Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Per provare le due affermazioni su DECIDERTM, useremo il teorema di Rice e una riduzione dal problema della fermata (HALTTM).

i. DECIDERTM non è Turing-riconoscibile (cioè, non è co-Turing-riconoscibile):

Il complemento di DECIDERTM è:

¬DECIDERTM = {<M> | M è una TM che non si ferma su almeno un input}

Supponiamo per assurdo che DECIDERTM sia Turing-riconoscibile. Allora esiste una TM R che riconosce DECIDERTM.

Possiamo usare R per costruire una TM S che riconosce ¬HALTTM come segue:

S = "Su input <M,w>:

1. Costruisci una TM M' che su input x:
   * Simula M su w.
   * Se M si ferma su w, entra in un loop infinito.
   * Se M non si ferma su w, si ferma.
2. Esegui R su <M'>.
3. Se R accetta, accetta. Altrimenti, rifiuta."

Se <M,w> ∈ ¬HALTTM, allora M non si ferma su w. Quindi M' si ferma su tutti gli input (perché simula M su w che non termina, e poi si ferma). Perciò <M'> ∈ DECIDERTM, e R accetta <M'>. Quindi S accetta <M,w>.

Se <M,w> ∉ ¬HALTTM, allora M si ferma su w. Quindi M' non si ferma su nessun input (perché simula M su w che termina, e poi entra in un loop infinito). Perciò <M'> ∉ DECIDERTM, e R non accetta <M'>. Quindi S rifiuta <M,w>.

Quindi S riconosce ¬HALTTM. Ma questo è impossibile perché sappiamo che ¬HALTTM non è Turing-riconoscibile (perché HALTTM non è decidibile). La contraddizione viene dall'aver assunto che DECIDERTM sia Turing-riconoscibile. Quindi DECIDERTM non può essere Turing-riconoscibile.

ii. DECIDERTM non è Turing-riconoscibile:

Supponiamo per assurdo che DECIDERTM sia Turing-riconoscibile. Allora esiste una TM R che riconosce DECIDERTM.

Possiamo usare R per costruire una TM S che decide HALTTM come segue:

S = "Su input <M,w>:

1. Costruisci una TM M' che su input x, ignora x e simula M su w.

2. Esegui R su <M'>.

3. Se R accetta, accetta. Altrimenti, rifiuta."

Se <M,w> ∈ HALTTM, allora M si ferma su w. Quindi M' si ferma su tutti gli input (ignorandoli sempre e simulando M su w che termina). Perciò <M'> ∈ DECIDERTM, e R accetta <M'>. Quindi S accetta <M,w>.

Se <M,w> ∉ HALTTM, allora M non si ferma su w. Quindi M' non si ferma su nessun input. Perciò <M'> ∉ DECIDERTM, e R non accetta <M'>. Quindi S rifiuta <M,w>.

Quindi S decide HALTTM. Ma questo è impossibile perché sappiamo che HALTTM è indecidibile. La contraddizione viene dall'aver assunto che DECIDERTM sia Turing-riconoscibile. Quindi DECIDERTM non può essere Turing-riconoscibile.



Per dimostrare che SREX è decidibile, dobbiamo fornire un algoritmo che, date due espressioni regolari R e S, determina se L(R) ⊆ L(S).

Ecco un possibile algoritmo:

1. Converti R e S in automi finiti non deterministici (NFA) equivalenti. Chiamiamoli NFA\_R e NFA\_S. Questo può essere fatto usando l'algoritmo standard di conversione da espressione regolare a NFA.

2. Converti NFA\_R in un automa finito deterministico (DFA) equivalente. Chiamiamolo DFA\_R. Questo può essere fatto usando l'algoritmo di subset construction.

3. Converti NFA\_S in un DFA equivalente. Chiamiamolo DFA\_S.

4. Costruisci un DFA, chiamiamolo DFA\_DIFF, che accetta il linguaggio L(DFA\_R) - L(DFA\_S), cioè tutte le stringhe che sono in L(DFA\_R) ma non in L(DFA\_S). Questo può essere fatto come segue:

- Gli stati di DFA\_DIFF sono coppie (q\_r, q\_s) dove q\_r è uno stato di DFA\_R e q\_s è uno stato di DFA\_S.

- Lo stato iniziale di DFA\_DIFF è (q\_r0, q\_s0) dove q\_r0 e q\_s0 sono gli stati iniziali di DFA\_R e DFA\_S rispettivamente.

- Gli stati finali di DFA\_DIFF sono tutte le coppie (q\_r, q\_s) dove q\_r è uno stato finale di DFA\_R e q\_s non è uno stato finale di DFA\_S.

- Per ogni transizione q\_r -a-> p\_r in DFA\_R e q\_s -a-> p\_s in DFA\_S, aggiungiamo una transizione (q\_r, q\_s) -a-> (p\_r, p\_s) in DFA\_DIFF.

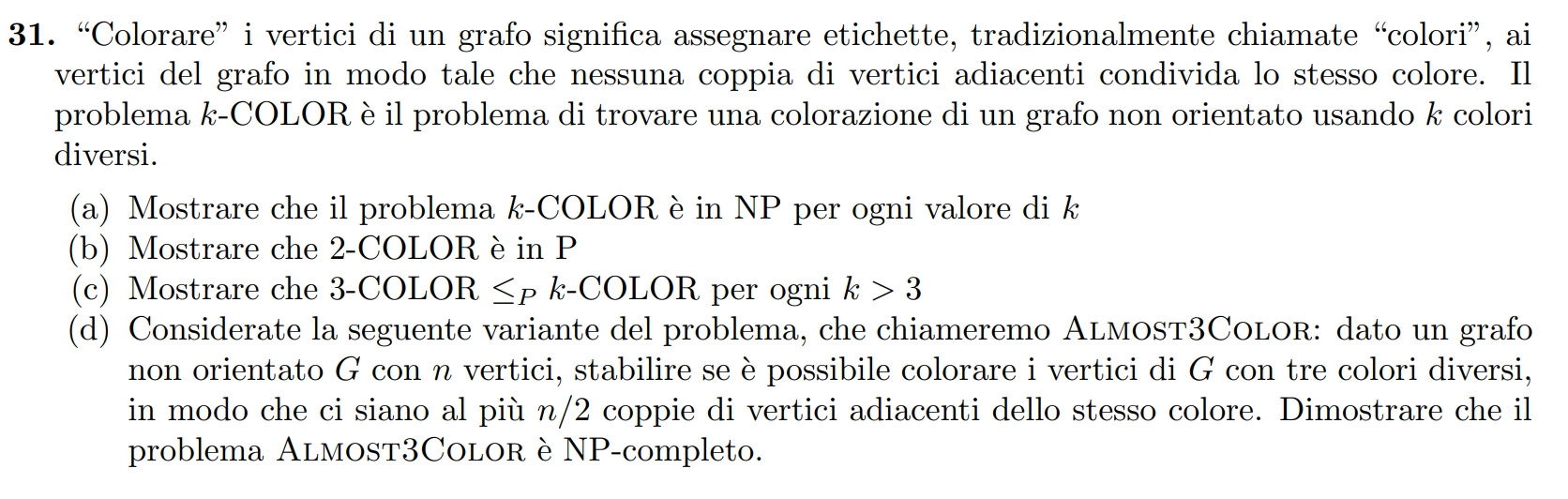
5. Controlla se il linguaggio accettato da DFA\_DIFF è vuoto. Questo può essere fatto controllando se DFA\_DIFF contiene stati finali raggiungibili dallo stato iniziale. Se il linguaggio è vuoto, allora L(R) ⊆ L(S), altrimenti no.

Perché funziona? L'idea chiave è che L(R) ⊆ L(S) se e solo se non esiste una stringa che è in L(R) ma non in L(S). DFA\_DIFF accetta esattamente queste stringhe. Quindi, se L(DFA\_DIFF) è vuoto, significa che non esistono tali stringhe, il che implica che L(R) ⊆ L(S).

Ogni passo di questo algoritmo può essere effettuato in tempo finito:

- Le conversioni da espressione regolare a NFA, da NFA a DFA, e la costruzione di DFA\_DIFF possono tutte essere fatte in un numero di passi proporzionale alla dimensione delle espressioni regolari.

- Controllare se il linguaggio di un DFA è vuoto può essere fatto in tempo lineare rispetto al numero di stati del DFA.

Quindi, questo algoritmo decide SREX, dimostrando che SREX è decidibile.

(a) Per mostrare che k-COLOR è in NP per ogni k:

- La soluzione (un assegnamento di colori) può essere verificata in tempo polinomiale controllando che vertici adiacenti abbiano colori diversi. Quindi k-COLOR è in NP.

(b) Per mostrare che 2-COLOR è in P:

- Esegui una ricerca in profondità (DFS) sul grafo.

- Assegna alternativamente due colori ai vertici man mano che vengono visitati dalla DFS.

- Se incontri un conflitto, il grafo non è 2-colorabile. Altrimenti, hai trovato una 2-colorazione valida.

- DFS richiede tempo O(V+E), quindi 2-COLOR è in P.

(c) Per mostrare che 3-COLOR ≤p k-COLOR per k>3:

- Data un'istanza di 3-COLOR (un grafo G), costruisci un'istanza identica per k-COLOR.

- Se il grafo è k-colorabile, è anche 3-colorabile (usando un sottoinsieme di k colori).

- Se è 3-colorabile, è banalmente anche k-colorabile.

- Questa riduzione richiede tempo O(1), quindi 3-COLOR ≤p k-COLOR.

(d) Per mostrare che ALMOST3COLOR è NP-completo:

- ALMOST3COLOR è chiaramente in NP (una 3-colorazione è verificabile in tempo polinomiale).

- Per mostrare la NP-completezza, riduciamo 3-COLOR ≤p ALMOST3COLOR.

- Data un'istanza di 3-COLOR (grafo G con n vertici), costruisci un grafo G' aggiungendo a G una clique di n/2 vertici, con ogni vertice della clique connesso ad ogni vertice originale di G.

- Se G è 3-colorabile, anche G' lo è (usando gli stessi colori di G più uno nuovo per la clique). Ci saranno esattamente n/2 spigoli monocromatici (nella clique).

- Se G' è almost-3-colorabile, rimuovendo la clique si ottiene una 3-colorazione propria per G (eventuali conflitti dovrebbero coinvolgere la clique per rispettare il bound di n/2).

- Questa riduzione richiede tempo polinomiale, quindi ALMOST3COLOR è NP-arduo. Essendo in NP, ALMOST3COLOR è NP-completo.

In sintesi, abbiamo mostrato che k-COLOR è in NP, 2-COLOR in P, 3-COLOR si riduce a k-COLOR, e che ALMOST3COLOR è NP-completo tramite una riduzione da 3-COLOR.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

a) Per dimostrare che entrambi i problemi SetPartitioning e SubsetSum sono in NP, dobbiamo mostrare che, data una soluzione candidata, è possibile verificare in tempo polinomiale se questa è effettivamente una soluzione valida.

Per SetPartitioning, una soluzione candidata consiste nella partizione di S in due sottoinsiemi disgiunti S1 e S2. Possiamo verificare in tempo polinomiale che la somma degli elementi di S1 sia uguale alla somma degli elementi di S2, calcolando le due somme e confrontandole.

Per SubsetSum, una soluzione candidata consiste in un sottoinsieme S' di S. Possiamo verificare in tempo polinomiale che la somma degli elementi di S' sia uguale a t, calcolando la somma e confrontandola con t.

Quindi, entrambi i problemi sono in NP.

b) Per dimostrare che SetPartitioning è NP-Hard usando SubsetSum come problema di riferimento, dobbiamo fornire una riduzione polinomiale da SubsetSum a SetPartitioning.

Data un'istanza <S, t> di SubsetSum, costruiamo un'istanza di SetPartitioning come segue: considerare l'insieme S' = S ∪ {t}. Ora, <S'> è un'istanza di SetPartitioning che ha soluzione se e solo se <S, t> ha soluzione per SubsetSum.

Se esiste un sottoinsieme S0 di S tale che la somma dei suoi elementi è t, allora S' può essere partizionato in S0 e S' \ S0, con la somma degli elementi in entrambi i sottoinsiemi uguale a t.

Viceversa, se S' può essere partizionato in due sottoinsiemi S1 e S2 con somma degli elementi uguale, allora t deve appartenere in uno dei due sottoinsiemi (altrimenti le somme non potrebbero essere uguali), e l'altro sottoinsieme rappresenta una soluzione per <S, t> in SubsetSum.

Questa riduzione può essere eseguita in tempo polinomiale, dimostrando che SetPartitioning è NP-Hard.

c) Per dimostrare che SubsetSum è NP-Hard usando SetPartitioning come problema di riferimento, dobbiamo fornire una riduzione polinomiale da SetPartitioning a SubsetSum.

Data un'istanza <S> di SetPartitioning, costruiamo un'istanza di SubsetSum come segue: consideriamo l'insieme S' = S e t = somma(S) / 2 (assumendo che somma(S) sia un numero pari, altrimenti il problema non ha soluzione). Ora, <S', t> è un'istanza di SubsetSum che ha soluzione se e solo se <S> ha soluzione per SetPartitioning.

Se S può essere partizionato in due sottoinsiemi S1 e S2 con somma degli elementi uguale, allora uno di questi sottoinsiemi (diciamo S1) rappresenta una soluzione per <S', t> in SubsetSum.

Viceversa, se esiste un sottoinsieme S0 di S' tale che la somma dei suoi elementi è t = somma(S) / 2, allora S0 e S' \ S0 formano una partizione di S con somma degli elementi uguale, risolvendo l'istanza di SetPartitioning.

Questa riduzione può essere eseguita in tempo polinomiale, dimostrando che SubsetSum è NP-Hard.

d) Per dimostrare che il problema QuasiPartitioning è NP-completo, dobbiamo provare che è in NP e che è NP-Hard.

QuasiPartitioning è in NP: data una soluzione candidata, cioè una partizione di S in due sottoinsiemi S1 e S2, possiamo verificare in tempo polinomiale che la somma degli elementi di S1 sia uguale alla somma degli elementi di S2 meno 1, calcolando le due somme e confrontandole.

Per dimostrare che QuasiPartitioning è NP-Hard, possiamo ridurre SetPartitioning a QuasiPartitioning in modo polinomiale.

Data un'istanza <S> di SetPartitioning, costruiamo un'istanza di QuasiPartitioning come segue: consideriamo l'insieme S' = S ∪ {1}. Ora, <S'> è un'istanza di QuasiPartitioning che ha soluzione se e solo se <S> ha soluzione per SetPartitioning.

Se S può essere partizionato in due sottoinsiemi S1 e S2 con somma degli elementi uguale, allora S1 e S2 ∪ {1} formano una partizione di S' con somma degli elementi che differisce di 1.

Viceversa, se S' può essere partizionato in due sottoinsiemi S1 e S2 con somma degli elementi che differisce di 1, allora l'elemento 1 deve appartenere in uno dei due sottoinsiemi (altrimenti le somme non potrebbero differire di 1), e l'altro sottoinsieme rappresenta una soluzione per <S> in SetPartitioning.

Questa riduzione può essere eseguita in tempo polinomiale, dimostrando che QuasiPartitioning è NP-Hard.

Poiché QuasiPartitioning è in NP e NP-Hard, ne consegue che è NP-completo.

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

(a) An example of a language that a 2-stack PDA can recognize, but no usual (single-stack) PDA can recognize, is the language L = {w c w^R | w is a string over {a, b}}, where w^R represents the reverse of the string w. This language consists of strings that have a middle section (represented by c) that separates a string w from its reverse w^R.

To recognize this language with a 2-stack PDA, we can use one stack to push the symbols of w while reading the input from left to right. After encountering the symbol c, we switch to the second stack and start popping symbols from the first stack and pushing them onto the second stack, effectively reversing the order of the symbols. We then continue reading the input from left to right and compare each symbol with the top of the second stack, popping and accepting only if they match. If we reach the end of the input and both stacks are empty, we accept the string.

(b) A 2-stack PDA can simulate a Turing machine. The idea is to use one stack to represent the non-blank portion of the Turing machine's tape (encoding both the symbols and the state information), and the other stack to store the remaining portion of the input. By carefully designing the transition function of the 2-stack PDA, we can mimic the behavior of the Turing machine, including reading and writing on the tape, moving the head, and transitioning between states.

(c) Yes, there is a language that a 3-stack PDA can recognize, but no 2-stack PDA can recognize. One such language is the language of palindromes over {a, b} with even-length middle segments delimited by c's. This language can be represented as L = {w c^(2n) c w^R | w is a string over {a, b}, n ≥ 1}.

To recognize this language with a 3-stack PDA, we can use one stack to push the symbols of w, and another stack to push the symbols of the middle segment c^(2n). After reading the second c, we reverse the contents of the first stack onto the third stack. We then compare the symbols popped from the third stack with the remaining input, accepting only if they match and the middle segment was of even length.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteHowever, no 2-stack PDA can recognize this language because it requires maintaining the count of the length of the middle segment (which could be arbitrarily large) while also storing the full string w and its reverse. A 2-stack PDA cannot keep track of all this information simultaneously, as it lacks the extra storage provided by the third stack.­­

a) Formuliamo questo problema come un linguaggio MUL\_TM. Sia Σ = {0, 1, #} l'alfabeto di input. Un'istanza di input valida per MUL\_TM è della forma x#y, dove x e y rappresentano due numeri binari. Il linguaggio MUL\_TM è definito come:

MUL\_TM = {x#y | x, y ∈ {0, 1}\* e x \* y = z in base binaria}

In altre parole, MUL\_TM è l'insieme di tutte le stringhe x#y tali che la moltiplicazione dei numeri binari rappresentati da x e y produce il numero binario z.

b) Per dimostrare che il linguaggio MUL\_TM è indecidibile, possiamo ridurre il problema della Macchina di Turing Universale (UTM) a MUL\_TM.

Il problema UTM è noto per essere indecidibile. UTM accetta una stringa w se e solo se la Macchina di Turing codificata da w si arresta su un input vuoto.

Possiamo costruire una funzione di riduzione f da UTM a MUL\_TM come segue:

1. Data una stringa w che rappresenta una Macchina di Turing, codifico w come un numero binario x.

2. Scelgo un numero binario y che rappresenta un'altra Macchina di Turing speciale M\_mul.

3. M\_mul è una Macchina di Turing che accetta se e solo se la Macchina di Turing codificata da x si arresta su un input vuoto.

4. La funzione di riduzione f(w) = x#y.

Se w ∈ UTM, allora la Macchina di Turing codificata da x si arresta su un input vuoto. Di conseguenza, M\_mul accetterà x, che significa che x \* y = z per qualche numero binario z. Quindi, f(w) = x#y ∈ MUL\_TM.

D'altra parte, se w ∉ UTM, allora la Macchina di Turing codificata da x non si arresta su un input vuoto. Di conseguenza, M\_mul non accetterà x, il che significa che non esiste alcun numero binario z tale che x \* y = z. Quindi, f(w) = x#y ∉ MUL\_TM.

Poiché la funzione di riduzione f è calcolabile in tempo polinomiale e UTM è indecidibile, ne consegue che MUL\_TM è indecidibile.

Quindi, abbiamo dimostrato che il linguaggio MUL\_TM, che rappresenta il problema della moltiplicazione binaria, è indecidibile.

Immagine che contiene testo, schermata, Carattere, informazione

Descrizione generata automaticamente

Una Turing machine con alfabeto ternario è una macchina di Turing deterministica a un singolo nastro dove l'alfabeto di input è Σ = {0, 1, 2} e l'alfabeto del nastro è Γ = {0, 1, 2, ⬜}. Questo tipo di macchina opera su stringhe di simboli sull'alfabeto ternario {0, 1, 2}, e i simboli non possono essere riscritti o cancellati, ma un simbolo ne marca i simboli sul nastro.

Per dimostrare che un linguaggio riconoscibile sull'alfabeto {0, 1, 2} può essere riconosciuto da una Turing machine con alfabeto ternario, possiamo utilizzare una codifica dei simboli dell'alfabeto ternario utilizzando una codifica binaria.

Specificamente, possiamo codificare:

0 → 00

1 → 01

2 → 11

Quindi, una stringa ternaria come "0122" può essere codificata come "000111001111" in binario.

Ora, una Turing machine con alfabeto binario può operare su questa codifica binaria seguendo le stesse regole di transizione della macchina ternaria originale, con la differenza che legge e scrive coppie di simboli binari invece di singoli simboli ternari.

Poiché una Turing machine con alfabeto binario può riconoscere qualsiasi linguaggio ricorsivamente enumerabile, e possiamo codificare le stringhe ternarie in stringhe binarie in modo efficiente, ne consegue che qualsiasi linguaggio riconoscibile da una Turing machine con alfabeto ternario può essere riconosciuto anche da una Turing machine con alfabeto binario.

Quindi, le Turing machine con alfabeto ternario non sono più potenti delle Turing machine con alfabeto binario standard, e non possono riconoscere alcun linguaggio aggiuntivo oltre a quelli riconoscibili dalle Turing machine binarie.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Una macchina di Turing ad albero binario usa un albero binario infinito come nastro, dove ogni cella nel nastro ha un figlio sinistro e un figlio destro. Ad ogni transizione, la testina si sposta dalla cella corrente al padre, al figlio sinistro oppure al figlio destro della cella corrente. Pertanto, la funzione di transizione di una tale macchina ha la forma:

δ: Q × Γ → Q × Γ × {P, L, R}

dove P indica lo spostamento verso il padre, L verso il figlio sinistro e R verso il figlio destro. La stringa di input viene fornita lungo il ramo sinistro dell'albero.

Per simulare una macchina di Turing ad albero binario con una macchina di Turing standard, possiamo codificare l'albero infinito su un nastro lineare infinito. Possiamo utilizzare una codifica in cui ogni cella del nastro lineare rappresenta una cella dell'albero, con informazioni aggiuntive per tenere traccia del percorso dalla radice.

Ecco come possiamo procedere:

1. Rappresentare ogni cella dell'albero come una tupla (simbolo, livello, posizione) sul nastro lineare.

2. La radice dell'albero è rappresentata come (simbolo\_radice, 0, 0).

3. Il figlio sinistro di una cella (s, l, p) è rappresentato come (s\_sinistro, l+1, 2p).

4. Il figlio destro di una cella (s, l, p) è rappresentato come (s\_destro, l+1, 2p+1).

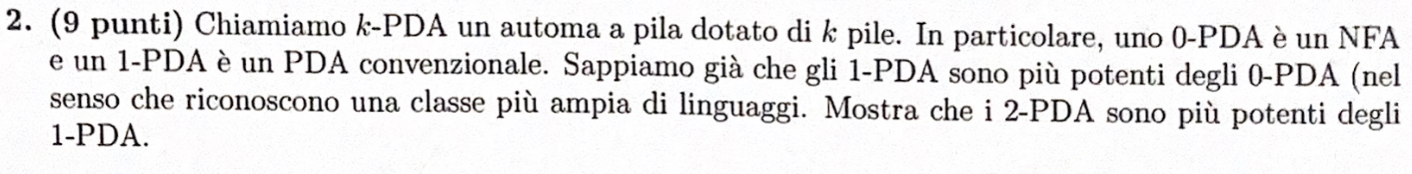
Con questa codifica, possiamo simulare i movimenti della testina dell'albero binario sul nastro lineare:

- Per spostarsi al padre, muovere la testina verso sinistra finché il livello non diminuisce di 1.

- Per spostarsi al figlio sinistro, muovere la testina verso destra fino a trovare la prima cella con livello maggiore e posizione pari.

- Per spostarsi al figlio destro, muovere la testina verso destra fino a trovare la prima cella con livello maggiore e posizione dispari.

Sebbene questa simulazione richieda spazio sul nastro esponenziale rispetto alla profondità dell'albero, dimostra che una macchina di Turing ad albero binario non è più potente di una macchina di Turing standard, poiché può essere simulata da quest'ultima.



Per dimostrare che i 2-PDA (automi a pila con 2 pile) sono più potenti degli 1-PDA (automi a pila con una sola pila), dobbiamo trovare un linguaggio che può essere riconosciuto da un 2-PDA, ma non da alcun 1-PDA. Un esempio di un tale linguaggio è:

L = {w c w^R | w è una stringa su {a, b}}

Dove w^R rappresenta la stringa inversa di w, e c è un simbolo speciale che separa w e w^R.

Questo linguaggio consiste di stringhe che hanno una parte centrale (rappresentata da c) che divide una stringa w dalla sua inversa w^R.

Possiamo costruire un 2-PDA per riconoscere L nel seguente modo:

1. Usando la prima pila, spingiamo i simboli di w man mano che li leggiamo da sinistra a destra.

2. Quando incontriamo il simbolo c, passiamo alla seconda pila.

3. Iniziamo a prelevare simboli dalla prima pila e a spingerli sulla seconda pila, effettivamente invertendo l'ordine dei simboli.

4. Continuiamo a leggere l'input da sinistra a destra e confrontiamo ogni simbolo con il simbolo in cima alla seconda pila, prelevando e accettando solo se corrispondono.

5. Se raggiungiamo la fine dell'input e entrambe le pile sono vuote, accettiamo la stringa.

Tuttavia, non esiste alcun 1-PDA che possa riconoscere questo linguaggio. Un 1-PDA non può memorizzare contemporaneamente la stringa w e la sua inversa w^R sulla singola pila, poiché avrebbe bisogno di invertire l'ordine dei simboli sulla pila per confrontarli con w^R.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteQuindi, abbiamo dimostrato che i 2-PDA sono più potenti degli 1-PDA, poiché possono riconoscere il linguaggio L, che non è riconoscibile da alcun 1-PDA.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente