici di G, $v_1, \ldots, v_m$ e che m sia minore o uguale al numero di vertici in G più uno. Se non lo è, rifiuta.
  2. Se la sequenza è di lunghezza minore o uguale a k, rifiuta.
  3. Controlla se $s = v_1$ e $t = v_m$. Se una delle due è falsa, rifiuta.
  4. Controlla se ci sono ripetizioni nella sequenza. Se ne trova una diversa da $v_1 = v_m$, rifiuta.
  5. 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.
  6. Se tutti i test sono stati superati, accetta."

  Per analizzare questo algoritmo e dimostrare che viene eseguito in tempo polinomiale, esaminiamo ogni sua fase. Ognuna delle fasi è un controllo sugli m elementi del certificato, e quindi richiede un tempo polinomiale rispetto ad m. Poiché l'algoritmo rifiuta immediatamente quando m è maggiore del numero di vertici di G più uno, allora possiamo concludere che l'algoritmo richiede un tempo polinomiale rispetto al numero di vertici del grafo.

- Dimostriamo che *LPATH* è NP-Hard per riduzione polinomiale da *HAMILTON* a *LPATH*. La funzione di riduzione polinomiale f prende in input un grafo $\langle G\rangle$ e produce come output la quadrupla $\langle G, s, s, n\rangle$ dove s è un vertice arbitrario di G e n è uguale al numero di vertici di G. Dimostriamo che la riduzione polinomiale è corretta:
  - Se $\langle G\rangle \in HAMILTON$, allora esiste un circuito Hamiltoniano in G. Dato un qualsiasi vertice s di G, possiamo costruire un cammino che parte da s e segue il circuito Hamiltoniano per tornare in s. Questo cammino attraversa tutti gli altri vertici di G prima di tornare in s e sarà quindi di lunghezza n. Di conseguenza $\langle G, s, s, n\rangle \in LPATH$.
  - Se $\langle G, s, s, n\rangle \in LPATH$, allora esiste un cammino semplice nel grafo G che inizia e termina in s ed è di lunghezza maggiore o uguale a n. Un cammino semplice non ha ripetizioni, ad eccezione dei vertici iniziali e finali. Un cammino semplice da s ad s di lunghezza n deve attraversare tutti gli altri nodi una sola volta prima di tornare in s, ed è quindi un circuito Hamiltoniano per G. Cammini semplici più lunghi non possono esistere perché dovrebbero ripetere dei vertici. Quindi l'esistenza di un cammino semplice nel grafo G che inizia e termina in s ed è di lunghezza maggiore o uguale a n implica l'esistenza di un circuito Hamiltoniano in G, ed abbiamo dimostrato che $\langle G\rangle \in HAMILTON$.

  La funzione di riduzione si limita ad aggiungere tre nuovi elementi dopo la codifica del grafo G: due vertici ed un numero, operazione che si può fare in tempo polinomiale.

Automi e Linguaggi Formali – a.a. 2020/21
Soluzione della prima prova intermedia

1. Per risolvere l'esercizio dobbiamo dimostrare che *(a)* ogni linguaggio riconosciuto da una tag-Turing machine è Turing-riconoscibile e *(b)* ogni linguaggio Turing-riconoscibile è riconosciuto da una tag-Turing machine.

(a) Mostriamo come convertire una tag-Turing machine M in una TM deterministica a nastro singolo S equivalente. S simula il comportamento di M tenendo traccia delle posizioni delle due testine marcando la cella dove si trova la testina di lettura con un pallino sopra il simbolo, e marcando la cella la cella dove si trova la testina di scrittura con un pallino sotto il simbolo. Per simulare il comportamento di M la TM S scorre il nastro e aggiorna le posizioni delle testine ed il contenuto delle celle come indicato dalla funzione di transizione di M.

$S =$ "Su input $w = w_1 w_2 \dots w_n$:

1. Scrivi un pallino sopra il primo simbolo di w e un pallino sotto la prima cella vuota dopo l'input, in modo che il nastro contenga
$$\overset{\bullet}{w}_1 w_2 \dots w_n \underset{\bullet}{\sqcup}.$$
2. Per simulare una transizione, S scorre il nastro per trovare la posizione della testina di lettura e determinare il simbolo letto da M. Se la funzione di transizione stabilisce che la testina di lettura deve spostarsi a destra, allora S sposta il pallino nella cella immediatamente a destra, altrimenti lo lascia dov'è. Successivamente S si sposta verso destra finché non trova la cella dove si trova la testina di scrittura, scrive il simbolo stabilito dalla funzione di transizione nella cella e sposta la marcatura nella cella immediatamente a destra.
3. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di M allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

(b) Mostriamo come convertire una TM deterministica a nastro singolo S in una tag-Turing machine M equivalente. M simula il comportamento di S memorizzando sul nastro una sequenza di configurazioni di S separate dal simbolo #. All'interno di ogni configurazione un pallino marca il simbolo sotto la testina di S. Per simulare il comportamento di S la tag-Turing machine M scorre la configurazione corrente e scrivendo man mano la prossima configurazione sul nastro.

$M =$ "Su input $w = w_1 w_2 \dots w_n$:

1. Scrive il simbolo # subito dopo l'input, seguito dalla configurazione iniziale, in modo che il nastro contenga
$$w_1 w_2 \dots w_n \# \overset{\bullet}{w}_1 w_2 \dots w_n \sqcup,$$
che la testina di lettura si trovi in corrispondenza del $\overset{\bullet}{w}_1$ e quella di lettura in corrispondenza del blank dopo la configurazione iniziale. Imposta lo stato corrente della simulazione st allo stato iniziale di S e memorizza l'ultimo simbolo letto $prec = \#$. L'informazione sui valori di st e $prec$ sono codificate all'interno degli stato di M.
2. Finché il simbolo sotto la testina di lettura non è marcato, scrive il simbolo precedente $prec$ e muove a destra. Aggiorna il valore di $prec$ con il simbolo letto.
3. Quando si trova un simbolo marcato $\overset{\bullet}{a}$ e $\delta(st, a) = (q, b, R)$:
   - aggiorna lo stato della simulazione $st = q$;
   - scrive $prec$ seguito da b, poi muove la testina di lettura a destra;
   - scrive il simbolo sotto la testina marcandolo con un pallino.
4. Quando si trova un simbolo marcato $\overset{\bullet}{a}$ e $\delta(st, a) = (q, b, L)$:
   - aggiorna lo stato della simulazione $st = q$;
   - scrive $\overset{\bullet}{prec}$; se $prec = \#$ scrive $\#\underset{\bullet}{\sqcup}$;
   - scrive b.
5. Copia il resto della configurazione fino al # escluso. Al termine della copia la testina di lettura di trova in corrispondenza della prima cella nella configurazione corrente, e quella di lettura sulla cella vuota dopo la configurazione.
6. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di S, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di M allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

2. (a) Dimostriamo che AlwaysDiverge è un linguaggio indecidibile mostrando che $\overline{A_{TM}}$ è riducibile ad AlwaysDiverge. La funzione di riduzione f è calcolata dalla seguente macchina di Turing:

$F =$ "su input $\langle M, w\rangle$, dove M è una TM e w una stringa:
1. Costruisci la seguente macchina M':
   $M' =$ "su input x:
   1. Se $x \neq w$, vai in loop.
   2. Se $x = w$, esegue M su input w.
   3. Se M accetta, *accetta*.
   4. Se M rifiuta, vai in loop."
2. Restituisci $\langle M'\rangle$."

Dimostriamo che f è una funzione di riduzione da $\overline{A_{TM}}$ ad AlwaysDiverge.

- Se $\langle M, w\rangle \in \overline{A_{TM}}$ allora la computazione di M su w non termina oppure termina rifiutando. Di conseguenza la macchina M' costruita dalla funzione non termina mai la computazione per qualsiasi input. Quindi $f(\langle M, w\rangle) = \langle M'\rangle \in$ AlwaysDiverge.
- Viceversa, se $\langle M, w\rangle \notin \overline{A_{TM}}$ allora la computazione di M su w termina con accettazione. Di conseguenza la macchina M' termina la computazione sull'input w e $f(\langle M, w\rangle) = \langle M'\rangle \notin$ AlwaysDiverge.

Per concludere, siccome abbiamo dimostrato che $\overline{A_{TM}} \leq_m$ AlwaysDiverge e sappiamo che $\overline{A_{TM}}$ è indecidibile, allora possiamo concludere che AlwaysDiverge è indecidibile.

(b) AlwaysDiverge è un linguaggio coTuring-riconoscibile. Dato un ordinamento $s_1, s_2, \ldots$ di tutte le stringhe in Σ^*, la seguente TM riconosce il complementare $\overline{\text{AlwaysDiverge}}$:

$D =$ "su input $\langle M\rangle$, dove M è una TM:
1. Ripeti per $i = 1, 2, 3, \ldots$:
2. Esegue M per i passi di computazione su ogni input $s_1, s_2, \ldots s_i$.
3. Se M termina la computazione su almeno uno, *accetta*. Altrimenti continua."

3. (a) WSP è in NP. La disposizione degli ospiti tra i tavoli è il certificato. Se gli ospiti sono numerati da 1 a n la possiamo rappresentare con un vettore T tale che $T[i]$ è il tavolo assegnato all'ospite i. Il seguente algoritmo è un verificatore per WSP:

V = "Su input $\langle\langle n, k, R\rangle, T\rangle$:

1. Controlla che T sia un vettore di n elementi dove ogni elemento ha un valore compreso tra 1 e k. Se non lo è, rifiuta.
2. Per ogni coppia i, j tale che $R[i, j] = 1$, controlla che $T[i] = T[j]$. Se il controllo fallisce, rifiuta.
3. Per ogni coppia i, j tale che $R[i, j] = -1$, controlla che $T[i] \neq T[j]$. Se il controllo fallisce, rifiuta.
4. Se tutti i test sono stati superati, accetta."

Per analizzare questo algoritmo e dimostrare che viene eseguito in tempo polinomiale, esaminiamo ogni sua fase. La prima fase è un controllo sugli n elementi del vettore T, e quindi richiede un tempo polinomiale rispetto alla dimensione dell'input. La seconda e terza fase controllano gli elementi della matrice R, operazione che si può fare in tempo polinomiale rispetto alla dimensione dell'input.

(b) Dimostriamo che WSP è NP-Hard per riduzione polinomiale da 3-COLOR a WSP. La funzione di riduzione polinomiale f prende in input un grafo $\langle G\rangle$ e produce come output la tripla $\langle n, 3, R\rangle$ dove n è il numero di vertici di G e la matrice R è definita in questo modo:

$$R[i, j] = \begin{cases} -1 & \text{se c'è un arco da } i \text{ a } j \text{ in } G \\ 0 & \text{altrimenti} \end{cases}$$

Dimostriamo che la riduzione polinomiale è corretta:

- Se $\langle G\rangle \in$ 3-COLOR, allora esiste un modo per colorare G con tre colori $1, 2, 3$. La disposizione degli ospiti che fa sedere l'ospite i nel tavolo corrispondente al colore del vertice i nel grafo è corretta:
  - ogni invitato siede ad un solo tavolo;
  - per ogni coppia di invitati rivali i, j si ha che $R[i, j] = -1$ e, per la definizione della funzione di riduzione, l'arco (i, j) appartiene a G. Quindi la colorazione assegna colori diversi ad i e j, ed i due ospiti siedono su tavoli diversi;
  - per la definizione della funzione di riduzione non ci sono ospiti che sono amici tra di loro.
- Se $\langle n, 3, R\rangle \in$ WSP, allora esiste una disposizione degli n ospiti su 3 tavoli dove i rivali siedono sempre su tavoli diversi. La colorazione che assegna al vertice i il colore corrispondente al tavolo dove siede l'ospite i è corretta: se c'è un arco tra i e j allora i due ospiti sono rivali e siederanno su tavoli diversi, cioè avranno colori diversi nella colorazione. Di conseguenza abbiamo dimostrato che $\langle G\rangle \in$ 3-COLOR.

La funzione di riduzione deve contare i vertici del grafo G e costruire la matrice R di dimensione $n \times n$, operazioni che si possono fare in tempo polinomiale.

Automi e Linguaggi Formali – 17/6/2021
Seconda parte – Decidibilità e complessità – Soluzione

1. *Una* macchina di Turing bidimensionale *utilizza una griglia bidimensionale infinita di celle come nastro. Ad ogni transizione, la testina può spostarsi dalla cella corrente ad una qualsiasi delle quattro celle adiacenti. La funzione di transizione di tale macchina ha la forma*

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\uparrow, \downarrow, \rightarrow, \leftarrow\},$$

dove le frecce indicano in quale direzione si muove la testina dopo aver scritto il simbolo sulla cella corrente.

Dimostra che ogni macchina di Turing bidimensionale può essere simulata da una macchina di Turing deterministica a nastro singolo.

Mostriamo come simulare una TM bidimensionale B con una TM deterministica a nastro singolo S. S memorizza il contenuto della griglia bidimensionale sul nastro come una sequenza di stringhe separate da #, ognuna delle quali rappresenta una riga della griglia. Due cancelletti consecutivi ## segnano l'inizio e la fine della rappresentazione della griglia. La posizione della testina di B viene indicata marcando la cella con $\hat{}$. Nelle altre righe, un pallino $^\bullet$ indica che la testina si trova su quella colonna della griglia, ma in una riga diversa. La TM a nastro singolo S funziona come segue:

$S =$ "su input w:

1. Sostituisce w con la configurazione iniziale $\#\#w\#\#$ e marca con $\hat{}$ il primo simbolo di w.
2. Scorre il nastro finché non trova la cella marcata con $\hat{}$.
3. Aggiorna il nastro in accordo con la funzione di transizione di B:
   - Se $\delta(r, a) = (s, b, \rightarrow)$, scrive b non marcato sulla cella corrente, sposta $\hat{}$ sulla cella immediatamente a destra. Poi sposta di una cella a destra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## finale, di una cella più a destra.
   - Se $\delta(r, a) = (s, b, \leftarrow)$, scrive b non marcato sulla cella corrente, sposta $\hat{}$ sulla cella immediatamente a sinistra. Poi sposta di una cella a sinistra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## iniziale, di una cella più a sinistra.
   - Se $\delta(r, a) = (s, b, \uparrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\hat{}$ sulla prima cella marcata con un pallino posta a sinistra della cella corrente. Se questa cella marcata non esiste, aggiunge una nuova riga composta solo da blank all'inizio della configurazione.
   - Se $\delta(r, a) = (s, b, \downarrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\hat{}$ sulla prima cella marcata con un pallino posta a destra della cella corrente. Se questa cella non esiste, aggiunge una nuova riga composta solo da blank alla fine della configurazione.
4. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di B, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di B allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

2. *Dimostra che il seguente linguaggio è indecidibile:*

$$A_{1010} = \{\langle M \rangle \mid M \text{ è una TM tale che } 1010 \in L(M)\}.$$

Dimostriamo che A_{1010} è un linguaggio indecidibile mostrando che A_{TM} è riducibile ad A_{1010}. La funzione di riduzione f è calcolata dalla seguente macchina di Turing:

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

1. Costruisci la seguente macchina M_w:
   $M_w =$ "su input x:
   1. Se $x \neq 1010$, rifiuta.
   2. Se $x = 1010$, esegue M su input w.
   3. Se M accetta, *accetta*.
   4. Se M rifiuta, *rifiuta*."
2. Restituisci $\langle M_w \rangle$."

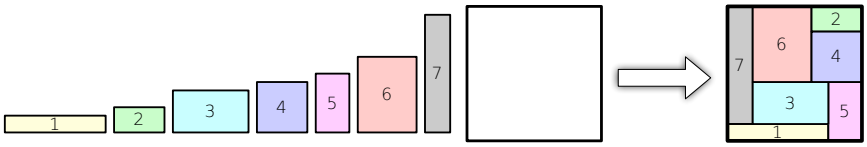
Dimostriamo che f è una funzione di riduzione da A_{TM} ad A_{1010}.

- Se $\langle M, w \rangle \in A_{TM}$ allora la TM M accetta w. Di conseguenza la macchina M_w costruita dalla funzione accetta la parola 1010. Quindi $f(\langle M, w \rangle) = \langle M_w \rangle \in A_{1010}$.
- Viceversa, se $\langle M, w \rangle \notin A_{TM}$ allora la computazione di M su w non termina o termina con rifiuto. Di conseguenza la macchina M_w rifiuta 1010 e $f(\langle M, w \rangle) = \langle M_w \rangle \notin A_{1010}$.

Per concludere, siccome abbiamo dimostrato che $A_{TM} \leq_m A_{1010}$ e sappiamo che A_{TM} è indecidibile, allora possiamo concludere che A_{1010} è indecidibile.

3. *Il problema* SetPartitioning *chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-hard.*

Il problema RectangleTiling *è definito come segue: dato un rettangolo grande e diversi rettangoli più piccoli, determinare se i rettangoli più piccoli possono essere posizionati all'interno del rettangolo grande senza sovrapposizioni e senza lasciare spazi vuoti.*



Un'istanza positiva di RectangleTiling.

Dimostra che RectangleTiling *è NP-hard, usando* SetPartitioning *come problema di riferimento.*

Dimostriamo che RectangleTiling è NP-Hard per riduzione polinomiale da SetPartitioning. La funzione di riduzione polinomiale prende in input un insieme di interi positivi $S = \{s_1, \ldots, s_n\}$ e costruisce un'istanza di RectangleTiling come segue:

- i rettangoli piccoli hanno altezza 1 e base uguale ai numeri in S moltiplicati per 3: $(3s_1, 1), \ldots, (3s_n, 1)$;
- il rettangolo grande ha altezza 2 e base $\frac{3}{2}N$, dove $N = \sum_{i=1}^{n} s_i$ è la somma dei numeri in S.

Dimostriamo che esiste un modo per suddividere S in due insiemi S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 se e solo se esiste un tiling corretto:

- Supponiamo esista un modo per suddividere S nei due insiemi S_1 e S_2. Posizioniamo i rettangoli che corrispondono ai numeri in S_1 in una fila orizzontale, ed i rettangoli che corrispondono ad S_2 in un'altra fila orizzontale. Le due file hanno altezza 1 e base $\frac{3}{2}N$, quindi formano un tiling corretto.
- Supponiamo che esista un modo per disporre i rettangoli piccoli all'interno del rettangolo grande senza sovrapposizioni né spazi vuoti. Moltiplicare le base dei rettangoli per 3 serve ad impedire che un rettangolo piccolo possa essere disposto in verticale all'interno del rettangolo grande. Quindi il tiling valido è composto da due file di rettangoli disposti in orizzontale. Mettiamo i numeri corrispondenti ai rettangoli in una fila in S_1 e quelli corrispondenti all'altra fila in S_2. La somma dei numeri in S_1 ed S_2 è pari ad $N/2$, e quindi rappresenta una soluzione per SetPartitioning.

Per costruire l'istanza di RectangleTiling basta scorrere una volta l'insieme S, con un costo polinomiale.

Automi e Linguaggi Formali
Soluzione dell'esame scritto del 1 Settembre 2021

1. (8 punti) *Considera il linguaggio*

$$L = \{0^m 1^n \mid m/n \text{ è un numero intero}\}.$$

Dimostra che L non è regolare.

Usiamo il Pumping Lemma per dimostrare che il linguaggio non è regolare.

Supponiamo per assurdo che L sia regolare:

- sia k la lunghezza data dal Pumping Lemma;
- consideriamo la parola $w = 0^{k+1}1^{k+1}$, che è di lunghezza maggiore di k ed appartiene ad L perché $(k+1)/(k+1) = 1$;
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$;
- poiché $|xy| \leq k$, allora x e y sono entrambe contenute nella sequenza di 0. Inoltre, siccome $y \neq \varepsilon$, abbiamo che $x = 0^q$ e $y = 0^p$ per qualche $q \geq 0$ e $p > 0$. z contiene la parte rimanente della stringa: $z = 0^{k+1-q-p}1^{k+1}$. Consideriamo l'esponente $i = 0$: la parola xy^0z ha la forma

  $$xy^0z = xz = 0^q 0^{k+1-q-p}1^{k+1} = 0^{k+1-p}1^{k+1}.$$

  Si può notare che $(k+1-p)/(k+1)$ è un numero strettamente compreso tra 0 e 1, e quindi non può essere un numero intero. Di conseguenza, la parola non appartiene al linguaggio L, in contraddizione con l'enunciato del Pumping Lemma.

2. (8 punti) *Per ogni linguaggio L, sia* $\text{prefix}(L) = \{u \mid uv \in L \text{ per qualche stringa } v\}$. *Dimostra che se L è un linguaggio context-free, allora anche* $\text{prefix}(L)$ *è un linguaggio context-free.*

Se L è un linguaggio context-free, allora esiste una grammatica G *in forma normale di Chomski* che lo genera. Possiamo costruire una grammatica G' che genera il linguaggio $\text{prefix}(L)$ in questo modo:

- per ogni variabile V di G, G' contiene sia la variabile V che una nuova variabile V'. La variabile V' viene usata per generare i prefissi delle parole che sono generate da V;
- tutte le regole di G sono anche regole di G';
- per ogni variabile V di G, le regole $V' \to V$ e $V' \to \varepsilon$ appartengono a G;
- per ogni regola $V \to AB$ di G, le regole $V' \to AB'$ e $V' \to A'$ appartengono a G';
- se S è la variabile iniziale di G, allora S' è la variabile iniziale di G'.

3. (8 punti) *Una* Turing machine con alfabeto binario *è una macchina di Turing deterministica a singolo nastro dove l'alfabeto di input è $\Sigma = \{0,1\}$ e l'alfabeto del nastro è $\Gamma = \{0,1,\sqcup\}$. Questo significa che la macchina può scrivere sul nastro solo i simboli $0,1$ e blank: non può usare altri simboli né marcare i simboli sul nastro.*

Dimostra che che le Turing machine con alfabeto binario machine riconoscono tutti e soli i linguaggi Turing-riconoscibili sull'alfabeto $\{0,1\}$.

Per risolvere l'esercizio dobbiamo dimostrare che *(a)* ogni linguaggio riconosciuto da una Turing machine con alfabeto binario è Turing-riconoscibile e *(b)* ogni linguaggio Turing-riconoscibile sull'alfabeto $\{0,1\}$ è riconosciuto da una Turing machine con alfabeto binario.

(a) Questo caso è semplice: una Turing machine con alfabeto binario è un caso speciale di Turing machine deterministica a nastro singolo. Quindi ogni linguaggio riconosciuto da una Turing machine con alfabeto binario è anche Turing-riconoscibile.

(b) Per dimostrare questo caso, consideriamo un linguaggio L Turing-riconoscibile, e sia M una Turing machine deterministica a nastro singolo che lo riconosce. Questa TM potrebbe avere un alfabeto del nastro Γ che contiene altri simboli oltre a $0,1$ e blank. Per esempio potrebbe contenere simboli marcati o separatori.
Per costruire una TM con alfabeto binario B che simula il comportamento di M dobbiamo come prima cosa stabilire una *codifica binaria* dei simboli nell'alfabeto del nastro Γ di M. Questa codifica è una funzione C che assegna ad ogni simbolo $a \in \Gamma$ una sequenza di k cifre binarie, dove k è un valore scelto in modo tale che ad ogni simbolo corrisponda una codifica diversa. Per esempio, se Γ contiene 4 simboli, allora $k = 2$, perché con 2 bit si rappresentano 4 valori diversi. Se Γ contiene 8 simboli, allora $k = 3$, e così via.

La TM con alfabeto binario B che simula M è definita in questo modo:

$B =$ "su input w:

1. Sostituisce $w = w_1 w_2 \dots w_n$ con la codifica binaria $C(w_1)C(w_2)\dots C(w_n)$, e riporta la testina sul primo simbolo di $C(w_1)$.
2. Scorre il nastro verso destra per leggere k cifre binarie: in questo modo la macchina stabilisce qual è il simbolo a presente sul nastro di M. Va a sinistra di k celle.
3. Aggiorna il nastro in accordo con la funzione di transizione di M:
   - Se $\delta(r, a) = (s, b, R)$, scrive la codifica binaria di b sul nastro.
   - Se $\delta(r, a) = (s, b, L)$, scrive la codifica binaria di b sul nastro e sposta la testina a sinistra di $2k$ celle.
4. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di M allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

4. (8 punti) *Supponiamo che un impianto industriale costituito da m linee di produzione identiche debba eseguire n lavori distinti. Ognuno dei lavori può essere svolto da una qualsiasi delle linee di produzione, e richiede un certo tempo per essere completato. Il* problema del bilanciamento del carico *(*LoadBalance*) chiede di trovare un* assegnamento *dei lavori alle linee di produzione che permetta di completare tutti i lavori entro un tempo limite k.*

Più precisamente, possiamo rappresentare l'input del problema con una tripla $\langle m, T, k\rangle$ dove:

- *m è il numero di linee di produzione;*
- *$T[1 \dots n]$ è un array di numeri interi positivi dove $T[j]$ è il tempo di esecuzione del lavoro j;*
- *k è un limite superiore al tempo di completamento di tutti i lavori.*

Per risolvere il problema vi si chiede di trovare un array $A[1 \dots n]$ con gli assegnamenti, dove $A[j] = i$ significa che il lavoro j è assegnato alla linea di produzione i. Il tempo di completamento (o makespan*) di A è il tempo massimo di occupazione di una qualsiasi linea di produzione:*

$$\text{makespan}(A) = \max_{1 \le i \le m} \sum_{A[j]=i} T[j]$$

Load Balance *è il problema di trovare un assegnamento con makespan minore o uguale al limite superiore k:*

$$\text{LoadBalance} = \{\langle m, T, k\rangle \mid \textit{esiste un assegnamento } A \textit{ degli } n \textit{ lavori su } m \textit{ linee di produzione tale che } \text{makespan}(A) \le k\}$$

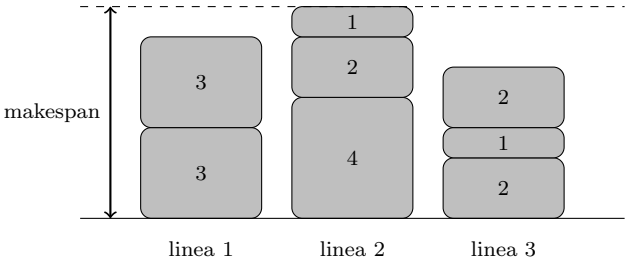


Figura 1: Esempio di assegnamento dei lavori $T = \{1, 1, 2, 2, 2, 3, 3, 4\}$ su 3 linee con makespan 7.

(a) Dimostra che LoadBalance *è un problema NP.*
(b) Dimostra che LoadBalance *è NP-hard, usando* SetPartitioning *come problema NP-hard di riferimento.*

(a) LoadBalance è in NP. L'array A con gli assegnamenti è il certificato. Il seguente algoritmo è un verificatore per LoadBalance:

$V =$ "Su input $\langle\langle m, T, k\rangle, A\rangle$:

1. Controlla che A sia un vettore di n elementi dove ogni elemento ha un valore compreso tra 1 e m. Se non lo è, rifiuta.
2. Calcola makespan(A): se è minore o uguale a k accetta, altrimenti rifiuta."

Per analizzare questo algoritmo e dimostrare che viene eseguito in tempo polinomiale, esaminiamo ogni sua fase. La prima fase è un controllo sugli n elementi del vettore A, e quindi richiede un tempo polinomiale rispetto alla dimensione dell'input. Per calcolare il makespan, la seconda fase deve calcolare il tempo di occupazione di ognuna delle m linee e poi trovare il massimo tra i tempi di occupazione, operazioni che si possono fare in tempo polinomiale rispetto alla dimensione dell'input.

(b) Dimostriamo che LoadBalance è NP-Hard per riduzione polinomiale da SetPartitioning a LoadBalance. La funzione di riduzione polinomiale f prende in input un insieme di numeri interi positivi $\langle T \rangle$ e produce come output la tripla $\langle 2, T, k \rangle$ dove k è uguale alla metà della somma dei valori in T:

$$k = \frac{1}{2} \sum_{1 \leq i \leq n} T[i]$$

Dimostriamo che la riduzione polinomiale è corretta:

- Se $\langle T \rangle \in$ SetPartitioning, allora esiste un modo per suddividere T in due sottoinsiemi T_1 e T_2 in modo tale che la somma dei valori contenuti in T_1 è uguale alla somma dei valori contenuti in T_2. Nota che questa somma deve essere uguale alla metà della somma dei valori in T, cioè uguale a k. Quindi assegnando i lavori contenuti in T_1 alla prima linea di produzione e quelli contenuti in T_2 alla seconda linea di produzione otteniamo una soluzione per LoadBalance con makespan uguale a k, come richiesto dal problema.
- Se $\langle 2, T, k \rangle \in$ LoadBalance, allora esiste un assegnamento dei lavori alle 2 linee di produzione con makespan minore o uguale a k. Siccome ci sono solo 2 linee, il makespan di questa soluzione non può essere minore della metà della somma dei valori in T, cioè di k. Quindi l'assegnamento ha makespan esattamente uguale a k, ed entrambe le linee di produzione hanno tempo di occupazione uguale a k. Quindi, inserendo i lavori assegnati alla prima linea in T_1 e quelli assegnati alla seconda linea in T_2 otteniamo una soluzione per SetPartitioning.

La funzione di riduzione deve sommare i valori in T e dividere per due, operazioni che si possono fare in tempo polinomiale.

Automi e Linguaggi Formali – 9/6/2022
Seconda prova intermedia – Primo Turno

1. (12 punti) Una *Macchina di Turing con inserimento* è una macchina di Turing deterministica a nastro singolo che può inserire nuove celle nel nastro. Formalmente la funzione di transizione è definita come

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R, I\}$$

dove L, R indicano gli spostamenti a sinistra e a destra della testina, e I indica l'inserimento di una nuova cella nella posizione corrente della testina. Dopo una operazione di inserimento, la cella inserita contiene il simbolo blank, mentre la cella che si trovava sotto la testina si trova immediatamente a destra della nuova cella.

Dimostra che qualsiasi macchina di Turing con inserimento può essere simulata da una macchina di Turing deterministica a nastro singolo.

Soluzione. Mostriamo come convertire una macchina di Turing con Inserimento M in una TM deterministica a nastro singolo S equivalente.

$S =$ "Su input w:

1. Inizialmente S mette il suo nastro in un formato che gli consente di implementare l'operazione di inserimento di una cella, segnando con il simbolo speciale # la fine della porzione di nastro usata dalla macchina. Se w è l'input della TM, la configurazione iniziale del nastro è $w\#$.
2. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a sinistra.
3. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a destra. Se lo spostamento a destra porta la testina sopra il # che marca la fine del nastro, S scrive un blank al posto del #, e scrive un # nella cella immediatamente più a destra. La simulazione continua con la testina in corrispondenza del blank.
4. Per simulare una mossa del tipo $\delta(q, a) = (r, b, I)$ la TM S scrive un blank marcato nella cella corrente e sposta il contenuto del nastro, dalla cella corrente fino al # di fine nastro, di una cella più a destra. Quindi riporta la testina in corrispondenza del blank marcato, toglie la marcatura e scrive b nella cella immediatamente più a destra. La simulazione continua con la testina in corrispondenza della cella inserita.
5. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

2. (12 punti) Data una Turing Machine M, definiamo

$$\text{HALTS}(M) = \{w \mid M \text{ termina la computazione su } w\}.$$

Considera il linguaggio

$$F = \{\langle M \rangle \mid \text{HALTS}(M) \text{ è un insieme finito}\}.$$

Dimostra che F è indecidibile.

Soluzione. La seguente macchina G calcola una riduzione $\overline{A_{TM}} \leq_m F$:

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

1. Costruisci la seguente macchina M':

   $M' =$ "Su input x:
   1. Esegue M su input w.
   2. Se M accetta, *accetta*.
   3. Se M rifiuta, va in loop."

2. Ritorna $\langle M' \rangle$."

Mostriamo che G calcola una funzione di riduzione g da $\overline{A_{TM}}$ a F, cioè una funzione tale che

$$\langle M, w \rangle \in \overline{A_{TM}} \text{ se e solo se } M' \in F.$$

- Se $\langle M, w\rangle \in \overline{A_{TM}}$ allora la macchina M su input w rifiuta oppure va in loop. In entrambi i casi la macchina M' va in loop su tutte le stringhe, quindi $\text{HALTS}(M) = \emptyset$ che è un insieme finito. Di conseguenza $M' \in F$.
- Viceversa, se $\langle M, w\rangle \notin \overline{A_{TM}}$ allora la macchina M accetta w. In questo caso la macchina M' accetta tutte le parole, quindi $\text{HALTS}(M) = \Sigma^*$ che è un insieme infinito. Di conseguenza $M' \notin F$.

3. (12 punti) Una 3-colorazione di un grafo non orientato G è una funzione che assegna a ciascun vertice di G un "colore" preso dall'insieme $\{1, 2, 3\}$, in modo tale che per qualsiasi arco $\{u, v\}$ i colori associati ai vertici u e v sono diversi. Una 3-colorazione è *bilanciata* se ogni colore è associato ad esattamente 1/3 dei vertici del grafo.

Balanced-3-Color è il problema di trovare una 3-colorazione bilanciata:

$$\text{Balanced-3-Color} = \{\langle G\rangle \mid G \text{ è un grafo che ammette una 3-colorazione bilanciata}\}$$

(a) Dimostra che Balanced-3-Color è un problema NP
(b) Dimostra che Balanced-3-Color è NP-hard, usando 3-Color come problema NP-hard di riferimento.

Soluzione.

(a) Il certificato è un vettore c dove ogni elemento $c[i]$ è il colore assegnato al vertice i. Il seguente algoritmo è un verificatore V per Balanced-3-Color:

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

1. Controlla se c è un vettore di n elementi, dove n è il numero di vertici di G.
2. Controlla se $c[i] \in \{1, 2, 3\}$ per ogni i.
3. Controlla se per ogni arco (i, j) di G, $c[i] \neq c[j]$.
4. Controlla se ogni colore compare esettamente in 1/3 degli elementi di c.
5. Se tutte le condizioni sono vere, *accetta*, altrimenti *rifiuta*."

Ognuno dei passi dell'algoritmo si può eseguire in tempo polinomiale.

(b) Dimostriamo che Balanced-3-Color è NP-Hard per riduzione polinomiale da 3-Color a Balanced-3-Color. La funzione di riduzione polinomiale f prende in input un grafo $\langle G\rangle$ e produce come output un grafo $\langle G'\rangle$. Se n è il numero di vertici di G, G' è costruito aggiungendo $2n$ vertici isolati a G. Un vertice è isolato se non ci sono archi che lo collegano ad altri vertici.

Dimostriamo che la riduzione è corretta:

- Se $\langle G\rangle \in$ 3-Color, allora esiste un modo per colorare G con tre colori $1, 2, 3$. Sia n il numero di vertici di G, e assumiamo che questa colorazione associ k_1 vertici al colore 1, k_2 vertici al colore 2 e k_3 vertici al colore 3. Posso costruire una colorazione bilanciata per il grafo G' come segue:
  - i colori dei vertici di G' che appartengono anche a G sono colorati come in G;
  - $n - k_1$ vertici isolati sono colorati con il colore 1;
  - $n - k_2$ vertici isolati sono colorati con il colore 2;
  - $n - k_3$ vertici isolati sono colorati con il colore 3.

  In questo modo ognuno dei tre colori compare in esattamente n vertici e la colorazione di G' è bilanciata.
- Se $\langle G'\rangle \in$ Balanced-3-Color, allora esiste una colorazione bilanciata di G'. Se elimino da G' i vertici isolati aggiunti dalla riduzione ottengo una 3-colorazione di G.

La funzione di riduzione deve contare i vertici del grafo G e aggiungere $2n$ vertici al grafo, operazioni che si possono fare in tempo polinomiale.

Automi e Linguaggi Formali – 9/6/2022
Seconda prova intermedia – Secondo Turno

1. (12 punti) Una *Macchina di Turing con eliminazione* è una macchina di Turing deterministica a nastro singolo che può eliminare celle dal nastro. Formalmente la funzione di transizione è definita come

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R, D\}$$

dove L, R indicano i normali spostamenti a sinistra e a destra della testina, e D indica l'eliminazione della cella sotto la posizione corrente della testina. Dopo una operazione di eliminazione, la testina si muove nella cella che si trovava immediatamente a destra della cella eliminata.

Dimostra che qualsiasi macchina di Turing con eliminazione può essere simulata da una macchina di Turing deterministica a nastro singolo.

Soluzione. Mostriamo come convertire una macchina di Turing con Inserimento M in una TM deterministica a nastro singolo S equivalente. La simulazione usa il simbolo speciale # per segnare le celle che vengono eliminate.

$S =$ "Su input w:

1. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a sinistra. Se lo spostamento a sinistra porta la testina sopra un #, allora continua a spostare la testina a sinistra finché non si arriva in una cella che contiene un simbolo diverso dal #.
2. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a destra. Se lo spostamento a destra porta la testina sopra un #, allora continua a spostare la testina a destra finché non si arriva in una cella che contiene un simbolo diverso dal #.
3. Per simulare una mossa del tipo $\delta(q, a) = (r, b, D)$ la TM S scrive # nella cella corrente e e muove la testina di una cella a destra. Se lo spostamento a destra porta la testina sopra un #, allora continua a spostare la testina a destra finché non si arriva in una cella che contiene un simbolo diverso dal #.
4. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

2. (12 punti) Data una Turing Machine M, definiamo

$$\text{HALTS}(M) = \{w \mid M \text{ termina la computazione su } w\}.$$

Considera il linguaggio

$$I = \{\langle M \rangle \mid \text{HALTS}(M) \text{ è un insieme infinito}\}.$$

Dimostra che I è indecidibile.

Soluzione. La seguente macchina F calcola una riduzione mediante funzione $A_{TM} \leq_m I$:

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

1. Costruisci la seguente macchina M':

   $M' =$ "Su input x:
   1. Esegue M su input w.
   2. Se M accetta, *accetta*.
   3. Se M rifiuta, va in loop."

2. Ritorna $\langle M' \rangle$."

Mostriamo che F calcola una funzione di riduzione f da A_{TM} a I, cioè una funzione tale che

$$\langle M, w \rangle \in A_{TM} \text{ se e solo se } M' \in I.$$

- Se $\langle M, w\rangle \in A_{TM}$ allora la macchina M accetta w. In questo caso la macchina M' accetta tutte le parole, quindi $\text{HALTS}(M) = \Sigma^*$ che è un insieme infinito. Di conseguenza $M' \in I$.
- Viceversa, se $\langle M, w\rangle \notin A_{TM}$, allora la macchina M su input w rifiuta oppure va in loop. In entrambi i casi la macchina M' va in loop su tutte le stringhe, quindi $\text{HALTS}(M) = \emptyset$ che è un insieme finito. Di conseguenza $M' \notin I$.

3. (12 punti) Una 3-colorazione di un grafo non orientato G è una funzione che assegna a ciascun vertice di G un "colore" preso dall'insieme $\{1, 2, 3\}$, in modo tale che per qualsiasi arco $\{u, v\}$ i colori associati ai vertici u e v sono diversi. Una 3-colorazione è *sbilanciata* se esiste un colore che colora più di metà dei vertici del grafo.

UNBALANCED-3-COLOR è il problema di trovare una 3-colorazione sbilanciata:

$$\text{UNBALANCED-3-COLOR} = \{\langle G\rangle \mid G \text{ è un grafo che ammette una 3-colorazione sbilanciata}\}$$

(a) Dimostra che UNBALANCED-3-COLOR è un problema NP
(b) Dimostra che UNBALANCED-3-COLOR è NP-hard, usando 3-COLOR come problema NP-hard di riferimento.

Soluzione.

(a) Il certificato è un vettore c dove ogni elemento $c[i]$ è il colore assegnato al vertice i. Il seguente algoritmo è un verificatore V per UNBALANCED-3-COLOR:
V = "Su input $\langle\langle G\rangle, c\rangle$:
1. Controlla se c è un vettore di n elementi, dove n è il numero di vertici di G.
2. Controlla se $c[i] \in \{1, 2, 3\}$ per ogni i.
3. Controlla se per ogni arco (i, j) di G, $c[i] \neq c[j]$.
4. Controlla se esiste un colore che compare in più di metà degli elementi di c.
5. Se tutte le condizioni sono vere, *accetta*, altrimenti *rifiuta*."

Ognuno dei passi dell'algoritmo si può eseguire in tempo polinomiale.

(b) Dimostriamo che UNBALANCED-3-COLOR è NP-Hard per riduzione polinomiale da 3-COLOR a UNBALANCED-3-COLOR. La funzione di riduzione polinomiale f prende in input un grafo $\langle G\rangle$ e produce come output un grafo $\langle G'\rangle$. Se n è il numero di vertici di G, G' è costruito aggiungendo $n + 1$ vertici isolati a G. Un vertice è isolato se non ci sono archi che lo collegano ad altri vertici.

Dimostriamo che la riduzione è corretta:

- Se $\langle G\rangle \in$ 3-COLOR, allora esiste un modo per colorare G con tre colori $1, 2, 3$. Sia n il numero di vertici di G. Posso costruire una colorazione sbilanciata per il grafo G' come segue:
  - i colori dei vertici di G' che appartengono anche a G sono colorati come in G;
  - i vertici isolati sono colorati tutti con il colore 1.

  In questo modo il colore 1 è assegnato ad almeno metà dei vertici di G' e la colorazione è sbilanciata.
- Se $\langle G'\rangle \in$ UNBALANCED-3-COLOR, allora esiste una colorazione sbilanciata di G'. Se elimino da G' i vertici isolati aggiunti dalla riduzione ottengo una 3-colorazione di G.

La funzione di riduzione deve contare i vertici del grafo G e aggiungere $n + 1$ vertici al grafo, operazioni che si possono fare in tempo polinomiale.

Automi e Linguaggi Formali – 15/6/2023
Seconda prova intermedia – Primo Turno

1. (12 punti) Una macchina di Turing "ecologica" (ETM) è uguale a una normale macchina di Turing deterministica a singolo nastro, ma può leggere e scrivere su entrambi i lati di ogni cella del nastro: fronte e retro. La testina di una TM ecologica può spostarsi a sinistra (L), a destra (R) o passare all'altro lato del nastro (F).

(a) Dai una definizione formale della funzione di transizione di una TM ecologica.

(b) Dimostra che le TM ecologiche riconoscono la classe dei linguaggi Turing-riconoscibili. Usa una descrizione a livello implementativo per definire le macchine di Turing.

Soluzione.

(a) $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R, F\}$

(b) Per dimostrare che TM ecologiche riconoscono la classe dei linguaggi Turing-riconoscibili dobbiamo dimostrare due cose: che ogni linguaggio Turing-riconoscibile è riconosciuto da una ETM, e che ogni linguaggio riconosciuto da una ETM è Turing-riconoscibile.

La prima dimostrazione è banale: le TM deterministiche a singolo nastro sono un caso particolare di ETM che usano solamente il lato di fronte del nastro e non effettuano mai la mossa F per passare dall'altro lato del nastro. Di conseguenza, ogni linguaggio Turing-riconoscibile è riconosciuto da una ETM.

Per dimostrare che ogni linguaggio riconosciuto da una ETM è Turing-riconoscibile, mostriamo come convertire una macchina di Turing ecologica M in una TM deterministica a due nastri S equivalente. Il primo nastro di S rappresenta il lato di fronte del nastro di M, il secondo nastro rappresenta il lato di retro.

S = "Su input w:

1. S usa lo stato per memorizzare lo stato di M ed il lato dove si trova la testina di M. All'inizio il lato corrente è "fronte" e lo stato di M è quello iniziale.
2. Se il lato corrente è "fronte": leggi il simbolo sotto la testina del primo nastro per stabilire la mossa da fare. Se il lato corrente è "retro", leggi il simbolo sotto la testina del secondo nastro per stabilire la mossa da fare.
3. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ scrive b sul primo nastro se il lato corrente è "fronte", e scrive b sul secondo nastro se il nastro corrente è "retro". Poi sposta entrambe le testine di una cella a sinistra.
4. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ scrive b sul primo nastro se il lato corrente è "fronte", e scrive b sul secondo nastro se il nastro corrente è "retro". Poi sposta entrambe le testine di una cella a destra.
5. Per simulare una mossa del tipo $\delta(q, a) = (r, b, F)$ la TM S scrive b sul primo nastro se il lato corrente è "fronte", poi cambia il valore del lato corrente in "retro". Se il lato corrente è "retro", scrive b sul secondo nastro, poi cambia il valore del lato corrente in "fronte". Sposta entrambe le testine di una cella a destra e poi una cella a sinistra, in modo da ritornare nella cella di partenza.
6. r diventa il nuovo stato corrente della simulazione. Se r è lo stato di accettazione di M, allora S termina con accettazione. Se r è lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

Per concludere, siccome sappiamo che le TM multinastro riconoscono i linguaggi Turing-riconoscibili, allora abbiamo dimostrato che ogni linguaggio riconosciuto da una ETM è Turing-riconoscibile.

2. (12 punti) Considera il seguente problema: dato un DFA D e un'espressione regolare R, il linguaggio riconosciuto da D è uguale al linguaggio generato da R?

(a) Formula questo problema come un linguaggio $EQ_{\text{DFA,REX}}$.

(b) Dimostra che $EQ_{\text{DFA,REX}}$ è decidibile.

Soluzione.

(a) $EQ_{\text{DFA,REX}} = \{\langle D, R\rangle \mid D \text{ è un DFA, } R \text{ è una espressione regolare e } L(D) = L(R)\}$

(b) La seguente macchina N usa la Turing machine M che decide EQ_{DFA} per decidere $EQ_{\text{DFA,REX}}$:

N = "su input $\langle D, R\rangle$, dove D è un DFA e R una espressione regolare:
1. Converti R in un DFA equivalente D_R
2. Esegui M su input $\langle D, D_R\rangle$, e ritorna lo stesso risultato di M."

Mostriamo che N è un decisore dimostrando che termina sempre e che ritorna il risultato corretto. Sappiamo che esiste un algoritmo per convertire ogni espressione regolare in un ε-NFA, ed un algoritmo per convertire ogni ε-NFA in un DFA. Il primo step di N si implementa eseguendo i due algoritmi in sequenza, e termina sempre perché entrambi gli algoritmi di conversione terminano. Il secondo step termina sempre perché sappiamo che EQ_{DFA} è un linguaggio decidibile. Quindi N termina sempre la computazione.

Vediamo ora che N dà la risposta corretta:

- Se $\langle D, R\rangle \in EQ_{\text{DFA,REX}}$ allora $L(D) = L(R)$, e di conseguenza $L(D) = L(D_R)$ perché D_R è un DFA equivalente ad R. Quindi $\langle D, D_R\rangle \in EQ_{\text{DFA}}$, e l'esecuzione di M terminerà con accettazione. N ritorna lo stesso risultato di M, quindi accetta.
- Viceversa, se $\langle D, R\rangle \notin EQ_{\text{DFA,REX}}$ allora $L(D) \neq L(R)$, e di conseguenza $L(D) \neq L(D_R)$ perché D_R è un DFA equivalente ad R. Quindi $\langle D, D_R\rangle \notin EQ_{\text{DFA}}$, e l'esecuzione di M terminerà con rifiuto. N ritorna lo stesso risultato di M, quindi rifiuta.

3. (12 point) Una macchina di Turing M accetta una stringa unaria se esiste una stringa $x \in \{1\}^*$ tale che M accetta x. Consideria il problema di determinare se una TM M accetta una stringa unaria.

(a) Formula questo problema come un linguaggio UA.

(b) Dimostra che il linguaggio UA è indecidibile.

Soluzione.

(a) $UA = \{\langle M\rangle \mid \text{esiste } x \in \{1\}^* \text{ tale che } M \text{ accetta } x\}$

(b) La seguente macchina F calcola una riduzione $A_{TM} \leq_m UA$:
F = "su input $\langle M, w\rangle$, dove M è una TM e w una stringa:
1. Costruisci la seguente macchina M':

   M' = "Su input x:
   1. Esegue M su input w.
   2. Se M accetta, *accetta*.
   3. Se M rifiuta, rifiuta."

2. Ritorna $\langle M'\rangle$."

Mostriamo che F calcola una funzione di riduzione da A_{TM} a UA, cioè una funzione tale che

$$\langle M, w\rangle \in A_{TM} \text{ se e solo se } \langle M'\rangle \in UA.$$

- Se $\langle M, w\rangle \in A_{TM}$ allora la macchina M accetta w. In questo caso la macchina M' accetta tutte le parole, quindi esiste almeno una parola unaria accettata da M', e di conseguenza $\langle M'\rangle \in UA$.
- Viceversa, se $\langle M, w\rangle \notin A_{TM}$ allora la macchina M su input w rifiuta o va in loop. Di conseguenza, la macchina M' rifiuta o va in loop su tutte le stringhe, quindi M' ha linguaggio vuoto e non esiste una stringa unaria che sia accettata da M'. Di conseguenza $\langle M'\rangle \notin UA$.

Automi e Linguaggi Formali – 15/6/2023
Seconda prova intermedia – Secondo Turno

1. (12 punti) Una macchina di Turing salva-nastro è simile a una normale macchina di Turing deterministica a nastro singolo semi-infinito, ma può spostare la testina al centro della parte non vuota del nastro. In particolare, se le prime s celle del nastro non sono vuote, allora la testina può spostarsi nella cella numero $\lfloor s/2 \rfloor$. A ogni passo, la testina della TM salva-nastro può spostarsi a sinistra di una cella (L), a destra di una cella (R) o al centro della parte non vuota del nastro (J).

(a) Dai una definizione formale della funzione di transizione di una TM salva-nastro.

(b) Dimostra che le TM salva-nastro riconoscono la classe dei linguaggi Turing-riconoscibili. Usa una descrizione a livello implementativo per definire le macchine di Turing.

Soluzione.

(a) $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R, J\}$

(b) Per dimostrare che TM salva-nastro riconoscono la classe dei linguaggi Turing-riconoscibili dobbiamo dimostrare due cose: che ogni linguaggio Turing-riconoscibile è riconosciuto da una TM salva-nastro, e che ogni linguaggio riconosciuto da una TM salva-nastro è Turing-riconoscibile.

La prima dimostrazione è banale: le TM deterministiche a singolo nastro sono un caso particolare di TM salva-nastro che non effettuano mai la mossa J per saltare al centro della parte non vuota del nastro. Di conseguenza, ogni linguaggio Turing-riconoscibile è riconosciuto da una TM salva-nastro.

Per dimostrare che ogni linguaggio riconosciuto da una TM salva-nastro è Turing-riconoscibile, mostriamo come convertire una macchina di Turing salva-nastro M in una TM deterministica a nastro singolo S equivalente.

S = "Su input w:

1. Inizialmente S mette il suo nastro in un formato che gli consente di implementare l'operazione di salto al centro della parte non vuota del nastro, usando il simbolo speciale # per marcare l'inizio del nastro. Se w è l'input della TM, la configurazione iniziale del nastro è $\#w$.
2. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a sinistra. Se lo spostamento a sinistra porta la testina sopra il # che marca l'inizio del nastro, S si muove immediatamente di una cella a destra, lasciando inalterato il #. La simulazione continua con la testina in corrispondenza del simbolo subito dopo il #.
3. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a destra.
4. Per simulare una mossa del tipo $\delta(q, a) = (r, b, J)$ la TM S scrive b nella cella corrente, e poi sposta la testina a sinistra fino a ritornare in corrispondenza del # che marca l'inizio del nastro. A questo punto si sposta a destra: se il simbolo dopo il # è un blank, la simulazione continua con la testina in corrispondenza del blank. Se il simbolo dopo il # è diverso dal blank, la TM lo marca con un pallino, poi si sposta a destra fino ad arrivare al primo blank. Dopodiché marca l'ultimo cella non vuota prima del blank e procede a zig-zag, marcando via via una cella all'inizio e una alla fine della porzione di nastro non vuota. Quando la prossima cella da marcare è una cella che è già stata marcata, allora la TM si sposta a sinistra, e marca la cella con un simbolo diverso, come una barra. Poi scorre il nastro per togliere tutti i pallini, e riprende la simulazione con la testina in corrispondenza della cella marcata con la barra.
5. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

2. (12 punti) Considera il problema di determinare se i linguaggi di due DFA sono l'uno il complemento dell'altro.

(a) Formula questo problema come un linguaggio $COMPLEMENT_{\text{DFA}}$.

(b) Dimostra che $COMPLEMENT_{\text{DFA}}$ è decidibile.

Soluzione.

(a) $COMPLEMENT_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ e } B \text{ sono DFA e } L(A) = \overline{L(B)}\}$

(b) La seguente macchina N usa la Turing machine M che decide EQ_{DFA} per decidere $COMPLEMENT_{\text{DFA}}$:

$N =$ "su input $\langle A, B\rangle$, dove A e B sono DFA:
1. Costruisci l'automa $\overline{B}$ che riconosce il complementare del linguaggio di B
2. Esegui M su input $\langle A, \overline{B}\rangle$, e ritorna lo stesso risultato di M."

Mostriamo che N è un decisore dimostrando che termina sempre e che ritorna il risultato corretto. Sappiamo che esiste un algoritmo per costruire il complementare di un DFA (basta invertire stati finali e stati non finali nella definizione dell'automa). Di conseguenza, il primo step di N termina sempre. Il secondo step termina sempre perché sappiamo che EQ_{DFA} è un linguaggio decidibile. Quindi N termina sempre la computazione.

Vediamo ora che N dà la risposta corretta:

- Se $\langle A, B\rangle \in COMPLEMENT_{\text{DFA}}$ allora $L(A) = \overline{L(B)}$, e di conseguenza $L(A) = L(\overline{B})$ perché $\overline{B}$ è il complementare di B. Quindi $\langle A, \overline{B}\rangle \in EQ_{\text{DFA}}$, e l'esecuzione di M terminerà con accettazione. N ritorna lo stesso risultato di M, quindi accetta.
- Viceversa, se $\langle A, B\rangle \notin COMPLEMENT_{\text{DFA}}$ allora $L(A) \neq \overline{L(B)}$, e di conseguenza $L(A) \neq L(\overline{B})$ perché $\overline{B}$ è il complementare di B. Quindi $\langle A, \overline{B}\rangle \notin EQ_{\text{DFA}}$, e l'esecuzione di M terminerà con rifiuto. N ritorna lo stesso risultato di M, quindi rifiuta.

3. (12 punti) Considera il seguente problema: data una TM M a nastro semi-infinito, determinare se esiste un input w su cui M sposta la testina alla destra della cella del nastro numero 2023.

(a) Formula questo problema come un linguaggio 2023_{TM}.

(b) Dimostra che il linguaggio 2023_{TM} è indecidibile.

Soluzione.

(a) $2023_{\text{TM}} = \{\langle M\rangle \mid M \text{ è una TM ed esiste input } w \text{ su cui } M \text{ sposta la testina a destra della cella numero } 2023\}$.

(b) **Il linguaggio 2023_{TM} è decidibile**. Questa non è una scelta voluta, ma la conseguenza di un errore nella definizione dell'esercizio. Di conseguenza, per il punto (b) sono stati valutati solo i criteri "sintattici" nella definizione della riduzione e la chiarezza espositiva, stralciando i criteri che valutano la correttezza della riduzione e della dimostrazione. Il punteggio totale dell'esercizio 3 rimane 12, in modo da mantenere 36 punti totali per il compito.

Automi e Linguaggi Formali – 26/6/2023
Primo appello – Seconda Parte: Macchine di Turing, decidibilità ed indecidibilità

1. **(12 punti)** Una macchina di Turing spreca-nastro è simile a una normale macchina di Turing deterministica a nastro singolo semi-infinito, ma può spostare la testina nella parte non ancora utilizzata del nastro. In particolare, se tutte le celle dopo la cella numero s del nastro sono vuote, e la cella s è non vuota, allora la testina può spostarsi nella cella numero $2s$. A ogni passo, la testina della TM spreca-nastro può spostarsi a sinistra di una cella (L), a destra di una cella (R) o dopo la parte non vuota del nastro (J).

   (a) Dai una definizione formale della funzione di transizione di una TM spreca-nastro.

   (b) Dimostra che le TM spreca-nastro riconoscono la classe dei linguaggi Turing-riconoscibili. Usa una descrizione a livello implementativo per definire le macchine di Turing.

Soluzione.

(a) $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R, J\}$

(b) Per dimostrare che TM spreca-nastro riconoscono la classe dei linguaggi Turing-riconoscibili dobbiamo dimostrare due cose: che ogni linguaggio Turing-riconoscibile è riconosciuto da una TM spreca-nastro, e che ogni linguaggio riconosciuto da una TM spreca-nastro è Turing-riconoscibile.

La prima dimostrazione è banale: le TM deterministiche a singolo nastro sono un caso particolare di TM spreca-nastro che non effettuano mai la mossa J per saltare oltre la parte non vuota del nastro. Di conseguenza, ogni linguaggio Turing-riconoscibile è riconosciuto da una TM spreca-nastro.

Per dimostrare che ogni linguaggio riconosciuto da una TM spreca-nastro è Turing-riconoscibile, mostriamo come convertire una macchina di Turing spreca-nastro M in una TM deterministica a nastro singolo S equivalente.

S = "Su input w:

1. Inizialmente S mette il suo nastro in un formato che gli consente di implementare l'operazione di salto oltre la parte non vuota del nastro, usando il simbolo speciale # per marcare l'inizio e la fine della porzione usata del nastro. Se w è l'input della TM, la configurazione iniziale del nastro è $\#w\#$.
2. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a sinistra. Se lo spostamento a sinistra porta la testina sopra il # che marca l'inizio del nastro, S si muove immediatamente di una cella a destra, lasciando inalterato il #. La simulazione continua con la testina in corrispondenza del simbolo subito dopo il #.
3. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a destra. Se lo spostamento a destra porta la testina sopra il # che marca la fine del nastro, S scrive un blank al posto del #, e scrive un # nella cella immediatamente più a destra. La simulazione continua con la testina in corrispondenza del blank.
4. Per simulare una mossa del tipo $\delta(q, a) = (r, b, J)$ la TM S scrive b nella cella corrente, e poi sposta la testina a destra fino ad arrivare in corrispondenza del # che marca la fine del nastro. A questo punto si sposta a sinistra finché non trova un simbolo diverso dal blank. Marca con un pallino il primo simbolo non blank che trova, poi si sposta di una cella a destra e la marca con il pallino. Continua procedendo a zig-zag, marcando via via una cella all'inizio e una alla fine della sequenza di pallini. Quando la prossima cella da marcare è il # all'inizio del nastro, la TM non la marca e inizia a spostarsi a destra, scorrendo il nastro per togliere tutti i pallini, e riprendere la simulazione con la testina in corrispondenza dell'ultima cella marcata. In questa ultima fase, se una delle celle marcate è il # alla fine del nastro, allora la TM lo sostituisce con un blank e scrive un # immediatamente a destra dell'ultima cella marcata prima di continuare con la simulazione.
5. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

2. (12 punti) Una variabile A in una grammatica context-free G è *persistente* se compare in ogni derivazione di ogni stringa w in $L(G)$. Data una grammatica context-free G e una variabile A, considera il problema di verificare se A è persistente.

(a) Formula questo problema come un linguaggio $PERSISTENT_{\text{CFG}}$.

(b) Dimostra che $PERSISTENT_{\text{CFG}}$ è decidibile.

Soluzione.

(a) $PERSISTENT_{\text{CFG}} = \{\langle G, A\rangle \mid G \text{ è una CFG, } A \text{ è una variabile persistente}\}$

(b) La seguente macchina N usa la Turing machine M che decide E_{CFG} per decidere $PERSISTENT_{\text{CFG}}$:

N = "su input $\langle G, A\rangle$, dove G è una CFG e A una variabile:

1. Verifica che A appartenga alle variabili di G. In caso negativo, rifiuta.
2. Costruisci una CFG G' eliminando tutte le regole dove compare A dalla grammatica G.
3. Esegui M su input $\langle G'\rangle$, e ritorna lo stesso risultato di M."

Mostriamo che N è un decisore dimostrando che termina sempre e che ritorna il risultato corretto. Verificare che una variabile appartenga alle variabili di G è una operazione che si può implementare scorrendo la codifica di G per controllare se A compare nella codifica. Il secondo passo si può implementare copiando la codifica di G senza riportare le regole dove compare A. Di conseguenza, il primo ed il secondo step terminano sempre. Anche il terzo step termina sempre perché sappiamo che E_{CFG} è un linguaggio decidibile. Quindi N termina sempre la computazione.
Vediamo ora che N dà la risposta corretta:

- Se $\langle G, A\rangle \in PERSISTENT_{\text{CFG}}$ allora A è una variabile persistente, quindi compare in ogni derivazione di ogni stringa $w \in L(G)$. Se la eliminiamo dalla grammatica, eliminando tutte le regole dove compare A, allora otteniamo una grammatica G' dove non esistono derivazioni che permettano di derivare una stringa di soli simboli terminali, e di conseguenza G' ha linguaggio vuoto. Quindi $\langle G'\rangle \in E_{\text{CFG}}$, e l'esecuzione di M terminerà con accettazione. N ritorna lo stesso risultato di M, quindi accetta.
- Viceversa, se $\langle G, A\rangle \in PERSISTENT_{\text{CFG}}$ allora A non è una variabile persistente, quindi esiste almeno una derivazione di una parola $w \in L(G)$ dove A non compare. Se eliminiamo A dalla grammatica, eliminando tutte le regole dove compare, allora otteniamo una grammatica G' che può derivare w, e di conseguenza G' ha linguaggio vuoto. Quindi $\langle G'\rangle \notin E_{\text{CFG}}$, e l'esecuzione di M terminerà con rifiuto. N ritorna lo stesso risultato di M, quindi rifiuta.

3. (12 punti) Considera le stringhe sull'alfabeto $\Sigma = \{1, 2, \ldots, 9\}$. Una stringa w di lunghezza n su Σ si dice *ordinata* se $w = w_1 w_2 \ldots w_n$ e tutti i caratteri $w_1, w_2, \ldots, w_n \in \Sigma$ sono tali che $w_1 \leq w_2 \leq \cdots \leq w_n$. Ad esempio, la stringa 1112778 è ordinata, ma le stringhe 5531 e 44427 non lo sono (la stringa vuota viene considerata ordinata). Diciamo che una Turing machine è *ossessionata dall'ordinamento* se ogni stringa che accetta è ordinata (ma non è necessario che accetti tutte queste stringhe). Considera il problema di determinare se una TM con alfabeto $\Sigma = \{1, 2, \ldots, 9\}$ è ossessionata dall'ordinamento.

(a) Formula questo problema come un linguaggio SO_{TM}.

(b) Dimostra che il linguaggio SO_{TM} è indecidibile.

Soluzione.

(a) $SO_{\text{TM}} = \{\langle M\rangle \mid M \text{ è una TM con alfabeto } \Sigma = \{1, 2, \ldots, 9\} \text{ che accetta solo parole ordinate}\}$

(b) La seguente macchina F calcola una riduzione $\overline{A_{TM}} \leq_m UA$:
F = "su input $\langle M, w\rangle$, dove M è una TM e w una stringa:

1. Costruisci la seguente macchina M':

   M' = "Su input x:
   1. Se $x = 111$, accetta.
   2. Se $x = 211$, esegue M su input w e ritorna lo stesso risultato di M.
   3. In tutti gli altri casi, rifiuta.

2. Ritorna $\langle M'\rangle$."

Mostriamo che F calcola una funzione di riduzione da $\overline{A_{TM}}$ a SO_{TM}, cioè una funzione tale che

$$\langle M, w\rangle \in \overline{A_{TM}} \text{ se e solo se } \langle M'\rangle \in SO_{\mathrm{TM}}.$$

- Se $\langle M, w\rangle \in \overline{A_{TM}}$ allora la macchina M rifiuta o va in loop su w. In questo caso la macchina M' accetta la parola ordinata 111 e rifiuta tutte le altre, quindi è ossessionata dall'ordinamento e di conseguenza $\langle M'\rangle \in SO_{\mathrm{TM}}$.
- Viceversa, se $\langle M, w\rangle \notin \overline{A_{TM}}$ allora la macchina M accetta w. Di conseguenza, la macchina M' accetta sia la parola ordinata 111 che la parola non ordinata 211, quindi non è ossessionata dall'ordinamento. Di conseguenza $\langle M'\rangle \notin SO_{\mathrm{TM}}$.

Automi e Linguaggi Formali – 10/7/2023
Secondo appello – Seconda Parte: Macchine di Turing, decidibilità ed indecidibilità

1. **(12 punti)** Una *R2-L3 Turing Machine* è una macchina di Turing deterministica a nastro semi-infinito che può effettuare solo due mosse: spostarsi a destra di due celle (R2), oppure spostarsi a sinistra di tre celle (L3). Se in uno spostamento a sinistra la macchina tenta di spostare la testina a sinistra dell'inizio del nastro, allora lo spostamento termina con la testina nella prima cella del nastro.

   (a) Dai una definizione formale della funzione di transizione di una R2-L3 Turing Machine.

   (b) Dimostra che le R2-L3 Turing Machine riconoscono la classe dei linguaggi Turing-riconoscibili. Usa una descrizione a livello implementativo per definire le macchine di Turing.

Soluzione.

(a) $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L3, R2\}$

(b) Per dimostrare che le R2-L3 Turing Machine riconoscono la classe dei linguaggi Turing-riconoscibili dobbiamo dimostrare due cose: che ogni linguaggio Turing-riconoscibile è riconosciuto da una R2-L3 Turing Machine, e che ogni linguaggio riconosciuto da una R2-L3 Turing Machine è Turing-riconoscibile.

Per la prima dimostrazione, mostriamo come convertire una macchina di Turing deterministica a nastro semi-infinito M in una R2-L3 Turing Machine S equivalente.

S = "Su input w:

1. Per simulare una mossa del tipo $\delta(q, a) = (r, b, L)$, S scrive b sul nastro e muove la testina di due celle a destra, e subito dopo di tre celle a sinistra. Se lo spostamento a sinistra porta la testina oltre l'inizio del nastro, allora vuol dire che la simulazione era partita dalla prima cella del nastro. In questo caso la simulazione riprende con la testina nella prima cella del nastro, come per le TM standard. Negli altri casi, la simulazione continua con la testina in corrispondenza della cella immediatamente a sinistra di quella di partenza.
2. Per simulare una mossa del tipo $\delta(q, a) = (r, b, R)$, S scrive b sul nastro e muove la testina di due celle a destra, di nuovo di due celle a destra, e subito dopo di tre celle a sinistra. La simulazione continua con la testina in corrispondenza della cella immediatamente a destra di quella di partenza.
3. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

Per dimostrare che ogni linguaggio riconosciuto da una R2-L3 Turing Machine è Turing-riconoscibile, mostriamo come convertire una R2-L3 Turing Machine S in una TM deterministica a nastro semi-infinito M equivalente.

M = "Su input w:

1. Per simulare una mossa del tipo $\delta(q, a) = (r, b, L3)$, M scrive b sul nastro e muove la testina di tre celle a sinistra. Se lo spostamento a sinistra porta la testina oltre l'inizio del nastro, allora lo sposta meno a sinistra si ferma con la testina nella prima cella del nastro, come per le R2-L3 Turing Machine.
2. Per simulare una mossa del tipo $\delta(q, a) = (r, b, R2)$, M scrive b sul nastro e muove la testina di due celle a destra.
3. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di S, allora M termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di S, allora M termina con rifiuto. Negli altri casi continua la simulazione dal punto 2."

2. (12 punti) Dati due DFA, considera il problema di determinare se esiste una stringa accettata da entrambi.

(a) Formula questo problema come un linguaggio $AGREE_{\text{DFA}}$.

(b) Dimostra che $AGREE_{\text{DFA}}$ è decidibile.

Soluzione.

(a) $AGREE_{\text{DFA}} = \{\langle A, B\rangle \mid A, B \text{ sono DFA, ed esiste una parola } w \text{ tale che } w \in L(A) \text{ e } w \in L(B)\}$

(b) La seguente macchina N usa la Turing machine M che decide E_{DFA} per decidere $AGREE_{\text{DFA}}$:

$N =$ "su input $\langle A, B\rangle$, dove A, B sono DFA:
1. Costruisci il DFA C che accetta l'intersezione dei linguaggi di A e B
2. Esegui M su input $\langle C\rangle$. Se M accetta, rifiuta, se M rifiuta, accetta."

Mostriamo che N è un decisore dimostrando che termina sempre e che ritorna il risultato corretto. Sappiamo che esiste un algoritmo per costruire l'intersezione di due DFA. Il primo step di N si implementa eseguendo questo algoritmo, e termina sempre perché la costruzione dell'intersezione termina. Il secondo step termina sempre perché sappiamo che E_{DFA} è un linguaggio decidibile. Quindi N termina sempre la computazione.
Vediamo ora che N dà la risposta corretta:

- Se $\langle A, B\rangle \in AGREE_{\text{DFA}}$ allora esiste una parola che viene accettata sia da A che da B, e quindi il linguaggio $L(A) \cap L(B)$ non può essere vuoto. Quindi $\langle C\rangle \notin E_{\text{DFA}}$, e l'esecuzione di M terminerà con rifiuto. N ritorna l'opposto di M, quindi accetta.
- Viceversa, se $\langle A, B\rangle \notin AGREE_{\text{DFA}}$ allora non esiste una parola che sia accettata sia da A che da B, e quindi il linguaggio $L(A) \cap L(B)$ è vuoto. Quindi $\langle C\rangle \in E_{\text{DFA}}$, e l'esecuzione di M terminerà con accettazione. N ritorna l'opposto di M, quindi rifiuta.

3. (12 punti) Data una Turing Machine M, considera il problema di determinare se esiste un input tale che M scrive "$xyzzy$" su cinque celle adiacenti del nastro. Puoi assumere che l'alfabeto di input di M non contenga i simboli x, y, z.

(a) Formula questo problema come un linguaggio $MAGIC_{\text{TM}}$.

(b) Dimostra che il linguaggio $MAGIC_{\text{TM}}$ è indecidibile.

Soluzione.

(a) $MAGIC_{\text{TM}} = \{\langle M\rangle \mid M \text{ è una TM che scrive } xyzzy \text{ sul nastro per qualche input } w\}$

(b) La seguente macchina F calcola una riduzione $A_{\text{TM}} \leq_m MAGIC_{\text{TM}}$:

$F =$ "su input $\langle M, w\rangle$, dove M è una TM e w una stringa:
1. Verifica che i simboli x, y, z non compaiano in w, né nell'alfabeto di input o nell'alfabeto del nastro di M. Se vi compaiono, sostituiscili con tre nuovi simboli X, Y, Z nella parola w e nella codifica di M.
2. Costruisci la seguente macchina M':

   $M' =$ "Su input x:
   1. Simula l'esecuzione di M su input w, senza usare i simboli x, y, z.
   2. Se M accetta, scrivi $xyzzy$ sul nastro, altrimenti rifiuta senza modificare il nastro.

3. Ritorna $\langle M'\rangle$."

Mostriamo che F calcola una funzione di riduzione da A_{TM} a $MAGIC_{\text{TM}}$, cioè una funzione tale che

$$\langle M, w\rangle \in A_{\text{TM}} \text{ se e solo se } \langle M'\rangle \in MAGIC_{\text{TM}}.$$

- Se $\langle M, w\rangle \in A_{\text{TM}}$ allora la macchina M accetta w. In questo caso la macchina M' scrive $xyzzy$ sul nastro per tutti gli input. Di conseguenza $\langle M'\rangle \in MAGIC_{\text{TM}}$.
- Viceversa, se $\langle M, w\rangle \notin A_{TM}$ allora la macchina M rifiuta o va in loop su w. Per tutti gli input, la macchina M' simula l'esecuzione di M su w senza usare i simboli x, y, z (perché sono stati tolti dalla definizione di M e di w se vi comparivano), e rifiuta o va in loop senza scrivere mai $xyzzy$ sul nastro. Di conseguenza $\langle M'\rangle \notin MAGIC_{\text{TM}}$.