Automi e Linguaggi Formali – a.a. 2020/2021
Seconda prova intermedia

1. Una *tag-Turing machine* è una macchina di Turing con un singolo nastro e due testine: una testina può solo leggere, l'altra può solo scrivere. All'inizio della computazione la testina di lettura si trova sopra il primo simbolo dell'input e la testina di scrittura si trova sopra la cella vuota posta immediatamente dopo la stringa di input. Ad ogni transizione, la testina di lettura può spostarsi di una cella a destra o rimanere ferma, mentre la testina di scrittura deve scrivere un simbolo nella cella corrente e spostarsi di una cella a destra. Nessuna delle due testine può spostarsi a sinistra.

Dimostra che che le tag-Turing machine riconoscono la classe dei linguaggi Turing-riconoscibili.

2. Considera il linguaggio AlwaysDiverge $= \{\langle M \rangle \mid M$ è una TM tale che per ogni $w \in \Sigma^*$ la computazione di M su input w non termina$\}$.

(a) Dimostra che il AlwaysDiverge è indecidibile.
(b) AlwaysDiverge è un linguaggio Turing-riconoscibile, coTuring-riconoscibile oppure né Turing-riconoscibile né coTuring-riconoscibile? Giustifica la tua risposta.

3. Considera il problema di pianificare la disposizione dei posti a sedere per un matrimonio con n invitati. Per disporre gli invitati hai a disposizione k tavoli, che possono ospitare un numero arbitrario di invitati. Alcuni degli invitati sono amici tra di loro, altri sono rivali ed altri sono indifferenti l'uno all'altro. Il tuo obiettivo è di trovare una disposizione degli n invitati sui k tavoli che rispetti i seguenti requisiti:

- ogni invitato siede ad un solo tavolo;
- due invitati che sono amici devono sedere allo stesso tavolo;
- due invitati che sono rivali devono sedere a tavoli diversi.

Possiamo rappresentare l'input del problema con una tripla $\langle n, k, R \rangle$ dove:

- n è il numero di invitati
- k è il numero di tavoli
- R è una matrice $n \times n$ che descrive le relazioni tra gli invitati:

$$R[i,j] = \begin{cases} 1 & \text{se gli ospiti } i \text{ e } j \text{ sono amici} \\ -1 & \text{se gli ospiti } i \text{ e } j \text{ sono rivali} \\ 0 & \text{se gli ospiti } i \text{ e } j \text{ sono indifferenti} \end{cases}$$

e definire il seguente linguaggio:

$$\text{WSP} = \{\langle n, k, R \rangle \mid \text{esiste una disposizione di } n \text{ ospiti su } k \text{ tavoli che rispetta le relazioni tra invitati } R\}$$

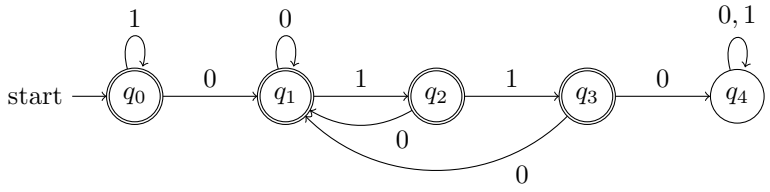
(a) Dimostra che WSP è un problema NP
(b) Dimostra che WSP è NP-hard, usando 3-COLOR come problema NP-hard di riferimento.[1]

[1] Vedi Esercizio 32 degli "Esercizi di preparazione all'esame" per la definizione del problema 3-COLOR.

Automi e Linguaggi Formali – 17/6/2021
Conferma del voto delle prove intermedie – secondo turno – Soluzione

1. *Definire un automa a stati finiti (di qualsiasi tipologia) che riconosca il linguaggio*

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ non contiene la sottostringa } 0110\}$$



2. *Definire una grammatica context-free che generi il linguaggio*

$$L_2 = \{0^n 1^m 2^{m+n} \mid n, m \leq 0\}$$

$$S \to 1S2 \mid T$$
$$T \to 0T2 \mid \varepsilon$$

3. *Fornisci una descrizione a livello implementativo di una TM deterministica a nastro singolo che decide il linguaggio*

$$L_3 = \{u\#w_1\# \ldots \#w_n \mid u, w_i \in \{0,1\}^* \text{ ed esiste } w_j \text{ tale che } u = w_j\}$$

$M =$ "Su input $u\#w_1\# \ldots \#w_n$:

1. Marca il simbolo più a sinistra dell'input. Se il simbolo è #, controlla che a destra ci sia un blank: se c'è *accetta*, altrimenti *rifiuta*.
2. Scorre a destra fino al primo # non marcato e marca il simbolo posto immediatamente a destra. Se non viene trovato nessun # non marcato prima di un blank, allora u è diverso da tutti i w_i, quindi *rifiuta*.
3. Procede a zig-zag confrontando i simboli di u con i simboli della stringa a destra del primo # non marcato. Se le due stringhe sono uguali, *accetta*.
4. Se le due stringhe sono diverse, marca il # e smarca tutti i simboli di u tranne il primo, poi ripete da 2."

4. Una 5-colorazione di un grafo non orientato G è una funzione che assegna a ciascun vertice di G un "colore" preso dall'insieme $\{0,1,2,3,4\}$, in modo tale che per qualsiasi arco $\{u,v\}$ i colori associati ai vertici u e v sono diversi. Una 5-colorazione è *accurata* se i colori assegnati ai vertici adiacenti sono distinti e con differenza maggiore di 1 *modulo 5*.

Fornisci un verificatore polinomiale per il seguente problema:

$$\textsc{Careful5Color} = \{\langle G \rangle \mid G \text{ è un grafo che ammette una 5-colorazione accurata}\}$$

Se i vertici sono numerati da 1 a n, il certificato è un vettore C tale che $C[i]$ è il colore del vertice i.

$V =$ "Su input $\langle G, C \rangle$, dove G è un grafo e C un vettore:

1. Controlla che C sia un vettore di n elementi dove ogni elemento ha valore in $\{0,1,2,3,4\}$.
2. Controlla che per ogni arco (i,j) la differenza tra $C[i]$ e $C[j]$ sia maggiore di 1 modulo 5.
3. Se tutti i test sono superati accetta, altrimenti rifiuta."

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:*

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

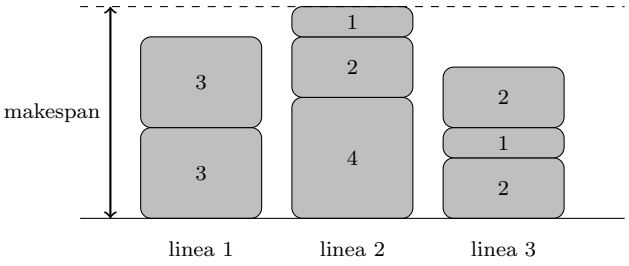


Figura 1: Esempio di assegnamento dei lavori $T = \{1, 1, 2, 2, 2, 3, 3, 4\}$ su 3 linee con makespan 7.

(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.

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.

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.*

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.

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.

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 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.

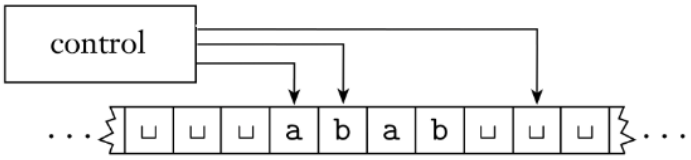
Una macchina di Turing a testine multiple è una macchina di Turing con un solo nastro ma con varie testine. Inizialmente, tutte le testine si trovano sopra alla prima cella dell'input. La funzione di transizione viene modificata per consentire la lettura, la scrittura e lo spostamento delle testine. Formalmente,

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

dove k è il numero delle testine. L'espressione

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli $a_1, \ldots, a_k$ allora la macchina va nello stato q_j, scrive i simboli $b_1, \ldots, b_k$ e muove le testine a destra e a sinistra come specificato.



Un esempio di TM con tre testine.

Mostriamo come simulare una TM a testine multiple B con una TM deterministica a nastro singolo S. Essendo una macchina che riporta molteplici testine su un nastro singolo, ciascuna di queste testine deve essere memorizzata in qualche modo posizionalmente parlando, per esempio assumiamo di separarle per mezzo di un pallino •, capendo dove è stata l'ultima scrittura, si sposta in avanti marcando la scrittura scrivendo l'insieme di simboli di b e prosegue in questo modo.
Ciascuna sequenza di stringhe memorizzata è separata da un # e tramite le marcature siamo in grado di capire dove e quando si sono avute scritture particolari.
La TM S a nastro singolo funziona come segue:

- si parte dalla prima cella e, avendo un certo numero di simboli da leggere, una delle testine si muove e non appena finisce la sua lettura, marca con un simbolo • la posizione
- le successive testine cominciano la lettura nello stesso modo, sapendo che le rispettive porzioni sono separate da # e quindi si ha la scansione di ciascuna porzione. In particolare, ogni testina, comincerà a riscrivere partendo da •
- la scrittura del nastro avviene secondo la funzione di transizione di B:
    - se δ(r,a) = (s,b,L), scrive *b* non marcato sulla cella corrente, dopodiché si sposta immediatamente a sinistra e sposta di una cella a sinistra tutte le marcature con il pallino, fino a che non raggiunge il #, al posto del quale aggiungerà un blank marcato, indicando che ha compiuto la sua scrittura a sinistra.
    - se δ(r,a) = (s,b,R) scrive *b* non marcato sulla cella corrente, dopodiché si sposta immediatamente a destra e sposta di una cella a destra tutte le marcature con il pallino, fino a che non raggiunge il #, al posto del quale aggiungerà un blank marcato, indicando che ha compiuto la sua scrittura a destra.

Se questa simulazione raggiunge lo stato di accettazione di B allora accetta, altrimenti se raggiunge lo stato di rifiuto di B, allora rifiuta. Se non dovesse essere nessuno dei due, si ripeterebbe la simulazione partendo dallo scorrimento del nastro.

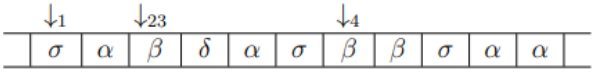
Una soluzione alternativa:

SOLUTION We must prove that the k head TM can be simulated by an ordinary TM. Since we already know that a multiple-tape TM can be simulated with an ordinary one, it suffices to simulate the k-head TM with a multiple-tape TM. We call K to the k-tape TM and M to the multiple-tape TM that simulates the former.

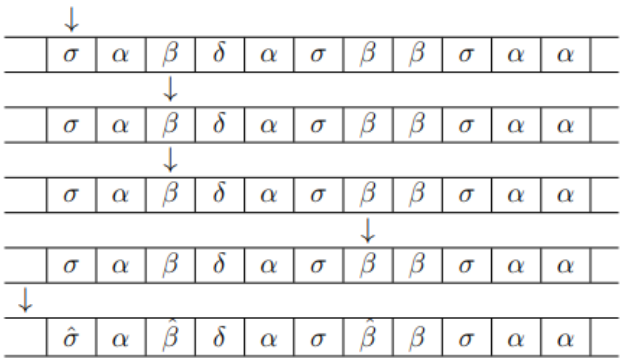
Assume we enumerate each head as h_i with $i = 1, \ldots, k$. We use a $k+1$-tape TM M to do the simulation. The first step is to copy the input tape of K into each tape of M. In the last tape, we call it 'extra', we add a mark to each symbol where there is a head in K (add a mark means that the alphabet of machine M is the union of the alphabet of the machine K plus the set of all the symbols in the alphabet, with a mark).

So, we get the following picture:

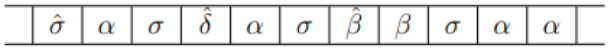
Let the machine K be in the following configuration:



Then the machine M is in the following configuration:



At each move in machine K, we do the same movement in the corresponding tape. Then, we check the last (extra) tape to see what are the changes with respect to each tape. We synchronize those changes one by one in the extra tape. For example, if in the tape 2th the 3th cell has been changed from β to σ and the head moves to the right, then after synchronizing such tape, the extra tape will looks as follows



Then when we have to synchronize the 3th tape we compare all the cells except those already synchronized (all except the 3th cell in our example), so the extra tape remain unchanged.

After having synchronized all the tapes. We replicate the extra tape in all the other tapes, which finish the synchronization process and the movement.

2. Dimostra che il seguente linguaggio è indecidibile:

$$L_2 = \{\langle M, w\rangle \mid M \text{ accetta la stringa } ww^R\}.$$

Partiamo dal costruire una TM M che effettivamente riconosca la stringa ww^R, quindi la stringa che abbiamo e similmente la stringa palindroma. Per fare ciò immaginiamo che:
M = Su input w:

- si ha il primo carattere della prima stringa, lo esaminiamo. Se appartiene alla stringa w, prosegue la computazione e si muove alla fine del nastro; altrimenti, rifiuta
- da questa parte si ha la stringa palindroma, dunque si esamina se tale carattere appartiene al palindromo, se così non fosse rifiuta.
- continuando a muoversi su e giù nella testina, la macchina *accetta* se esaurisce in questo modo tutta la computazione, *rifiuta* altrimenti.

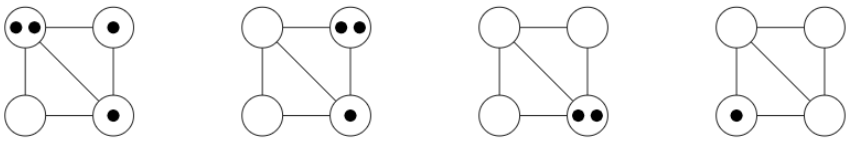
Abbiamo quindi creato un decisore. Ora creiamo una funzione di riduzione f, che prende in input la TM M appena creata ed una stringa *w*, dimostrando quindi che $A_{TM} \leq_m L_2$ e se e solo se f<M, w> = <M> $\in L_2$

La funzione di decisione f

- esegue M sull'input w
- se M accetta, *accetta*
- se M rifiuta, allora *rifiuta*

Se l'input appartiene correttamente al linguaggio, la TM rifiuta. La funzione calcola correttamente tutti gli input e si ferma se avesse una cosa del tipo (01), rispetto ad una stringa accettabile come (0110). Siccome f è computabile in un numero finito di passi, si fermerà su tutti gli input ed f rappresenta una riduzione del linguaggio, rifiutando tutte le stringhe che non centrano.
Tuttavia, sappiamo che L_2 è indecidibile ed è una contraddizione, dunque non può esistere un decisore.

3. Pebbling è un solitario giocato su un grafo non orientato G, in cui ogni vertice ha zero o più ciottoli. Una mossa del gioco consiste nel rimuovere due ciottoli da un vertice v e aggiungere un ciottolo ad un vertice u adiacente a v (il vertice v deve avere almeno due ciottoli all'inizio della mossa). Il problema PEBBLEDESTRUCTION chiede, dato un grafo $G = (V, E)$ ed un numero di ciottoli $p(v)$ per ogni vertice v, di determinare se esiste una sequenza di mosse che rimuove tutti i sassolini tranne uno.



Una soluzione in 3 mosse di PEBBLEDESTRUCTION.

Dimostra che PEBBLEDESTRUCTION è NP-hard usando il problema del Circuito Hamiltoniano come problema NP-hard noto (un circuito Hamiltoniano è un ciclo che attraversa ogni vertice di G esattamente una volta).

Per dimostrare che PEBBLEDESTRUCTION/PB è NP-Hard, dobbiamo dimostrare che è NP e che è riducibile attraverso l'idea del circuito hamiltoniano.
Descriviamo il certificato per questo problema, tale che si abbia SI in tempo polinomiale.
Esso deve verificare le seguenti condizioni:

- l'insieme dei ciottoli p deve essere collegato ad ogni vertice v
- per ogni vertice, verifica se esiste un arco collegato
- esiste almeno un ciottolo che contiene un sassolino, mentre tutti gli altri non ne contengono nessuno

Dovendo esaminare a coppie i vertici del grafo in PB si ha che la verifica viene eseguita in tempo polinomiale. Per dimostrare che PB è NP-Hard si deve descrivere un algoritmo per risolvere PB attraverso l'uso del Circuito Hamiltoniano/HAMPATH/HM e dimostrare che la riduzione è corretta, tale da trasformare istanze buone di HM in istanze buone di PB e dimostrare che l'istanza è corretta.

Partendo dal circuito hamiltoniano H, sappiamo che esso deve avere tutti i vertici collegati in un ciclo esattamente una volta. Supponiamo quindi che per ogni vertice si abbia almeno un sassolino, tranne un vertice che ne presenta 2, cioè p(v) = 2.
Se abbiamo una riduzione, quindi, l'istanza buona S deve verificare che ogni singolo vertice sia connesso agli altri e che presenti almeno un sassolino; se ciò accade siamo all'interno di un circuito hamiltoniano.
Detto ciò, è possibile che, aggiungendo all'ultimo vertice percorso un sassolino, siamo in un'istanza buona di HM.
Se invece vogliamo verificare se l'istanza S' sia buona per PB, allora prendiamo l'ultimo vertice, che presenta un grado di pebbles/sassolini uguale a 2. Allora è possibile, verificare, togliendo dall'ultimo vertice il sassolino in più e verificando che tutti gli altri vertici contengano un solo sassolino; nel qual caso, abbiamo percorso tutto il ciclo e correttamente abbiamo che si ha un circuito hamiltoniano.

Per verificare la correttezza dell'algoritmo rispetto all'idea iniziale, si ripercorre tutto il circuito, lasciando un solo sassolino all'ultimo vertice e togliendo tutti gli altri con la verifica hamiltoniana.
Se abbiamo che il vertice utilizzato era lo stesso di partenza (indichiamo il vertice u che discutevo prima e v quello attuale dopo l'ultimo ciclo), tale che u = v, allora abbiamo un circuito hamiltoniano che arriva nello stesso punto in cui iniziava il pebble.
Quindi, tutto viene eseguito in tempo polinomiale. Se avessimo anche solo un vertice di troppo o rimanesse un sassolino, correttamente non si avrebbe più un circuito hamiltoniano e neanche il pebble sarebbe risolto. Dunque, PB è un problema NP-Completo e può essere correttamente ridotto ad H.

15. Un *automa a coda* è simile ad un automa a pila con la differenza che la pila viene sostituita da una coda. Una *coda* è un nastro che permette di scrivere solo all'estremità sinistra del nastro e di leggere solo all'estremità destra. Ogni operazione di scrittura (*push*) aggiunge un simbolo all'estremità sinistra della coda e ogni operazione di lettura (*pull*) legge e rimuove un simbolo all'estremità destra. Come per un PDA, l'input è posizionato su un nastro a sola lettura separato, e la testina sul nastro di lettura può muoversi solo da sinistra a destra. Il nastro di input contiene una cella con un blank che segue l'input, in modo da poter rilevare la fine dell'input. Un automa a coda accetta l'input entrando in un particolare stato di accettazione in qualsiasi momento. Mostra che un linguaggio può essere riconosciuto da un automa deterministico a coda se e solo se è Turing-riconoscibile.

L'equivalenza tra un automa a coda ed una TM deve essere mostrata; quindi, che il linguaggio può essere riconosciuto dall'automa e dunque dalla TM. Si dimostra creando una simulazione che fa capire che l'automa Q si comporti esattamente come la TM M.

La simulazione di Q come M funziona nel modo seguente:

si consideri l'intero nastro come una coda. Ciascun simbolo viene modificato e il movimento del nastro avviene a destra. Quando più di un numero viene pushato nella coda, si shiftano tutti i valori a destra. Se si raggiunge la fine del nastro, allora il simbolo più a sinistra viene considerato e si capisce di essere arrivati alla fine.

La simulazione di M come Q funziona nel modo seguente:

l'alfabeto della macchina M viene espanso aggiungendo un simbolo extra e un simbolo marcatore a sinistra viene inserito nella coda. I simboli sono pushati a sinistra e letti (pop) a destra.

4. (8 punti) "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. Considera la seguente variante del problema 4-COLOR. Oltre al grafo G, l'input del problema comprende anche un *colore proibito* f_v per ogni vertice v del grafo. Per esempio, il vertice 1 non può essere rosso, il vertice 2 non può essere verde, e così via. Il problema che dobbiamo risolvere è stabilire se possiamo colorare il grafo G con 4 colori in modo che nessun vertice sia colorato con il colore proibito.

$$\text{CONSTRAINED-4-COLOR} = \{\langle G, f_1, \ldots, f_n\rangle \mid \text{esiste una colorazione } c_1, \ldots, c_n \text{ degli } n \text{ vertici tale che } c_v \neq f_v \text{ per ogni vertice } v\}$$

(a) Dimostra che CONSTRAINED-4-COLOR è un problema NP.
(b) Dimostra che CONSTRAINED-4-COLOR è NP-hard, usando k-COLOR come problema NP-hard di riferimento, per un opportuno valore di k.

a) Per dimostrare che CONSTRAINED-4-COLOR/C4C è in NP deve esistere un verificatore in tempo polinomiale. Tale verificatore, prendendo in input il grafo G, l'insieme dei colori proibiti *f* e l'insieme di tutti i colori *c* verifica che:
- ogni vertice contenga un colore tra i 4 *f*
- verifica che ogni vertice sia collegato ad un vertice con un colore che non sia proibito per lui (come si vede, ai 4 colori accettati, corrispondo 4 colori proibiti), quindi controllando non stia in *c*
- controlla che tutti i vertici siano adiacenti ai colori permessi *(f)* e che le coppie di vertici adiacenti non contengano lo stesso colore (dunque a due a due non stiano in *c*); se tutti i test sono superati accetta, altrimenti rifiuta. Essendo una verifica fatta in tempo polinomiale, C4C sta in NP.

b) Per dimostrare che C4C è NP-Hard, dimostriamo che si può usare C4C come istanza per risolvere k-Color/KC.
La riduzione deve prendere l'insieme dei colori buoni *f* che sta in C4C ed usarli per risolvere KC.
Dimostriamo quindi che:

- sia S un'istanza buona di KC. Allora è possibile associare ad ogni vertice un colore compreso in *f* in tempo polinomiale (in quanto i colori *f* fanno parte di *k*). Per fare ciò consideriamo un grafo G che duplica ogni singolo vertice e forziamo tutti i vertici duplicati ad un colore fisso (che sono i 4 colori di *f,* quindi per k=4). Aggiungendo ogni colore tra i 4 di *f* ad un vertice, otteniamo per certo un grafo in KC e con colori che vanno bene a C4C.

- sia S' un'istanza buon di C4C. Tra tutti i colori di k, sappiamo che esiste il sottoinsieme che va bene (C_1, … C_n) e l'insieme che non va bene (f_1, … , f_m). Dobbiamo verificare a coppie che i vertici adiacenti siano diversi dall'insieme *f*. Dato che ogni coppia di vertici ha colore diverso, verifichiamo che ogni coppia abbia un colore *c* diverso dal corrispondente colore proibito in *f*; siccome abbiamo usato 4 colori, alla fine si riduce ad un controllo polinomiale facendo in modo di verificare che ogni singolo vertice contenga esattamente 4 colori; se ciò avviene, abbiamo un'istanza corretta di C4C.

Il se e solo se si ha dato che, usando 4 colori, di sicuro fanno parte dei k e similmente, partendo da k colori, verifico che siano tutti diversi e ciò avviene avendo usato effettivamente 4 colori diversi in partenza.

Pertanto, la riduzione è corretta ed è stata eseguita correttamente in tempo polinomiale.

3. (8 punti) Una *Turing machine con alfabeto ternario* è una macchina di Turing deterministica a singolo nastro dove l'alfabeto di input è $\Sigma = \{0, 1, 2\}$ e l'alfabeto del nastro è $\Gamma = \{0, 1, 2, \sqcup\}$. 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 ogni linguaggio Turing-riconoscibile sull'alfabeto $\{0, 1, 2\}$ può essere riconosciuto da una Turing machine con alfabeto ternario.

Per risolvere l'esercizio dobbiamo dimostrare che
a) ogni linguaggio riconosciuto da una Turing machine con alfabeto ternario è Turing-riconoscibile
b) ogni linguaggio Turing-riconoscibile sull'alfabeto {0, 1, 2} è riconosciuto da una Turing machine con alfabeto binario

Quindi:

a) Questo caso è semplice: una Turing machine con alfabeto ternario 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, 2 e blank. Per esempio, potrebbe contenere simboli marcati o separatori. Per costruire una TM con alfabeto ternario B che simula il comportamento di M dobbiamo come prima cosa stabilire una codifica ternario 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 9 simboli, allora k = 2, perché con 3 bit si rappresentano 9 valori diversi.

La TM con alfabeto ternario B che simula M è definita in questo modo:
B = Su input w:

- Sostituisce $w = w_1 w_2 \ldots w_n$ con la codifica ternaria $C(w_1)C(w_2)\ldots C(w_n)$, e riporta la testina sul primo simbolo di $C(w_1)$.
- 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.
- Aggiorna il nastro in accordo con la funzione di transizione di M:
    a. Se δ(r, a) = (s, b, R), scrive la codifica ternaria di b sul nastro.
    b. Se δ(r, a) = (s, b, L), scrive la codifica ternaria di b sul nastro e sposta la testina a sinistra di 3k celle.
- 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) "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. Considera la seguente variante del problema 4-COLOR. Oltre al grafo G, l'input del problema comprende anche un *colore proibito* f_v per ogni vertice v del grafo. Per esempio, il vertice 1 non può essere rosso, il vertice 2 non può essere verde, e così via. Il problema che dobbiamo risolvere è stabilire se possiamo colorare il grafo G con 4 colori in modo che nessun vertice sia colorato con il colore proibito.

$$\text{CONSTRAINED-4-COLOR} = \{\langle G, f_1, \ldots, f_n\rangle \mid \text{esiste una colorazione } c_1, \ldots, c_n \text{ degli } n \text{ vertici tale che } c_v \neq f_v \text{ per ogni vertice } v\}$$

(a) Dimostra che CONSTRAINED-4-COLOR è un problema NP.
(b) Dimostra che CONSTRAINED-4-COLOR è NP-hard, usando k-COLOR come problema NP-hard di riferimento, per un opportuno valore di k.

a) Per dimostrare che CONSTRAINED-4-COLOR/C4C è in NP deve esistere un verificatore in tempo polinomiale. Tale verificatore, prendendo in input il grafo G, l'insieme dei colori proibiti *f* e l'insieme di tutti i colori *c* verifica che:
- ogni vertice contenga un colore tra i 4 *f*
- verifica che ogni vertice sia collegato ad un vertice con un colore che non sia proibito per lui (come si vede, ai 4 colori accettati, corrispondo 4 colori proibiti), quindi controllando non stia in *c*
- controlla che tutti i vertici siano adiacenti ai colori permessi *(f)* e che le coppie di vertici adiacenti non contengano lo stesso colore (dunque a due a due non stiano in *c*); se tutti i test sono superati accetta, altrimenti rifiuta. Essendo una verifica fatta in tempo polinomiale, C4C sta in NP.

b) Per dimostrare che C4C è NP-Hard, dimostriamo che si può usare C4C come istanza per risolvere k-Color/KC.
La riduzione deve prendere l'insieme dei colori buoni *f* che sta in C4C ed usarli per risolvere KC.
Dimostriamo quindi che:

- sia S un'istanza buona di KC. Allora è possibile associare ad ogni vertice un colore compreso in *f* in tempo polinomiale (in quanto i colori *f* fanno parte di *k*). Per fare ciò consideriamo un grafo G che duplica ogni singolo vertice e forziamo tutti i vertici duplicati ad un colore fisso (che sono i 4 colori di *f,* quindi per k=4). Aggiungendo ogni colore tra i 4 di *f* ad un vertice, otteniamo per certo un grafo in KC e con colori che vanno bene a C4C.
- sia S' un'istanza buon di C4C. Tra tutti i colori di k, sappiamo che esiste il sottoinsieme che va bene (C_1, … C_n) e l'insieme che non va bene (f_1, … , f_m). Dobbiamo verificare a coppie che i vertici adiacenti siano diversi dall'insieme *f*. Dato che ogni coppia di vertici ha colore diverso, verifichiamo che ogni coppia abbia un colore *c* diverso dal corrispondente colore proibito in *f*; siccome abbiamo usato 4 colori, alla fine si riduce ad un controllo polinomiale facendo in modo di verificare che ogni singolo vertice contenga esattamente 4 colori; se ciò avviene, abbiamo un'istanza corretta di C4C.

Il se e solo se si ha dato che, usando 4 colori, di sicuro fanno parte dei k e similmente, partendo da k colori, verifico che siano tutti diversi e ciò avviene avendo usato effettivamente 4 colori diversi in partenza.
Pertanto, la riduzione è corretta ed è stata eseguita correttamente 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 $\langle M_w \rangle$."

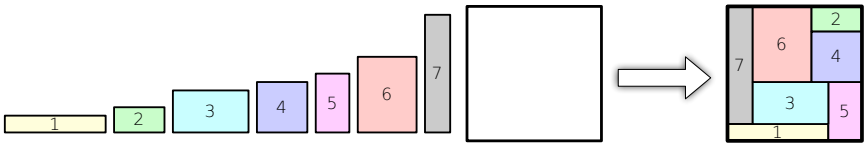
Dimostriamo che f è una funzione di riduzione da A_{TM} ad A_{1010}.

- 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 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
Seconda prova intermedia

1. Una *macchina di Turing con reset a sinistra* è una variante delle comuni macchine di Turing, dove la funzione di transizione ha la forma:

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{R, RESET\}.$$

Se $\delta(q, a) = (r, b, RESET)$, quando la macchina si trova nello stato q e legge a, la testina scrive b sul nastro, salta all'estremità sinistra del nastro ed entra nello stato r. Per sapere su quale cella saltare la macchina usa il simbolo speciale $\triangleright$ per identificare l'estremità di sinistra del nastro. Questo simbolo si può trovare solo in una cella del nastro, e non può essere sovrascritto o cancellato. La computazione di una macchina di Turing con reset a sinistra sulla parola w inizia con $\triangleright w$ sul nastro. Si noti che queste macchine non hanno la solità capacità di muovere la testina di una cella a sinistra.

Mostrare che le macchine di Turing con reset a sinistra riconoscono la classe dei linguaggi Turing-riconoscibili.

2. (a) Mostrare che A è Turing-riconoscibile *se e solo se* $A \leq_m A_{TM}$.
(b) Mostrare che A è decidibile *se e solo se* $A \leq_m 0^*1^*$.

3. Sia G un grafo non orientato, e si consideri il seguente problema[1]:

$$LPATH = \{\langle G, s, t, k\rangle \mid G \text{ contiene un cammino semplice di lunghezza almeno } k \text{ da } s \text{ a } t\}$$

Un cammino semplice è un cammino nel grafo senza ripetizioni di vertici, con la possibile eccezione dei vertici iniziale e finale che possono coincidere. La sua lunghezza è data dal numero di archi che lo compongono.

Mostrare che *LPATH* è NP-completo, usando il problema del circuito Hamiltoniano come problema NP-hard di riferimento.

[1] *LPATH* è la *versione decisionale* del problema del cammino massimo tra i nodi s e t, in cui la lunghezza del cammino k diventa uno degli input del problema. In questo modo si trasforma un problema di ottimizzazione dove l'output è la lunghezza del cammino in un problema di decisione con risposta binaria vero/falso.