Tutorato di Automi e Linguaggi Formali

Soluzioni Homework 8: Varianti delle Macchine di Turing e Decidibilità

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

Tutorato 8 - 12-05-2025

1 Varianti Avanzate delle Macchine di Turing

Esercizio 1. Una macchina di Turing a nastri multipli è una TM con k nastri, ciascuno con la propria testina. Formalmente, viene definita come una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, dove tutti i componenti sono definiti come per una TM standard, eccetto la funzione di transizione:

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

a) Dimostrate che una TM a k nastri è equivalente a una TM standard a singolo nastro in termini di potere computazionale, fornendo una costruzione dettagliata della simulazione.

b) Analizzate la complessità temporale della simulazione. Se una TM a k nastri opera in tempo $T(n)$, qual è il tempo richiesto dalla TM a nastro singolo che la simula?

c) Fornite un esempio concreto di un problema che può essere risolto più efficientemente usando una TM a nastri multipli rispetto a una TM a nastro singolo.

Soluzione. a) Equivalenza computazionale

Teorema 2. *Una macchina di Turing a k nastri (MTkN) è equivalente a una macchina di Turing standard a singolo nastro (MTS) in termini di potere computazionale.*

Proof. Per dimostrare l'equivalenza, dobbiamo mostrare che:

- Una MTS può simulare una MTkN
- Una MTkN può simulare una MTS

La seconda direzione è triviale, poiché una MTkN può semplicemente utilizzare solo il primo nastro e ignorare gli altri, comportandosi esattamente come una MTS.

Per la prima direzione, costruiamo una MTS che simula una MTkN come segue:

1. **Rappresentazione dei nastri:** Il nastro della MTS viene diviso in k tracce, ciascuna corrispondente a un nastro della MTkN. Possiamo rappresentare questo usando un alfabeto esteso: $\Gamma' = (\Gamma \cup \{\#\})^k$, dove $\#$ è un nuovo simbolo che indica la posizione della testina.

2. **Codifica dell'informazione:** Per ogni cella del nastro della MTS, memorizziamo una k-tupla contenente il simbolo presente in quella posizione in ciascuno dei k nastri della MTkN. Inoltre, per tenere traccia della posizione di ciascuna testina, utilizziamo k simboli speciali che marcano la posizione di ciascuna testina nei rispettivi nastri.

3. **Simulazione di un passo:** Per simulare un singolo passo della MTkN:

   - La MTS scansiona l'intero nastro da sinistra a destra per identificare la posizione di ciascuna testina.
   - Basandosi sullo stato corrente della MTkN e sui simboli letti da ciascuna testina, la MTS determina la nuova configurazione.
   - La MTS aggiorna i simboli e le posizioni delle testine effettuando un'altra scansione completa del nastro.

Costruzione formale:

Sia $M_k = (Q_k, \Sigma, \Gamma_k, \delta_k, q_0, q_{accept}, q_{reject})$ una MTkN. Costruiamo una MTS $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0', q_{accept}', q_{reject}')$ che la simula:

- $\Gamma_1 = (\Gamma_k \times \{0,1\})^k \cup \{\triangleright\}$, dove $\triangleright$ è un simbolo per marcare l'inizio del nastro, e $\{0,1\}$ indica se la testina è presente (1) o assente (0) in quella posizione per quel nastro.

- Q_1 include stati per gestire le varie fasi della simulazione:

  - Stati di scansione per trovare le posizioni delle testine
  - Stati per memorizzare i simboli letti
  - Stati per aggiornare le posizioni delle testine e i simboli
  - Stati che corrispondono direttamente agli stati di Q_k

- La funzione di transizione δ_1 implementa l'algoritmo di simulazione descritto sopra.

La MTS così costruita può simulare esattamente il comportamento della MTkN, includendo movimenti arbitrari delle testine e cambiamenti di stato. Pertanto, qualsiasi linguaggio riconosciuto da una MTkN può essere riconosciuto anche da una MTS adeguatamente costruita. □

b) Analisi della complessità temporale

Teorema 3. *Se una MTkN opera in tempo $T(n)$, allora la MTS che la simula richiede un tempo $O(T(n)^2)$.*

Proof. Per analizzare la complessità temporale della simulazione, consideriamo i passi necessari per simulare un singolo passo della MTkN:

1. **Fase di scansione:** La MTS deve scansionare l'intero nastro per identificare la posizione di ciascuna testina. Nel caso peggiore, questa scansione richiede $O(T(n))$ passi, poiché dopo $T(n)$ passi di computazione, la MTkN può aver utilizzato al massimo $O(T(n))$ celle del nastro.

2. **Fase di aggiornamento:** Dopo aver determinato la nuova configurazione, la MTS deve effettuare un'altra scansione completa per aggiornare le posizioni delle testine e i simboli. Anche questa fase richiede $O(T(n))$ passi.

Pertanto, simulare un singolo passo della MTkN richiede $O(T(n))$ passi nella MTS. Poiché la MTkN esegue $T(n)$ passi, la simulazione completa richiede $O(T(n) \cdot T(n)) = O(T(n)^2)$ passi.

Analisi dettagliata:

In realtà, possiamo essere più precisi riguardo al fattore costante nella notazione $O(T(n)^2)$. Simulare un singolo passo della MTkN richiede:

- k scansioni per trovare la posizione di ciascuna testina (o un'unica scansione se le posizioni sono codificate direttamente)

- 1 scansione per aggiornare i simboli e le posizioni delle testine

Quindi, il numero esatto di passi è dell'ordine di $c \cdot T(n) \cdot T(n)$, dove c è una costante che dipende dal numero di nastri k e dai dettagli dell'implementazione.

È importante notare che questa è solo un'analisi del caso peggiore. In pratica, ottimizzazioni nella simulazione potrebbero ridurre il tempo effettivo, ma la complessità asintotica rimane $O(T(n)^2)$. □

c) Esempio di problema risolto più efficientemente con nastri multipli

Un esempio concreto di problema che può essere risolto più efficientemente usando una TM a nastri multipli rispetto a una TM a nastro singolo è il problema del **riconoscimento di palindromi**.

Problema: Data una stringa $w \in \{0,1\}^*$, determinare se w è una palindroma, cioè se $w = w^R$ (dove w^R è il reverso di w).

Soluzione con TM a nastro singolo: Con una TM a nastro singolo, l'algoritmo richiede movimenti avanti e indietro ripetuti:

1. Leggere il primo simbolo a_1 della stringa

2. Muoversi fino alla fine della stringa

3. Confrontare l'ultimo simbolo a_n con a_1

4. Se $a_1 \neq a_n$, rigettare

5. Altrimenti, marcare sia a_1 che a_n come "visitati"

6. Muoversi al secondo simbolo a_2

7. Ripetere le operazioni per confrontare a_2 con a_{n-1}, e così via

La complessità temporale è $O(n^2)$, dove n è la lunghezza della stringa, poiché per ogni coppia di simboli da confrontare (ce ne sono $n/2$), la testina deve attraversare quasi l'intera stringa (lunghezza $O(n)$).

Soluzione con TM a 2 nastri: Con una TM a 2 nastri, l'algoritmo diventa molto più efficiente:

1. Copiare l'input dal nastro 1 al nastro 2 (richiede $O(n)$ passi)

2. Riportare la testina del nastro 1 all'inizio e posizionare la testina del nastro 2 alla fine

3. Confrontare simultaneamente i simboli nei due nastri, muovendo la testina del nastro 1 verso destra e la testina del nastro 2 verso sinistra

4. Se tutti i simboli corrispondono, accettare; altrimenti rigettare

La complessità temporale è $O(n)$, dove n è la lunghezza della stringa:

- $O(n)$ passi per copiare l'input
- $O(n)$ passi per posizionare le testine
- $O(n/2)$ passi per il confronto parallelo

Altri esempi: Altri problemi che beneficiano significativamente dell'uso di nastri multipli includono:

- **Riconoscimento del linguaggio** $\{ww \mid w \in \{0,1\}^*\}$: Con nastri multipli, possiamo copiare l'input, determinare il punto medio, e confrontare le due metà in tempo lineare.
- **Ordinamento di sequenze**: Con nastri multipli, possiamo implementare algoritmi di ordinamento più efficienti, utilizzando nastri separati per diverse partizioni o fasi dell'algoritmo.
- **Simulazione di algoritmi paralleli**: I nastri multipli permettono di simulare computazioni che avvengono in parallelo, riducendo significativamente il tempo rispetto a implementazioni sequenziali.

In generale, i nastri multipli offrono un vantaggio significativo quando è necessario accedere simultaneamente a diverse parti dell'input o mantenere strutture dati ausiliarie separate, riducendo la necessità di spostamenti ripetuti della testina che sono inevitabili con un singolo nastro.

Esercizio 4. Una macchina di Turing a nastro circolare è una TM in cui il nastro forma un anello, permettendo alla testina di muoversi continuamente verso destra (o sinistra) senza mai raggiungere la fine del nastro. La lunghezza del nastro è finita ma può essere espansa quando necessario.

a) Dimostrate che una TM a nastro circolare è equivalente a una TM standard.

b) Descrivete un algoritmo che permetta a una TM a nastro circolare di simulare una TM standard.

c) Ideate un algoritmo per il riconoscimento di stringhe palindrome che sfrutti l'architettura a nastro circolare in modo efficiente.

Soluzione. a) Equivalenza tra TM a nastro circolare e TM standard

Teorema 5. *Una macchina di Turing a nastro circolare (TMNC) è equivalente a una macchina di Turing standard (TMS) in termini di potere computazionale.*

Proof. Per dimostrare l'equivalenza, dobbiamo mostrare che:

- Una TMS può simulare una TMNC
- Una TMNC può simulare una TMS

Parte 1: TMS può simulare TMNC
La simulazione di una TMNC da parte di una TMS è relativamente semplice:

- La TMS mantiene sul suo nastro infinito una rappresentazione della configurazione del nastro circolare.
- La TMS tiene traccia della posizione corrente della testina sulla rappresentazione del nastro circolare.
- Se la TMNC espande il nastro circolare, la TMS aggiorna la sua rappresentazione di conseguenza.

Poiché il nastro della TMS è infinito, può sempre rappresentare un nastro circolare di qualsiasi dimensione finita.

Parte 2: TMNC può simulare TMS
Questa direzione è più complessa, poiché dobbiamo simulare un nastro potenzialmente infinito con un nastro circolare finito che può essere espanso:

1. **Rappresentazione del nastro:** Il nastro circolare contiene una porzione del nastro infinito della TMS, centrata intorno alla posizione corrente della testina.
2. **Gestione dei limiti:** Quando la testina della TMNC si avvicina al "confine" della porzione rappresentata, la TMNC espande il nastro circolare per includere più celle in quella direzione.
3. **Marcatura di confini:** Per distinguere tra le celle che fanno parte del nastro della TMS e le celle "non ancora visitate", possiamo usare un simbolo speciale (ad esempio, #) per marcare i confini della porzione rappresentata.

Dettagli della simulazione:

1. Inizialmente, il nastro circolare contiene l'input della TMS, con simboli # che marcano i confini a sinistra e a destra.

2. Quando la testina raggiunge un simbolo #, la TMNC espande il nastro circolare inserendo nuove celle (inizializzate con il simbolo blank) in quella direzione e sposta il marcatore #.

3. La TMNC mantiene lo stesso stato interno della TMS simulata e applica le stesse transizioni, intercettando solo i movimenti verso i marcatori # per gestire l'espansione.

Questa simulazione garantisce che la TMNC possa rappresentare e manipolare qualsiasi porzione finita del nastro infinito della TMS che viene effettivamente utilizzata durante la computazione. Poiché ogni TMS che termina utilizza solo una porzione finita del nastro, la TMNC può simulare completamente il suo comportamento.

Pertanto, TMS e TMNC hanno lo stesso potere computazionale. □

b) Algoritmo per simulare una TM standard con una TM a nastro circolare

Presentiamo ora un algoritmo dettagliato che permette a una TMNC di simulare una TMS:

Correttezza dell'algoritmo: Questo algoritmo garantisce che la TMNC possa simulare fedelmente qualsiasi computazione della TMS:

- Per le transizioni standard (che non coinvolgono i confini), la TMNC si comporta esattamente come la TMS.

- Quando la TMS tenta di accedere a una cella oltre il confine rappresentato, la TMNC espande il nastro circolare, inserisce il simbolo blank appropriato, e continua la simulazione.

- L'espansione può avvenire un numero arbitrario di volte, permettendo alla TMNC di simulare nastri di qualsiasi lunghezza effettivamente utilizzata dalla TMS.

c) Algoritmo per il riconoscimento di stringhe palindrome con TM a nastro circolare

Progettiamo ora un algoritmo efficiente per il riconoscimento di stringhe palindrome che sfrutti l'architettura a nastro circolare:

Vantaggi del nastro circolare per questo algoritmo:

1. **Efficienza nei movimenti:** Il nastro circolare permette di passare rapidamente dall'inizio alla fine della stringa semplicemente proseguendo nella stessa direzione, senza dover invertire la direzione della testina.

2. **Eliminazione del bisogno di marcatori di fine:** Poiché il nastro è circolare, possiamo facilmente identificare l'inizio e la fine della stringa con un solo marcatore, e ciclare attraverso l'intera stringa con un movimento continuo.

3. **Implementazione di "puntatori duali":** L'algoritmo può mantenere efficacemente due "puntatori" a diverse posizioni della stringa semplicemente muovendosi in senso orario o antiorario sul nastro circolare.

Analisi della complessità:

- Senza nastro circolare, il riconoscimento di palindromi richiede $O(n^2)$ passi con una TM a singolo nastro.

Input: Una TMS $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ e una stringa di input $w = w_1 w_2 \dots w_n$.
Output: Una TMNC $M' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject})$ che simula M.

1. **Definizione dell'alfabeto esteso:** $\Gamma' = \Gamma \cup \{\#\}$, dove # è un simbolo speciale che marca i confini del nastro rappresentato.

2. **Inizializzazione del nastro circolare:**

   - Inizialmente, il nastro circolare contiene $\#w_1 w_2 \dots w_n \#$
   - La testina è posizionata su w_1

3. **Definizione degli stati:** $Q' = Q \cup \{q_{espandi\_sx}, q_{espandi\_dx}, q_{riposiziona\_sx}, q_{riposiziona\_dx}\} \cup \{q_{copia\_sx}, q_{copia\_dx}\}$, dove gli stati aggiuntivi servono per gestire l'espansione del nastro.

4. **Simulazione delle transizioni standard:** Per ogni transizione $\delta(q, a) = (q', b, D)$ in M dove $a \neq \#$ e $D \in \{L, R\}$:

   - Se $D = R$, definiamo $\delta'(q, a) = (q', b, R)$
   - Se $D = L$, definiamo $\delta'(q, a) = (q', b, L)$

5. **Gestione del confine sinistro:** Per ogni stato $q \in Q$ e transizione $\delta(q, a) = (q', b, L)$ in M:

   - Definiamo $\delta'(q, \#) = (q_{espandi\_sx}, \#, L)$

   Lo stato $q_{espandi\_sx}$ inizia la procedura di espansione a sinistra:

   - $\delta'(q_{espandi\_sx}, *) = (q_{copia\_sx}, *, R)$ per ogni simbolo $* \in \Gamma'$

   Lo stato q_{copia_sx} sposta tutti i simboli di una posizione a destra:

   - $\delta'(q_{copia\_sx}, a) = (q_{copia\_sx}, \sqcup, R)$ per ogni $a \in \Gamma'$
   - Quando raggiunge il confine destro #, torna indietro: $\delta'(q_{copia\_sx}, \#) = (q_{riposiziona\_sx}, \#, L)$

   Lo stato $q_{riposiziona\_sx}$ torna all'inizio:

   - $\delta'(q_{riposiziona\_sx}, *) = (q_{riposiziona\_sx}, *, L)$ per ogni $* \in \Gamma'$
   - Quando raggiunge il confine sinistro: $\delta'(q_{riposiziona\_sx}, \#) = (q', b, R)$ (completando la transizione originale)

6. **Gestione del confine destro:** Per ogni stato $q \in Q$ e transizione $\delta(q, a) = (q', b, R)$ in M:

   - Definiamo $\delta'(q, \#) = (q_{espandi\_dx}, \#, R)$

   Analogamente, definiamo stati e transizioni per espandere il nastro a destra, inserendo un nuovo simbolo blank e spostando il marcatore #.

7. **Stati finali:**

   - $q'_{accept} = q_{accept}$
   - $q'_{reject} = q_{reject}$

Input: Una stringa $w = w_1 w_2 \dots w_n \in \{0,1\}^*$.
Output: Accettazione se w è una palindroma, rigetto altrimenti.

1. **Inizializzazione:**
   - Il nastro circolare contiene inizialmente $w = w_1 w_2 \dots w_n$
   - Aggiungiamo due marcatori speciali: $\$w_1 w_2 \dots w_n\#$
   - La testina è posizionata su \$
2. **Prima fase - Marcatura dei simboli esterni:**
   - La macchina si sposta a destra fino al primo simbolo non marcato w_i
   - Memorizza w_i nello stato e marca w_i con un simbolo X
   - Si sposta a destra fino al marcatore #
   - Si sposta a sinistra fino al primo simbolo non marcato w_j partendo dalla fine
   - Confronta w_j con il simbolo memorizzato w_i
   - Se $w_i \neq w_j$, rigetta
   - Altrimenti, marca w_j con un simbolo Y e torna al marcatore \$
3. **Iterazione:**
   - Ripeti la fase 1 finché tutti i simboli sono marcati o fino alla convergenza delle marcature
4. **Verifica finale:**
   - Se tutti i simboli sono stati marcati senza rilevare differenze, accetta
   - Se è stata trovata anche solo una differenza, rigetta

- Con un nastro circolare, possiamo ridurre la complessità a $O(n)$ grazie alla capacità di muoversi tra l'inizio e la fine della stringa in un singolo movimento circolare.

Questa implementazione dimostra come l'architettura a nastro circolare possa essere particolarmente vantaggiosa per algoritmi che richiedono confronti tra elementi distanti nell'input, eliminando la necessità di attraversamenti ripetuti che sarebbero inevitabili con un nastro lineare.

Esercizio 6. 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) Dimostrate che questa variante della macchina di Turing è meno potente della TM standard, ovvero esiste un linguaggio riconoscibile da una TM standard che non può essere riconosciuto da una TM con "ferma" invece di "sinistra".

b) Caratterizzate la classe di linguaggi riconoscibili da una TM con "ferma" invece di "sinistra" in termini di una classe nota di linguaggi.

c) Fornite un esempio di un linguaggio semplice che può essere riconosciuto da una TM standard ma non da questa variante.

Soluzione. a) Dimostrazione che la variante è meno potente

Teorema 7. *Una macchina di Turing con "ferma" invece di "sinistra" (TMFS) è strettamente meno potente di una macchina di Turing standard (TMS).*

Proof. Per dimostrare che le TMFS sono meno potenti delle TMS, dobbiamo mostrare che:

1. Ogni linguaggio riconosciuto da una TMFS può essere riconosciuto da una TMS.
2. Esiste almeno un linguaggio riconosciuto da una TMS che non può essere riconosciuto da una TMFS.

La prima condizione è banale, poiché una TMS può facilmente simulare una TMFS semplicemente non utilizzando mai la mossa a sinistra.

Per la seconda condizione, consideriamo la seguente limitazione fondamentale delle TMFS: una volta che la testina si è spostata dalla posizione i alla posizione $i + 1$, non può mai tornare alla posizione i o a qualsiasi posizione a sinistra di i.

Questa limitazione ha conseguenze significative:

- Ogni cella del nastro può essere letta al massimo una volta.
- La macchina può solo esaminare un prefisso continuo dell'input, procedendo da sinistra a destra.
- Non può confrontare simboli distanti dell'input o tornare a porzioni precedentemente esaminate.

Ora, consideriamo la riconoscibilità del linguaggio delle palindromi $L_{pal} = \{w \in \{0,1\}^* \mid w = w^R\}$. Per verificare se una stringa è una palindroma, è necessario confrontare il primo e l'ultimo simbolo, il secondo e il penultimo, e così via. Questo richiede inevitabilmente di rivisitare posizioni precedenti del nastro, cosa impossibile per una TMFS.

Più formalmente, possiamo dimostrare che L_{pal} non può essere riconosciuto da una TMFS utilizzando un argomento di pumping o di teoria degli automi. In particolare, possiamo dimostrare che i linguaggi riconoscibili da TMFS sono esattamente i linguaggi regolari, e L_{pal} è noto per essere un linguaggio non regolare.

Pertanto, esiste almeno un linguaggio (ovvero L_{pal}) che può essere riconosciuto da una TMS ma non da una TMFS, il che dimostra che le TMFS sono strettamente meno potenti delle TMS. □

b) Caratterizzazione della classe di linguaggi riconoscibili

Teorema 8. *I linguaggi riconoscibili da una macchina di Turing con "ferma" invece di "sinistra" (TMFS) sono esattamente i linguaggi regolari.*

Proof. Dimostriamo questa caratterizzazione in due parti:

Parte 1: Ogni linguaggio regolare è riconoscibile da una TMFS.

Dato un linguaggio regolare L, esiste un automa a stati finiti determinisitco (DFA) $A = (Q, \Sigma, \delta_A, q_0, F)$ che lo riconosce. Possiamo costruire una TMFS $M = (Q', \Sigma, \Gamma, \delta_M, q_0', q_{accept}, q_{reject})$ che simula A come segue:

- $Q' = Q \cup \{q_{accept}, q_{reject}\}$
- $\Gamma = \Sigma \cup \{\sqcup\}$
- Per ogni transizione $\delta_A(q, a) = q'$ in A, definiamo $\delta_M(q, a) = (q', a, R)$
- Per ogni stato $q \in Q$, definiamo $\delta_M(q, \sqcup) = (q_f, \sqcup, S)$ dove $q_f = q_{accept}$ se $q \in F$, altrimenti $q_f = q_{reject}$

Questa costruzione garantisce che M accetta esattamente le stesse stringhe di A, procedendo da sinistra a destra e accettando o rigettando una volta raggiunta la fine dell'input.

Parte 2: Ogni linguaggio riconoscibile da una TMFS è regolare.

Sia $M = (Q, \Sigma, \Gamma, \delta_M, q_0, q_{accept}, q_{reject})$ una TMFS che riconosce un linguaggio L. Costruiamo un DFA $A = (Q', \Sigma, \delta_A, q_0', F)$ che riconosce lo stesso linguaggio:

- $Q' = Q \times \Gamma$ (ogni stato del DFA tiene traccia sia dello stato della TMFS sia del simbolo corrente sul nastro)
- $q_0' = (q_0, \sqcup)$ (lo stato iniziale corrisponde allo stato iniziale della TMFS con il simbolo blank)
- $F = \{(q, a) \in Q' \mid q = q_{accept}\}$ (gli stati di accettazione corrispondono agli stati in cui la TMFS accetta)

La funzione di transizione δ_A è definita come segue:

- Per ogni transizione $\delta_M(q, a) = (q', b, S)$ in M, definiamo $\delta_A((q, c), a) = (q', b)$

- Per ogni transizione $\delta_M(q, a) = (q', b, R)$ in M, definiamo $\delta_A((q, c), a) = (q', \sqcup)$

L'intuizione dietro questa costruzione è che il DFA simula la TMFS leggendo un simbolo alla volta e aggiornando il suo stato interno per riflettere sia lo stato della TMFS sia il contenuto della cella corrente. Poiché la TMFS non può mai tornare indietro, ogni input viene elaborato in modo strettamente sequenziale, esattamente come fa un DFA.

Questa costruzione dimostra che ogni linguaggio riconosciuto da una TMFS può essere riconosciuto anche da un DFA, e quindi è un linguaggio regolare. □

c) Esempio di linguaggio riconoscibile da TMS ma non da TMFS

Un esempio semplice di linguaggio che può essere riconosciuto da una TMS ma non da una TMFS è il linguaggio $L = \{0^n 1^n \mid n \geq 1\}$, ovvero il linguaggio delle stringhe composte da un numero uguale di 0 seguiti da 1.

Teorema 9. *Il linguaggio $L = \{0^n 1^n \mid n \geq 1\}$ può essere riconosciuto da una TMS ma non da una TMFS.*

Proof. **Parte 1: L può essere riconosciuto da una TMS**

Una TMS può riconoscere L con il seguente algoritmo:

1. Verificare che l'input sia nella forma 0^*1^* (tutti gli 0 seguiti da tutti gli 1)
2. Marcare il primo 0 con un simbolo speciale X
3. Spostarsi a destra fino a trovare il primo 1
4. Marcare questo 1 con un simbolo speciale Y
5. Tornare indietro all'inizio dell'input
6. Ripetere i passi 2-5 finché tutti gli 0 e 1 sono stati marcati
7. Accettare se tutti i simboli sono stati marcati, altrimenti rigettare

Questa procedura garantisce che la TMS accetta solo se il numero di 0 è uguale al numero di 1, e rigetta altrimenti.

Parte 2: L non può essere riconosciuto da una TMFS

Per dimostrare che L non può essere riconosciuto da una TMFS, utilizziamo il fatto che i linguaggi riconoscibili da TMFS sono esattamente i linguaggi regolari, come dimostrato nel punto (b).

$L = \{0^n 1^n \mid n \geq 1\}$ è un classico esempio di linguaggio non regolare, come può essere dimostrato utilizzando il lemma del pumping per i linguaggi regolari:

Sia $p > 0$ la costante di pumping. Consideriamo la stringa $z = 0^p 1^p \in L$. Secondo il lemma del pumping, se L fosse regolare, z potrebbe essere scomposta come $z = uvw$ con le seguenti proprietà:

- $|uv| \leq p$ (quindi uv consiste solo di 0)
- $|v| > 0$ (quindi v contiene almeno un 0)
- Per ogni $i \geq 0$, $uv^i w \in L$

Consideriamo la stringa $uv^0w = uw$. Questa stringa ha meno 0 che 1, quindi $uw \notin L$, contraddicendo il lemma del pumping.

Pertanto, L non è un linguaggio regolare e non può essere riconosciuto da una TMFS. □

Altri esempi di linguaggi che possono essere riconosciuti da una TMS ma non da una TMFS includono:

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

In generale, qualsiasi linguaggio che richieda il conteggio o il confronto di simboli distanti nell'input non può essere riconosciuto da una TMFS, poiché queste operazioni richiedono la capacità di tornare indietro o di mantenere informazioni complesse, cosa impossibile per un dispositivo che può solo procedere in avanti.

Esercizio 10. 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) Dimostrate che questa variante di macchina di Turing è equivalente alla macchina di Turing standard, fornendo una costruzione dettagliata.

b) Descrivete un algoritmo efficiente per simulare una TM standard usando una TM a singola scrittura.

c) Discutete quali limitazioni pratiche questa restrizione impone all'implementazione di algoritmi ed eventualmente come possono essere superate.

Soluzione. a) Dimostrazione dell'equivalenza

Teorema 11. *Una macchina di Turing a singola scrittura (TMSS) è equivalente a una macchina di Turing standard (TMS) in termini di potere computazionale.*

Proof. Per dimostrare l'equivalenza, dobbiamo mostrare che:

- Una TMS può simulare una TMSS
- Una TMSS può simulare una TMS

La prima direzione è triviale: una TMS può facilmente simulare una TMSS semplicemente limitandosi a scrivere una sola volta in ogni cella. Questa è una restrizione che la TMS può auto-imporre.

La seconda direzione richiede una costruzione più elaborata. L'idea chiave è che una TMSS può simulare una TMS mantenendo una "cronologia" delle modifiche effettuate su ogni cella:

Costruzione:

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una TMS. Costruiremo una TMSS $M' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept},$ che simula M come segue:

1. **Rappresentazione del nastro:** Invece di memorizzare solo il simbolo corrente in ogni cella, la TMSS utilizza un formato speciale per rappresentare la "cronologia" delle scritture:
   - Ogni cella logica del nastro di M è rappresentata da un gruppo di celle fisiche nel nastro di M'.
   - Ogni gruppo contiene: un marcatore di inizio cella, il simbolo corrente, un indicatore di posizione della testina, e spazio per registrare le future modifiche.
2. **Simulazione di un passo:** Per simulare un passo di M che riscrive una cella:
   - M' trova la cella logica corrispondente
   - Invece di sovrascrivere il simbolo esistente, M' aggiunge il nuovo simbolo alla cronologia della cella
   - M' aggiorna l'indicatore del simbolo corrente per puntare al nuovo simbolo
3. **Gestione del movimento:** Per simulare il movimento della testina:
   - M' aggiorna gli indicatori di posizione della testina nelle celle logiche corrispondenti
   - M' sposta la sua testina fisica alla posizione corrispondente nel nuovo formato

Dettagli della rappresentazione:
Per essere più specifici, ogni cella logica del nastro di M è rappresentata nel nastro di M' come una sequenza di celle fisiche nel formato:

$$\langle a_0, a_1, a_2, \ldots, a_k, p, h \rangle$$

dove:

- $\langle$ e $\rangle$ sono marcatori di inizio e fine cella
- a_0 è il simbolo iniziale della cella
- $a_1, a_2, \ldots, a_k$ sono i simboli scritti successivamente nella cella
- p è un puntatore che indica quale dei simboli a_i è il simbolo corrente
- h è un flag che indica se la testina di M è attualmente su questa cella

Quando M esegue una transizione $\delta(q, a) = (q', b, D)$ che riscrive una cella da a a b, M' aggiunge b alla cronologia e aggiorna il puntatore p per indicare che b è ora il simbolo corrente, invece di sovrascrivere direttamente.

Questa costruzione garantisce che M' possa simulare fedelmente qualsiasi computazione di M, pur rispettando il vincolo di singola scrittura per ogni cella fisica. □

b) Algoritmo efficiente per la simulazione

Descriviamo ora un algoritmo più dettagliato ed efficiente per simulare una TMS usando una TMSS:

Ottimizzazioni:

Per rendere l'algoritmo più efficiente:

Input: Una TMS $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ e una stringa di input w.
Output: Una TMSS $M' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject})$ che simula M.

1. **Inizializzazione:**
   - La TMSS inizia convertendo l'input w nel formato esteso:
   $$\triangle\langle w_1, \square, 1, 1\rangle\langle w_2, \square, 1, 0\rangle \ldots \langle w_n, \square, 1, 0\rangle\langle \sqcup, \square, 1, 0\rangle \ldots$$
   - Dove $\triangle$ è un marcatore speciale di inizio nastro, $\langle$ e $\rangle$ sono delimitatori di cella, $\square$ indica spazio per future scritture, il primo 1 indica che il simbolo attuale è il primo della lista, e l'ultimo bit indica se la testina è presente (1) o assente (0).
2. **Algoritmo di simulazione principale:**
   (a) **Localizzazione della testina:**
   - La TMSS scansiona il nastro per trovare la cella logica con indicatore di testina = 1
   - Legge il simbolo corrente di questa cella (indicato dal puntatore)

   (b) **Esecuzione della transizione:**
   - Basandosi sullo stato corrente e sul simbolo letto, la TMSS determina la transizione da eseguire
   - Per una transizione $\delta(q, a) = (q', b, D)$ della TMS:
     – La TMSS aggiunge b alla lista di simboli nella cella corrente
     – Aggiorna il puntatore per indicare che b è ora il simbolo corrente
     – Imposta l'indicatore di testina della cella corrente a 0

   (c) **Movimento della testina:**
   - Se $D = R$, la TMSS si sposta alla cella logica successiva
   - Se $D = L$, la TMSS si sposta alla cella logica precedente
   - Imposta l'indicatore di testina della nuova cella a 1

   (d) **Creazione di nuove celle:**
   - Se la testina si sposta oltre l'estremità rappresentata del nastro, la TMSS crea una nuova cella logica nel formato appropriato

   (e) **Iterazione:**
   - La TMSS ripete i passi precedenti finché non raggiunge uno stato finale
3. **Accettazione/Rigetto:**
   - La TMSS accetta se la simulazione della TMS raggiunge lo stato q_{accept}
   - La TMSS rigetta se la simulazione della TMS raggiunge lo stato q_{reject}

1. **Rappresentazione compatta:**
    - Invece di allocare spazio fisso per le future scritture, la TMSS può utilizzare un formato dinamico che espande la rappresentazione solo quando necessario.
    - La cronologia può essere rappresentata in modo più compatto, memorizzando solo le modifiche effettive.
2. **Caching degli accessi frequenti:**
    - Per celle che vengono accedute frequentemente, la TMSS può mantenere una "cache" di celle logiche in posizioni facilmente accessibili.
    - Questo riduce il numero di scansioni necessarie per trovare celle specifiche.
3. **Gestione intelligente dello spazio:**
    - La TMSS può organizzare il nastro in modo da ridurre la distanza media tra celle logiche consecutive.
    - Celle che vengono accedute frequentemente insieme possono essere posizionate vicine sul nastro fisico.

Questo algoritmo garantisce che la TMSS simuli correttamente la TMS, mantenendo la restrizione di singola scrittura per ogni cella fisica.

c) Limitazioni pratiche e possibili soluzioni

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

1. Aumento dell'occupazione di spazio:

- **Limitazione:** Per ogni cella logica che viene modificata k volte nella TMS originale, la TMSS richiede $O(k)$ celle fisiche.
- **Impatto:** Algoritmi che modificano ripetutamente le stesse posizioni (come contatori o strutture dati dinamiche) diventano inefficienti in termini di spazio.
- **Soluzione:** Utilizzare codifiche più compatte per la cronologia e tecniche di garbage collection per recuperare spazio quando possibile.

2. Overhead temporale:

- **Limitazione:** Accedere al contenuto corrente di una cella logica richiede la scansione della sua rappresentazione estesa.
- **Impatto:** Operazioni che nella TMS originale richiedono tempo costante possono richiedere tempo lineare nella TMSS.
- **Soluzione:** Utilizzare tecniche di indicizzazione e caching per accelerare gli accessi frequenti.

3. Complessità algoritmica:

- **Limitazione:** Algoritmi che fanno affidamento su aggiornamenti in-place diventano complessi da implementare.

- **Impatto:** Operazioni come l'ordinamento in-place o la manipolazione di grafi richiedono riprogettazione significativa.
- **Soluzione:** Adottare paradigmi algoritmici che favoriscono operazioni immutabili, come la programmazione funzionale.

4. Frammentazione del nastro:

- **Limitazione:** Con il tempo, la rappresentazione delle celle logiche può diventare sparsa sul nastro fisico.
- **Impatto:** Aumenta il tempo necessario per attraversare celle logiche adiacenti.
- **Soluzione:** Implementare periodicamente operazioni di compattazione del nastro.

5. Difficoltà nella gestione di backtracking:

- **Limitazione:** Algoritmi che richiedono backtracking e ripristino di stati precedenti sono difficili da implementare.
- **Impatto:** Problemi di ricerca non deterministica diventano più complessi.
- **Soluzione:** Utilizzare tecniche di "trail" o "journal" per registrare e riprodurre sequenze di operazioni invece di stati.

Strategie generali per superare queste limitazioni:

1. **Approccio incrementale/differenziale:**
   - Invece di memorizzare l'intero stato in ogni momento, memorizzare solo le differenze rispetto a uno stato base.
   - Questo riduce lo spazio richiesto per rappresentare stati simili.
2. **Elaborazione a flussi:**
   - Progettare algoritmi che elaborano i dati in modo sequenziale, minimizzando la necessità di riscritture.
   - Favorire algoritmi che fanno un singolo passaggio attraverso i dati.
3. **Strutture dati persistenti:**
   - Utilizzare strutture dati che preservano le versioni precedenti quando modificate.
   - Queste strutture si adattano naturalmente al vincolo di singola scrittura.
4. **Tecniche di lazy evaluation:**
   - Ritardare le computazioni fino a quando i risultati sono effettivamente necessari.
   - Questo può ridurre il numero totale di scritture richieste.

5. **Memorizzazione:**

   - Memorizzare i risultati di calcoli frequenti per evitare di doverli ricalcolare.
   - Particolarmente utile per funzioni pure che mappano gli stessi input agli stessi output.

In pratica, mentre la restrizione di singola scrittura impone sfide significative, molti problemi possono essere riformulati in modi che sono naturalmente compatibili con questa restrizione. L'adozione di paradigmi di programmazione che enfatizzano l'immutabilità (come la programmazione funzionale) può fornire modelli utili per la progettazione di algoritmi efficienti per TMSS.

2 Decidibilità di Problemi su Automi e Grammatiche

Esercizio 12. Sia $INFINITE_{PDA} = \{\langle P \rangle \mid P \text{ è un PDA ed } L(P) \text{ è un linguaggio infinito}\}$.

a) Dimostrate che $INFINITE_{PDA}$ è decidibile, fornendo un algoritmo dettagliato.

b) Descrivete come l'algoritmo utilizza la proprietà di pumping per i linguaggi context-free per decidere se un linguaggio è infinito.

c) Confrontate questo problema con il caso dei linguaggi regolari. Come si può decidere se un DFA riconosce un linguaggio infinito?

Soluzione. a) Decidibilità di $INFINITE_{PDA}$

Teorema 13. *Il problema $INFINITE_{PDA} = \{\langle P \rangle \mid P$ è un PDA ed $L(P)$ è un linguaggio infinito$\}$ è decidibile.*

Proof. Per dimostrare la decidibilità di $INFINITE_{PDA}$, forniremo un algoritmo che, dato un PDA P, determina se $L(P)$ è infinito o meno.

L'algoritmo sfrutta due proprietà fondamentali dei linguaggi context-free:

1. Un linguaggio context-free è infinito se e solo se contiene stringhe arbitrariamente lunghe.
2. Il lemma di pumping per i linguaggi context-free garantisce che ogni stringa sufficientemente lunga in un linguaggio context-free contiene una sottostringa che può essere "pompata" per generare infinite stringhe nel linguaggio.

Algoritmo:

1. **Conversione del PDA in una grammatica libera dal contesto (CFG):**

   - Dato un PDA P, convertiamo P in una grammatica libera dal contesto G tale che $L(P) = L(G)$.
   - Questa conversione è nota ed è sempre possibile (algoritmo standard della teoria degli automi).

2. **Eliminazione di produzioni inutili e non raggiungibili:**
   - Eliminiamo da G tutte le variabili che non producono stringhe terminali.
   - Eliminiamo anche tutte le variabili che non sono raggiungibili dal simbolo iniziale.
   - Questo passaggio garantisce che ogni variabile rimanente in G contribuisca effettivamente a generare stringhe nel linguaggio.
3. **Identificazione di cicli nella derivazione:**
   - Costruiamo il grafo di dipendenza delle variabili di G:
     - I nodi sono le variabili di G.
     - C'è un arco da A a B se esiste una produzione $A \to \alpha B \beta$ in G.
   - Cerchiamo cicli nel grafo di dipendenza utilizzando un algoritmo standard di rilevamento di cicli (come la ricerca in profondità).
4. **Analisi dei cicli:**
   - Per ogni ciclo trovato, esaminiamo le produzioni coinvolte.
   - Se esiste un ciclo $A \Rightarrow^+ \alpha A \beta$ dove almeno uno tra α e β non è la stringa vuota (cioè, la derivazione aumenta effettivamente la lunghezza), allora G può generare stringhe arbitrariamente lunghe.
   - In tal caso, $L(P)$ è infinito e l'algoritmo restituisce "SI".
5. **Analisi in assenza di cicli "produttivi":**
   - Se non vengono trovati cicli che aumentano la lunghezza, allora esiste una lunghezza massima per le stringhe generabili da G.
   - In questo caso, $L(P)$ è finito e l'algoritmo restituisce "NO".

La correttezza dell'algoritmo si basa sul fatto che un linguaggio context-free è infinito se e solo se la sua grammatica contiene un ciclo di derivazione che aumenta la lunghezza delle stringhe generate. Questo è una conseguenza diretta del lemma di pumping per i linguaggi context-free.

Per quanto riguarda la complessità temporale, la conversione da PDA a CFG può essere realizzata in tempo polinomiale rispetto alla dimensione del PDA. L'eliminazione delle produzioni inutili e il rilevamento dei cicli sono anch'essi operazioni con complessità polinomiale rispetto alla dimensione della grammatica. Pertanto, l'intero algoritmo ha complessità polinomiale rispetto alla dimensione dell'input $\langle P \rangle$. □

b) Utilizzo della proprietà di pumping

Il lemma di pumping per i linguaggi context-free è uno strumento fondamentale per comprendere la struttura di questi linguaggi e viene utilizzato implicitamente nell'algoritmo descritto.

Lemma di Pumping per i linguaggi context-free: Per ogni linguaggio context-free L, esiste una costante $p > 0$ (la "costante di pumping") tale che ogni stringa $z \in L$ con $|z| \geq p$ può essere decomposta come $z = uvwxy$ con le seguenti proprietà:

- $|vx| > 0$ (almeno una delle due sottostringhe v o x non è vuota)
- $|vwx| \leq p$ (la porzione centrale non è troppo lunga)
- Per ogni $i \geq 0$, $uv^i wx^i y \in L$ (possiamo "pompare" v e x ripetendole qualsiasi numero di volte, anche zero)

Come l'algoritmo sfrutta implicitamente la proprietà di pumping:

1. **Individuazione della struttura ricorsiva:**
   - I cicli nel grafo di dipendenza delle variabili corrispondono a strutture ricorsive nella grammatica.
   - Queste strutture ricorsive sono esattamente quelle che permettono di "pompare" parti delle stringhe, come descritto dal lemma di pumping.
2. **Analisi dell'effetto del pumping sulla lunghezza:**
   - Quando identifichiamo un ciclo $A \Rightarrow^+ \alpha A \beta$ dove α o β non sono vuoti, stiamo essenzialmente trovando un modo per applicare il pumping.
   - Per ogni derivazione che contiene A, possiamo applicare più volte la sostituzione ricorsiva, producendo stringhe sempre più lunghe.
3. **Connessione diretta con il pumping:**
   - Se consideriamo una derivazione $S \Rightarrow^* uAy \Rightarrow^* u\alpha A\beta y \Rightarrow^* u\alpha\alpha A\beta\beta y \Rightarrow^* \ldots$, possiamo vedere che questo corrisponde esattamente alla decomposizione $uvwxy$ del lemma di pumping.
   - In particolare, v corrisponde a α e x corrisponde a β nella notazione del lemma.

Illustrazione con un esempio:
Consideriamo la grammatica con le produzioni:

$$S \to aAb$$
$$A \to aAb \mid c$$

Questa grammatica genera il linguaggio $L = \{a^n c^n b^n \mid n \geq 1\}$.

Il grafo di dipendenza ha un ciclo da A a se stesso attraverso la produzione $A \to aAb$. Questa produzione aumenta la lunghezza della stringa (aggiunge 'a' e 'b'), quindi il linguaggio è infinito.

Secondo il lemma di pumping, possiamo decomporre una stringa lunga in L, ad esempio $z = a^p c^p b^p$ (per p sufficientemente grande), come $z = uvwxy$ dove v e x corrispondono rispettivamente a alcuni dei simboli 'a' e alcuni dei simboli 'b' generati attraverso il ciclo. Pompando v e x (cioè, aumentando il numero di iterazioni del ciclo), otteniamo stringhe più lunghe che sono ancora in L.

c) Confronto con i linguaggi regolari

Il problema di decidere se un linguaggio regolare (rappresentato da un DFA) è infinito è concettualmente simile, ma significativamente più semplice rispetto al caso dei linguaggi context-free.

Algoritmo per decidere se $L(A)$ è infinito, dove A è un DFA:

1. **Costruzione del grafo di raggiungibilità:**

   - Costruiamo un grafo G dove i nodi sono gli stati del DFA A e c'è un arco da uno stato q a uno stato r se esiste una transizione da q a r in A.

2. **Analisi del grafo:**

   - Identifichiamo tutti gli stati accessibili dallo stato iniziale e tutti gli stati co-accessibili (cioè, da cui è possibile raggiungere uno stato finale).
   - Cerchiamo cicli nel sottografo degli stati che sono sia accessibili che co-accessibili.

3. **Decisione:**

   - Se esiste almeno un ciclo nel sottografo degli stati accessibili e co-accessibili, allora $L(A)$ è infinito, e l'algoritmo restituisce "SI".
   - Altrimenti, $L(A)$ è finito, e l'algoritmo restituisce "NO".

Confronto tra i due problemi:

1. **Semplicità strutturale:**

   - Per i DFA, l'infinitezza dipende solo dalla presenza di cicli accessibili e co-accessibili, una proprietà del grafo di transizione che è facile da verificare.
   - Per i PDA, dobbiamo considerare non solo i cicli, ma anche l'interazione tra le transizioni di stato e le operazioni sullo stack.

2. **Complessità algoritmica:**

   - Per i DFA, l'algoritmo ha complessità temporale $O(|Q|^2)$, dove $|Q|$ è il numero di stati.
   - Per i PDA, la complessità è significativamente maggiore a causa della necessità di convertire il PDA in una grammatica e analizzare i cicli nelle produzioni.

3. **Fondamento teorico:**

   - Per i DFA, l'infinitezza è direttamente collegata alla presenza di cicli nel grafo di transizione.
   - Per i PDA, l'infinitezza è collegata alla presenza di cicli di derivazione che aumentano la lunghezza, che corrisponde alla proprietà di pumping.

4. **Semplicità di implementazione:**

   - L'algoritmo per i DFA può essere implementato direttamente utilizzando algoritmi standard di ricerca in grafo.
   - L'algoritmo per i PDA richiede implementazioni più complesse per la conversione da PDA a CFG e l'analisi delle produzioni.

In entrambi i casi, l'idea fondamentale è la stessa: un linguaggio è infinito se e solo se contiene cicli che permettono di generare stringhe arbitrariamente lunghe. La differenza principale sta nella complessità strutturale dei modelli coinvolti: i DFA hanno una struttura molto più semplice rispetto ai PDA, il che rende l'algoritmo per i linguaggi regolari significativamente più efficiente e diretto.

Esercizio 14. Dato un automa a pila (PDA), definiamo uno stato inutile come uno stato che non viene mai inserito in alcuna computazione che accetta una stringa di input. Sia $USELESS_{PDA} = \{\langle P \rangle \mid P \text{ è un PDA che ha almeno uno stato inutile}\}$.

a) Formalizzate la definizione di stato inutile in un PDA.

b) Dimostrate che $USELESS_{PDA}$ è decidibile, fornendo un algoritmo per determinare quali stati di un PDA sono inutili.

c) Discutete come l'algoritmo proposto potrebbe essere utilizzato per ottimizzare un PDA rimuovendo tutti gli stati inutili.

Soluzione. a) Formalizzazione della definizione di stato inutile in un PDA
Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un automa a pila (PDA), dove:

- Q è un insieme finito di stati
- Σ è l'alfabeto di input
- Γ è l'alfabeto dello stack
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathrm{P}(Q \times \Gamma^*)$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $Z_0 \in \Gamma$ è il simbolo iniziale dello stack
- $F \subseteq Q$ è l'insieme degli stati finali

Definizione 1 (Configurazione)**.** Una configurazione di un PDA P è una tripla (q, w, α) dove:

- $q \in Q$ è lo stato corrente
- $w \in \Sigma^*$ è l'input rimanente da leggere
- $\alpha \in \Gamma^*$ è il contenuto corrente dello stack (con il primo simbolo che rappresenta il top dello stack)

Definizione 2 (Relazione di mossa)**.** La relazione di mossa $\vdash$ è definita su coppie di configurazioni come segue. Per ogni $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$:
$(q, aw, Z\alpha) \vdash (q', w, \beta\alpha)$ se e solo se $(q', \beta) \in \delta(q, a, Z)$

Definizione 3 (Relazione di mossa estesa)**.** La relazione di mossa estesa $\vdash^*$ è la chiusura riflessiva e transitiva di $\vdash$.

Definizione 4 (Computazione accettante)**.** Una computazione accettante di P su input w è una sequenza di configurazioni $C_0, C_1, \ldots, C_n$ tali che:

- $C_0 = (q_0, w, Z_0)$ (la configurazione iniziale)
- $C_i \vdash C_{i+1}$ per ogni $0 \leq i < n$ (ogni configurazione porta alla successiva secondo la relazione di mossa)
- $C_n = (q_f, \varepsilon, \alpha)$ per qualche $q_f \in F$ e $\alpha \in \Gamma^*$ (la configurazione finale ha uno stato di accettazione e ha consumato tutto l'input)

Definizione 5 (Stato accessibile)**.** Uno stato $q \in Q$ è accessibile se esiste una stringa $w \in \Sigma^*$ e una sequenza di configurazioni $C_0, C_1, \ldots, C_n$ tali che:

- $C_0 = (q_0, w, Z_0)$
- $C_i \vdash C_{i+1}$ per ogni $0 \leq i < n$
- $C_n = (q, w', \alpha)$ per qualche $w' \in \Sigma^*$ e $\alpha \in \Gamma^*$

In altre parole, uno stato è accessibile se può essere raggiunto dalla configurazione iniziale attraverso una sequenza di mosse valide.

Definizione 6 (Stato co-accessibile)**.** Uno stato $q \in Q$ è co-accessibile se esiste una stringa $w \in \Sigma^*$ e una sequenza di configurazioni $C_0, C_1, \ldots, C_n$ tali che:

- $C_0 = (q, w, \alpha)$ per qualche $\alpha \in \Gamma^*$
- $C_i \vdash C_{i+1}$ per ogni $0 \leq i < n$
- $C_n = (q_f, \varepsilon, \beta)$ per qualche $q_f \in F$ e $\beta \in \Gamma^*$

In altre parole, uno stato è co-accessibile se da esso è possibile raggiungere uno stato finale consumando completamente qualche stringa di input.

Definizione 7 (Stato utile)**.** Uno stato $q \in Q$ è utile se è sia accessibile che co-accessibile, ovvero:

- q può essere raggiunto dalla configurazione iniziale
- Da q è possibile raggiungere uno stato finale con uno stack opportuno

Definizione 8 (Stato inutile)**.** Uno stato $q \in Q$ è inutile se non è utile, cioè se:

- q non è accessibile, oppure
- q non è co-accessibile

In altre parole, uno stato è inutile se non compare in alcuna computazione accettante del PDA.

In termini più intuitivi, uno stato è inutile se si verifica almeno una delle seguenti condizioni:

1. Non è possibile raggiungere lo stato partendo dallo stato iniziale (stato inaccessibile)
2. Non è possibile raggiungere uno stato finale partendo da questo stato (stato "trappola" o "morto")

b) Dimostrazione della decidibilità di $USELESS_{PDA}$

Teorema 15. *Il problema $USELESS_{PDA} = \{\langle P\rangle \mid P$ è un PDA che ha almeno uno stato inutile$\}$ è decidibile.*

Proof. Per dimostrare la decidibilità di $USELESS_{PDA}$, forniremo un algoritmo che, dato un PDA P, determina se P ha almeno uno stato inutile.

L'algoritmo si basa sulla caratterizzazione degli stati inutili come stati che non sono sia accessibili che co-accessibili. Pertanto, l'algoritmo calcola separatamente gli insiemi di stati accessibili e co-accessibili, e poi identifica gli stati inutili come quelli che non appartengono all'intersezione di questi due insiemi.

Algoritmo:

1. **Calcolo degli stati accessibili:**
   - Inizializziamo un insieme $A = \{q_0\}$ contenente solo lo stato iniziale.
   - Ripetiamo finché non vengono aggiunti nuovi stati ad A:
     - Per ogni stato $q \in A$, ogni simbolo $a \in \Sigma \cup \{\varepsilon\}$ e ogni simbolo di stack $Z \in \Gamma$:
       - Per ogni transizione $(q', \beta) \in \delta(q, a, Z)$, aggiungiamo q' ad A.
   - Al termine, A contiene tutti e soli gli stati accessibili.
2. **Calcolo degli stati co-accessibili:**
   - Questa è la parte più complessa, poiché la co-accessibilità dipende non solo dallo stato, ma anche dal contenuto dello stack.
   - Convertiamo il PDA P in una grammatica libera dal contesto G tale che $L(P) = L(G)$.
   - Eliminiamo da G tutte le variabili non generative (che non producono alcuna stringa terminale).
   - Identifichiamo gli stati q per cui esiste una variabile corrispondente in G che è generativa.
   - Questi sono gli stati co-accessibili, li memorizziamo nell'insieme C.
3. **Identificazione degli stati utili e inutili:**
   - Gli stati utili sono quelli nell'intersezione $U = A \cap C$.
   - Gli stati inutili sono quelli nel complemento $I = Q \setminus U$.
4. **Decisione:**
   - Se I è non vuoto, allora P ha almeno uno stato inutile, e l'algoritmo restituisce "SI".

- Altrimenti, P non ha stati inutili, e l'algoritmo restituisce "NO".

Correttezza:
La correttezza dell'algoritmo si basa su due affermazioni chiave:

1. L'insieme A calcolato nel passo 1 contiene esattamente gli stati accessibili.

2. L'insieme C calcolato nel passo 2 contiene esattamente gli stati co-accessibili.

La prima affermazione è abbastanza immediata dalla definizione di accessibilità. Per la seconda, la conversione da PDA a grammatica libera dal contesto preserva la struttura delle computazioni accettanti, e l'eliminazione delle variabili non generative corrisponde all'eliminazione degli stati da cui non è possibile raggiungere uno stato finale.

Complessità:
Il calcolo degli stati accessibili può essere realizzato in tempo polinomiale rispetto alla dimensione del PDA, utilizzando un algoritmo di ricerca in ampiezza o profondità.

La conversione da PDA a grammatica e l'eliminazione delle variabili non generative hanno anch'esse complessità polinomiale rispetto alla dimensione dell'input.

Pertanto, l'intero algoritmo ha complessità polinomiale rispetto alla dimensione dell'input $\langle P \rangle$. □

c) Ottimizzazione di un PDA rimuovendo stati inutili

L'algoritmo proposto per determinare gli stati inutili di un PDA può essere utilizzato come base per un processo di ottimizzazione che rimuove tutti gli stati inutili, producendo un PDA equivalente ma più efficiente.

Algoritmo di ottimizzazione:

1. **Identificazione degli stati utili:**

   - Utilizzare l'algoritmo descritto nel punto (b) per identificare l'insieme U degli stati utili.

2. **Costruzione del PDA ottimizzato:**

   - Dato il PDA originale $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, costruiamo un nuovo PDA $P' = (Q', \Sigma, \Gamma, \delta', q_0, Z_0, F')$ dove:
     - $Q' = U$ (solo gli stati utili)
     - $F' = F \cap U$ (solo gli stati finali che sono anche utili)
     - δ' è la restrizione di δ a $Q' \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$, ovvero manteniamo solo le transizioni che coinvolgono stati utili

Vantaggi dell'ottimizzazione:

1. **Riduzione della dimensione:**

   - Il PDA ottimizzato ha generalmente meno stati e transizioni, il che può ridurre la memoria richiesta per la sua rappresentazione.

2. **Miglioramento delle prestazioni:**

- Con meno stati e transizioni da considerare, l'esecuzione del PDA può essere più efficiente.
- In particolare, l'eliminazione degli stati inaccessibili evita di allocare risorse per stati che non verranno mai raggiunti.
- L'eliminazione degli stati "trappola" (non co-accessibili) previene computazioni che non porteranno mai all'accettazione.

3. **Semplificazione dell'analisi:**

   - Un PDA con meno stati è generalmente più facile da analizzare, debuggare e comprendere.
   - La rimozione degli stati inutili può evidenziare strutture importanti che erano nascoste nella versione non ottimizzata.

4. **Base per ulteriori ottimizzazioni:**

   - La rimozione degli stati inutili può essere un primo passo in un processo di ottimizzazione più ampio.
   - Ad esempio, dopo aver rimosso gli stati inutili, potrebbe essere più facile identificare stati equivalenti che possono essere fusi.

Preservazione della semantica:
È importante notare che il PDA ottimizzato P' riconosce esattamente lo stesso linguaggio del PDA originale P, cioè $L(P') = L(P)$. Questo è garantito dal fatto che gli stati rimossi non contribuiscono ad alcuna computazione accettante.

Estensioni dell'algoritmo:
L'algoritmo di ottimizzazione può essere esteso in vari modi:

1. **Eliminazione di transizioni inutili:**

   - Oltre agli stati inutili, possiamo identificare e rimuovere transizioni che non contribuiscono mai a computazioni accettanti.

2. **Riorganizzazione della funzione di transizione:**

   - Dopo la rimozione degli stati inutili, possiamo riorganizzare la funzione di transizione per migliorare l'efficienza delle ricerche.

3. **Minimizzazione del PDA:**

   - Sebbene non esista un algoritmo generale per la minimizzazione dei PDA (a differenza dei DFA), la rimozione degli stati inutili può essere combinata con altre tecniche euristiche per ridurre ulteriormente la dimensione del PDA.

Esempio pratico:
Consideriamo un PDA per il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ con alcuni stati inutili aggiunti artificialmente:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ (dove q_3, q_4, q_5 sono stati inutili)

- $F = \{q_2\}$

L'algoritmo identifica q_3 come inaccessibile (non può essere raggiunto da q_0), q_4 come accessibile ma non co-accessibile (una volta raggiunto, non può portare all'accettazione), e q_5 come sia inaccessibile che non co-accessibile.

Il PDA ottimizzato avrà:

- $Q' = \{q_0, q_1, q_2\}$
- $F' = \{q_2\}$

Questo PDA ottimizzato è più semplice da analizzare e più efficiente da eseguire, pur riconoscendo lo stesso linguaggio dell'originale.

Esercizio 16. Sia $REV_{DFA} = \{\langle A \rangle \mid A \text{ è un DFA che accetta } w^R \text{ in qualsiasi caso accetti } w\}$, dove w^R indica il reverse di w.

a) Dimostrate che REV_{DFA} è decidibile, fornendo un algoritmo dettagliato.

b) Caratterizzate la classe di linguaggi riconosciuti da DFA per cui vale questa proprietà.

c) Fornite un esempio non banale di un linguaggio regolare che appartiene a questa classe e uno che non vi appartiene.

Soluzione. a) Decidibilità di REV_{DFA}

Teorema 17. *Il problema $REV_{DFA} = \{\langle A \rangle \mid A$ è un DFA che accetta w^R in qualsiasi caso accetti $w\}$ è decidibile.*

Proof. Per dimostrare la decidibilità di REV_{DFA}, forniremo un algoritmo che, dato un DFA A, determina se A accetta w^R ogni volta che accetta w.

L'algoritmo si basa sull'osservazione che A accetta w^R ogni volta che accetta w se e solo se $L(A) = L(A)^R$, dove $L(A)^R = \{w^R \mid w \in L(A)\}$ è il linguaggio reverso di $L(A)$.

Algoritmo:

1. **Costruzione del DFA per il linguaggio reverso:**
   - Dato il DFA $A = (Q, \Sigma, \delta, q_0, F)$, costruiamo un NFA A_R che riconosce $L(A)^R$ come segue:
     - $A_R = (Q, \Sigma, \delta_R, F, \{q_0\})$ dove:
       * Gli stati iniziali di A_R sono gli stati finali di A
       * L'unico stato finale di A_R è lo stato iniziale di A
       * La funzione di transizione δ_R è definita come $\delta_R(q, a) = \{p \in Q \mid \delta(p, a) = q\}$ (cioè, le transizioni sono invertite)
   - Convertiamo l'NFA A_R in un DFA equivalente B utilizzando l'algoritmo di subset construction.
2. **Verificare l'equivalenza dei linguaggi:**

- Verifichiamo se $L(A) = L(B)$ utilizzando l'algoritmo standard per l'equivalenza di DFA:
  - Costruiamo il prodotto cartesiano degli stati di A e B
  - Verifichiamo se esiste una coppia (q, q') raggiungibile dalla coppia iniziale (q_0, q_0') tale che q è uno stato finale in A ma q' è uno stato finale in B o viceversa.
  - Se tale coppia esiste, allora $L(A) \neq L(B)$ e l'algoritmo restituisce "NO".
  - Se nessuna tale coppia esiste, allora $L(A) = L(B)$ e l'algoritmo restituisce "SI".

Correttezza:
La correttezza dell'algoritmo si basa sulle seguenti osservazioni:

1. A accetta w^R ogni volta che accetta w se e solo se $L(A) \subseteq L(A)^R$.
2. A accetta w ogni volta che accetta w^R se e solo se $L(A)^R \subseteq L(A)$.
3. Combinando queste due condizioni, $A \in REV_{DFA}$ se e solo se $L(A) = L(A)^R$.
4. L'NFA A_R costruito nel passo 1 riconosce esattamente $L(A)^R$.
5. Il DFA B ottenuto da A_R è equivalente a A_R, quindi $L(B) = L(A)^R$.
6. Pertanto, verificare se $L(A) = L(B)$ equivale a verificare se $L(A) = L(A)^R$, che è esattamente ciò che definisce REV_{DFA}.

Complessità:
L'algoritmo ha complessità polinomiale rispetto alla dimensione dell'input:

- La costruzione di A_R richiede tempo $O(|Q|^2 \cdot |\Sigma|)$.
- La conversione da NFA a DFA può richiedere tempo esponenziale rispetto al numero di stati dell'NFA, quindi $O(2^{|Q|})$ nel caso peggiore.
- La verifica dell'equivalenza dei DFA richiede tempo $O(|Q| \cdot 2^{|Q|})$.

Sebbene la complessità nel caso peggiore sia esponenziale, il problema rimane decidibile in quanto l'algoritmo termina sempre con la risposta corretta in tempo finito. □

b) Caratterizzazione della classe di linguaggi
I linguaggi riconosciuti da DFA per cui vale la proprietà di reversibilità (cioè, $L = L^R$) sono detti **linguaggi palindromi** o **linguaggi simmetrici**.

Teorema 18. *Un linguaggio regolare L soddisfa $L = L^R$ se e solo se può essere rappresentato come un'unione finita di linguaggi della forma xy^R, dove x e y sono stringhe finite (possibilmente vuote).*

Sketch della dimostrazione. ($\Rightarrow$) Se $L = L^R$, allora L può essere espresso come un'unione di linguaggi di base della forma xy^R.

La dimostrazione utilizza la rappresentazione algebrica dei linguaggi regolari come unione finita di linguaggi di base. Per ogni DFA che soddisfa $L(A) = L(A)^R$, possiamo decomporre il linguaggio in componenti elementari che preservano la proprietà di reversibilità.

($\Leftarrow$) Se L è un'unione finita di linguaggi della forma xy^R, allora $L = L^R$.

Questo segue dal fatto che $(xy^R)^R = yx^R = (x^R)^R(y^R)^R = xy^R$ quando x e y sono stringhe singole. Estendendo questa proprietà alle unioni finite, otteniamo che $L = L^R$. □

Proprietà dei linguaggi palindromi regolari:

1. **Chiusura sotto unione, intersezione e complemento:**
   - Se L_1 e L_2 sono linguaggi palindromi regolari, allora anche $L_1 \cup L_2$, $L_1 \cap L_2$ e $\overline{L_1}$ sono linguaggi palindromi regolari.
2. **Non chiusura sotto concatenazione:**
   - La concatenazione di due linguaggi palindromi regolari non è necessariamente un linguaggio palindromo regolare.
   - Ad esempio, $\{a\}$ e $\{a\}$ sono entrambi palindromi, ma $\{a\} \cdot \{a\} = \{aa\}$ non è palindromo perché $(aa)^R = aa$ ma $\{aa\}^R = \{aa\}$.
3. **Rappresentazione mediante espressioni regolari:**
   - I linguaggi palindromi regolari possono essere rappresentati mediante espressioni regolari specifiche che rispettano la proprietà di simmetria.
4. **Relazione con gli automi:**
   - Un DFA A riconosce un linguaggio palindromo se e solo se l'automa ottenuto invertendo le transizioni di A e scambiando stati iniziali e finali riconosce lo stesso linguaggio.

c) Esempi di linguaggi

Esempio di linguaggio regolare che appartiene alla classe:

Consideriamo il linguaggio $L_1 = \{w \in \{a, b\}^* \mid w$ ha un numero pari di a e un numero pari di $b\}$.

Questo linguaggio è chiaramente regolare, poiché può essere riconosciuto da un DFA con 4 stati che tiene traccia della parità del numero di a e b letti.

L_1 è anche un linguaggio palindromo, perché:

- Una stringa w ha un numero pari di a e un numero pari di b se e solo se w^R ha un numero pari di a e un numero pari di b.
- Questo è vero perché invertire l'ordine dei simboli in una stringa non cambia il conteggio totale di ciascun simbolo.
- Quindi, $w \in L_1$ se e solo se $w^R \in L_1$, il che implica $L_1 = L_1^R$.

Possiamo verificare questa proprietà costruendo esplicitamente un DFA per L_1 e verificando che soddisfa la condizione descritta nell'algoritmo.

Esempio di linguaggio regolare che non appartiene alla classe:

Consideriamo il linguaggio $L_2 = \{w \in \{a, b\}^* \mid w \text{ inizia con } a\}$.

Questo linguaggio è chiaramente regolare, poiché può essere riconosciuto da un DFA con 2 stati.

L_2 non è un linguaggio palindromo, perché:

- La stringa a appartiene a L_2 perché inizia con a.
- La stringa ab appartiene a L_2 perché inizia con a.
- Ma $(ab)^R = ba$ non appartiene a L_2 perché non inizia con a.
- Quindi, esiste almeno una stringa $w \in L_2$ tale che $w^R \notin L_2$, il che implica $L_2 \neq L_2^R$.

In generale, qualsiasi linguaggio regolare che dipende dalla posizione dei simboli (come quelli che richiedono che una certa proprietà valga all'inizio o alla fine della stringa) difficilmente sarà un linguaggio palindromo, a meno che la proprietà stessa non sia simmetrica.

3 Decidibilità di Algoritmi e Problemi Computazionali

Esercizio 19. Dato un algoritmo ricorsivo in pseudocodice, consideriamo il problema di determinare se l'algoritmo termina per ogni possibile input. Sia $TERM_{RC} = \{\langle A \rangle \mid A \text{ è un algoritmo ricorsivo che termina per ogni input}\}$.

a) Descrivete un insieme di condizioni sufficienti a garantire che un algoritmo ricorsivo termini per ogni input.

b) Date queste condizioni, progettate un algoritmo che analizza un algoritmo ricorsivo per verificare se soddisfa tali condizioni.

c) Discutete i limiti dell'approccio proposto, specificando tipi di algoritmi ricorsivi per cui il vostro metodo potrebbe non funzionare.

Soluzione. a) Condizioni sufficienti per la terminazione

La terminazione di un algoritmo ricorsivo è una proprietà indecidibile in generale (riducibile al problema della fermata). Tuttavia, possiamo identificare un insieme di condizioni sufficienti (ma non necessarie) che, se soddisfatte, garantiscono la terminazione dell'algoritmo per ogni input.

Condizioni sufficienti per la terminazione di algoritmi ricorsivi:

1. **Esistenza di casi base:**
    - L'algoritmo deve avere almeno un caso base, cioè una condizione sotto la quale la funzione restituisce un risultato senza ulteriori chiamate ricorsive.

- Questo caso base deve essere raggiungibile per ogni possibile input attraverso ripetute chiamate ricorsive.

2. **Funzione di misura decrescente:**
   - Deve esistere una funzione di misura $\mu : D \to \mathbb{N}$ che associa a ogni input un numero naturale.
   - Ad ogni chiamata ricorsiva, il valore di μ applicato al nuovo input deve essere strettamente minore del valore di μ applicato all'input corrente.
   - Formalmente, se $f(x)$ chiama ricorsivamente $f(y)$, allora $\mu(y) < \mu(x)$.
3. **Limite inferiore per la funzione di misura:**
   - Il codominio della funzione di misura deve avere un limite inferiore (tipicamente 0 per i numeri naturali).
   - Il caso base deve essere raggiunto quando la funzione di misura raggiunge questo limite inferiore.
4. **Non-alterazione delle variabili di controllo durante le chiamate ricorsive:**
   - Le variabili che controllano il flusso ricorsivo (cioè quelle utilizzate nella funzione di misura) non devono essere modificate in modo imprevedibile durante l'esecuzione.
   - In particolare, le variabili di controllo non devono essere modificate in modi che potrebbero interferire con la diminuzione monotona della funzione di misura.
5. **Numero finito di chiamate ricorsive per passo:**
   - A ogni passo, l'algoritmo deve effettuare un numero finito di chiamate ricorsive.
   - Questo garantisce che, anche se l'algoritmo utilizza ricorsione ramificata, la terminazione può ancora essere garantita se tutte le ramificazioni riducono la funzione di misura.

Queste condizioni costituiscono un'istanza di ciò che in teoria della computabilità e verifica dei programmi è noto come "ordinamento ben fondato" o "misura di terminazione". Se possiamo dimostrare che esiste un ordinamento ben fondato rispetto al quale gli input delle chiamate ricorsive sono sempre "più piccoli" dell'input originale, allora possiamo garantire la terminazione.

b) Algoritmo per verificare le condizioni di terminazione

Di seguito presentiamo un algoritmo che analizza un algoritmo ricorsivo in pseudocodice e verifica se soddisfa le condizioni sufficienti per la terminazione descritte sopra.

Esempi di applicazione dell'algoritmo:

Esempio 1: Fattoriale “‘ function factorial(n): if n <= 1: return 1 else: return n * factorial(n-1) “‘

- Caso base: ‘n <= 1‘
- Parametro di ricorsione: ‘n‘

Input: Un algoritmo ricorsivo A rappresentato come un albero sintattico astratto (AST) o in una forma simile analizzabile.
Output: "SI" se l'algoritmo soddisfa le condizioni sufficienti per la terminazione, "NO" altrimenti.

1. **Identificazione dei casi base:**
   - Analizzare il codice per identificare tutti i percorsi di esecuzione che non comportano chiamate ricorsive.
   - Verificare che esista almeno un tale percorso (caso base).
   - Se non viene trovato alcun caso base, restituire "NO".
2. **Identificazione dei parametri di ricorsione:**
   - Identificare i parametri della funzione che vengono modificati nelle chiamate ricorsive.
   - Questi sono i candidati per le variabili di controllo della ricorsione.
3. **Definizione di una possibile funzione di misura:**
   - Basandosi sui parametri di ricorsione identificati, proporre una o più possibili funzioni di misura μ.
   - Tipicamente, per algoritmi che operano su numeri naturali, μ potrebbe essere il valore stesso del parametro (per ricorsioni decrescenti) o la differenza tra il parametro e un valore limite (per ricorsioni crescenti).
   - Per algoritmi su strutture dati come liste o alberi, μ potrebbe essere la lunghezza della lista o l'altezza dell'albero.
4. **Verifica della monotonia decrescente:**
   - Per ogni chiamata ricorsiva individuata, verificare che $\mu(args_{nuovo}) < \mu(args_{corrente})$.
   - Questo può richiedere l'analisi statica del flusso del programma e/o l'utilizzo di tecniche di dimostrazione automatica.
   - Se viene trovata una chiamata ricorsiva che non diminuisce la funzione di misura, restituire "NO".
5. **Verifica della raggiungibilità dei casi base:**
   - Verificare che, per ogni possibile valore iniziale della funzione di misura, ripetute diminuzioni porteranno eventualmente a un valore per cui è definito un caso base.
   - In particolare, verificare che i casi base coprano tutti i possibili valori minimi della funzione di misura.
6. **Verifica dell'immutabilità delle variabili di controllo:**
   - Analizzare il codice per assicurarsi che le variabili utilizzate nella funzione di misura non vengano modificate in modi che potrebbero interferire con la diminuzione monotona.
   - Se viene trovata una tale modifica, restituire "NO".
7. **Verifica del numero finito di chiamate ricorsive:**

- Funzione di misura: $\mu(n) = n$
- Monotonia: $\mu(n-1) = n-1 < n = \mu(n)$
- Raggiungibilità del caso base: Per ogni $n > 1$, ripetute sottrazioni di 1 porteranno eventualmente a 1 o 0.
- Risultato: "SI"

Esempio 2: Algoritmo di Ackermann “‘ function ackermann(m, n): if m == 0: return n + 1 else if n == 0: return ackermann(m-1, 1) else: return ackermann(m-1, ackermann(m, n-1)) “‘

- Casi base: ‘m == 0‘
- Parametri di ricorsione: ‘m‘ e ‘n‘
- Funzione di misura possibile: $\mu(m, n) = (m, n)$ con ordine lessicografico
- Verifica: Per la chiamata ‘ackermann(m-1, ...)‘ abbiamo $(m-1, \_) <_{lex} (m, n)$, e per ‘ackermann(m, n-1)‘ abbiamo $(m, n-1) <_{lex} (m, n)$.
- Risultato: "SI"

c) Limiti dell’approccio proposto

L’approccio proposto ha diversi limiti significativi, che riflettono la difficoltà intrinseca del problema della terminazione.

1. Incompletezza:

- L’algoritmo verifica solo condizioni sufficienti, non necessarie, per la terminazione.
- Esistono algoritmi ricorsivi che terminano per ogni input ma non soddisfano le condizioni che verifichiamo.
- Questo è inevitabile a causa dell’indecidibilità del problema generale della terminazione (teorema di Rice).

2. Difficoltà nell’identificazione della funzione di misura:

- Per algoritmi complessi, può essere difficile o impossibile identificare automaticamente una funzione di misura appropriata.
- In alcuni casi, la funzione di misura potrebbe esistere ma essere troppo complessa per essere scoperta dal nostro algoritmo.

3. Limiti dell’analisi statica:

- L’analisi statica del codice può non essere in grado di determinare con certezza se una chiamata ricorsiva diminuisce effettivamente la funzione di misura.
- Questo è particolarmente vero per algoritmi che utilizzano chiamate ricorsive condizionali o indirette.

4. Ricorsione mutuale:

- L'algoritmo potrebbe non gestire efficacemente la ricorsione mutuale, dove più funzioni si chiamano reciprocamente.
- In tali casi, è necessaria un'analisi più sofisticata che consideri il grafo delle chiamate nel suo complesso.

5. Ricorsione non strutturale:

- L'algoritmo funziona meglio per la ricorsione strutturale (dove gli argomenti sono sottostrutture dirette dell'input originale).
- Per la ricorsione non strutturale, dove gli argomenti hanno relazioni più complesse con l'input originale, l'algoritmo potrebbe fallire.

6. Algoritmi con terminazione dipendente dai dati:

- Algoritmi la cui terminazione dipende da proprietà complesse dei dati di input (come la convergenza di una serie o la raggiungibilità in un grafo) potrebbero essere difficili da analizzare.

7. Algoritmi con ricorsione dinamica:

- Algoritmi che costruiscono dinamicamente le chiamate ricorsive (ad esempio, utilizzando puntatori a funzione o valutazione di espressioni) sfuggono all'analisi statica.

8. Algoritmi con correttezza parziale:

- Alcuni algoritmi sono progettati per terminare solo su un sottoinsieme del dominio di input possibile.
- Il nostro approccio potrebbe rifiutare tali algoritmi anche se sono corretti nel loro dominio previsto.

Esempi di algoritmi ricorsivi per cui il metodo potrebbe fallire:
1. Algoritmo di Collatz (congettura di Syracuse):

Listing 1: Algoritmo di Collatz (congettura di Syracuse)

```
function collatz(n):
    if n == 1:
        return 1
    else if n % 2 == 0:
        return 1 + collatz(n/2)
    else:
        return 1 + collatz(3*n + 1)
```

Per questo algoritmo, non è noto se termina per ogni input (è un problema aperto in matematica). La funzione di misura non è ovvia, e il nostro algoritmo probabilmente restituirebbe "NO" nonostante si congetturi che l'algoritmo termini per ogni input positivo.

2. Algoritmo con ricorsione basata su una funzione complessa:

Listing 2: Algoritmo con ricorsione basata su una funzione complessa

```
function complex_recursion(n):
    if n <= 0:
        return 0
    else:
        next_n = some_complex_function(n)
        return 1 + complex_recursion(next_n)
```

3. Algoritmo con ricorsione mutuale:

Listing 3: Algoritmo con ricorsion emutuale)

```
function even(n):
    if n == 0:
        return True
    else:
        return odd(n-1)

function odd(n):
    if n == 0:
        return False
    else:
        return even(n-1)
```

Questo algoritmo richiede un'analisi del sistema di funzioni nel suo complesso piuttosto che delle singole funzioni isolatamente.

In conclusione, sebbene l'approccio proposto possa identificare con successo molti algoritmi ricorsivi che terminano, è intrinsecamente limitato dall'indecidibilità del problema generale della terminazione. Per algoritmi complessi, potrebbero essere necessarie tecniche più sofisticate di analisi statica o addirittura dimostrazioni manuali di terminazione.

Esercizio 20. Consideriamo il problema di determinare se un grafo diretto aciclico (DAG) è un albero. Sia $TREE_{DAG} = \{\langle G \rangle \mid G \text{ è un DAG che è anche un albero}\}$.

a) Formalizzate la definizione di albero nel contesto dei grafi diretti.

b) Dimostrate che $TREE_{DAG}$ è decidibile, fornendo un algoritmo dettagliato.

c) Analizzate la complessità temporale e spaziale dell'algoritmo proposto.

Soluzione. a) Definizione formale di albero nel contesto dei grafi diretti

Nel contesto dei grafi diretti, un albero ha una definizione specifica che differisce leggermente dalla definizione classica per grafi non diretti.

Definizione 9 (Albero diretto)**.** Un grafo diretto $G = (V, E)$ è un **albero diretto** se soddisfa le seguenti condizioni:

1. G è un grafo diretto aciclico (DAG).
2. Esiste esattamente un nodo $r \in V$, chiamato **radice**, tale che non ha archi entranti (cioè, non esiste alcun nodo $v \in V$ tale che $(v, r) \in E$).

3. Ogni nodo $v \in V \setminus \{r\}$ ha esattamente un arco entrante (cioè, esiste esattamente un nodo $u \in V$ tale che $(u, v) \in E$).

4. Da ogni nodo $v \in V$ esiste un cammino diretto dalla radice r a v.

Questa definizione cattura le proprietà essenziali di un albero diretto:

- La proprietà 1 garantisce che non ci siano cicli.
- La proprietà 2 stabilisce l'esistenza di una radice unica.
- La proprietà 3 assicura che ogni nodo (tranne la radice) abbia esattamente un "genitore".
- La proprietà 4 garantisce che l'albero sia connesso, cioè che ogni nodo sia raggiungibile dalla radice.

Osservazioni importanti:

- In un albero diretto, la direzione degli archi va tipicamente dalla radice verso le foglie.
- Gli alberi diretti sono una sottoclasse dei DAG.
- Un albero diretto con n nodi ha esattamente $n - 1$ archi.
- Le condizioni 2, 3 e 4 insieme implicano che il grafo è debolmente connesso (cioè, il grafo non diretto sottostante è connesso).

b) Dimostrazione della decidibilità di $TREE_{DAG}$

Teorema 21. *Il problema $TREE_{DAG} = \{\langle G \rangle \mid G$ è un DAG che è anche un albero$\}$ è decidibile.*

Proof. Per dimostrare la decidibilità di $TREE_{DAG}$, forniremo un algoritmo che, dato un grafo diretto G, determina se G è un DAG che è anche un albero.

Algoritmo:

1. **Verifica che G sia un DAG:**
   - Utilizzare un algoritmo di rilevamento di cicli (come la ricerca in profondità con tracciamento delle visite) per verificare che G non contenga cicli.
   - Se G contiene cicli, restituire "NO".
2. **Identificazione dei candidati radice:**
   - Calcolare, per ogni nodo $v \in V$, il numero di archi entranti (grado entrante).
   - Identificare tutti i nodi con grado entrante 0 come candidati radice.
   - Se non ci sono candidati radice o ce ne sono più di uno, restituire "NO".
3. **Verifica del numero di archi entranti per nodi non radice:**

- Per ogni nodo $v \in V$ che non è la radice, verificare che abbia esattamente un arco entrante.
- Se viene trovato un nodo con zero o più di un arco entrante, restituire "NO".

4. **Verifica della raggiungibilità dalla radice:**
   - Eseguire una ricerca in ampiezza (BFS) o in profondità (DFS) a partire dalla radice.
   - Verificare che tutti i nodi del grafo siano visitati durante la ricerca.
   - Se ci sono nodi non visitati, restituire "NO".
5. **Verifica del conteggio degli archi:**
   - Verificare che il numero totale di archi sia esattamente $|V| - 1$.
   - Se il numero di archi è diverso, restituire "NO".
6. **Decisione finale:**
   - Se tutte le verifiche sopra hanno esito positivo, restituire "SI" (il grafo è un DAG che è anche un albero).
   - Altrimenti, restituire "NO".

Correttezza dell'algoritmo:
L'algoritmo verifica direttamente tutte le condizioni della definizione di albero diretto:

- Il passo 1 verifica che G sia un DAG.
- I passi 2 e 3 verificano che esista esattamente una radice e che ogni altro nodo abbia esattamente un arco entrante.
- Il passo 4 verifica che ogni nodo sia raggiungibile dalla radice.
- Il passo 5 è una verifica ridondante che può essere utile per l'implementazione pratica, poiché se un grafo è un albero diretto, deve avere esattamente $|V| - 1$ archi.

Se tutte queste condizioni sono soddisfatte, allora G è un DAG che è anche un albero diretto. Se una qualsiasi delle condizioni fallisce, allora G non è un albero diretto. □

c) Analisi della complessità
Complessità temporale:
Analizziamo la complessità temporale di ciascuna fase dell'algoritmo, assumendo che il grafo sia rappresentato mediante liste di adiacenza:

1. **Verifica che G sia un DAG:**
   - La rilevazione di cicli con DFS richiede $O(|V| + |E|)$ tempo.
2. **Identificazione dei candidati radice:**
   - Calcolare il grado entrante di tutti i nodi richiede $O(|V| + |E|)$ tempo.

- Identificare i nodi con grado entrante 0 richiede $O(|V|)$ tempo.

3. **Verifica del numero di archi entranti per nodi non radice:**
   - Questa verifica è già stata effettuata nel passo 2 e non richiede tempo aggiuntivo.
4. **Verifica della raggiungibilità dalla radice:**
   - BFS o DFS a partire dalla radice richiede $O(|V| + |E|)$ tempo.
5. **Verifica del conteggio degli archi:**
   - Contare gli archi richiede $O(|E|)$ tempo.

In totale, la complessità temporale dell'algoritmo è $O(|V| + |E|)$, che è lineare rispetto alla dimensione dell'input.

Complessità spaziale:
Per quanto riguarda la complessità spaziale:

1. **Rappresentazione del grafo:**
   - La rappresentazione del grafo mediante liste di adiacenza richiede $O(|V| + |E|)$ spazio.
2. **Strutture dati ausiliarie:**
   - Per la rilevazione di cicli e le ricerche BFS/DFS, sono necessari array di marcatura che richiedono $O(|V|)$ spazio.
   - Per calcolare i gradi entranti, è necessario un array che richiede $O(|V|)$ spazio.

In totale, la complessità spaziale dell'algoritmo è $O(|V| + |E|)$, che è lineare rispetto alla dimensione dell'input.

Ottimizzazioni possibili:
Alcune ottimizzazioni possono essere applicate per migliorare l'efficienza pratica dell'algoritmo:

1. **Verifiche combinate:**
   - Le verifiche della radice e dei gradi entranti possono essere combinate in una singola scansione del grafo.
2. **Terminazione anticipata:**
   - L'algoritmo può terminare non appena viene rilevata una violazione di una qualsiasi delle condizioni.
3. **Riduzione delle verifiche ridondanti:**
   - Se sappiamo che il grafo ha esattamente $|V| - 1$ archi e che ogni nodo non radice ha esattamente un arco entrante, la verifica della raggiungibilità dalla radice è automaticamente soddisfatta (assumendo che il grafo sia connesso).

In conclusione, l'algoritmo proposto è efficiente sia in termini di tempo che di spazio, con complessità lineare rispetto alla dimensione dell'input. Questo rende il problema $TREE_{DAG}$ non solo decidibile ma anche praticamente risolvibile per grafi di grandi dimensioni.

Esercizio 22. Dato un sistema di equazioni lineari $Ax = b$, consideriamo il problema di determinare se il sistema ha almeno una soluzione. Sia $SOLVABLE_{LE} = \{\langle A, b\rangle \mid A$ è una matrice e b è un vettore tali che il sistema $Ax = b$ ha almeno una soluzione$\}$.

a) Dimostrate che $SOLVABLE_{LE}$ è decidibile, descrivendo un algoritmo basato sull'eliminazione gaussiana.

b) Analizzate la complessità temporale dell'algoritmo proposto.

c) Estendete l'algoritmo per decidere se il sistema ha esattamente una soluzione, infinite soluzioni o nessuna soluzione.

Soluzione. a) Dimostrazione della decidibilità di $SOLVABLE_{LE}$

Teorema 23. *Il problema $SOLVABLE_{LE} = \{\langle A, b\rangle \mid A$ è una matrice e b è un vettore tali che il siste b ha almeno una soluzione$\}$ è decidibile.*

Proof. Per dimostrare la decidibilità di $SOLVABLE_{LE}$, forniremo un algoritmo che, data una matrice A e un vettore b, determina se il sistema di equazioni lineari $Ax = b$ ha almeno una soluzione.

L'algoritmo si basa sul metodo dell'eliminazione gaussiana, che trasforma il sistema in forma a scalini (o forma di Gauss-Jordan) per determinare se il sistema ha soluzioni.

Algoritmo di eliminazione gaussiana per decidere $SOLVABLE_{LE}$:

1. **Costruzione della matrice aumentata:**
   - Sia A una matrice $m \times n$ e b un vettore colonna di dimensione m.
   - Costruire la matrice aumentata $[A|b]$ di dimensione $m \times (n + 1)$ aggiungendo b come ultima colonna di A.
2. **Riduzione a forma a scalini mediante eliminazione gaussiana:**
   - Iniziare con la prima colonna e la prima riga.
   - Per ogni colonna j (da 1 a n):
     - Trovare la prima riga $i \geq j$ tale che $[A|b]_{i,j} \neq 0$.
     - Se tale riga non esiste, passare alla colonna successiva.
     - Altrimenti, scambiare la riga i con la riga j (se $i \neq j$).
     - Dividere tutti gli elementi della riga j per $[A|b]_{j,j}$ per ottenere un pivot uguale a 1.
     - Per ogni riga $k \neq j$, sottrarre $[A|b]_{k,j}$ volte la riga j dalla riga k per annullare tutti gli altri elementi nella colonna j.
3. **Verifica della consistenza del sistema:**

- Esaminare la forma a scalini della matrice aumentata.
- Se esiste una riga con tutti zeri nelle prime n colonne ma un valore non zero nella colonna $n+1$ (corrispondente a b), allora il sistema è inconsistente.
- In simboli, se esiste una riga i tale che $[A|b]_{i,j} = 0$ per ogni $j \in \{1, 2, \ldots, n\}$ ma $[A|b]_{i,n+1} \neq 0$, allora il sistema non ha soluzioni.

4. **Decisione:**
   - Se non viene rilevata alcuna inconsistenza nel passo 3, allora il sistema ha almeno una soluzione, e l'algoritmo restituisce "SI".
   - Altrimenti, il sistema non ha soluzioni, e l'algoritmo restituisce "NO".

Correttezza dell'algoritmo:
La correttezza dell'algoritmo si basa sul teorema fondamentale dell'algebra lineare che stabilisce che un sistema di equazioni lineari ha soluzioni se e solo se il rango della matrice dei coefficienti è uguale al rango della matrice aumentata.

L'eliminazione gaussiana trasforma la matrice aumentata in una forma equivalente (cioè, con lo stesso insieme di soluzioni) ma più facile da analizzare. In particolare, se dopo la riduzione a forma a scalini esiste una riga con tutti zeri nelle colonne corrispondenti alle variabili ma un valore non zero nella colonna dei termini noti, allora il sistema è inconsistente (cioè, non ha soluzioni).

Questo perché una tale riga corrisponderebbe a un'equazione della forma $0 = c$ con $c \neq 0$, che è chiaramente insoddisfacibile.

Pertanto, l'algoritmo determina correttamente se il sistema $Ax = b$ ha almeno una soluzione. □

b) Analisi della complessità temporale
Complessità temporale dell'algoritmo di eliminazione gaussiana:
L'analisi della complessità temporale dell'algoritmo proposto si concentra sul processo di eliminazione gaussiana, che è il passaggio computazionalmente più costoso.

Assumiamo che A sia una matrice $m \times n$ e b sia un vettore di dimensione m:

1. **Costruzione della matrice aumentata:**
   - La costruzione della matrice aumentata $[A|b]$ richiede $O(m)$ operazioni (aggiungere una colonna).

2. **Riduzione a forma a scalini:**
   - Per ogni colonna j (da 1 a n):
     - Trovare il pivot non nullo richiede $O(m)$ confronti nel caso peggiore.
     - Scambiare le righe (se necessario) richiede $O(n)$ operazioni.
     - Normalizzare la riga del pivot richiede $O(n)$ operazioni.
     - Eliminare i valori sotto il pivot richiede $O(m \cdot n)$ operazioni (per ogni riga $k \neq j$, dobbiamo modificare ogni elemento).
   - In totale, per ogni colonna, il costo è $O(m \cdot n)$.

- Poiché ci sono n colonne, il costo totale per questa fase è $O(n \cdot m \cdot n) = O(m \cdot n^2)$.

3. **Verifica della consistenza:**

   - La verifica della consistenza richiede $O(m \cdot n)$ operazioni nel caso peggiore.

La complessità temporale complessiva dell'algoritmo è quindi $O(m \cdot n^2)$, dominata dal costo della riduzione a forma a scalini.

Considerazioni aggiuntive:

- Se la matrice A è sparsa (cioè, contiene molti zeri), possono essere impiegate tecniche specializzate per accelerare l'eliminazione gaussiana.
- In pratica, l'implementazione può essere ottimizzata utilizzando algoritmi numericamente stabili come la fattorizzazione LU con pivoting parziale.
- La complessità teorica può essere ridotta utilizzando algoritmi più avanzati come l'algoritmo di Strassen per la moltiplicazione di matrici, ma questi algoritmi sono generalmente utili solo per matrici molto grandi a causa del loro overhead costante.

c) Estensione dell'algoritmo

Estendiamo l'algoritmo per determinare se il sistema $Ax = b$ ha esattamente una soluzione, infinite soluzioni o nessuna soluzione.

Algoritmo esteso:

1. **Riduzione a forma a scalini mediante eliminazione gaussiana:**

   - Applicare l'eliminazione gaussiana alla matrice aumentata $[A|b]$ come descritto in precedenza.

2. **Verifica della consistenza:**

   - Se esiste una riga con tutti zeri nelle prime n colonne ma un valore non zero nella colonna $n + 1$, allora il sistema è inconsistente (nessuna soluzione).

3. **Calcolo del rango:**

   - Calcolare il rango r della matrice A contando il numero di righe non nulle nella forma a scalini ridotta.

4. **Determinazione del tipo di soluzione:**

   - Se il sistema è consistente (passo 2):
     - Se $r = n$ (il rango è uguale al numero di variabili), allora il sistema ha esattamente una soluzione.
     - Se $r < n$ (il rango è minore del numero di variabili), allora il sistema ha infinite soluzioni.
   - Se il sistema è inconsistente (passo 2), allora il sistema non ha soluzioni.

Giustificazione dell'algoritmo esteso:
Il numero di soluzioni di un sistema lineare $Ax = b$ è determinato dal rango della matrice A e dalla consistenza del sistema:

- **Caso 1: Sistema inconsistente**
  - Se il rango della matrice aumentata $[A|b]$ è maggiore del rango di A, allora il sistema è inconsistente e non ha soluzioni.
  - Questo si verifica quando c'è una riga nella forma a scalini con tutti zeri nelle colonne di A ma un valore non zero nella colonna di b.
- **Caso 2: Sistema consistente con rango completo**
  - Se il sistema è consistente e il rango di A è uguale al numero di variabili ($r = n$), allora il sistema ha esattamente una soluzione.
  - In questo caso, ogni variabile è determinata in modo unico dalle equazioni.
- **Caso 3: Sistema consistente con rango non completo**
  - Se il sistema è consistente ma il rango di A è minore del numero di variabili ($r < n$), allora il sistema ha infinite soluzioni.
  - In questo caso, ci sono $n - r$ variabili libere che possono assumere qualsiasi valore, generando infinite soluzioni.

Esempio illustrativo:
Consideriamo il sistema $Ax = b$ con:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 5 & 7 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 12 \\ 15 \end{pmatrix}$$

Applicando l'eliminazione gaussiana alla matrice aumentata $[A|b]$, otteniamo:

$$[A|b] = \begin{pmatrix} 1 & 2 & 3 & 6 \\ 2 & 4 & 6 & 12 \\ 3 & 5 & 7 & 15 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & -2 & -3 \end{pmatrix}$$

Non ci sono righe con tutti zeri nelle prime 3 colonne ma un valore non zero nella colonna 4, quindi il sistema è consistente.

Il rango di A è 2 (ci sono 2 righe linearmente indipendenti), che è minore del numero di variabili (3). Pertanto, il sistema ha infinite soluzioni.

Complessità dell'algoritmo esteso:
La complessità temporale dell'algoritmo esteso rimane $O(m \cdot n^2)$, poiché i passaggi aggiuntivi (calcolo del rango e determinazione del tipo di soluzione) richiedono solo $O(m)$ operazioni, che sono dominate dal costo dell'eliminazione gaussiana.

In conclusione, l'algoritmo esteso non solo decide se un sistema di equazioni lineari ha soluzioni, ma fornisce anche informazioni complete sul numero di soluzioni (nessuna, esattamente una, o infinite), mantenendo la stessa complessità temporale dell'algoritmo originale.