Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteThe Post Correspondence Problem is to determine whether a collection of dominos has a match. This problem is unsolvable by algorithms.

The proof in book shows that the Post Correspondence Problem (PCP) is undecidable by reducing the acceptance problem for Turing machines (ATM) to PCP. The key steps are:

1. From any Turing machine M and input w, construct an instance P of PCP such that P has a match if and only if M accepts w. This establishes that if PCP were decidable, ATM would also be decidable.

2. The constructed PCP instance P simulates the computation of M on w:

- Each domino in P corresponds to a step in the computation

- A match of the dominos represents an accepting computation history of M on w

- The match forces the Turing machine simulation to occur by linking positions in one configuration to the corresponding ones in the next configuration

3. First construct an instance P' of a Modified PCP (MPCP) where the match must start with the first domino. This is done by carefully designing dominos corresponding to the initial config, transitions, copying tape symbols, halting, etc.

4. Then convert the MPCP instance P' to a standard PCP instance P by adding separator symbols (\*) around the original symbols. This forces the match to start with the first domino without explicitly requiring it.

5. If the resulting PCP instance P has a match, it corresponds to an accepting computation history of M on w. Conversely, if M accepts w, the constructed P will have a match mimicking that accepting computation.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamenteTherefore, since ATM is known to be undecidable, and we reduced it to PCP, PCP must also be undecidable. The key idea is designing the PCP dominos to force a simulation of the Turing machine's computation.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

The proof describes a reduction from the Domino problem (D1) to the Eulerian Path problem (EULER) in graph theory. The key steps are:

1. Define the Domino problem (D1):

- Input: A set B of domino tiles

- Question: Is there an alignment that uses all the tiles?

2. Reduce D1 to a problem on graphs (EULER) for which a polynomial-time algorithm is known to exist.

3. Define a graph:

- An undirected graph G is a pair (V, E), where:

- V = {v1, v2, ..., vn} is a finite, non-empty set of vertices

- E ⊆ {{u, v} | u, v ∈ V} is a set of unordered pairs, each corresponding to an edge of the graph

4. Construct the domino graph:

- Vertices: The numbers on the domino tiles, V = {①, ②, ③, ④}

- Edges: The domino tiles themselves, E = {①|②, ②|③, ③|④, ②|④, ④|①}

5. Define the Eulerian Path problem (EULER):

- Input: A graph G

- Question: Does G contain an Eulerian path (a path that traverses each edge exactly once)?

6. Note that EULER is a classic graph theory problem with known polynomial-time algorithms, such as Fleury's algorithm.

Fleury's Algorithm:

- Input: An undirected graph G

- Steps:

1. Choose a vertex with odd degree (or any vertex if all degrees are even)

2. Choose an edge such that its removal does not disconnect the graph. If no such edge exists, REJECT.

3. Move to the vertex at the other end of the chosen edge

4. Remove the edge from the graph

5. Repeat from step 2 until all edges are removed

6. If all edges have been removed, ACCEPT

- Complexity: On a graph with n edges, Fleury's algorithm runs in O(n^2) time

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteTherefore, the reduction shows that D1 ≤m EULER, and since EULER can be solved in polynomial time (e.g., by Fleury's algorithm), D1 can also be solved in polynomial time using this reduction.

Immagine che contiene diagramma, testo, linea, schizzo

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Il discorso è:

* il problema non è risolvibile
* è invece facile *verificare* se la soluzione è corretta

I problemi per i quali esiste un algoritmo polinomiale vengono considerati *trattabili*, mentre quelli che richiedono un algoritmo più che polinomiale sono detti *intrattabili*

Stabilire con precisione qual è il confine tra problemi trattabili ed intrattabili è piuttosto difficile

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteA verifier uses additional information to verify that a string w is a member of A. This information is called a *certificate*, or proof, of membership in A.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, linea, diagramma

Descrizione generata automaticamente

Per fornire un verificatore per PebbleDestruction, procediamo come segue:

1. L'input è una tripla (G, p, S) dove:
   * G = (V, E) è un grafo non orientato
   * p è una funzione che assegna un numero di ciottoli a ogni vertice
   * S è una sequenza di mosse
2. Per ogni mossa (v, u) in S:
   * a. Verifica che v abbia almeno 2 ciottoli (altrimenti rifiuta)
   * b. Rimuovi 2 ciottoli da v
   * c. Aggiungi 1 ciottolo a u (se u != v)
3. Dopo aver processato tutte le mosse, verifica che ogni vertice abbia 0 o 1 ciottoli
4. Immagine che contiene testo, schermata, Carattere, linea

   Descrizione generata automaticamenteSe tutte le verifiche hanno successo, accetta. Altrimenti, rifiuta.

*Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, documento

Descrizione generata automaticamenteImmagine che contiene Line art, disegno, schizzo, illustrazione

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, documento

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamenteImmagine che contiene testo, diagramma, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

Per dimostrare che 3-Color è un problema in NP, dobbiamo:

1. Definire cos'è un certificato per 3-Color

2. Definire un verificatore polinomiale per 3-Color

Un certificato per 3-Color è semplicemente un'assegnazione di 3 colori ai vertici del grafo dato, in modo che vertici adiacenti abbiano colori diversi. La dimensione del certificato è O(n), dove n è il numero di vertici, poiché specifica un colore per ogni vertice.

Un verificatore polinomiale per 3-Color, dato un grafo G e un certificato (assegnazione di colori), fa quanto segue:

1. Controlla che il certificato usi solo 3 colori. Questo richiede tempo O(n).

2. Per ogni coppia di vertici adiacenti in G, controlla che abbiano colori diversi nel certificato. Poiché G può avere al massimo O(n^2) coppie di vertici adiacenti, questo passaggio richiede tempo O(n^2).

Se entrambi i controlli vanno a buon fine, il verificatore accetta, altrimenti rifiuta. Il tempo totale è O(n^2), quindi polinomiale nella dimensione dell'input.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Per dimostrare che SubsetSum è un problema in NP, dobbiamo:

1. Definire cos'è un certificato per SubsetSum

2. Definire un verificatore polinomiale per SubsetSum

Un certificato per SubsetSum è un sottoinsieme S' dell'insieme dato S tale che la somma dei numeri in S' sia uguale al target t. La dimensione del certificato è O(n), dove n è la dimensione di S, poiché nel caso peggiore S' potrebbe includere tutti gli elementi di S.

Un verificatore polinomiale per SubsetSum, dato S, t, e un certificato S', fa quanto segue:

1. Controlla che S' sia un sottoinsieme di S. Questo può essere fatto in tempo O(n^2) confrontando ogni elemento di S' con ogni elemento di S.

2. Somma tutti i numeri in S' e controlla se la somma è uguale a t. Questo richiede tempo O(n).

Se entrambi i controlli vanno a buon fine, il verificatore accetta, altrimenti rifiuta. Il tempo totale è O(n^2), quindi polinomiale nella dimensione dell'input.

Nell'esempio dato, un certificato sarebbe S' = {4, 21}, e il verificatore controllerebbe:

1. {4, 21} è un sottoinsieme di {4, 11, 16, 21, 27}

2. 4 + 21 = 25, che è uguale al target t = 25

Quindi, il verificatore accetterebbe questo certificato.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamentePer dimostrare che un certo problema è NP-hard si procede tipicamente con una dimostrazione per riduzione polinomiale.

1. Start with a SAT instance φ in Conjunctive Normal Form (CNF), which is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals.

2. For each clause C in φ:

a. If C has 1, 2, or 3 literals, leave it unchanged.

b. If C has more than 3 literals, e.g., (x₁ ∨ x₂ ∨ ... ∨ xₙ) where n > 3, replace it with the following clauses:

- (x₁ ∨ x₂ ∨ y₁)

- (¬y₁ ∨ x₃ ∨ y₂)

- (¬y₂ ∨ x₄ ∨ y₃)

- ...

- (¬yₙ₋₃ ∨ xₙ₋₁ ∨ xₙ)

Here, y₁, y₂, ..., yₙ₋₃ are new variables introduced to split the clause.

3. The resulting formula φ' is an instance of 3SAT, as each clause now has at most 3 literals.

The reduction preserves satisfiability because:

- If φ is satisfiable, then φ' is also satisfiable. Any satisfying assignment for φ can be extended to a satisfying assignment for φ' by assigning appropriate values to the new variables.

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente- If φ' is satisfiable, then φ is also satisfiable. Any satisfying assignment for φ' can be used to construct a satisfying assignment for φ by simply ignoring the new variables.

To reduce the 3-Colorability Problem (3COLOR) to the 3-Satisfiability Problem (3SAT), we need to convert any given instance of 3COLOR into an equivalent instance of 3SAT. The main idea is to use boolean variables to represent the color assignments and create clauses that enforce the constraints of the 3-Colorability Problem.

1. Given a graph G = (V, E) as an instance of 3COLOR, where V is the set of vertices and E is the set of edges, we create a 3SAT formula φ as follows:

2. For each vertex v ∈ V, create three boolean variables: r\_v, g\_v, and b\_v, representing the three possible colors (red, green, and blue) that can be assigned to v.

3. Add the following clauses to φ for each vertex v ∈ V to ensure that each vertex is assigned at least one color:

(r\_v ∨ g\_v ∨ b\_v)

4. Add the following clauses to φ for each vertex v ∈ V to ensure that each vertex is not assigned more than one color:

(¬r\_v ∨ ¬g\_v)

(¬r\_v ∨ ¬b\_v)

(¬g\_v ∨ ¬b\_v)

5. For each edge (u, v) ∈ E, add the following clauses to φ to ensure that adjacent vertices are not assigned the same color:

(¬r\_u ∨ ¬r\_v)

(¬g\_u ∨ ¬g\_v)

(¬b\_u ∨ ¬b\_v)

6. The resulting formula φ is an instance of 3SAT, as each clause has at most 3 literals.

The reduction preserves satisfiability because:

- If G is 3-colorable, then φ is satisfiable. Any valid 3-coloring of G can be used to create a satisfying assignment for φ by setting the corresponding boolean variables (r\_v, g\_v, or b\_v) to true for each vertex v based on its assigned color.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente- If φ is satisfiable, then G is 3-colorable. Any satisfying assignment for φ can be used to construct a valid 3-coloring of G by assigning colors to vertices based on the truth values of the corresponding boolean variables.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, informazione

Descrizione generata automaticamente

Per dimostrare che sia il problema del circuito Toniano che quello del circuito quasi Hamiltoniano sono NP-completi, possiamo usare una riduzione dal problema del circuito Hamiltoniano che è noto essere NP-completo.

(a) Problema del circuito Toniano:

1. Il problema è chiaramente in NP: data una soluzione (un ciclo), è possibile verificare in tempo polinomiale che attraversa almeno metà dei vertici senza ripetizioni.

2. Riduzione dal problema del circuito Hamiltoniano: dato un grafo G=(V,E) di cui vogliamo sapere se contiene un circuito Hamiltoniano, costruiamo un nuovo grafo G' aggiungendo a G un numero di nuovi vertici scollegati pari a |V|. Così G' ha 2|V| vertici.

Se G ha un circuito Hamiltoniano, anche G' ce l'ha (è lo stesso ciclo) e quindi attraversa almeno |V| vertici che sono la metà di quelli di G'.

Viceversa, se G' ha un circuito Toniano che attraversa almeno metà dei suoi vertici (cioè |V|), deve per forza passare per tutti i vertici di G (altrimenti non arriverebbe a |V| vertici essendo quelli nuovi scollegati). Quindi questo circuito è un circuito Hamiltoniano in G.

(b) Problema del circuito quasi Hamiltoniano:

1. Anche questo problema è in NP perché, data una soluzione, la si può verificare in tempo polinomiale.

2. Riduzione dal problema del circuito Hamiltoniano: dato un grafo G=(V,E), costruiamo un nuovo grafo G' aggiungendo a G un nuovo vertice v collegato a tutti i vertici di G. Chiaramente se G ha un circuito Hamiltoniano, G' ha un circuito quasi Hamiltoniano (lo stesso ciclo di G che salta v). Se invece G' ha un circuito quasi Hamiltoniano, siccome v è collegato a tutti deve per forza essere v il vertice non visitato dal circuito, che quindi è un ciclo Hamiltoniano in G.

Queste riduzioni dimostrano che risolvere i problemi del circuito Toniano o quasi Hamiltoniano è almeno tanto difficile quanto risolvere il problema del circuito Hamiltoniano. Essendo quest'ultimo NP-completo, anche gli altri due problemi lo sono.

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

Per dimostrare che SetPartitioning e SubsetSum sono NP-completi, dobbiamo mostrare che sono in NP e che sono NP-hard. Procediamo step by step:

(a) Entrambi i problemi sono in NP:

SetPartitioning: data una soluzione (cioè una partizione S1, S2), possiamo verificare in tempo polinomiale che S1 e S2 sono disgiunti e che la somma dei loro elementi è uguale.

SubsetSum: data una soluzione (cioè un sottoinsieme S'), possiamo verificare in tempo polinomiale che S' è un sottoinsieme di S e che la somma dei suoi elementi è uguale a t.

(b) SetPartitioning è NP-hard usando SubsetSum:

Riduzione da SubsetSum a SetPartitioning:

Dato un'istanza <S, t> di SubsetSum, creiamo un'istanza di SetPartitioning S' = S ∪ {-t}, dove -t è l'intero negativo di t.

Se <S, t> è un'istanza yes di SubsetSum, allora esiste un sottoinsieme S1 ⊆ S tale che la somma dei suoi elementi è t. Poniamo S2 = S - S1 ∪ {-t}. Allora S1 e S2 formano una partizione di S' tale che la somma in entrambi i sottoinsiemi è zero (perché la somma in S1 è t e in S2 è -t più qualche altro intero). Quindi S' è un'istanza yes di SetPartitioning.

Viceversa, se S' è un'istanza yes di SetPartitioning, allora esiste una partizione S1, S2 di S' tale che le somme in S1 e S2 sono uguali. Uno dei due sottoinsiemi (diciamo S1) deve contenere -t, altrimenti le somme non possono essere uguali. Allora S1 - {-t} è un sottoinsieme di S la cui somma è t. Quindi <S, t> è un'istanza yes di SubsetSum.

(c) SubsetSum è NP-hard usando SetPartitioning:

Riduzione da SetPartitioning a SubsetSum:

Data un'istanza S di SetPartitioning, sia t la metà della somma di tutti gli elementi in S. Creiamo un'istanza <S, t> di SubsetSum.

Se S è un'istanza yes di SetPartitioning, allora esiste una partizione S1, S2 di S tale che le somme in S1 e S2 sono uguali. Questa somma comune deve necessariamente essere t (metà della somma totale). Quindi S1 (o S2) è un sottoinsieme di S la cui somma è t, e quindi <S, t> è un'istanza yes di SubsetSum.

Viceversa, se <S, t> è un'istanza yes di SubsetSum, allora esiste un sottoinsieme S1 ⊆ S tale che la somma dei suoi elementi è t. Sia S2 = S - S1. Allora S1 e S2 formano una partizione di S, e le somme in S1 e S2 sono entrambe t (perché la somma totale è 2t). Quindi S è un'istanza yes di SetPartitioning.

Queste riduzioni mostrano che SetPartitioning e SubsetSum sono entrambi NP-hard. Poiché abbiamo anche dimostrato che sono in NP, possiamo concludere che entrambi i problemi sono NP-completi.

Per dimostrare che J e il suo complemento J non sono Turing-riconoscibili, possiamo utilizzare il metodo di riduzione all'assurdo.

Prima di tutto, notiamo che J è l'unione di due insiemi:

- L1 = {0x | x ∈ ¬ATM}

- L2 = {1y | y ∈ ¬ATM}

Supponiamo per assurdo che J sia Turing-riconoscibile. Allora esiste una Macchina di Turing M che riconosce J.

Costruiamo una nuova Macchina di Turing M1 che riconosce ¬ATM nel seguente modo:

1. Su input x, M1 costruisce la stringa 0x.

2. M1 simula M su input 0x.

3. Se M accetta, allora M1 accetta. Altrimenti, M1 rifiuta.

Se x ∈ ¬ATM, allora 0x ∈ L1 ⊆ J, quindi M accetta 0x, e di conseguenza M1 accetta x.

Se x ∉ ¬ATM, allora 0x ∉ L1, e poiché inizia con 0, non può nemmeno essere in L2. Quindi 0x ∉ J, M rifiuta 0x, e M1 rifiuta x.

Quindi M1 riconosce ¬ATM. Ma sappiamo che ¬ATM non è Turing-riconoscibile (perché ATM non è Turing-decidibile). Questa è una contraddizione, generata dall'assunzione che J sia Turing-riconoscibile. Quindi J non può essere Turing-riconoscibile.

Ora supponiamo per assurdo che J sia Turing-riconoscibile. Allora esiste una Macchina di Turing M che riconosce J.

Costruiamo una nuova Macchina di Turing M2 che riconosce ¬ATM nel seguente modo:

1. Su input y, M2 costruisce la stringa 1y.

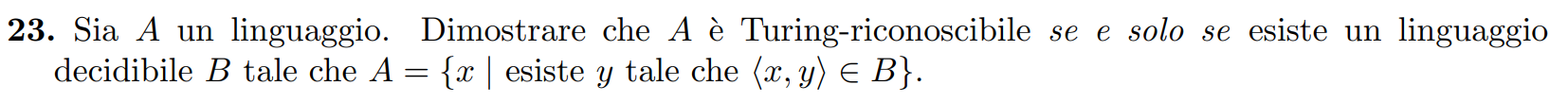
2. M2 simula M su input 1y.

3. Se M accetta, allora M2 accetta. Altrimenti, M2 rifiuta.

Se y ∈ ¬ATM, allora 1y ∈ L2 ⊆ J, quindi 1y ∉ J, M rifiuta 1y, e M2 accetta y.

Se y ∉ ¬ATM, allora 1y ∉ L2, e poiché inizia con 1, non può nemmeno essere in L1. Quindi 1y ∈ J, M accetta 1y, e M2 rifiuta y.

Quindi M2 riconosce ¬ATM, il che è impossibile. Questa contraddizione dimostra che J non può essere Turing-riconoscibile.

In conclusione, abbiamo dimostrato che sia J che J non sono Turing-riconoscibili.

Per dimostrare questa affermazione, dobbiamo provare due direzioni:

1. Se A è Turing-riconoscibile, allora esiste un linguaggio decidibile B tale che A = {x | esiste y tale che <x, y> ∈ B}.

2. Se esiste un linguaggio decidibile B tale che A = {x | esiste y tale che <x, y> ∈ B}, allora A è Turing-riconoscibile.

Direzione 1: Supponiamo che A sia Turing-riconoscibile. Allora esiste una Macchina di Turing M che riconosce A. Costruiamo un linguaggio B nel seguente modo:

B = {<x, y> | M accetta x in al più |y| passi}

Chiaramente, A = {x | esiste y tale che <x, y> ∈ B}, perché se x ∈ A, allora M accetta x in un certo numero di passi, e possiamo prendere y come una codifica binaria di quel numero di passi.

Inoltre, B è decidibile. Ecco una Macchina di Turing M' che decide B:

1. Su input <x, y>, M' simula M su x per al più |y| passi.

2. Se M accetta x entro questi passi, M' accetta <x, y>. Altrimenti, M' rifiuta <x, y>.

Questa simulazione richiede un numero di passi finito (al più |y|), quindi M' decide B.

Direzione 2: Supponiamo che esista un linguaggio decidibile B tale che A = {x | esiste y tale che <x, y> ∈ B}. Poiché B è decidibile, esiste una Macchina di Turing M che decide B. Costruiamo una Macchina di Turing M' che riconosce A:

1. Su input x, M' genera sistematicamente tutte le possibili stringhe y (in ordine di lunghezza crescente).

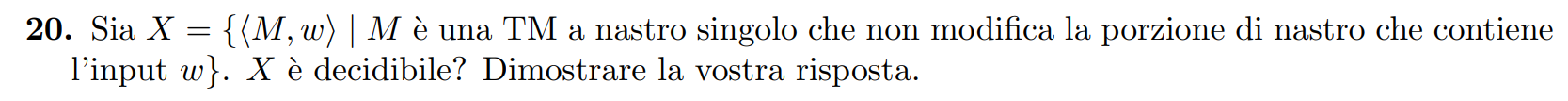
2. Per ogni y generata, M' costruisce la coppia <x, y> e simula M su <x, y>.

3. Se M accetta <x, y>, allora M' accetta x. Se M rifiuta <x, y>, M' passa alla prossima y.

Se x ∈ A, allora esiste una stringa y tale che <x, y> ∈ B, quindi M accetterà <x, y> quando M' la genera, e M' alla fine accetterà x.

Se x ∉ A, allora non esiste alcuna y tale che <x, y> ∈ B, quindi M rifiuterà tutte le coppie <x, y> generate da M', e M' non accetterà mai x.

Quindi M' riconosce A, il che significa che A è Turing-riconoscibile.

Queste due direzioni insieme dimostrano l'affermazione: A è Turing-riconoscibile se e solo se esiste un linguaggio decidibile B tale che A = {x | esiste y tale che <x, y> ∈ B}.

Per determinare se X è decidibile, cerchiamo di costruire una Macchina di Turing che decida X. Se non riusciamo a costruire una tale macchina, o se troviamo una riduzione da un problema indecidibile a X, allora possiamo concludere che X è indecidibile.

Costruiamo una Macchina di Turing M che tenta di decidere X:

1. Su input <M', w>, M simula M' su w.

2. Durante la simulazione, M confronta il contenuto del nastro nella porzione che conteneva inizialmente w con w stesso, ad ogni passo di simulazione.

3. Se in qualsiasi momento la porzione di nastro che conteneva inizialmente w differisce da w, M rifiuta.

4. Se M' si ferma senza aver modificato la porzione di nastro che conteneva inizialmente w, M accetta.

Se <M', w> ∈ X, allora M' non modifica la porzione di nastro che contiene w durante il suo calcolo. Quindi, M accetterà <M', w>.

Se <M', w> ∉ X, allora M' modifica la porzione di nastro che contiene w in qualche momento durante il suo calcolo. Quindi, M rifiuterà <M', w>.

Tuttavia, c'è un problema con questa costruzione. Se M' non si ferma su w, allora M non si fermerà mai, perché continuerà a simulare M' indefinitamente. Quindi, M non decide X, perché non si ferma su tutti gli input.

Infatti, X è indecidibile. Possiamo dimostrarlo riducendo il problema della fermata (HALT) a X. Ricordiamo che HALT = {<M, w> | M è una TM che si ferma su input w}, ed è noto essere indecidibile.

Ecco una riduzione da HALT a X:

- Data un'istanza <M, w> di HALT, costruiamo una nuova Macchina di Turing M' che, su input w, simula M su w. Se M si ferma, M' entra in un loop infinito senza modificare il nastro.

Se <M, w> ∈ HALT, allora M si ferma su w, quindi M' entrerà in un loop infinito senza modificare w. Quindi <M', w> ∈ X.

Se <M, w> ∉ HALT, allora M non si ferma su w, quindi M' non entrerà mai nel loop infinito e non modificherà w. Quindi <M', w> ∈ X.

Questa è una riduzione valida da HALT a X. Quindi, se X fosse decidibile, anche HALT lo sarebbe (perché potremmo decidere HALT decidendo X sulla Macchina di Turing costruita dalla riduzione). Ma sappiamo che HALT non è decidibile. Quindi, per contraddizione, X non può essere decidibile.

Immagine che contiene testo, schermata, Carattere, informazione

Descrizione generata automaticamenteIn conclusione, X è indecidibile.

(a) La funzione di transizione δ di una ETM è definita come:

δ: Q × (Γ × {F,B}) → Q × (Γ × {F,B}) × {L,R,F}

Dove:

- Q è l'insieme finito degli stati

- Γ è l'alfabeto del nastro che include il simbolo blank (▢)

- {F,B} indica il lato del nastro (F per fronte, B per retro)

- {L,R,F} indica il movimento della testina (L per sinistra, R per destra, F per girare sull'altro lato)

Quindi, δ(q,a,s) = (p,b,t,m) significa che se la ETM è nello stato q, legge il simbolo a sul lato s del nastro, allora passa nello stato p, scrive il simbolo b sul lato t del nastro, e muove la testina secondo m.

(b) Per mostrare che le ETM riconoscono esattamente i linguaggi Turing-riconoscibili, dobbiamo provare due direzioni:

1. Ogni linguaggio riconosciuto da una ETM è Turing-riconoscibile.

2. Ogni linguaggio Turing-riconoscibile è riconosciuto da una ETM.

Direzione 1: Data una ETM M, possiamo costruire una TM ordinaria M' che simula M. M' usa tre nastri:

- Il primo nastro simula il lato frontale del nastro di M.

- Il secondo nastro simula il lato posteriore del nastro di M.

- Il terzo nastro tiene traccia della posizione e del lato della testina di M.

Ad ogni passo, M' legge lo stato corrente di M dal suo stato, il simbolo dal nastro appropriato (primo o secondo) in base alla posizione e al lato della testina, applica la funzione di transizione di M, scrive il nuovo simbolo sul nastro appropriato, aggiorna lo stato e muove la sua testina sui tre nastri in base al movimento della testina di M.

M' accetta se e solo se M accetta. Quindi, ogni linguaggio riconosciuto da una ETM è Turing-riconoscibile.

Direzione 2: Data una TM ordinaria M, possiamo costruire una ETM M' che simula M. M' usa solo il lato frontale del suo nastro per simulare il nastro di M, e non usa mai il movimento F.

Ad ogni passo, M' legge il suo stato corrente e il simbolo sul lato frontale del nastro, applica la funzione di transizione di M, scrive il nuovo simbolo sul lato frontale e muove la sua testina a sinistra o a destra in base al movimento della testina di M.

M' accetta se e solo se M accetta. Quindi, ogni linguaggio Turing-riconoscibile è riconosciuto da una ETM.

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamenteQueste due direzioni insieme dimostrano che le ETM riconoscono esattamente la classe dei linguaggi Turing-riconoscibili.

(a) Per formulare il problema come un linguaggio, definiamo ENTHUSIASTICTM come:

ENTHUSIASTICTM = {<M> | M è una TM che su input vuoto non scrive mai "332" su tre celle adiacenti del suo nastro}

Dove <M> è la codifica della Macchina di Turing M.

(b) Per dimostrare che ENTHUSIASTICTM è indecidibile, possiamo ridurre da HALTTM, che sappiamo essere indecidibile.

Costruiamo una riduzione f da HALTTM a ENTHUSIASTICTM:

Per una data istanza <M, w> di HALTTM, definiamo f(<M, w>) = <M'>, dove M' è una TM che funziona come segue:

1. M' scrive w sul suo nastro.

2. M' simula M su w.

3. Se M si ferma, M' scrive "332" su tre celle adiacenti del nastro e poi si ferma.

Se <M, w> ∈ HALTTM, allora M si ferma su w, quindi M' scriverà "332" e <M'> ∉ ENTHUSIASTICTM.

Se <M, w> ∉ HALTTM, allora M non si ferma su w, quindi M' non scriverà mai "332" e <M'> ∈ ENTHUSIASTICTM.

Quindi HALTTM ≤m ENTHUSIASTICTM. Poiché HALTTM è indecidibile, anche ENTHUSIASTICTM deve essere indecidibile.

(c) Poiché ENTHUSIASTICTM è il complemento di {<M> | M è una TM che su input vuoto scrive "332" su tre celle adiacenti del suo nastro}, e abbiamo dimostrato che ENTHUSIASTICTM è indecidibile, segue dal Teorema di Rice (per ogni proprietà non banale delle funzioni parziali calcolabili, l'insieme delle Macchine di Turing che calcolano una funzione con quella proprietà è indecidibile) che ENTHUSIASTICTM non è Turing-riconoscibile.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente(d) ENTHUSIASTICTM non è Turing-riconoscibile, come dimostrato in (c).

(a) Per mostrare che OLTM non è Turing-riconoscibile usando una riduzione da ¬ATM, possiamo procedere come segue:

Dato un input <M, w> per ¬ATM, costruiamo una Macchina di Turing M' che funziona così:

- Su input x, M' simula M su w.

- Se M non accetta w, M' accetta x se |x| è dispari e rifiuta x se |x| è pari.

- Se M accetta w, M' rifiuta x.

Se <M,w> ∈ ¬ATM, allora M' accetta tutte e sole le stringhe di lunghezza dispari, quindi <M'> ∈ OLTM.

Se <M,w> ∉ ¬ATM, allora M' rifiuta tutte le stringhe, quindi <M'> ∉ OLTM.

Questa è una riduzione da ¬ATM a OLTM. Gli input sono istanze di ¬ATM, l'output è un'istanza di OLTM.

(b) La Macchina di Turing che computa la riduzione funziona come segue:

Su input <M, w>:

1. Costruisce la Macchina di Turing M' descritta in (a).

2. Genera in output <M'>, ovvero la codifica di M'.

Questa Macchina di Turing computa effettivamente la funzione di riduzione f(<M,w>) = <M'>.

(c) Se OLTM fosse Turing-riconoscibile, potremmo riconoscere ¬ATM nel seguente modo:

1. Dato un input <M,w> per ¬ATM, applichiamo la riduzione per ottenere un'istanza <M'> di OLTM.

2. Eseguiamo il riconoscitore per OLTM su <M'>.

3. Se il riconoscitore accetta <M'>, accettiamo <M,w>. Altrimenti, rifiutiamo <M,w>.

Questo funzionerebbe perché, per la proprietà della riduzione, <M,w> ∈ ¬ATM se e solo se <M'> ∈ OLTM.

Ma sappiamo che ¬ATM non è Turing-riconoscibile (perché ATM non è decidibile). Quindi OLTM non può essere Turing-riconoscibile, altrimenti potremmo riconoscere ¬ATM.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente(d) La riduzione da ¬ATM a OLTM descritta sopra dimostra che OLTM non è Turing-riconoscibile (cioè, non è co-Turing-riconoscibile).

Possiamo formulare il problema AGREEDFA come il linguaggio:

AGREEDFA = {<A, B, w> | A e B sono DFA che entrambi accettano la stringa w}

ii. Per mostrare che AGREEDFA è decidibile, possiamo usare il decider per EQDFA (l'equivalenza di DFA) dal Capitolo 4.

Data un'istanza <A, B, w> di AGREEDFA, costruiamo due nuovi DFA A' e B' come segue:

A' simula A su w. Se A accetta w, A' accetta tutte le stringhe. Altrimenti, A' rifiuta tutte le stringhe.

B' simula B su w. Se B accetta w, B' accetta tutte le stringhe. Altrimenti, B' rifiuta tutte le stringhe.

Ora, <A, B, w> ∈ AGREEDFA se e solo se A' e B' sono equivalenti (cioè, accettano esattamente le stesse stringhe).

Quindi, possiamo decidere AGREEDFA eseguendo il decisore per EQDFA su <A', B'>.

iii. Per mostrare che AGREEDFA è decidibile testando direttamente i DFA su stringhe fino a una certa lunghezza:

Sia n la somma degli stati in A e B. Qualsiasi stringa di lunghezza n o superiore che è accettata da A o B deve fare entrare A o B in un loop (per il pumping lemma).

Quindi, se esiste una stringa accettata sia da A che da B, ce ne deve essere una di lunghezza inferiore a n.

Possiamo quindi decidere AGREEDFA generando tutte le stringhe fino alla lunghezza n-1 e testandole su entrambi i DFA. Se troviamo una stringa accettata da entrambi, accettiamo. Altrimenti, rifiutiamo.

(b)

i. Possiamo formulare REVDFA come:

REVDFA = {<A, B> | A e B sono DFA e L(A) = L(B)^R}

dove L(A) è il linguaggio accettato da A e L(B)^R è il linguaggio formato invertendo ogni stringa in L(B).

Per mostrare che REVDFA è decidibile, data un'istanza <A,B>, costruiamo un DFA B' che inverte ogni stringa accettata da B (scambiando stati finali e iniziali e invertendo le transizioni). Quindi eseguiamo il decider per EQDFA su <A, B'>.

ii. Se provassimo a decidere REVPDA analogamente, usando un decider per l'equivalenza di PDA invece di EQDFA, falliremmo perché non esiste un tale decider. L'equivalenza di PDA è indecidibile.

Intuitivamente, invertire un PDA è molto più complicato che invertire un DFA a causa dello stack. Non possiamo semplicemente scambiare stati finali/iniziali e invertire le transizioni; dovremmo anche invertire in qualche modo le operazioni di stack, che è impossibile in generale.