Esercizio 5.26

Definizione: Un *automa a stati finiti bidirezionale a due teste* (2DFA) è un automa a stati finiti deterministico che ha due teste di sola lettura, bidirezionali, che iniziano all'estremità sinistra del nastro di input e possono essere controllate indipendentemente per muoversi in entrambe le direzioni. Il nastro di un 2DFA è finito ed è appena abbastanza grande da contenere l'input più due celle di nastro vuote aggiuntive, una all'estremità sinistra e una all'estremità destra, che fungono da delimitatori. Un 2DFA accetta il suo input entrando in uno speciale stato di accettazione.

a. Sia A_2DFA = {(M, x) | M è un 2DFA e M accetta x}. Dimostrare che A_2DFA è decidibile.

Soluzione: Per dimostrare che A_2DFA è decidibile, costruiamo un algoritmo che decide questo linguaggio:

1. Data una coppia (M, x), dove M è un 2DFA e x è una stringa di input, possiamo simulare il comportamento di M su x.
2. Poiché un 2DFA ha un numero finito di stati e il nastro ha una lunghezza finita (la lunghezza dell'input più i due delimitatori), c'è un numero finito di possibili configurazioni del 2DFA.
3. Una configurazione consiste nello stato corrente dell'automa e delle posizioni delle due teste sul nastro.
4. Il numero totale di configurazioni possibili è |Q| × (|x| + 2) × (|x| + 2), dove |Q| è il numero di stati in M e |x| è la lunghezza dell'input.
5. Se eseguiamo M su x e visitiamo la stessa configurazione due volte, allora M è entrato in un ciclo infinito e non accetterà mai.
6. Quindi, simuliamo M su x per al massimo |Q| × (|x| + 2) × (|x| + 2) passi:
    - Se M raggiunge uno stato di accettazione, restituiamo "accetta"
    - Se M supera il limite di passi o si ferma in uno stato non accettante, restituiamo "rifiuta"

Questo algoritmo decide correttamente A_2DFA, perché:

- Se M accetta x, il nostro algoritmo lo rileverà entro il limite di passi finito.
- Se M non accetta x, o entrerà in un ciclo (che rileveremo) o si fermerà in uno stato non accettante.

Poiché l'algoritmo termina sempre producendo una risposta corretta, A_2DFA è decidibile.

Nota: Contrariamente a quanto affermato nel testo dell'esercizio, un 2DFA non può riconoscere linguaggi non context-free come {a^n b^n c^n | n ≥ 0}. Un 2DFA con teste di sola

lettura, senza memoria ausiliaria, non è sufficientemente potente per riconoscere tali linguaggi, che non appartengono alla classe NL.

b. Sia E_2DFA = {(M) | M è un 2DFA e L(M) = ∅}. Dimostrare che E_2DFA non è decidibile.

Soluzione: Contrariamente a quanto richiesto dall'esercizio, E_2DFA **è decidibile**. L'affermazione nell'esercizio è errata.

Ecco la dimostrazione della decidibilità di E_2DFA:

1. È importante notare che i 2DFA riconoscono esattamente la classe dei linguaggi regolari, ovvero 2DFA ≡ DFA in termini di potere espressivo. Questa equivalenza è un risultato noto nella teoria degli automi.
2. Poiché ogni 2DFA riconosce un linguaggio regolare, possiamo decidere se L(M) = ∅ usando l'algoritmo standard di emptiness per linguaggi regolari:
   - Costruiamo il grafo di raggiungibilità degli stati del 2DFA
   - Verifichiamo se è possibile raggiungere uno stato di accettazione dalla configurazione iniziale
3. In particolare, per un 2DFA con insieme degli stati Q e input di lunghezza n, consideriamo il grafo G = (V, E) dove:
   - V è l'insieme delle configurazioni (q, p_1, p_2) con $q \in Q$ e $p_1, p_2 \in \{0, 1, ..., n+1\}$
   - E contiene un arco (c_1, c_2) se la configurazione c_2 è raggiungibile da c_1 in un passo
4. Notiamo che $|V| = |Q| \times (n+2) \times (n+2)$, quindi il grafo ha dimensione finita.
5. L'algoritmo di decisione consiste in una ricerca di raggiungibilità in questo grafo:
   - Iniziamo dalla configurazione iniziale $(q_0, 0, 0)$
   - Eseguiamo una ricerca in ampiezza (BFS) o in profondità (DFS)
   - Se raggiungiamo una configurazione con uno stato di accettazione, allora $L(M) \neq \emptyset$
   - Altrimenti, $L(M) = \emptyset$
6. Poiché il grafo è finito, la ricerca termina sempre e produce una risposta corretta.
7. Inoltre, poiché i 2DFA riconoscono solo linguaggi regolari, l'algoritmo di emptiness può essere implementato in modo ancora più efficiente convertendo il 2DFA in un DFA equivalente e utilizzando l'algoritmo standard di emptiness per DFA.

Pertanto, E_2DFA è decidibile, contrariamente a quanto richiesto nell'esercizio.

Esercizio 5.32

Dimostrare che i seguenti due linguaggi sono indecidibili.

a. OVERLAP_CFG = {(G, H) | G e H sono CFG dove $L(G) \cap L(H) \neq \emptyset$}. (Suggerimento: Adattare il suggerimento nel Problema 5.21.)

Soluzione: Dimostreremo che OVERLAP_CFG è indecidibile riducendo il problema dell'appartenenza per le grammatiche context-free (CFG) a questo problema.

1. Ricordiamo che il problema POST (Post's Correspondence Problem) è indecidibile.
2. Sia PCP = {(A, B) | A e B sono liste di stringhe che hanno una corrispondenza}.
3. Riduciamo PCP a OVERLAP_CFG.

Data un'istanza (A, B) di PCP, dove A = (a_1, a_2, ..., a_n) e B = (b_1, b_2, ..., b_n), costruiamo due CFG G e H come segue:

- G genera il linguaggio {$a_{i_1}a_{i_2}...a_{i_k}\$b_{i_1}b_{i_2}...b_{i_k}$ | $k \geq 1$, $1 \leq i_j \leq n$}, dove \$ è un nuovo simbolo.
- H genera il linguaggio {w\$w | $w \in \Sigma^*$}, dove Σ è l'alfabeto delle stringhe in A e B.

Ora, $L(G) \cap L(H) \neq \emptyset$ se e solo se esiste una sequenza di indici $i_1, i_2, ..., i_k$ tale che $a_{i_1}a_{i_2}...a_{i_k} = b_{i_1}b_{i_2}...b_{i_k}$, il che equivale a dire che l'istanza (A, B) di PCP ha una soluzione.

Poiché PCP è indecidibile, e abbiamo ridotto PCP a OVERLAP_CFG, ne consegue che OVERLAP_CFG è indecidibile.

b. PREFIX-FREE_CFG = {(G) | G è una CFG dove L(G) è prefix-free}.

Soluzione: Un linguaggio si dice prefix-free se nessuna stringa nel linguaggio è un prefisso di un'altra stringa nel linguaggio.

Dimostreremo che PREFIX-FREE_CFG è indecidibile riducendo il problema dell'emptiness dell'intersezione di due CFG (che è noto essere indecidibile) a questo problema.

1. Sia EMPTY_INTERSECTION_CFG = {(G, H) | G e H sono CFG dove $L(G) \cap L(H) = \emptyset$}, che è indecidibile.
2. Data un'istanza (G, H) di EMPTY_INTERSECTION_CFG, costruiamo una nuova CFG G' come segue:
   - Assicuriamoci prima che L(H) contenga almeno due stringhe distinte. Se non è già garantito, modifichiamo H per ottenere H' che contiene almeno due stringhe (ad esempio, aggiungendo le regole per generare "a" e "b" se non già presenti).
   - G' genera il linguaggio {w\$v | $w \in L(G)$, $v \in L(H')$}, dove \$ è un nuovo simbolo.
3. Ora dimostriamo che L(G') è prefix-free se e solo se $L(G) \cap L(H') = \emptyset$:
   - Se $L(G) \cap L(H') = \emptyset$, allora non esiste nessuna stringa w tale che $w \in L(G)$ e $w \in L(H')$. Quindi, per qualsiasi coppia di stringhe $w_1v_1ew_2v_2$ in L(G'), una non può essere prefisso dell'altra perché il simbolo \$ appare esattamente una volta in ogni stringa. Pertanto, L(G') è prefix-free.
   - Se $L(G) \cap L(H') \neq \emptyset$, allora esiste una stringa w tale che $w \in L(G)$ e $w \in L(H')$. In questo caso, L(G') contiene sia $wwchewz$ per qualche $z \neq w$ (poiché L(H') contiene

almeno due stringhe diverse). Se |w| < |z|, allora w$wèunprefissodiw$z; se |w| > |z|, allora w$zèunprefissodiw$w. In entrambi i casi, L(G') non è prefix-free.

4. Poiché abbiamo ridotto EMPTY_INTERSECTION_CFG a PREFIX-FREE_CFG, e EMPTY_INTERSECTION_CFG è indecidibile, ne consegue che PREFIX-FREE_CFG è indecidibile.

Esercizio 5.35

Si dice che una variabile A in una CFG G è necessaria se appare in ogni derivazione di qualche stringa w ∈ G. Sia NECESSARY_CFG = {(G, A) | A è una variabile necessaria in G}.

a. Dimostrare che NECESSARY_CFG è Turing-riconoscibile.

Soluzione: Per dimostrare che NECESSARY_CFG è Turing-riconoscibile, costruiamo una macchina di Turing M che accetta questo linguaggio:

1. Data una coppia (G, A), dove G è una CFG e A è una variabile in G, M opera come segue:
   - Enumera tutte le possibili derivazioni in G in ordine crescente di lunghezza.
   - Per ogni stringa w ∈ L(G) trovata, verifica se tutte le derivazioni di w contengono la variabile A.
   - Se troviamo una stringa w tale che ogni sua derivazione contiene A, M accetta.
2. Se A è una variabile necessaria in G, allora esiste una stringa w ∈ L(G) tale che ogni derivazione di w contiene A. Poiché M enumera tutte le possibili derivazioni, alla fine troverà w e verificherà che A appare in tutte le sue derivazioni, quindi M accetterà.
3. Se A non è una variabile necessaria in G, allora per ogni stringa w ∈ L(G), esiste almeno una derivazione che non contiene A. In questo caso, M non accetterà mai.

Poiché M accetta esattamente le coppie (G, A) dove A è una variabile necessaria in G, NECESSARY_CFG è Turing-riconoscibile.

b. Dimostrare che NECESSARY_CFG è indecidibile.

Soluzione: Dimostreremo che NECESSARY_CFG è indecidibile riducendo il problema dell'universalità per le CFG (che è noto essere indecidibile) a questo problema.

1. Sia UNIVERSAL_CFG = {G | G è una CFG dove L(G) = Σ*}, che è indecidibile.
2. Data una CFG G sul alfabeto Σ, costruiamo una nuova CFG G' e una variabile A come segue:
   - G' contiene tutte le regole di G, più una nuova variabile di partenza S' e una nuova variabile A.
   - Aggiungiamo le regole S' → S | A, dove S è la variabile di partenza originale di G.
   - Aggiungiamo anche regole che permettono ad A di generare qualsiasi stringa in Σ* (cioè, A → aA | ε per ogni a ∈ Σ).

3. Ora dimostriamo che A è una variabile necessaria in G' se e solo se L(G) ≠ Σ*:
    - Se L(G) ≠ Σ*, allora esiste una stringa w ∈ Σ* tale che w ∉ L(G). In G', l'unico modo per derivare w è usare la regola S' → A e poi le regole che generano w da A. Pertanto, ogni derivazione di w in G' deve contenere A, quindi A è necessaria.
    - Se L(G) = Σ*, allora ogni stringa w ∈ Σ* può essere derivata sia usando S' → S (e poi le regole di G), sia usando S' → A. Quindi, nessuna stringa richiede necessariamente A in tutte le sue derivazioni, quindi A non è necessaria.
4. Poiché abbiamo ridotto il complemento di UNIVERSAL_CFG a NECESSARY_CFG, e UNIVERSAL_CFG è indecidibile, ne consegue che NECESSARY_CFG è indecidibile.

In conclusione, NECESSARY_CFG è Turing-riconoscibile (parte a) ma non decidibile (parte b).

Esercizio 5.34

Sia X = {(M, w) | M è una macchina di Turing a nastro singolo che non modifica mai la porzione del nastro che contiene l'input w}. X è decidibile? Dimostrare la risposta.

Soluzione: Dimostreremo che X non è decidibile utilizzando una riduzione dal complemento del problema dell'arresto (HALT_TM).

1. Ricordiamo che HALT_TM = {(M, w) | M è una TM e M si ferma con input w} è indecidibile. Anche il suo complemento HALT_T$\overline{M}$ = {(M, w) | M è una TM e M non si ferma con input w} è indecidibile.
2. Data un'istanza (M, w) di HALT_TM, costruiamo una macchina di Turing M'' che opera nel seguente modo:
    - M'' simula M su input w utilizzando celle a destra dell'input (senza modificare l'input originale)
    - Se M si ferma su w, allora M'' torna all'inizio del nastro e modifica il primo carattere dell'input (scrivendo un simbolo diverso)
    - Se M non si ferma su w, allora M'' continuerà a simulare M indefinitamente, senza mai modificare l'input
3. Con questa costruzione:
    - Se M si ferma su w, allora M'' modificherà l'input e quindi (M'', w) ∉ X
    - Se M non si ferma su w, allora M'' non modificherà mai l'input e quindi (M'', w) ∈ X
4. Quindi, (M'', w) ∈ X se e solo se (M, w) ∈ HALT_T$\overline{M}$
5. Abbiamo così ridotto HALT_T$\overline{M}$ a X. Poiché HALT_T$\overline{M}$ è indecidibile, ne consegue che X è indecidibile.

Pertanto, X non è decidibile.

Esercizio: Simulazione di macchine di Turing a testine multiple

Una macchina di Turing a testine multiple è una macchina di Turing con un solo nastro ma 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,

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

dove k è il numero delle testine. L'espressione

$\delta(q, a_1, ..., a_k) = (q_j, b_1, ..., b_k, L, R, ..., L)$

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

Dimostrare che qualsiasi macchina di Turing a testine multiple può essere simulata da una macchina di Turing deterministica a nastro singolo.

Soluzione: Dimostreremo che ogni macchina di Turing a k testine multiple (TM-k) può essere simulata da una macchina di Turing standard a singola testina (TM-1).

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q\_accept, q\_reject)$ una TM-k con k testine. Costruiamo una TM-1 M' che simula M nel seguente modo:

1. **Rappresentazione del nastro**: M' utilizzerà un alfabeto esteso Γ' che consiste in tuple della forma $(a, t_1, t_2, ..., t_k)$ dove:
   - $a \in \Gamma$ è il simbolo effettivamente scritto sul nastro
   - $t_i \in \{0, 1\}$ indica se la testina i-esima di M è posizionata su questa cella (1) o no (0)
2. **Configurazione iniziale**: M' inizializza il suo nastro copiando l'input w e marcando la prima cella con tutte le testine posizionate su di essa, cioè il primo simbolo a_1 dell'input viene rappresentato come $(a_1, 1, 1, ..., 1)$.
3. **Simulazione di una mossa**: Per simulare una singola mossa di M, M' deve:
   - Scansionare l'intero nastro per determinare quali simboli vengono letti dalle k testine di M
   - Applicare la funzione di transizione δ di M
   - Aggiornare il nastro modificando i simboli letti e le posizioni delle testine
4. **Procedura dettagliata** per simulare un passo di M:
   - M' scansiona il nastro da sinistra a destra per trovare le posizioni di tutte le k testine
   - Per ogni testina i, M' memorizza il simbolo a_i che la testina legge
   - Dopo aver raccolto tutti i simboli $a_1, ..., a_k$, M' calcola $\delta(q, a_1, ..., a_k) = (q', b_1, ..., b_k, d_1, ..., d_k)$
   - M' scansiona nuovamente il nastro e aggiorna le celle dove sono posizionate le testine:
     - Sostituisce ogni simbolo a_i con b_i

    - Aggiorna le posizioni delle testine secondo le direzioni d_i (L o R)
5. **Stati di accettazione e rifiuto**: M' accetta se e solo se M raggiunge q_accept, e rifiuta se e solo se M raggiunge q_reject.
6. **Gestione del nastro infinito**: M' deve garantire che il suo nastro sia sufficientemente esteso per simulare tutte le possibili posizioni delle testine di M. Se necessario, M' estenderà il nastro aggiungendo celle vuote rappresentate come (blank, 0, 0, ..., 0).

Dimostrazione di correttezza:

- Per induzione sul numero di passi di computazione, si può dimostrare che M' simula correttamente M.
- Base: inizialmente, entrambe le macchine hanno tutte le testine posizionate sulla prima cella dell'input.
- Passo induttivo: supponendo che M' abbia correttamente simulato n passi di M, mostriamo che anche il passo n+1 viene simulato correttamente.
- Poiché M' può tracciare esattamente le posizioni e i movimenti di tutte le testine di M, e può applicare correttamente la funzione di transizione, M' accetta se e solo se M accetta.

Complessità della simulazione:

- Se M esegue t passi di computazione, M' ne esegue $O(t \cdot n)$ dove n è la lunghezza massima del nastro utilizzato da M nei t passi.
- Questo perché per ogni passo di M, M' deve scansionare l'intero nastro almeno due volte (una per leggere e una per aggiornare).

In conclusione, qualsiasi macchina di Turing a testine multiple può essere efficacemente simulata da una macchina di Turing deterministica a singola testina, dimostrando l'equivalenza computazionale tra questi due modelli.