1. **(12 punti)** Una *Macchina di Turing con stack di nastri* possiede due azioni aggiuntive che modificano il suo nastro di lavoro, oltre alle normali operazioni di scrittura di singole celle e spostamento a destra o a sinistra della testina: può salvare l'intero nastro inserendolo in uno stack (operazione di Push) e può ripristinare l'intero nastro estraendolo dallo stack (operazione di Pop). Il ripristino del nastro riporta il contenuto di ogni cella al suo contenuto quando il nastro è stato salvato. Il salvataggio e il ripristino del nastro non modificano lo stato della macchina o la posizione della sua testina. Se la macchina tenta di "ripristinare" il nastro quando lo stack è vuoto, la macchina va nello stato di rifiuto. Se ci sono più copie del nastro salvate nello stack, la macchina ripristina l'ultima copia inserita nello stack, che viene quindi rimossa dallo stack.

   Mostra che qualsiasi macchina di Turing con stack di nastri può essere simulata da una macchina di Turing standard. *Suggerimento:* usa una macchina multinastro per la simulazione.

Per la simulazione usiamo una macchina di turing a nastro singolo. In una parte del nastro inseriamo un simbolo marcatore che rappresenta inizio e fine della pila (**##**) e inseriamo un altro simbolo delimitatore (**#**) che separa l'inizio di un altro nastro nella pila.

S = "su input w:

1) δ(q, a) = (r, b, PUSH) allora la TM legge tutti i valori presenti nel nastro fino a quando non trova **##** e, una volta letti scrive traslando a destra l'intero nastro i valori subito dopo **##**. Una volta scritto aggiunge un delimitatore subito dopo **##** ovvero **#** per separare il futuro nastro da quello attuale dentro la pila.
2) δ(q, a) = (r, b, POP) inizialmente inserisce un * all'inizio del nastro. Scorre il nastro finchè non trova **##**. Se sono presenti in tutto **####** allora la pila è vuota e termina l'esecuzione con rifiuto, altrimenti continua a leggere cella per cella i valori subito dopo **##** fino a quando non trova **#** e a partire dal * scrive i valori rimuovendoli da dove gli ha appena letti e sostituendoli appunto da *. Una volta rimossi (ovvero aggiunto dei blank) trasla il nastro a sinistra sostituendo i blank con i valori del prossimo nastro presente nella pila.
3) Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi ricomincia la simulazione.

2. (12 punti) Considera il linguaggio

$$\textsc{Forty-Two} = \{\langle M, w\rangle \mid M \text{ termina la computazione su } w \text{ avendo solo 42 sul nastro}\}.$$

Dimostra che FORTY-TWO è indecidibile.

Per dimostrare che Forty-Two è indecidibile, useremo il noto metodo di riduzione dall'halting problem.

Supponiamo per assurdo che esista un algoritmo A che decide se una coppia <M, w> appartiene al linguaggio Forty-Two. Costruiamo un algoritmo B che risolve l'halting problem, cioè decide se una macchina di Turing M termina su un input w arbitrario. L'algoritmo B lavora come segue:

B(w):

1. Costruisci una nuova macchina di Turing M' che lavora come segue:

   - Se l'input è diverso da w, allora M' entra in un loop infinito.

   - Altrimenti, M' simula la macchina di Turing M su w.

   Se M termina la computazione su w avendo solo 42 sul nastro, allora M' accetta, altrimenti M' entra in un loop infinito.

2. Applica l'algoritmo A alla coppia <M', λ>.

3. Se A accetta, allora B accetta, altrimenti B rifiuta.

L'algoritmo B costruisce una nuova macchina di Turing M' che simula la macchina di Turing M su w e verifica se M termina la computazione su w avendo solo 42 sul nastro. Se sì, allora M' accetta, altrimenti entra in un loop infinito. Quindi, l'algoritmo B applica l'algoritmo A alla coppia <M', λ> per verificare se M' accetta la stringa vuota avendo solo 42 sul nastro.

Supponiamo che l'algoritmo B sia corretto. Allora, se B accetta w, significa che M termina la computazione su w avendo solo 42 sul nastro, cioè <M, w> appartiene al linguaggio Forty-Two. Al contrario, se B rifiuta w, significa che M non termina la computazione su w o termina la computazione su w ma non ha solo 42 sul nastro, quindi <M, w> non appartiene al linguaggio Forty-Two.

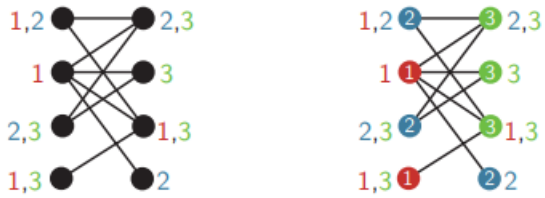
Ma questo implica che possiamo usare l'algoritmo B per risolvere l'halting problem, che è noto essere indecidibile. Infatti, se avessimo un algoritmo per risolvere l'halting problem, potremmo usarlo per costruire l'algoritmo B e quindi risolvere il problema di Forty-Two, che è un assurdo.

Pertanto, abbiamo ottenuto una contraddizione e quindi Forty-Two non è decidibile.

3. (12 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 chiamata LIST-COLORING. Oltre al grafo G, l'input del problema comprende anche una *lista di colori ammissibili* $L(v)$ per ogni vertice v del grafo.

Il problema che dobbiamo risolvere è stabilire se possiamo colorare il grafo G in modo che ogni vertice v sia colorato con un colore preso dalla lista di colori ammissibili $L(v)$.

$$\text{LIST-COLORING} = \{\langle G, L\rangle \mid \text{esiste una colorazione } c_1, \ldots, c_n \text{ degli } n \text{ vertici tale che } c_v \in L(v) \text{ per ogni vertice } v\}$$



Esempio di istanza (a sinistra) e soluzione (a destra) di LIST-COLORING.

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

(a) Per dimostrare che List-Coloring è un problema NP, dobbiamo mostrare che esiste un algoritmo deterministico polinomiale che verifica se una soluzione proposta di List-Coloring è corretta. In altre parole, dobbiamo mostrare che se abbiamo una colorazione proposta del grafo G in cui ogni vertice v è colorato con un colore preso dalla lista di colori ammissibili L(v), possiamo verificare in tempo polinomiale se questa colorazione è corretta.

Per verificare se una colorazione proposta è corretta, dobbiamo controllare che nessuna coppia di vertici adiacenti condivida lo stesso colore e che ogni vertice sia colorato con un colore presente nella sua lista di colori ammissibili. Questo controllo richiede un tempo polinomiale rispetto alla dimensione del grafo e delle liste di colori, quindi List-Coloring è un problema NP.

(b) Dimostriamo che List-Coloring è NP-hard riducendo 3-Color a List-Coloring. Supponiamo di avere un'istanza di 3-Color, ovvero un grafo G. Costruiamo un'istanza di List-Coloring come segue: per ogni vertice v di G, creiamo una lista di colori ammissibili L(v) contenente tutti e tre i colori possibili per 3-Color. In questo modo, List-Coloring richiede di trovare una colorazione del grafo G in cui ogni vertice è colorato con uno dei tre colori ammissibili.

Se esiste una soluzione per l'istanza di 3-Color, allora esiste anche una soluzione per l'istanza di List-Coloring. Infatti, se il grafo G è 3-colorabile, allora possiamo colorare ogni vertice di G con un colore diverso tra i tre colori ammissibili nella sua lista di colori. In questo modo, ogni vertice di G sarà colorato correttamente e List-Coloring avrà una soluzione.

D'altra parte, se esiste una soluzione per l'istanza di List-Coloring, allora esiste anche una soluzione per l'istanza di 3-Color. Infatti, se abbiamo una colorazione del grafo G in cui ogni vertice è colorato con un colore preso dalla sua lista di colori ammissibili, allora ogni vertice di G ha almeno un colore ammissibile in comune con i suoi vertici adiacenti, altrimenti non esisterebbe una colorazione corretta. Pertanto, possiamo selezionare un colore comune a tutti i vertici adiacenti di G per ottenere una 3-colorazione del grafo.

In questo modo, abbiamo dimostrato che List-Coloring è NP-hard riducendo 3-Color a List-Coloring. Poiché 3-Color è un problema NP-hard di riferimento, segue che List-Coloring è NP-hard.