Tutorato di Automi e Linguaggi Formali

Soluzioni Homework 7: Macchine di Turing (TMs), varianti ed esempi

Gabriel Rovesti
Corso di Laurea in Informatica - Università degli Studi di Padova

Tutorato 7 - 05-05-2025

1 Macchine di Turing e Descrizioni Implementative

Esercizio 1. Sia data la seguente definizione formale di una Macchina di Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:

$$Q = \{q_0, q_1, q_2, q_{accept}, q_{reject}\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \{0, 1, \sqcup\}$$
$$\delta = \text{tabella di transizione sottostante}$$

Stato	Simbolo letto	Nuovo stato	Azione
q_0	0	q_1	(1, R)
q_0	1	q_0	(1, R)
q_0	$\sqcup$	q_2	($\sqcup$, L)
q_1	0	q_1	(0, R)
q_1	1	q_0	(1, R)
q_1	$\sqcup$	q_{accept}	($\sqcup$, R)
q_2	0	q_2	(0, L)
q_2	1	q_{reject}	(1, L)
q_2	$\sqcup$	q_{accept}	($\sqcup$, R)

a) Determina l'esito della computazione di M sulle seguenti stringhe: (i) 010, (ii) 1101, (iii) 00

b) Descrivi informalmente quale linguaggio riconosce la macchina M

c) Disegna il diagramma degli stati della macchina M

Soluzione. a) Esito della computazione sulle stringhe date

(i) Stringa 010

Passo	Configurazione	Transizione applicata
0	$q_0\underline{0}10$	$\delta(q_0, 0) = (q_1, 1, R)$
1	$1q_1\underline{1}0$	$\delta(q_1, 1) = (q_0, 1, R)$
2	$11q_0\underline{0}$	$\delta(q_0, 0) = (q_1, 1, R)$
3	$111q_1\underline{\sqcup}$	$\delta(q_1, \sqcup) = (q_{accept}, \sqcup, R)$
4	$111 \sqcup q_{accept}$	Accettazione

La macchina M accetta la stringa 010, trasformandola in 111.

(ii) Stringa 1101

Passo	Configurazione	Transizione applicata
0	$q_0\underline{1}101$	$\delta(q_0, 1) = (q_0, 1, R)$
1	$1q_0\underline{1}01$	$\delta(q_0, 1) = (q_0, 1, R)$
2	$11q_0\underline{0}1$	$\delta(q_0, 0) = (q_1, 1, R)$
3	$111q_1\underline{1}$	$\delta(q_1, 1) = (q_0, 1, R)$
4	$1111q_0\underline{\sqcup}$	$\delta(q_0, \sqcup) = (q_2, \sqcup, L)$
5	$111q_2\underline{1}$	$\delta(q_2, 1) = (q_{reject}, 1, L)$
6	$11q_{reject}\underline{1}1$	Rigetto

La macchina M rigetta la stringa 1101.

(iii) Stringa 00

Passo	Configurazione	Transizione applicata
0	$q_0\underline{0}0$	$\delta(q_0, 0) = (q_1, 1, R)$
1	$1q_1\underline{0}$	$\delta(q_1, 0) = (q_1, 0, R)$
2	$10q_1\underline{\sqcup}$	$\delta(q_1, \sqcup) = (q_{accept}, \sqcup, R)$
3	$10 \sqcup q_{accept}$	Accettazione

La macchina M accetta la stringa 00, trasformandola in 10.

b) Descrizione informale del linguaggio riconosciuto

Analizzando il comportamento della macchina, possiamo osservare che:

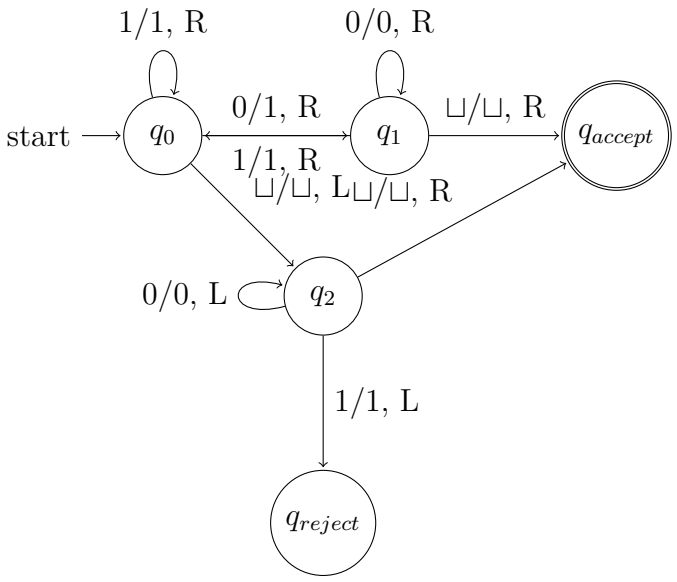
1. La macchina M sostituisce ogni 0 che legge con un 1 2. La macchina verifica che l'ultimo simbolo prima del blank non sia seguito da altri simboli 1 3. Accetta se l'ultimo simbolo prima del blank non è un 1, altrimenti rigetta

Il linguaggio riconosciuto da M può essere descritto formalmente come:

$$L(M) = \{w \in \{0,1\}^* \mid w \text{ non termina con } 1\}$$

Ovvero, M accetta tutte le stringhe che non terminano con il simbolo 1. Equivalentemente, M accetta le stringhe che terminano con 0 o sono vuote.

c) Diagramma degli stati



Esercizio 2. Per ciascuno dei seguenti linguaggi, fornire una descrizione a livello implementativo della TM in grado di accettare il linguaggio proposto:

a) $L_1 = \{a^n b^m c^n \mid n, m \geq 1\}$

b) $L_2 = \{w \mid w \text{ contiene un numero uguale di 0 e di 1}\}$

c) $L_3 = \{w \mid w \text{ contiene un numero di 0 doppio rispetto al numero degli 1}\}$

d) $L_4 = \{w \mid w \text{ non contiene il doppio degli 0 rispetto al numero degli 1}\}$

Nota: Per "descrizione a livello implementativo" si intende specificare in dettaglio il comportamento della macchina, descrivendo come si muove la testina, quali simboli scrive sul nastro e come cambia gli stati per riconoscere il linguaggio.

Soluzione. a) $L_1 = \{a^n b^m c^n \mid n, m \geq 1\}$

Descrizione implementativa della TM per L_1:

La macchina deve verificare che il numero di a all'inizio della stringa sia uguale al numero di c alla fine, e che ci sia almeno una b in mezzo. Adottiamo un approccio di marcatura per contare: cancelleremo progressivamente una a all'inizio e una c alla fine finché non avremo elaborato tutte le a e le c.

1. La macchina inizia nello stato q_0 e legge il primo simbolo:
   - Se il primo simbolo è a, la macchina lo marca con un simbolo speciale, ad esempio X, e si sposta verso destra cercando l'ultimo c.
   - Se il primo simbolo non è a, la macchina rigetta immediatamente.
2. La macchina si muove verso destra, ignorando qualsiasi simbolo, fino a trovare un c o un simbolo blank:
   - Se trova un blank prima di trovare un c, rigetta.
   - Se trova un c, lo marca con un simbolo Y e torna all'inizio della stringa.

3. La macchina si sposta all'inizio del nastro (o al primo simbolo non marcato):

   - Se trova una a non marcata, ripete dal passo 1.
   - Se trova una b o un X, passa allo stato successivo per verificare che tutti i a e c siano stati marcati e che ci sia almeno una b.

4. La macchina verifica la struttura residua:

   - Si assicura che ci siano solo X, b e Y sul nastro (in quest'ordine) con almeno una b.
   - Se trova una a o c non marcata, rigetta.
   - Se non ci sono b tra gli X e gli Y, rigetta.
   - Altrimenti, accetta.

Formalmente, la TM $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \{a, b, c, X, Y, \sqcup\}
\end{aligned}
$$

La funzione di transizione δ implementa la logica descritta sopra. Ad esempio:

$$
\begin{aligned}
&\delta(q_0, a) = (q_1, X, R) \quad \text{(Marca la prima } a \text{ e passa allo stato } q_1) \\
&\delta(q_0, b) = (q_{reject}, b, R) \quad \text{(Rigetta se il primo simbolo è } b) \\
&\delta(q_0, c) = (q_{reject}, c, R) \quad \text{(Rigetta se il primo simbolo è } c)
\end{aligned}
$$

e così via per completare tutte le transizioni necessarie.

b) $L_2 = \{w \mid w$ **contiene un numero uguale di 0 e di 1**$\}$

Descrizione implementativa della TM per L_2:

Per questo linguaggio, dobbiamo contare il numero di 0 e 1 nella stringa e verificare che siano uguali. Utilizziamo un approccio di conteggio con un contatore che incrementiamo per ogni 1 e decrementiamo per ogni 0.

1. La macchina inizia con un contatore (implicito nello stato) impostato a 0.
2. La macchina legge la stringa da sinistra a destra, un simbolo alla volta:

   - Se legge un 1, incrementa il contatore.
   - Se legge un 0, decrementa il contatore.

3. Dopo aver letto l'intera stringa (raggiungendo il simbolo blank), la macchina verifica il valore del contatore:

   - Se il contatore è 0, accetta la stringa.
   - Altrimenti, rigetta la stringa.

Poiché il contatore potrebbe assumere valori arbitrariamente grandi (sia positivi che negativi), implementiamo una soluzione più pratica utilizzando il nastro per tenere traccia del contatore:

1. La macchina inizia nello stato q_0 e si sposta all'estrema sinistra del nastro.
2. La macchina copia la stringa di input su una porzione del nastro, lasciando spazio a sinistra.
3. La macchina torna all'inizio dell'input e inizia l'elaborazione:
   - Per ogni 1 incontrato, la macchina aggiunge un simbolo X nella regione del contatore.
   - Per ogni 0 incontrato, la macchina rimuove un simbolo X dalla regione del contatore (se presente).
4. Alla fine dell'input, la macchina verifica la regione del contatore:
   - Se la regione è vuota (nessun X), accetta.
   - Altrimenti, rigetta.

Formalmente, la TM $M_2 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\} \\
\Sigma &= \{0, 1\} \\
\Gamma &= \{0, 1, X, \sqcup\}
\end{aligned}
$$

c) $L_3 = \{w \mid w$ **contiene un numero di 0 doppio rispetto al numero degli 1**$\}$
Descrizione implementativa della TM per L_3:
Per questo linguaggio, dobbiamo verificare che il numero di 0 sia esattamente il doppio del numero di 1. Utilizziamo un approccio simile al punto precedente, ma con un conteggio diverso.

1. La macchina copia l'input in una regione di lavoro, sostituendo ogni simbolo con un simbolo marcato (per indicare che è già stato elaborato).
2. Per ogni 1 incontrato, la macchina aggiunge due simboli Y in una regione separata del nastro.
3. Per ogni 0 incontrato, la macchina rimuove un simbolo Y dalla regione di conteggio (se presente).
4. Alla fine dell'input, la macchina verifica la regione di conteggio:
   - Se la regione è vuota (tutti gli Y sono stati rimossi), accetta.
   - Altrimenti, rigetta.

Alternativamente, possiamo usare due contatori separati:

1. La macchina scansiona l'input e conta il numero di 0 e il numero di 1 in due regioni separate.
2. Per ogni 1, aggiunge un simbolo X nella prima regione.
3. Per ogni 0, aggiunge un simbolo Y nella seconda regione.
4. Dopo aver scansionato l'intero input, la macchina verifica che per ogni simbolo X ci siano esattamente due simboli Y:
   - La macchina rimuove un X e due Y in sequenza.
   - Se alla fine tutti i simboli sono stati rimossi, accetta.
   - Se rimangono simboli non accoppiati, rigetta.

Formalmente, la TM $M_3 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{accept}, q_{reject}\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \{0, 1, X, Y, \sqcup\}$$

d) $L_4 = \{w \mid w$ **non contiene il doppio degli 0 rispetto al numero degli 1**$\}$
Descrizione implementativa della TM per L_4:
Questo linguaggio è il complemento di L_3. Possiamo quindi utilizzare la stessa TM progettata per L_3, ma invertendo gli stati di accettazione e rigetto.
Alternativamente, possiamo implementare direttamente la logica per L_4:

1. La macchina conta il numero di 0 e 1 nell'input:
   - Per ogni 0, aggiunge un simbolo X in una regione dedicata.
   - Per ogni 1, aggiunge un simbolo Y in un'altra regione.
2. La macchina verifica se il numero di 0 è diverso dal doppio del numero di 1:
   - La macchina confronta le due regioni, rimuovendo un simbolo Y e due simboli X alla volta.
   - Se alla fine ci sono simboli X o Y non accoppiati, accetta (non è vero che $0 = 2 * 1$).
   - Se tutti i simboli sono stati accoppiati correttamente, rigetta (è vero che $0 = 2 * 1$).

Formalmente, la TM $M_4 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{accept}, q_{reject}\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \{0, 1, X, Y, \sqcup\}$$

Esercizio 3. Progettare una macchina di Turing che implementi l'operazione di moltiplicazione tra due numeri naturali codificati in unario. In particolare, la macchina deve accettare input nella forma $a^n \# a^m$ e trasformare il nastro in modo che contenga $a^{n \cdot m}$.

a) Fornire una descrizione dettagliata del funzionamento della macchina

b) Descrivere lo stato del nastro in ogni fase dell'esecuzione per l'input $a^3\#a^2$

c) Discutere le possibili ottimizzazioni per rendere l'algoritmo più efficiente

Soluzione. a) Descrizione dettagliata del funzionamento della macchina

Progetteremo una macchina di Turing che implementa la moltiplicazione in unario, trasformando l'input $a^n\#a^m$ in $a^{n \cdot m}$. L'algoritmo utilizzerà un approccio di addizione ripetuta, dove viene sommato n a se stesso m volte.

Alfabeto e simboli:

- $\Sigma = \{a, \#\}$ (alfabeto di input)
- $\Gamma = \{a, \#, X, Y, Z, \sqcup\}$ (alfabeto del nastro)

Descrizione dell'algoritmo:

1. **Fase di inizializzazione:**
   - Verifica che l'input sia nella forma corretta $a^n\#a^m$.
   - Prepara il nastro per l'elaborazione, creando una copia di lavoro del primo operando a^n.
2. **Fase di moltiplicazione:**
   - Per ogni a presente nel secondo operando (a^m), aggiungi una copia del primo operando (a^n) al risultato.
   - Marco i simboli del secondo operando con Y man mano che vengono processati.
3. **Fase di pulizia:**
   - Rimuovi tutti i simboli ausiliari (X, Y, $\#$) e riporta il nastro nella forma finale $a^{n \cdot m}$.

Descrizione dettagliata dei passi:

1. **Fase di inizializzazione:**
   - La macchina inizia nello stato q_0 e scansiona l'input $a^n\#a^m$.
   - La macchina marca il primo a con un simbolo Z per indicare l'inizio del risultato.
   - Dopo aver marcato il primo a, la macchina si sposta a destra fino al simbolo $\#$.
2. **Fase di moltiplicazione:**
   - La macchina si sposta a destra del $\#$ e verifica se ci sono simboli a non marcati.
   - Se c'è un a non marcato, lo marca con Y e inizia la fase di copia.
   - La fase di copia:

- La macchina torna all'inizio della stringa e conta quanti a ci sono prima del #, marcandoli temporaneamente con X.
- Per ogni a marcato con X, la macchina aggiunge un a alla fine del nastro (dopo tutti gli a già presenti).
- Dopo aver aggiunto tutti gli a, la macchina ripristina i simboli X a simboli a.

- La macchina ripete questi passi per ogni a nel secondo operando.

3. **Fase di pulizia:**

- Dopo aver processato tutti gli a del secondo operando, la macchina rimuove i simboli #, Y e Z.
- La macchina lascia solo i simboli a sul nastro, che rappresentano il risultato $a^{n \cdot m}$.

b) Stato del nastro per l'input $a^3\#a^2$

Vediamo l'evoluzione del nastro durante l'esecuzione dell'algoritmo per l'input $a^3\#a^2$:

Fase	Stato del nastro	Descrizione
Input iniziale	$\underline{a}aa\#aa$	Input iniziale $a^3\#a^2$
Inizio fase 1	$\underline{Z}aa\#aa$	Marca il primo a con Z
Fine fase 1	$Z\underline{a}a\#aa$	Spostamento a destra
Inizio fase 2 (primo ciclo)	$Zaa\#\underline{Y}a$	Marca il primo a del secondo operando con
Durante fase 2 (primo ciclo)	$\underline{Z}Xa\#Ya$	Torna all'inizio e marca il primo a con X
Durante fase 2 (primo ciclo)	$ZX\underline{X}\#Ya$	Marca il secondo a con X
Durante fase 2 (primo ciclo)	$ZXX\#\underline{Y}a$	Marca il terzo a con X
Durante fase 2 (primo ciclo)	$ZXX\#Y\underline{a}$	Continua a destra verso la fine
Durante fase 2 (primo ciclo)	$ZXX\#Ya\underline{\sqcup}$	Raggiunge la fine e inizia ad aggiungere a
Durante fase 2 (primo ciclo)	$ZXX\#Ya\underline{a}$	Aggiunge il primo a (per il primo X)
Durante fase 2 (primo ciclo)	$ZXX\#Ya\underline{a}a$	Aggiunge il secondo a (per il secondo X)
Durante fase 2 (primo ciclo)	$ZXX\#Yaa\underline{a}$	Aggiunge il terzo a (per il terzo X)
Durante fase 2 (primo ciclo)	$Z\underline{a}a\#Yaaa$	Ripristina i simboli X a simboli a
Durante fase 2 (primo ciclo)	$Za\underline{a}\#Yaaa$	Continua a ripristinare i simboli
Fine fase 2 (primo ciclo)	$Zaa\#\underline{Y}aaaa$	Completa il primo ciclo di moltiplicazione
Inizio fase 2 (secondo ciclo)	$Zaa\#Y\underline{Y}aaa$	Marca il secondo a del secondo operando co
Durante fase 2 (secondo ciclo)	$\underline{Z}Xa\#YYaaa$	Torna all'inizio e marca nuovamente
Durante fase 2 (secondo ciclo)	$ZX\underline{X}\#YYaaa$	Continua a marcare
Durante fase 2 (secondo ciclo)	$ZXX\#\underline{Y}Yaaa$	Completa la marcatura
Durante fase 2 (secondo ciclo)	$ZXX\#YY\underline{a}aaa$	Si sposta verso la fine del nastro
Durante fase 2 (secondo ciclo)	$ZXX\#YYaaa\underline{a}$	Aggiunge altri tre a (uno per ogni X)
Durante fase 2 (secondo ciclo)	$ZXX\#YYaaaa\underline{a}a$	Continua ad aggiungere
Durante fase 2 (secondo ciclo)	$Z\underline{a}a\#YYaaaaaa$	Ripristina i simboli X
Fine fase 2 (secondo ciclo)	$Zaa\#\underline{Y}Yaaaaaa$	Completa il secondo ciclo
Inizio fase 3	$\underline{a}aa\#YYaaaaaa$	Inizia la pulizia rimuovendo Z
Durante fase 3	$aaa\underline{\#}YYaaaaaa$	Continua la pulizia
Durante fase 3	$aaa\underline{a}aaaaa$	Rimuove # e Y
Risultato finale	$\underline{a}aaaaaa$	Risultato finale: a^6 (3×2=6)

Alla fine dell'esecuzione, il nastro contiene a^6, che è il prodotto di 3×2.

c) Possibili ottimizzazioni

L'algoritmo presentato, basato sull'addizione ripetuta, è funzionale ma può essere ottimizzato in diversi modi:

1. **Ottimizzazione dello spazio**:
   - Invece di utilizzare una regione separata per accumulare il risultato, possiamo riciclare le celle occupate dal secondo operando, sovrascrivendole con il risultato.
   - Questo riduce lo spazio necessario, soprattutto per moltiplicazioni con operandi grandi.
2. **Ottimizzazione del tempo**:
   - Possiamo ridurre il numero di scansioni complete del nastro, memorizzando temporaneamente il valore di n (lunghezza del primo operando) in una codifica più efficiente.
   - Ad esempio, potremmo usare una rappresentazione binaria o unaria compatta con simboli diversi per ridurre il numero di passaggi.
3. **Utilizzo di più nastri**:
   - Se si utilizza una macchina di Turing multi-nastro, possiamo dedicare nastri separati per gli operandi e per il risultato.
   - Un nastro potrebbe essere utilizzato come contatore, riducendo la necessità di marcare e ripristinare gli elementi ripetutamente.
4. **Algoritmo più sofisticato**:
   - Invece dell'addizione ripetuta (che ha complessità $O(m \cdot n)$), potremmo implementare algoritmi di moltiplicazione più efficienti.
   - Ad esempio, un algoritmo basato su raddoppio successivo potrebbe ridurre la complessità a $O(m \log n)$.
5. **Riduzione del cambio di direzione**:
   - Ogni inversione di direzione della testina aumenta il tempo di esecuzione.
   - Possiamo ottimizzare l'algoritmo per minimizzare i cambi di direzione, ad esempio elaborando gruppi di simboli in un'unica passata.
6. **Parallelizzazione**:
   - In una variante di macchina di Turing con più testine, potremmo parallelizzare alcune operazioni.
   - Ad esempio, una testina potrebbe copiare il primo operando mentre un'altra calcola la posizione dove inserire le nuove copie.

L'algoritmo ottimale dipenderà dalle specifiche limitazioni e dalle varianti della macchina di Turing disponibile. In generale, per macchine di Turing a nastro singolo, la complessità temporale minima per la moltiplicazione unaria sarà $O(n \cdot m)$, poiché è necessario generare $n \cdot m$ simboli e ciascuno richiede almeno un passo di computazione.

2 Varianti delle Macchine di Turing

Esercizio 4. Una macchina di Turing con nastro doppiamente infinito è simile ad una macchina di Turing ordinaria, ma il nastro è infinito sia a sinistra che a destra. Il nastro è inizialmente riempito di blank, eccetto che per la porzione che contiene l'input. La computazione è definita come al solito, eccetto che la testina non incontra mai la fine del nastro mentre si muove a sinistra.

a) Dimostrare che questo tipo di macchina di Turing riconosce la stessa classe di linguaggi delle macchine di Turing standard

b) Descrivere esplicitamente come simulare una macchina di Turing con nastro doppiamente infinito usando una macchina di Turing standard

c) Discutere se la simulazione comporta un aumento di complessità temporale o spaziale

Soluzione. a) Dimostrazione dell'equivalenza

Teorema 5. *Una macchina di Turing con nastro doppiamente infinito (MTDI) riconosce esattamente la stessa classe di linguaggi delle macchine di Turing standard (MTS).*

Proof. La dimostrazione procede in due parti:

Parte 1: Ogni linguaggio riconosciuto da una MTS è riconosciuto da una MTDI.

Questa direzione è ovvia, poiché una MTDI può facilmente simulare una MTS semplicemente non utilizzando mai la porzione di nastro a sinistra della posizione iniziale. Questo equivale a imporre un limite artificiale sul lato sinistro, trasformando di fatto la MTDI in una MTS.

Parte 2: Ogni linguaggio riconosciuto da una MTDI è riconosciuto da una MTS.

Per dimostrare questa direzione, dobbiamo mostrare come una MTS possa simulare una MTDI. Questa simulazione verrà descritta in dettaglio nel punto (b).

L'idea principale è che una MTS può codificare un nastro doppiamente infinito su un nastro infinito solo a destra, utilizzando una codifica che rappresenta entrambe le direzioni. Una volta stabilita questa codifica, la MTS può simulare ogni passo della MTDI, mantenendo la corrispondenza tra le configurazioni.

Poiché ogni passo della MTDI può essere simulato dalla MTS, se la MTDI accetta (o rigetta) un input, anche la MTS corrispondente accetterà (o rigetterà) lo stesso input. Quindi, i linguaggi riconosciuti sono gli stessi. □

b) Simulazione di una MTDI usando una MTS

Per simulare una MTDI usando una MTS, possiamo utilizzare la seguente costruzione:

1. **Codifica del nastro**: Dividiamo il nastro della MTS in due tracce. La traccia superiore rappresenta la parte destra del nastro della MTDI (inclusa la posizione 0), mentre la traccia inferiore rappresenta la parte sinistra del nastro della MTDI in ordine inverso.

2. **Rappresentazione dell'input**: L'input viene inizialmente posizionato sulla traccia superiore, a partire dalla posizione 0.

3. **Rappresentazione della testina**: La posizione della testina della MTDI è rappresentata dalla posizione della testina nella MTS, insieme a un bit (memorizzato nello stato) che indica se la testina è sulla parte superiore o inferiore della traccia.

4. **Simulazione del movimento**:

   - Quando la MTDI sposta la testina a destra dalla posizione i, la MTS sposta la sua testina a destra se è sulla traccia superiore, o a sinistra se è sulla traccia inferiore.
   - Quando la MTDI sposta la testina a sinistra dalla posizione i, la MTS sposta la sua testina a sinistra se è sulla traccia superiore, o a destra se è sulla traccia inferiore.
   - Se la testina della MTS si trova nella posizione 0 della traccia superiore e deve spostarsi a sinistra, passa alla traccia inferiore (ancora in posizione 0).
   - Se la testina della MTS si trova nella posizione 0 della traccia inferiore e deve spostarsi a sinistra, passa alla traccia superiore (ancora in posizione 0).

Formalmente, data una MTDI $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0, q_{accept}, q_{reject})$, costruiamo una MTS $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q'_0, q'_{accept}, q'_{reject})$ come segue:

- $Q_2 = Q_1 \times \{U, L\} \cup \{q'_0, q'_{accept}, q'_{reject}\}$ dove U e L indicano rispettivamente le tracce superiore e inferiore.
- $\Gamma_2 = \Gamma_1 \times \Gamma_1 \cup \{(\sqcup, \sqcup)\}$
- La funzione di transizione δ_2 simula δ_1 come descritto sopra, gestendo i passaggi tra le tracce e il movimento della testina.

c) Analisi della complessità

La simulazione di una MTDI mediante una MTS comporta un certo overhead in termini di complessità:

Complessità temporale:

- Ogni passo della MTDI è simulato da un numero costante di passi (tipicamente 2-3) della MTS.
- Questo implica che la complessità temporale della simulazione è $O(T)$, dove T è il tempo di esecuzione della MTDI originale.
- Il fattore costante di overhead dipende dai dettagli dell'implementazione, ma non modifica la classe di complessità.

Complessità spaziale:

- La MTS utilizza esattamente lo stesso spazio della MTDI, ma codificato diversamente.
- Per ogni cella utilizzata dalla MTDI, la MTS utilizza una posizione nel suo nastro (con due tracce).
- Pertanto, la complessità spaziale rimane $O(S)$, dove S è lo spazio utilizzato dalla MTDI.

Overhead aggiuntivo:

- L'insieme degli stati della MTS è più grande (per un fattore costante) rispetto a quello della MTDI, per tenere traccia della posizione (traccia superiore o inferiore).
- Ciò non influisce sulla complessità asintotica, ma può rendere la macchina più complessa da descrivere.
- La necessità di spostare la testina tra le tracce può introdurre un overhead costante in scenari di "confine" (quando la testina attraversa la posizione 0).

In conclusione, la simulazione di una MTDI mediante una MTS mantiene la stessa classe di complessità sia temporale che spaziale dell'originale, con solo un fattore costante di overhead. Questo conferma ulteriormente l'equivalenza computazionale tra questi due modelli.

Esercizio 6. Una macchina di Turing con nastro di sola lettura è una macchina di Turing che, oltre al nastro di lavoro standard, ha un nastro aggiuntivo di sola lettura che contiene l'input e sul quale la testina può solo leggere, senza scrivere.

a) Dimostrare che questo tipo di macchina di Turing riconosce la stessa classe di linguaggi delle macchine di Turing standard

b) Fornire una descrizione dettagliata di come simulare una macchina di Turing standard usando una macchina di Turing con nastro di sola lettura

c) Implementare una macchina di Turing con nastro di sola lettura che riconosca il linguaggio $L = \{ww^R \mid w \in \{0,1\}^*\}$, dove w^R è il reverse di w

Soluzione. a) Dimostrazione dell'equivalenza

Teorema 7. *Una macchina di Turing con nastro di sola lettura (MTSL) riconosce esattamente la stessa classe di linguaggi delle macchine di Turing standard (MTS).*

Proof. La dimostrazione procede in due parti:

Parte 1: Ogni linguaggio riconosciuto da una MTS è riconosciuto da una MTSL.
Possiamo costruire una MTSL che simula una MTS nel seguente modo:

- La MTSL prima copia l'intero input dal nastro di sola lettura al nastro di lavoro.
- Successivamente, simula la MTS lavorando esclusivamente sul nastro di lavoro, ignorando completamente il nastro di sola lettura.

Poiché la MTSL ha la capacità di copiare l'intero input sul nastro di lavoro e poi operare su di esso come farebbe una MTS, può riconoscere qualsiasi linguaggio riconosciuto da una MTS.

Parte 2: Ogni linguaggio riconosciuto da una MTSL è riconosciuto da una MTS.

Questa direzione è ancora più diretta. Una MTS può simulare una MTSL semplicemente suddividendo il suo nastro in due sezioni:

- La prima sezione contiene una copia dell'input originale e non viene mai modificata durante la computazione (simulando il nastro di sola lettura).
- La seconda sezione viene utilizzata come nastro di lavoro.

La MTS può mantenere traccia della posizione in entrambe le sezioni e simulare il comportamento della MTSL, assicurandosi di non modificare mai la prima sezione. Poiché una MTS può simulare una MTSL con questa costruzione, ogni linguaggio riconosciuto da una MTSL è anche riconosciuto da una MTS.

In conclusione, MTSL e MTS riconoscono esattamente la stessa classe di linguaggi. □

b) Simulazione di una MTS usando una MTSL

La simulazione di una MTS utilizzando una MTSL richiede una strategia più elaborata, poiché la MTSL non può scrivere direttamente sul nastro di input. Ecco una descrizione dettagliata:

Setup iniziale:

1. La MTSL ha due nastri: un nastro di sola lettura contenente l'input e un nastro di lavoro (inizialmente vuoto).
2. La prima operazione è copiare l'intero input dal nastro di sola lettura al nastro di lavoro.

Processo di simulazione:

1. La MTSL mantiene sul nastro di lavoro una rappresentazione dell'intera configurazione della MTS, incluso:
    - Il contenuto del nastro della MTS
    - La posizione corrente della testina (marcata con un simbolo speciale)
    - Lo stato corrente (codificato utilizzando simboli speciali adiacenti alla posizione della testina)
2. Per ogni passo di simulazione:
    - La MTSL esamina il simbolo sotto la testina marcata nel nastro di lavoro e lo stato corrente.
    - Basandosi sulla funzione di transizione della MTS, determina il nuovo simbolo da scrivere, il movimento della testina e il nuovo stato.
    - Aggiorna la configurazione sul nastro di lavoro, modificando il simbolo, spostando il marcatore della testina e aggiornando l'indicatore di stato.

3. La MTSL accetta o rigetta in base allo stato finale della simulazione della MTS.

Gestione del nastro:

- Se la MTS richiede più spazio di quello inizialmente allocato (spostando la testina oltre la fine del nastro attualmente rappresentato), la MTSL estende la rappresentazione aggiungendo simboli blank.
- Per efficienza, la rappresentazione può essere compattata periodicamente, rimuovendo i simboli blank inutili alle estremità del nastro.

Rappresentazione formale:
Data una MTS $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0, q_{accept}, q_{reject})$, costruiamo una MTSL $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q'_0, q'_{accept}, q'_{reject})$ dove:

- $\Gamma_2 = \Gamma_1 \cup \{[q, a] \mid q \in Q_1, a \in \Gamma_1\}$ (dove $[q, a]$ rappresenta la testina in stato q sopra il simbolo a)
- Q_2 contiene stati per gestire la copia iniziale e tutte le fasi della simulazione
- δ_2 implementa la logica di simulazione descritta sopra

c) Implementazione per $L = \{ww^R \mid w \in \{0,1\}^*\}$
La macchina di Turing con nastro di sola lettura che riconosce il linguaggio $L = \{ww^R \mid w \in \{0,1\}^*\}$ può essere implementata in modo più efficiente rispetto a una MTS standard, sfruttando il nastro di sola lettura.
Descrizione dell'algoritmo:

1. La MTSL inizia leggendo il primo simbolo dal nastro di input e lo copia sul nastro di lavoro.
2. La MTSL continua a copiare l'input fino a quando non ha esaminato circa metà dell'input (questo richiede di determinare la lunghezza totale dell'input prima).
3. La MTSL segna la posizione centrale sul nastro di lavoro con un simbolo speciale.
4. Successivamente, la MTSL confronta i simboli rimanenti sull'input con i simboli sul nastro di lavoro in ordine inverso:
   - La testina sul nastro di sola lettura si muove da destra a sinistra (dalla posizione centrale verso la fine).
   - La testina sul nastro di lavoro si muove da sinistra a destra (dall'inizio verso la posizione centrale).
5. Se tutti i simboli corrispondono, la MTSL accetta. Altrimenti, rigetta.

Descrizione dettagliata dell'implementazione:

1. **Fase 1: Determinazione della lunghezza dell'input**
   - La MTSL scansiona l'intero nastro di input da sinistra a destra, contando i simboli.

- Il conteggio viene mantenuto in binario sul nastro di lavoro.
- Una volta raggiunta la fine dell'input, la MTSL divide il conteggio per 2 per trovare la posizione centrale.

2. **Fase 2: Copia della prima metà dell'input**
   - La MTSL ripristina la posizione della testina all'inizio dell'input.
   - Copia i primi $n/2$ simboli (dove n è la lunghezza totale) sul nastro di lavoro.
   - Marca la fine della copia con un simbolo speciale.
3. **Fase 3: Verifica della simmetria**
   - La MTSL posiziona la testina del nastro di input all'inizio della seconda metà.
   - Posiziona la testina del nastro di lavoro alla fine della sequenza copiata.
   - Per ogni simbolo rimanente nell'input, la MTSL confronta:
     - Il simbolo corrente sul nastro di input
     - Il simbolo corrente sul nastro di lavoro
   - La testina del nastro di input si muove verso destra.
   - La testina del nastro di lavoro si muove verso sinistra.
   - Se in qualsiasi momento i simboli non corrispondono, la MTSL rigetta.
   - Se tutti i confronti hanno successo, la MTSL accetta.

Gestione dei casi particolari:

- Per gestire stringhe di lunghezza dispari, la MTSL verifica che il simbolo centrale sia confrontato con se stesso.
- Per la stringa vuota (che è in L poiché $\varepsilon\varepsilon^R = \varepsilon$), la MTSL la accetta direttamente.

Questa implementazione dimostra come una MTSL possa riconoscere il linguaggio $\{ww^R \mid w \in \{0,1\}^*\}$ in modo naturale ed efficiente.

Esercizio 8. Una macchina di Turing a singola scrittura è una TM a nastro singolo che può modificare ogni cella del nastro al più una volta (inclusa la parte di input del nastro).

a) Mostrare che questa variante di macchina di Turing è equivalente alla macchina di Turing standard

b) Fornire una descrizione dettagliata di come simulare una macchina di Turing standard usando una macchina di Turing a singola scrittura

c) Discutere quali limitazioni pratiche questa restrizione impone all'implementazione di algoritmi

Soluzione. a) Dimostrazione dell'equivalenza

Teorema 9. *Una macchina di Turing a singola scrittura (MTSS) riconosce esattamente la stessa classe di linguaggi delle macchine di Turing standard (MTS).*

Proof. La dimostrazione procede in due parti:

Parte 1: Ogni linguaggio riconosciuto da una MTSS è riconosciuto da una MTS.

Questa direzione è ovvia, poiché una MTS può sempre simulare una MTSS semplicemente scegliendo di non riscrivere mai su celle già modificate. La restrizione di singola scrittura è una limitazione che una MTS può auto-imporre senza perdere potere espressivo.

Parte 2: Ogni linguaggio riconosciuto da una MTS è riconosciuto da una MTSS.

Questa direzione richiede di mostrare come una MTSS possa simulare una MTS, nonostante la restrizione di scrittura singola. La chiave è utilizzare una codifica che permetta di registrare la storia completa delle modifiche su ogni cella. Descriveremo questa tecnica in dettaglio nel punto (b).

L'idea generale è che la MTSS utilizza un alfabeto esteso per rappresentare non solo il simbolo corrente di ogni cella nella simulazione della MTS, ma anche la posizione relativa della testina e una "cronologia" di tutti i simboli precedentemente scritti sulla cella. Questo permette alla MTSS di simulare le scritture multiple della MTS pur rispettando il vincolo di scrittura singola.

Dato che ogni passo della MTS può essere simulato dalla MTSS, se la MTS accetta (o rigetta) un input, anche la MTSS corrispondente accetterà (o rigetterà) lo stesso input. Quindi, i linguaggi riconosciuti sono gli stessi. □

b) Simulazione di una MTS usando una MTSS

Per simulare una MTS usando una MTSS, usiamo una tecnica nota come "simulazione cronologica". La chiave è rappresentare non solo il contenuto attuale del nastro, ma anche la sequenza di operazioni eseguite su ciascuna cella.

Idea della simulazione:

1. La MTSS utilizza un nastro suddiviso in celle logiche, dove ogni cella logica è composta da più celle fisiche.
2. Ogni cella logica contiene:
   - Il simbolo corrente della cella corrispondente nella MTS
   - Un indicatore di "testina presente/assente"
   - Lo stato corrente (se la testina è presente)
   - Una lista cronologica dei simboli precedentemente scritti in questa cella
3. La MTSS mantiene una traccia separata che rappresenta l'intera configurazione della MTS in un dato momento.

Descrizione dettagliata dell'implementazione:

1. **Rappresentazione delle celle logiche:** Ogni cella logica nel nastro della MTSS è costituita da una sequenza di celle fisiche che rappresentano:
   - Un simbolo di delimitazione di inizio cella $\langle$
   - Il simbolo corrente della MTS in quella posizione
   - Un indicatore di presenza della testina (0=assente, 1=presente)

- Lo stato corrente della MTS (se la testina è presente)
- Una sequenza di simboli che rappresenta la cronologia delle scritture
- Un simbolo di delimitazione di fine cella $\rangle$

2. **Simulazione dell'inizializzazione:** La MTSS inizializza il suo nastro in modo che ogni cella logica contenga:
   - Il simbolo di input corrispondente (o blank)
   - Un indicatore di testina (1 solo per la posizione iniziale, 0 altrove)
   - Lo stato iniziale q_0 (solo nella posizione iniziale)
   - Una cronologia vuota
3. **Simulazione di un passo della MTS:** Per simulare un passo della MTS $\delta(q, a) = (q', b, D)$ (dove $D \in \{L, R\}$):
   (a) La MTSS cerca la cella logica con indicatore di testina = 1
   (b) Legge lo stato corrente q e il simbolo a
   (c) Crea una nuova cella logica che contiene:
      - Il nuovo simbolo b
      - Indicatore di testina = 0
      - La cronologia aggiornata (aggiungendo a alla cronologia)

   (d) Si sposta nella direzione D (una cella logica a sinistra o a destra)
   (e) Crea una nuova cella logica che contiene:
      - Lo stesso simbolo della cella originale
      - Indicatore di testina = 1
      - Il nuovo stato q'
      - La stessa cronologia della cella originale
4. **Gestione degli stati di accettazione/rigetto:**
   - Se la MTS entra nello stato q_{accept}, la MTSS passa al suo stato di accettazione.
   - Se la MTS entra nello stato q_{reject}, la MTSS passa al suo stato di rigetto.

Rappresentazione formale:
Data una MTS $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0, q_{accept}, q_{reject})$, costruiamo una MTSS $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q'_0, q'_{accept}, q'_{reject})$ dove:

- Γ_2 contiene simboli per rappresentare la struttura delle celle logiche, inclusi i delimitatori, indicatori di testina, stati e la cronologia.
- Q_2 contiene stati per gestire tutte le fasi della simulazione, inclusa l'inizializzazione, la ricerca della posizione della testina, l'aggiornamento delle celle e la gestione degli stati di accettazione/rigetto.

- δ_2 implementa la logica di simulazione descritta sopra, rispettando sempre il vincolo di scrittura singola su ogni cella.

La chiave per rispettare il vincolo di singola scrittura è che, invece di modificare una cella esistente, la MTSS crea sempre una nuova rappresentazione della cella aggiungendo alla cronologia. Questo garantisce che ogni cella fisica venga scritta al massimo una volta durante l'intera computazione.

c) Limitazioni pratiche

La restrizione di singola scrittura impone diverse limitazioni pratiche all'implementazione di algoritmi, tra cui:

1. **Esplosione dello spazio:**
   - Per simulare una MTS che scrive k volte sulla stessa cella, una MTSS deve utilizzare $O(k)$ celle.
   - Per algoritmi che accedono ripetutamente alle stesse posizioni (come contatori), lo spazio utilizzato cresce linearmente con il numero di accessi.
   - Algoritmi che richiedono spazio S in una MTS potrebbero richiedere spazio $O(S \cdot T)$ in una MTSS, dove T è il tempo di esecuzione.
2. **Aumento della complessità temporale:**
   - La rappresentazione espansa delle celle logiche aumenta il tempo necessario per accedere e manipolare le informazioni.
   - Per simulare un singolo passo della MTS, la MTSS potrebbe dover attraversare più celle fisiche.
   - Operazioni come il ripristino di una cella allo stato originale (comune in molti algoritmi) richiedono un approccio completamente diverso.
3. **Difficoltà nell'implementazione di strutture dati comuni:**
   - Strutture di dati come stack, code e contatori, che tipicamente aggiornano lo stesso spazio di memoria, diventano inefficienti.
   - Algoritmi iterativi che modificano ripetutamente gli stessi valori richiedono riscritture significative.
   - Operazioni di ordinamento in-place diventano particolarmente problematiche.
4. **Complessità algoritmica:**
   - Molti algoritmi efficienti per MTS standard diventano inefficienti quando implementati su MTSS.
   - Algoritmi che fanno affidamento su aggiornamenti frequenti (come algoritmi di ricerca iterativi) richiedono riprogettazione completa.
   - La restrizione favorisce algoritmi "write-once" che scrivono risultati in nuove posizioni piuttosto che aggiornare le posizioni esistenti.
5. **Gestione delle computazioni non deterministiche:**

- Simulare computazioni non deterministiche, che tipicamente richiedono backtracking e riscrittura di stati precedenti, diventa complicato.
- Algoritmi che esplorano diversi rami di calcolo devono duplicare interi stati piuttosto che modificare selettivamente parti dello stato.

Esempi di adattamenti necessari:

1. **Contatori**: Invece di incrementare un contatore nella stessa posizione, una MTSS deve rappresentare ogni valore del contatore in una nuova posizione, creando una sequenza di valori.

2. **Attraversamento bidirezionale**: Gli algoritmi che richiedono attraversamenti ripetuti dello stesso input (come il confronto di palindromi) devono essere riprogettati per utilizzare più copie o rappresentazioni alternative dell'input.

3. **Ordinamento**: Algoritmi di ordinamento come il QuickSort, che tipicamente scambiano elementi in posizione, devono essere sostituiti con versioni che creano nuove sequenze ordinate invece di modificare la sequenza originale.

Nonostante queste limitazioni, la dimostrazione di equivalenza con le MTS standard conferma che qualsiasi algoritmo implementabile su una MTS può essere adattato per funzionare su una MTSS, anche se potenzialmente con un significativo costo in termini di efficienza.

Esercizio 10. Una macchina di Turing con "ferma" invece di "sinistra" è simile a una macchina di Turing ordinaria, ma la funzione di transizione ha la forma $\delta : Q \times \Gamma \to Q \times \Gamma \times \{S, R\}$. In ogni punto, la macchina può spostare la testina a destra (R) o lasciarla nella stessa posizione (S).

a) Dimostrare che questa variante della macchina di Turing non è equivalente alla versione usuale

b) Quale classe di linguaggi riconoscono queste macchine?

c) Fornire un esempio di linguaggio che può essere riconosciuto da una macchina di Turing standard ma non da una macchina di Turing con "ferma" invece di "sinistra"

Soluzione. a) Dimostrazione della non equivalenza

Teorema 11. *Una macchina di Turing con "ferma" invece di "sinistra" (MTFS) non è computazionalmente equivalente a una macchina di Turing standard (MTS).*

Proof. Per dimostrare che due modelli di calcolo non sono equivalenti, è sufficiente trovare un problema che può essere risolto in un modello ma non nell'altro.

Una MTFS non può mai muovere la sua testina a sinistra. Questo significa che una volta che la testina ha visitato una cella e si è spostata a destra, quella cella non può mai essere rivisitata. Questo limita drasticamente la capacità della macchina di elaborare l'input in modi complessi che richiedono attraversamenti ripetuti.

Consideriamo la seguente osservazione fondamentale: se una MTFS inizia con la testina nella posizione più a sinistra del nastro (come è convenzionale), allora la testina

può solo spostarsi verso destra o rimanere ferma. Questo significa che la macchina può esaminare ogni cella del nastro al massimo una volta.

Questa restrizione impedisce alla MTFS di implementare molti algoritmi che richiedono attraversamenti multipli dell'input o comparazioni tra porzioni dell'input che si trovano a distanze arbitrarie. Un esempio concreto sarà fornito nel punto (c).

Un'implicazione immediata è che una MTFS non può simulare una MTS per input generici, poiché la MTS può richiedere movimenti arbitrari a sinistra che la MTFS non può replicare. Pertanto, le due varianti non sono computazionalmente equivalenti. □

b) Classe di linguaggi riconosciuti

Le macchine di Turing con "ferma" invece di "sinistra" riconoscono esattamente la classe di linguaggi chiamati "linguaggi regolari" (o "linguaggi di tipo 3" nella gerarchia di Chomsky). Questi sono i linguaggi che possono essere riconosciuti da automi a stati finiti (FSA).

Teorema 12. *Una MTFS riconosce esattamente la classe dei linguaggi regolari.*

Dimostrazione (sketch). La dimostrazione procede in due parti:

Parte 1: Ogni linguaggio regolare può essere riconosciuto da una MTFS.

Questa parte è diretta. Poiché ogni automa a stati finiti può essere simulato da una MTS, può anche essere simulato da una MTFS. La simulazione è semplice: la MTFS legge l'input da sinistra a destra, un carattere alla volta, e cambia stato in base alla funzione di transizione dell'automa. Poiché gli automi a stati finiti non richiedono mai di tornare indietro nell'input, la restrizione della MTFS non comporta limitazioni.

Parte 2: Ogni linguaggio riconosciuto da una MTFS è regolare.

Per questa direzione, osserviamo che una MTFS, a causa della sua incapacità di muoversi a sinistra, può visitare ogni cella dell'input al massimo una volta. Questo significa che la macchina può essere in uno dei suoi stati finiti $q \in Q$ per ogni posizione dell'input.

Possiamo costruire un automa a stati finiti equivalente come segue:

- Gli stati dell'automa includono tutti gli stati della MTFS, più informazioni sul simbolo appena scritto (se applicabile).
- Le transizioni dell'automa simulano le transizioni della MTFS, tenendo conto del fatto che la testina può rimanere ferma.

Poiché l'automa può simulare ogni passo della MTFS e viceversa, i linguaggi riconoscibili sono gli stessi.

Questa equivalenza con gli automi a stati finiti stabilisce che le MTFS riconoscono esattamente i linguaggi regolari. □

c) Esempio di linguaggio riconoscibile da MTS ma non da MTFS

Un esempio classico di linguaggio che può essere riconosciuto da una MTS ma non da una MTFS è il linguaggio delle palindromi, definito come:

$$L_{pal} = \{w \in \{0,1\}^* \mid w = w^R\}$$

dove w^R è il reverso di w. Questo linguaggio contiene tutte le stringhe che rimangono invariate quando lette all'indietro.

Teorema 13. *Il linguaggio L_{pal} può essere riconosciuto da una MTS ma non da una MTFS.*

Proof. **Parte 1:** L_{pal} può essere riconosciuto da una MTS.

Un algoritmo semplice per una MTS che riconosce L_{pal} è il seguente:

1. La MTS marca il primo carattere, si sposta all'estrema destra della stringa e verifica se corrisponde al primo carattere.
2. Se corrisponde, marca anche l'ultimo carattere e si sposta a sinistra di una posizione.
3. La MTS continua questo processo, comparando il secondo carattere con il penultimo, il terzo con il terzultimo, e così via.
4. Se tutti i confronti hanno esito positivo fino al centro della stringa, la MTS accetta. Altrimenti, rigetta.

Questo algoritmo richiede movimenti ripetuti sia a destra che a sinistra, che una MTS può eseguire senza problemi.

Parte 2: L_{pal} non può essere riconosciuto da una MTFS.

Per dimostrare questa affermazione, utilizziamo il teorema del pumping per i linguaggi regolari. Se L_{pal} fosse regolare (e quindi riconoscibile da una MTFS), esisterebbe una costante di pumping $p > 0$ tale che ogni stringa $z \in L_{pal}$ con $|z| \geq p$ potrebbe essere scomposta come $z = uvw$ dove:

- $|uv| \leq p$
- $|v| > 0$
- Per ogni $i \geq 0$, $uv^i w \in L_{pal}$

Consideriamo la palindromo $z = 0^p 10^p$. Secondo il lemma del pumping, possiamo scomporre z come uvw dove v è una sequenza di simboli 0 (poiché $|uv| \leq p$, quindi v è contenuta nella prima metà della stringa).

Se proviamo con $i = 0$, otteniamo $uw = 0^{p-|v|}10^p$, che non è una palindromo poiché il numero di 0 a sinistra dell'1 è diverso dal numero di 0 a destra. Questo contraddice l'assunzione che L_{pal} sia regolare.

Pertanto, L_{pal} non può essere riconosciuto da una MTFS. □

Altri esempi di linguaggi che possono essere riconosciuti da MTS ma non da MTFS includono:

- $L = \{a^n b^n \mid n \geq 0\}$ (linguaggio delle stringhe con un numero uguale di a e b)
- $L = \{ww \mid w \in \{0,1\}^*\}$ (linguaggio delle stringhe ripetute)
- $L = \{a^n b^m c^n \mid n, m \geq 0\}$ (linguaggio con lo stesso numero di a e c)

In generale, qualsiasi linguaggio che richieda il confronto di porzioni distanti dell'input o conteggi e bilanciamenti che non possono essere gestiti da un automa a stati finiti non sarà riconoscibile da una MTFS.

3 Enumeratori e Altre Varianti

Esercizio 14. Un enumeratore è una macchina di Turing speciale che inizia con un nastro vuoto e stampa su un nastro separato (usando una stampante) le stringhe di un certo linguaggio. L'enumeratore può generare le stringhe in qualsiasi ordine, anche con ripetizioni.

a) Spiegare la connessione tra enumeratori e macchine di Turing che riconoscono linguaggi

b) Progettare un enumeratore per il linguaggio $L = \{0^n 1^n \mid n \geq 1\}$

c) Progettare un enumeratore per il linguaggio $L = \{w \mid w \text{ contiene lo stesso numero di 0 e 1}\}$

Soluzione. a) Connessione tra enumeratori e macchine di Turing riconoscitrici

La connessione tra enumeratori e macchine di Turing che riconoscono linguaggi è fondamentale nella teoria della computabilità e rivela caratteristiche importanti dei linguaggi formali.

Definizioni formali:

- Una **macchina di Turing riconoscitrice (MTR)** è una macchina che, dato un input w, accetta se $w \in L$ e rigetta (o non termina) se $w \notin L$.
- Un **enumeratore (E)** è una macchina di Turing che, partendo da un nastro vuoto, genera (stampa) tutte le stringhe di un linguaggio L, possibilmente con ripetizioni e in qualsiasi ordine.

Connessioni fondamentali:

1. **Linguaggi ricorsivamente enumerabili (r.e.):**
   - Un linguaggio L è ricorsivamente enumerabile se e solo se esiste una MTR che lo accetta.
   - Un linguaggio L è ricorsivamente enumerabile se e solo se esiste un enumeratore che genera esattamente le stringhe di L.
2. **Costruzione di un enumeratore da una MTR:** Data una MTR M per un linguaggio L, possiamo costruire un enumeratore E per L come segue:
   - E genera sistematicamente tutte le possibili stringhe in ordine lessicografico: $\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$
   - Per ogni stringa generata w, E simula M su input w per un numero limitato di passi.
   - Se M accetta w entro quel limite, E stampa w.
   - E incrementa progressivamente il limite di passi per garantire che ogni stringa in L venga eventualmente stampata.
3. **Costruzione di una MTR da un enumeratore:** Dato un enumeratore E per un linguaggio L, possiamo costruire una MTR M che riconosce L:

- Data una stringa di input w, M simula E e confronta ogni stringa generata con w.
- Se E genera una stringa uguale a w, M accetta.
- Se $w \notin L$, M continuerà a eseguire senza mai accettare (possibilmente non terminando).

Implicazioni teoriche:

1. **Decidibilità:**
   - Un linguaggio L è decidibile (ricorsivo) se esiste una MTR che accetta e termina per ogni input, accettando se $w \in L$ e rigettando se $w \notin L$.
   - Se L è decidibile, possiamo costruire un enumeratore che genera L in ordine lessicografico senza ripetizioni.
2. **Non decidibilità:**
   - Esistono linguaggi ricorsivamente enumerabili che non sono decidibili.
   - Per questi linguaggi, esiste un enumeratore, ma non esiste una MTR che garantisce la terminazione per tutti gli input.
3. **Complemento:**
   - Se sia L che il suo complemento sono ricorsivamente enumerabili, allora L è decidibile.
   - Ciò equivale a dire che se esistono enumeratori sia per L che per il suo complemento, allora esiste una MTR che decide L.

Importanza nella teoria della computabilità: La connessione tra enumeratori e riconoscitori è fondamentale per caratterizzare i limiti della computabilità. Molti teoremi sulla decidibilità e sulla classificazione dei problemi computazionali si basano su questa connessione. Ad esempio, il famoso problema della fermata è ricorsivamente enumerabile ma non decidibile, il che significa che esiste un enumeratore per esso, ma non una MTR che garantisce la terminazione per tutti gli input.

b) Enumeratore per $L = \{0^n1^n \mid n \geq 1\}$

Progettiamo un enumeratore che genera il linguaggio $L = \{0^n1^n \mid n \geq 1\}$, cioè il linguaggio di tutte le stringhe composte da un numero uguale di 0 seguiti da 1.

Algoritmo dell'enumeratore:

1. L'enumeratore inizia con un contatore $n = 1$.
2. Per ogni valore di n, l'enumeratore genera la stringa 0^n1^n (n zeri seguiti da n uni) e la stampa sul nastro di output.
3. L'enumeratore incrementa n di 1 e ripete dal passo 2.

Descrizione formale dell'enumeratore E:

1. **Inizializzazione:**
   - E inizia con un nastro di lavoro vuoto.
   - E scrive un simbolo speciale (ad esempio #) seguito da un singolo 1 sul nastro di lavoro, per rappresentare $n = 1$.
2. **Generazione della stringa corrente:**
   - E legge il valore di n dal nastro di lavoro (contando il numero di 1 dopo il simbolo #).
   - E genera n zeri seguiti da n uni sul nastro di output (usando la stampante).
3. **Incremento del contatore:**
   - E aggiunge un altro 1 dopo la sequenza di 1 sul nastro di lavoro, incrementando così n.
4. **Ripetizione:**
   - E torna al passo 2 e continua indefinitamente.

Questo enumeratore genera le stringhe del linguaggio nell'ordine: $01, 0011, 000111, 00001111, \ldots$

L'enumeratore genera esattamente tutte le stringhe in L in ordine crescente di lunghezza, senza ripetizioni.

c) Enumeratore per $L = \{w \mid w$ contiene lo stesso numero di 0 e 1$\}$

Progettiamo un enumeratore per il linguaggio $L = \{w \mid w$ contiene lo stesso numero di 0 e 1$\}$, cioè il linguaggio di tutte le stringhe che hanno lo stesso numero di 0 e 1, indipendentemente dall'ordine.

Questo linguaggio è più complesso del precedente, poiché include stringhe come 01, 10, 0011, 0101, 1010, 1001, ecc.

Algoritmo dell'enumeratore:

L'approccio sarà generare tutte le possibili stringhe di lunghezza $2n$ (per ogni $n \geq 1$) che contengono esattamente n zeri e n uni.

1. L'enumeratore inizia con un contatore $n = 1$.
2. Per ogni valore di n, l'enumeratore genera tutte le possibili permutazioni di n zeri e n uni.
3. L'enumeratore incrementa n di 1 e ripete dal passo 2.

Descrizione formale dell'enumeratore E:

1. **Inizializzazione:**
   - E inizia con un nastro di lavoro vuoto.
   - E scrive un simbolo speciale (ad esempio #) seguito da un singolo 1 sul nastro di lavoro, per rappresentare $n = 1$.

2. **Generazione di tutte le permutazioni:** Per generare tutte le permutazioni di n zeri e n uni, l'enumeratore E utilizza un approccio sistematico:
   - E inizia con un secondo contatore $k = 0$ (che rappresenta il numero binario delle posizioni in cui inserire gli 1).
   - Per ogni k da 0 a $\binom{2n}{n} - 1$:
     - E converte k in una rappresentazione binaria con $2n$ bit.
     - Per ogni posizione i da 1 a $2n$:
       - Se il bit i di k è 1, E stampa 1 sul nastro di output.
       - Se il bit i di k è 0, E stampa 0 sul nastro di output.
     - E conta il numero di 1 nella stringa generata.
     - Se la stringa contiene esattamente n uni (e quindi n zeri), E la accetta e passa al valore successivo di k.
     - Altrimenti, scarta la stringa e passa al valore successivo di k.
3. **Incremento di n:**
   - Dopo aver generato tutte le permutazioni per un dato n, E incrementa n di 1 sul nastro di lavoro.
   - E torna al passo 2 e continua indefinitamente.

Approccio alternativo più efficiente:
Un approccio più efficiente che evita di generare e scartare stringhe non valide è il seguente:

1. **Inizializzazione:** E inizia con $n = 1$.
2. **Generazione sistematica:** Per ogni n, E genera tutte le stringhe di lunghezza $2n$ con esattamente n zeri e n uni utilizzando un algoritmo combinatorio:
   - E inizializza una stringa di $2n$ bit con i primi n bit impostati a 0 e i successivi n bit impostati a 1.
   - E stampa questa stringa (inizialmente 0^n1^n).
   - E genera sistematicamente tutte le permutazioni utilizzando l'algoritmo di "next permutation":
     - Trova il più grande indice i tale che la stringa$[i] <$ stringa$[i+1]$.
     - Se non esiste tale indice, tutte le permutazioni sono state generate.
     - Trova il più grande indice j tale che stringa$[i] <$ stringa$[j]$.
     - Scambia stringa$[i]$ e stringa$[j]$.
     - Inverte la sottostringa stringa$[i+1:]$.
     - Stampa la nuova permutazione.
     - Ripeti fino a quando non vengono generate tutte le permutazioni.
3. **Incremento di n:** Dopo aver generato tutte le permutazioni per un dato n, E incrementa n di 1 e ripete dal passo 2.

Questo enumeratore genera tutte le stringhe del linguaggio L raggruppate per lunghezza. Per $n = 1$, genera $\{01, 10\}$. Per $n = 2$, genera $\{0011, 0101, 0110, 1001, 1010, 1100\}$, e così via.

Entrambi gli enumeratori descritti generano esattamente tutte le stringhe del linguaggio L senza omissioni e senza includere stringhe che non appartengono al linguaggio.