Preparazione esame - Seconda parte

Decidibilità, Macchine di Turing e Complessità Computazionale

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

Aprile 2025

1 Macchine di Turing e varianti

14 1. Un automa a coda è simile ad un automa a pila con la differenza che la pila viene sostituita da una coda. Una coda è un nastro che permette di scrivere solo all'estremità sinistra del nastro e di leggere solo all'estremità destra. Ogni operazione di scrittura (push) aggiunge un simbolo all'estremità sinistra della coda e ogni operazione di lettura (pull) legge e rimuove un simbolo all'estremità destra. Come per un PDA, l'input è posizionato su un nastro a sola lettura separato, e la testina sul nastro di lettura può muoversi solo da sinistra a destra. Il nastro di input contiene una cella con un blank che segue l'input, in modo da poter rilevare la fine dell'input. Un automa a coda accetta l'input entrando in un particolare stato di accettazione in qualsiasi momento. Mostra che un linguaggio può essere riconosciuto da un automa deterministico a coda se e solo se è Turing-riconoscibile.

Soluzione. Dimostriamo l'equivalenza tra gli automi deterministici a coda e le macchine di Turing in termini di potenza computazionale, mostrando entrambe le direzioni.

Direzione $\Rightarrow$: Un linguaggio riconosciuto da un automa deterministico a coda è Turing-riconoscibile.

Questa direzione è immediata: possiamo simulare un automa a coda con una macchina di Turing standard usando due porzioni del nastro, una per memorizzare l'input e una per simulare la coda. La macchina di Turing può facilmente implementare le operazioni di push e pull richieste dall'automa a coda, utilizzando marker speciali per indicare le estremità della coda. L'operazione di push consiste nel muovere tutti i simboli della coda di una posizione a destra e inserire il nuovo simbolo nella prima posizione. L'operazione di pull consiste nel leggere l'ultimo simbolo e rimuoverlo. Pertanto, se un linguaggio è riconosciuto da un automa deterministico a coda, è anche Turing-riconoscibile.

Direzione $\Leftarrow$: Un linguaggio Turing-riconoscibile può essere riconosciuto da un automa deterministico a coda.

Per questa direzione, mostreremo come un automa a coda può simulare una macchina di Turing. L'idea chiave è che possiamo codificare la configurazione completa di una macchina di Turing nella coda dell'automa e aggiornare questa configurazione passo passo.

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una macchina di Turing che riconosce un linguaggio L. Costruiamo un automa a coda D che riconosce lo stesso linguaggio.

La codifica di una configurazione di M nella coda di D includerà:

- I contenuti del nastro di M
- La posizione della testina di M
- Lo stato corrente di M

Il funzionamento di D sarà il seguente:

1. D inizia leggendo l'input w e inizializza la sua coda con la configurazione iniziale di M su input w, ovvero q_0w.
2. Successivamente, D itera il seguente processo:
   (a) Legge l'intera coda, simbolo per simbolo, fino a quando non ha letto l'intera configurazione attuale.
   (b) Basandosi sulla configurazione letta, D calcola la prossima configurazione di M secondo la funzione di transizione δ.
   (c) Scrive la nuova configurazione nella coda.
   (d) Se la nuova configurazione contiene lo stato q_{accept}, D accetta.
   (e) Se la nuova configurazione contiene lo stato q_{reject}, D continua indefinitamente senza accettare.

Per gestire il nastro potenzialmente infinito di M, D può utilizzare un simbolo speciale # per marcare i confini del nastro attualmente in uso e estendere questi confini quando necessario.

In questo modo, D simula M passo per passo. Se M accetta w, D alla fine raggiungerà una configurazione in cui M è nello stato q_{accept} e quindi accetterà. Se M non accetta w (sia che rifiuti o entri in un loop), D continuerà la simulazione indefinitamente.

Questo dimostra che ogni linguaggio Turing-riconoscibile può essere riconosciuto da un automa deterministico a coda.

Combinando entrambe le direzioni, abbiamo dimostrato che un linguaggio può essere riconosciuto da un automa deterministico a coda se e solo se è Turing-riconoscibile.

15 2. Una Macchina di Turing a sola 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). Mostrare che questa variante di macchina di Turing è equivalente alla macchina di Turing standard.

Soluzione. Dimostriamo che una Macchina di Turing a sola scrittura (WOTM - Write-Once Turing Machine) è equivalente a una Macchina di Turing standard (TM) in termini di potenza computazionale.

Direzione $\Rightarrow$: Una WOTM può essere simulata da una TM standard.

Questa direzione è ovvia: una TM standard può facilmente simulare una WOTM semplicemente rispettando la restrizione di modificare ogni cella al più una volta. Quindi, ogni linguaggio riconosciuto da una WOTM è anche riconosciuto da una TM standard.

Direzione $\Leftarrow$: Una TM standard può essere simulata da una WOTM.

Per questa direzione, dobbiamo mostrare come una WOTM può simulare una TM standard nonostante la limitazione di poter modificare ogni cella al più una volta. L'idea chiave è utilizzare una codifica che permetta alla WOTM di "modificare" virtualmente una cella più volte, anche se fisicamente ogni cella viene scritta al più una volta.

Costruiamo una WOTM M' che simula una TM standard $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ come segue:

1. M' utilizzerà un nastro che è concettualmente diviso in tracce multiple. Inizialmente, l'input sarà nella prima traccia.

2. M' inizia copiando l'input dalla prima traccia alla seconda traccia, lasciando la prima traccia intatta. Questo è necessario perché l'input potrebbe essere modificato durante la computazione.

3. La computazione principale avverrà sulla traccia attiva corrente (inizialmente la seconda traccia). Per ogni cella sulla traccia attiva, M' mantiene anche un puntatore alla prossima traccia da utilizzare se questa cella deve essere modificata.

4. Quando M vuole modificare una cella che M' ha già scritto, M' crea una nuova istanza della configurazione corrente nella traccia successiva, con la modifica appropriata alla cella in questione.

5. Per gestire questo, M' manterrà un registro di quali celle sono già state modificate e su quale traccia si trova la versione più recente di ciascuna cella.

Per implementare questa strategia, M' utilizzerà un alfabeto esteso che include, per ogni simbolo $\gamma \in \Gamma$, una tripla (γ, t, p) dove: - γ è il simbolo attuale nella cella - t è un indice che indica la traccia attiva corrente per questa cella - p è un puntatore alla prossima traccia da utilizzare se questa cella deve essere modificata

Quando M' deve modificare una cella che è già stata scritta, invece di sovrascriverla (cosa che non può fare), M' crea una nuova versione della configurazione corrente nella traccia indicata dal puntatore p, aggiorna i puntatori pertinenti, e continua la simulazione sulla nuova traccia.

Questo approccio garantisce che M' modifichi ogni cella al più una volta, pur permettendo di simulare multiple scritture sulla stessa cella da parte di M.

L'accettazione e il rifiuto seguono lo stesso comportamento di M: M' accetta se e solo se M accetta, e similmente per il rifiuto.

Anche se questa simulazione è meno efficiente rispetto a una TM standard (richiede più spazio e tempo), dimostra che una WOTM può riconoscere tutti i linguaggi riconoscibili da una TM standard.

Combinando entrambe le direzioni, concludiamo che le WOTM sono equivalenti alle TM standard in termini di potenza computazionale.

16 3. Chiamiamo k-PDA un automa a pila dotato di k pile.

(a) Mostrare che i 2-PDA sono più potenti degli 1-PDA

(b) Mostrare che i 2-PDA riconoscono esattamente la classe dei linguaggi Turing-riconoscibili

(c) Mostrare che i 3-PDA non sono più potenti dei 2-PDA

Soluzione. (a) Mostrare che i 2-PDA sono più potenti degli 1-PDA

Per dimostrare che i 2-PDA sono più potenti degli 1-PDA, dobbiamo mostrare che: 1. Ogni linguaggio riconoscibile da un 1-PDA è anche riconoscibile da un 2-PDA (ovvio,

basta usare solo una delle due pile). 2. Esiste almeno un linguaggio riconoscibile da un 2-PDA che non è riconoscibile da alcun 1-PDA.

Per il punto 2, consideriamo il linguaggio $L = \{a^n b^n c^n \mid n \geq 1\}$. È noto che questo linguaggio non è context-free, e quindi non è riconoscibile da un 1-PDA (poiché i linguaggi riconoscibili da 1-PDA sono esattamente i linguaggi context-free).

Possiamo costruire un 2-PDA che riconosce L come segue: 1. Legge gli a dall'input e li inserisce nella prima pila. 2. Legge i b dall'input e, per ogni b letto, estrae un a dalla prima pila. Se la prima pila diventa vuota prima di aver letto tutti i b, rifiuta. 3. Legge i c dall'input e li inserisce nella seconda pila. 4. Verifica che il numero di c inseriti nella seconda pila sia uguale al numero di b letti (che è uguale al numero di a originariamente nella prima pila). Può farlo confrontando la dimensione della seconda pila con il numero di b letti, che può essere tenuto traccia durante il passo 2. 5. Accetta se e solo se tutte le precedenti condizioni sono soddisfatte e l'input è stato completamente letto.

Poiché L non è context-free ma è riconoscibile da un 2-PDA, concludiamo che i 2-PDA sono più potenti degli 1-PDA.

(b) Mostrare che i 2-PDA riconoscono esattamente la classe dei linguaggi Turing-riconoscibili

Dimostriamo che i 2-PDA sono equivalenti alle macchine di Turing in termini di potenza computazionale.

Direzione $\Rightarrow$: Ogni linguaggio riconoscibile da un 2-PDA è Turing-riconoscibile.

Questa direzione è diretta: una macchina di Turing può facilmente simulare un 2-PDA usando due porzioni del suo nastro per simulare le due pile. Per ogni operazione del 2-PDA sulle pile, la macchina di Turing esegue le operazioni corrispondenti sulle porzioni del nastro che rappresentano le pile. Quindi, ogni linguaggio riconoscibile da un 2-PDA è anche Turing-riconoscibile.

Direzione $\Leftarrow$: Ogni linguaggio Turing-riconoscibile è riconoscibile da un 2-PDA.

Per questa direzione, mostreremo come un 2-PDA può simulare una macchina di Turing. L'idea chiave è che le due pile possono essere utilizzate per simulare il nastro bidirezionale di una macchina di Turing.

Sia M una macchina di Turing. Costruiamo un 2-PDA P che simula M come segue: 1. P utilizzerà le due pile per rappresentare il nastro di M: la prima pila conterrà la porzione del nastro a sinistra della testina (in ordine inverso, con il simbolo più vicino alla testina in cima), e la seconda pila conterrà la porzione del nastro a destra della testina (incluso il simbolo corrente, in cima). 2. Per simulare un passo di M, P esamina il simbolo in cima alla seconda pila (che rappresenta il simbolo corrente letto da M), determina l'azione di M in base allo stato corrente e al simbolo letto, e aggiorna le pile di conseguenza. 3. Se M si sposta a destra, P estrae il simbolo dalla cima della seconda pila e lo inserisce nella prima pila (dopo averlo eventualmente modificato). 4. Se M si sposta a sinistra, P estrae il simbolo dalla cima della prima pila e lo inserisce nella seconda pila (dopo aver estratto e eventualmente modificato il simbolo corrente della seconda pila). 5. P accetta se e solo se M accetta.

Questa costruzione permette a P di simulare fedelmente M, quindi ogni linguaggio Turing-riconoscibile è riconoscibile da un 2-PDA.

Combinando entrambe le direzioni, concludiamo che i 2-PDA riconoscono esattamente la classe dei linguaggi Turing-riconoscibili.

(c) Mostrare che i 3-PDA non sono più potenti dei 2-PDA

Dalla parte (b), abbiamo dimostrato che i 2-PDA sono equivalenti alle macchine di Turing in termini di potenza computazionale. Poiché le macchine di Turing rappresentano il massimo della potenza computazionale secondo la tesi di Church-Turing, i 3-PDA non possono essere più potenti dei 2-PDA.

Formalmente, ogni linguaggio riconoscibile da un 3-PDA è anche riconoscibile da una macchina di Turing (poiché una macchina di Turing può simulare un qualsiasi numero di pile). Ma dalla parte (b), sappiamo che ogni linguaggio Turing-riconoscibile è anche riconoscibile da un 2-PDA. Quindi, ogni linguaggio riconoscibile da un 3-PDA è anche riconoscibile da un 2-PDA.

Pertanto, i 3-PDA non sono più potenti dei 2-PDA.

2 Decidibilità e indecidibilità

17 4. Sia $ALLDFA = \{\langle A\rangle \mid A \text{ è un DFA e } L(A) = \Sigma^*\}$. Mostrare che $ALLDFA$ è decidibile.

Soluzione. Per dimostrare che $ALLDFA$ è decidibile, costruiremo una macchina di Turing M che, dato in input la codifica $\langle A\rangle$ di un DFA $A = (Q, \Sigma, \delta, q_0, F)$, decide se $L(A) = \Sigma^*$.

L'idea chiave è che $L(A) = \Sigma^*$ se e solo se A accetta tutte le possibili stringhe su Σ, cioè se e solo se non esiste alcuna stringa che A rifiuta. Equivalentemente, $L(A) = \Sigma^*$ se e solo se $L(\overline{A}) = \emptyset$, dove $\overline{A}$ è il DFA che accetta il complemento di $L(A)$.

La macchina di Turing M opera come segue:

1. Dato in input $\langle A\rangle$, M verifica se $\langle A\rangle$ è una codifica valida di un DFA. Se non lo è, M rifiuta.

2. M costruisce il DFA $\overline{A} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ che accetta il complemento di $L(A)$, semplicemente complementando l'insieme degli stati finali di A.

3. M determina se $L(\overline{A}) = \emptyset$, cioè se non esiste alcuna stringa accettata da $\overline{A}$. Questo può essere fatto verificando se nessuno stato finale di $\overline{A}$ è raggiungibile dallo stato iniziale q_0. La raggiungibilità può essere determinata utilizzando un algoritmo standard di ricerca dei cammini, come la BFS (Breadth-First Search) o la DFS (Depth-First Search). Se $L(\overline{A}) = \emptyset$, allora $L(A) = \Sigma^*$ e M accetta. Altrimenti, M rifiuta.

L'algoritmo di raggiungibilità opera come segue:

1. Inizializza un insieme R di stati raggiungibili con $\{q_0\}$.

2. Iterativamente, aggiungi a R tutti gli stati che possono essere raggiunti in un singolo passo dagli stati già in R. Formalmente, in ogni iterazione, aggiungi a R tutti gli stati $q' \in Q$ tali che esiste $q \in R$ e $a \in \Sigma$ per cui $\delta(q, a) = q'$.

3. Ripeti il passo 2 fino a quando R non cambia più.

4. Se $R \cap (Q \setminus F) = \emptyset$, allora $L(\overline{A}) = \emptyset$ e quindi $L(A) = \Sigma^*$. Altrimenti, $L(\overline{A}) \neq \emptyset$ e quindi $L(A) \neq \Sigma^*$.

Poiché un DFA ha un numero finito di stati, l'algoritmo termina dopo un numero finito di passi. Inoltre, l'algoritmo è corretto perché un DFA accetta una stringa se e solo

se, dopo aver letto la stringa, si trova in uno stato finale. Se nessuno stato finale di $\overline{A}$ è raggiungibile dallo stato iniziale, allora non esiste alcuna stringa che $\overline{A}$ accetta, il che significa che $L(\overline{A}) = \emptyset$ e quindi $L(A) = \Sigma^*$.

Quindi, M decide correttamente se $L(A) = \Sigma^*$, il che dimostra che $ALLDFA$ è decidibile.

18 5. Sia $A\varepsilon CFG = \{\langle G \rangle \mid G \text{ è una CFG che genera } \varepsilon\}$. Mostrare che $A\varepsilon CFG$ è decidibile.

Soluzione. Per dimostrare che $A\varepsilon CFG$ è decidibile, costruiremo una macchina di Turing M che, dato in input la codifica $\langle G \rangle$ di una grammatica context-free $G = (V, \Sigma, R, S)$, decide se $\varepsilon \in L(G)$, cioè se G genera la stringa vuota.

L'idea chiave è che $\varepsilon \in L(G)$ se e solo se la variabile iniziale S è annullabile, cioè può derivare la stringa vuota. Una variabile è annullabile se e solo se può derivare ε attraverso una sequenza di applicazioni delle regole di produzione.

La macchina di Turing M opera come segue:

1. Dato in input $\langle G \rangle$, M verifica se $\langle G \rangle$ è una codifica valida di una CFG. Se non lo è, M rifiuta.

2. M inizializza un insieme $Null$ di variabili annullabili come l'insieme vuoto.

3. M applica iterativamente le seguenti regole fino a quando $Null$ non cambia più: a. Se esiste una regola di produzione $A \to \varepsilon$ in R, aggiungi A a $Null$. b. Se esiste una regola di produzione $A \to X_1X_2 \ldots X_n$ in R tale che ogni X_i è in $Null$, aggiungi A a $Null$.

4. Dopo che il processo ha raggiunto un punto fisso, M verifica se la variabile iniziale S è in $Null$. Se lo è, allora $\varepsilon \in L(G)$ e M accetta. Altrimenti, $\varepsilon \notin L(G)$ e M rifiuta.

Più formalmente, l'algoritmo opera come segue:

DecideEmptyString($G = (V, \Sigma, R, S)$):

```
1: Null ← ∅                                   ▷ Inizializza l'insieme delle variabili annullabili
2: changed ← true
3: while changed do
4:     changed ← false
5:     for ogni regola di produzione A → α in R do
6:         if α = ε o α contiene solo variabili in Null then
7:             if A ∉ Null then
8:                 Null ← Null ∪ {A}
9:                 changed ← true
10:            end if
11:        end if
12:    end for
13: end while
14: if S ∈ Null then return ACCETTA
15: elsereturn RIFIUTA
16: end if
```

Poiché una CFG ha un numero finito di variabili, il ciclo while termina dopo un numero finito di iterazioni. Al termine, $Null$ contiene esattamente l'insieme delle variabili annullabili, cioè quelle che possono derivare ε. Quindi, $S \in Null$ se e solo se $\varepsilon \in L(G)$.

L'algoritmo è chiaramente decidibile (termina sempre con una risposta corretta), quindi $A\varepsilon CFG$ è decidibile.

19 6. Sia $SREX = \{\langle R, S\rangle \mid R, S \text{ sono espressioni regolari tali che } L(R) \subseteq L(S)\}$. Mostrare che $SREX$ è decidibile.

Soluzione. Per dimostrare che $SREX$ è decidibile, costruiremo una macchina di Turing M che, dato in input le codifiche $\langle R, S\rangle$ di due espressioni regolari R e S, decide se $L(R) \subseteq L(S)$.

L'idea chiave è che $L(R) \subseteq L(S)$ se e solo se $L(R) \cap L(S)^C = \emptyset$, dove $L(S)^C$ è il complemento di $L(S)$. In altre parole, $L(R) \subseteq L(S)$ se e solo se non esiste alcuna stringa che sia in $L(R)$ ma non in $L(S)$.

La macchina di Turing M opera come segue:

1. Dato in input $\langle R, S\rangle$, M verifica se $\langle R\rangle$ e $\langle S\rangle$ sono codifiche valide di espressioni regolari. Se non lo sono, M rifiuta.
2. M costruisce un NFA A_R che riconosce $L(R)$ usando la costruzione standard da espressione regolare a NFA (costruzione di Thompson).
3. M costruisce un NFA A_S che riconosce $L(S)$ usando la stessa costruzione.
4. M converte A_S in un DFA B_S usando l'algoritmo di subset construction.
5. M complementa B_S per ottenere un DFA C_S che riconosce $L(S)^C$.
6. M costruisce un NFA D che riconosce $L(R) \cap L(S)^C$ come il prodotto cartesiano di A_R e C_S.
7. M determina se $L(D) = \emptyset$ utilizzando l'algoritmo standard per verificare se un linguaggio riconosciuto da un NFA è vuoto. Se $L(D) = \emptyset$, allora $L(R) \subseteq L(S)$ e M accetta. Altrimenti, $L(R) \not\subseteq L(S)$ e M rifiuta.

Il passo 7 può essere implementato come segue:

1. Costruisci il grafo di raggiungibilità di D, dove i nodi sono gli stati di D e c'è un arco da uno stato q a uno stato q' se è possibile passare da q a q' leggendo un simbolo.
2. Utilizza un algoritmo di ricerca dei cammini (come BFS o DFS) per verificare se esiste un cammino dallo stato iniziale a uno stato finale nel grafo di raggiungibilità.
3. Se non esiste un tale cammino, allora $L(D) = \emptyset$, altrimenti $L(D) \neq \emptyset$.

Tutti i passi sopra descritti possono essere eseguiti da una macchina di Turing in un numero finito di passi. Inoltre, l'algoritmo è corretto perché: - $L(R) \subseteq L(S)$ se e solo se $L(R) \cap L(S)^C = \emptyset$. - $L(D) = L(R) \cap L(S)^C$. - Quindi, $L(R) \subseteq L(S)$ se e solo se $L(D) = \emptyset$.

Poiché M decide correttamente se $L(R) \subseteq L(S)$, concludiamo che $SREX$ è decidibile.

20 7. Sia $X = \{\langle M, w\rangle \mid M$ è una TM a nastro singolo che non modifica la porzione di nastro che contiene l'input $w\}$. X è decidibile? Dimostrare la vostra risposta.

Soluzione. Dimostriamo che X è indecidibile.

Prima di tutto, notiamo che il linguaggio X chiede di determinare se, data una TM M e un input w, la computazione di M su w non modifica mai la porzione del nastro che contiene l'input originale w.

Per dimostrare che X è indecidibile, useremo una riduzione dal problema dell'arresto (halting problem), che è noto essere indecidibile. Ricordiamo che il problema dell'arresto è definito come: $HALT = \{\langle M, w\rangle \mid M$ è una TM ed M si ferma su input $w\}$.

Data una macchina di Turing M e un input w, costruiremo una nuova macchina di Turing M' tale che $\langle M', w\rangle \in X$ se e solo se $\langle M, w\rangle \notin HALT$ (cioè, M non si ferma su input w).

La macchina di Turing M' opera come segue su un input x: 1. M' verifica se $x = w$. Se $x \neq w$, M' si ferma senza modificare il nastro. 2. Se $x = w$, M' simula M su w. 3. Se la simulazione di M su w termina, M' modifica la prima cella del nastro (cambia il primo simbolo di w) e poi si ferma.

Analizziamo il comportamento di M' su input w: - Se M si ferma su input w, allora M' modificherà la prima cella del nastro, che contiene il primo simbolo di w. Quindi, $\langle M', w\rangle \notin X$. - Se M non si ferma su input w, allora M' continuerà a simulare M indefinitamente e non modificherà mai il nastro. Quindi, $\langle M', w\rangle \in X$.

Quindi, $\langle M', w\rangle \in X$ se e solo se $\langle M, w\rangle \notin HALT$.

Supponiamo ora, per assurdo, che X sia decidibile. Allora esisterebbe una macchina di Turing T che decide X. Potremmo utilizzare T per costruire una macchina di Turing che decide il complemento di $HALT$: dato $\langle M, w\rangle$, la macchina costruisce $\langle M', w\rangle$ come descritto sopra, e poi esegue T su $\langle M', w\rangle$. Se T accetta, la macchina accetta (indicando che M non si ferma su w). Se T rifiuta, la macchina rifiuta (indicando che M si ferma su w).

Ma questo è impossibile, poiché il complemento di $HALT$ è indecidibile. Quindi, la nostra supposizione che X sia decidibile deve essere falsa.

Pertanto, X è indecidibile.

21 8. Sia $ETM = \{\langle M\rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$. Mostrare che $\overline{ETM}$, il complemento di ETM, è Turing-riconoscibile.

Soluzione. Per dimostrare che $\overline{ETM}$ è Turing-riconoscibile, dobbiamo costruire una macchina di Turing T che, dato in input la codifica $\langle M\rangle$ di una TM M, accetta se $L(M) \neq \emptyset$ (cioè, se M accetta almeno una stringa) e non si ferma altrimenti.

Ricordiamo che $\overline{ETM} = \{\langle M\rangle \mid M \text{ è una TM tale che } L(M) \neq \emptyset\}$.

La macchina di Turing T opera come segue:

1. Dato in input $\langle M\rangle$, T verifica se $\langle M\rangle$ è una codifica valida di una TM. Se non lo è, T rifiuta.

2. T enumera sistematicamente tutte le possibili stringhe in Σ^* in ordine di lunghezza crescente: $\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$ (assumendo che $\Sigma = \{0, 1\}$, ma il ragionamento si generalizza a qualsiasi alfabeto finito).

3. Per ogni stringa w enumerata, T simula M su input w per un numero di passi limitato, aumentando il limite ad ogni iterazione.

4. Se M accetta w entro il limite di passi, T accetta.

5. Se M non accetta w entro il limite di passi, T passa alla prossima stringa o alla stessa stringa con un limite di passi aumentato.

In modo più dettagliato, T può essere implementata come segue:

Questa macchina di Turing T accetta $\langle M\rangle$ se e solo se M accetta almeno una stringa, cioè, se $L(M) \neq \emptyset$. Se $L(M) = \emptyset$, allora T non si fermerà mai (ma questo è accettabile per una macchina che riconosce un linguaggio).

Quindi, T riconosce correttamente $\overline{ETM}$, il che dimostra che $\overline{ETM}$ è Turing-riconoscibile.

22 9. Mostrare che se A è Turing-riconoscibile e $\overline{A} \leq_m A$, allora A è decidibile.

RICONOSCI($\langle M \rangle$):

1: $i \leftarrow 0$ ▷ Indice della stringa attuale nell'enumerazione
2: **while** true **do**
3: $w \leftarrow$ la i-esima stringa nell'enumerazione di Σ^*
4: **for** $j \leftarrow 1$ to i **do**
5: Simula M su w per j passi
6: **if** M accetta w entro j passi **then return** ACCETTA
7: **end if**
8: **end for**
9: $i \leftarrow i + 1$
10: **end while**

Soluzione. Vogliamo dimostrare che se A è Turing-riconoscibile e $\overline{A} \leq_m A$, allora A è decidibile.

Dato che $\overline{A} \leq_m A$, esiste una funzione calcolabile f tale che per ogni stringa x: $x \in \overline{A}$ se e solo se $f(x) \in A$.

Possiamo usare questa funzione f per costruire una macchina di Turing che decide A. Sia M_A una macchina di Turing che riconosce A (sappiamo che esiste perché A è Turing-riconoscibile). Costruiamo una macchina di Turing D che decide A come segue:

D su input x: 1. Calcola $y = f(x)$. 2. Esegui in parallelo M_A su x e M_A su y. 3. Se M_A accetta x, accetta. 4. Se M_A accetta y, rifiuta.

L'esecuzione "in parallelo" può essere implementata eseguendo un passo di M_A su x, poi un passo di M_A su y, e così via.

Analizziamo il comportamento di D: - Se $x \in A$, allora M_A accetterà x (perché M_A riconosce A), quindi D accetterà. - Se $x \notin A$, allora $x \in \overline{A}$, il che implica che $f(x) \in A$ (per la proprietà di f). Quindi, M_A accetterà $f(x)$, e D rifiuterà.

Pertanto, D accetta se e solo se $x \in A$, ovvero D decide A.

Rimane da dimostrare che D si ferma sempre. Sia x una stringa arbitraria. Abbiamo esattamente uno dei due casi: 1. $x \in A$: In questo caso, M_A accetterà x e D si fermerà. 2. $x \notin A$: In questo caso, $x \in \overline{A}$, quindi $f(x) \in A$. Quindi, M_A accetterà $f(x)$ e D si fermerà.

In entrambi i casi, D si ferma, quindi D è una macchina di Turing che decide A.

Concludiamo che se A è Turing-riconoscibile e $\overline{A} \leq_m A$, allora A è decidibile.

23 10. Sia A un linguaggio. Dimostrare che A è Turing-riconoscibile se e solo se esiste un linguaggio decidibile B tale che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y \rangle \in B\}$.

Soluzione. Dimostriamo che A è Turing-riconoscibile se e solo se esiste un linguaggio decidibile B tale che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y \rangle \in B\}$.

Direzione $\Rightarrow$: Se A è Turing-riconoscibile, allora esiste un linguaggio decidibile B tale che $A = \{x \mid$ esiste y tale che $\langle x, y \rangle \in B\}$.

Sia M una macchina di Turing che riconosce A. Definiamo il linguaggio B come: $B = \{\langle x, y \rangle \mid M \text{ accetta } x \text{ entro } |y| \text{ passi}\}$

Dimostriamo che B è decidibile e che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y \rangle \in B\}$.

Per decidere B, una macchina di Turing D opera come segue su input $\langle x, y \rangle$: 1. Simula M su input x per esattamente $|y|$ passi. 2. Se M accetta x entro $|y|$ passi, D accetta. Altrimenti, D rifiuta.

Poiché la simulazione è limitata a $|y|$ passi, D terminerà sempre. Quindi, B è decidibile.

Ora verifichiamo che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$: - Se $x \in A$, allora M accetta x dopo un certo numero n di passi. Sia y una stringa di lunghezza almeno n. Allora, $\langle x, y\rangle \in B$, quindi $x \in \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$. - Se $x \in \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$, allora esiste y tale che $\langle x, y\rangle \in B$, il che significa che M accetta x entro $|y|$ passi. Quindi, M accetta x, ovvero $x \in A$.

Pertanto, $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$.

Direzione $\Leftarrow$: Se esiste un linguaggio decidibile allora A è Turing-riconoscibile.

Sia B un linguaggio decidibile tale che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$, e sia D una macchina di Turing che decide B.

Costruiamo una macchina di Turing M che riconosce A come segue:

M su input x: 1. Enumera sistematicamente tutte le stringhe y in ordine di lunghezza crescente: $\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$ 2. Per ogni stringa y enumerata, esegui D su input $\langle x, y\rangle$. 3. Se D accetta $\langle x, y\rangle$ per qualche y, M accetta x.

Analizziamo il comportamento di M: - Se $x \in A$, allora esiste y tale che $\langle x, y\rangle \in B$. Quando M enumera questa stringa y (o una qualsiasi altra stringa y' tale che $\langle x, y'\rangle \in B$), D accetterà $\langle x, y\rangle$ (o $\langle x, y'\rangle$), e M accetterà x. - Se $x \notin A$, allora non esiste alcuna stringa y tale che $\langle x, y\rangle \in B$. Quindi, D rifiuterà $\langle x, y\rangle$ per ogni y, e M non accetterà mai x.

Quindi, M accetta x se e solo se $x \in A$, ovvero M riconosce A. Pertanto, A è Turing-riconoscibile.

Combinando entrambe le direzioni, abbiamo dimostrato che A è Turing-riconoscibile se e solo se esiste un linguaggio decidibile B tale che $A = \{x \mid \text{esiste } y \text{ tale che } \langle x, y\rangle \in B\}$.

24 11. $A \leq_m B$ e B è un linguaggio regolare implica che A è un linguaggio regolare? Perché sì o perché no?

Soluzione. La risposta è no, $A \leq_m B$ e B è un linguaggio regolare non implica necessariamente che A sia un linguaggio regolare. Dimostriamo questo con un controesempio.

Sia $B = \{1\}$, che è chiaramente un linguaggio regolare (è un linguaggio finito).

Sia $A = \{a^n b^n \mid n \geq 1\}$, che è un linguaggio context-free ma non regolare.

Definiamo una funzione $f : \{a, b\}^* \to \{0, 1\}^*$ come segue:

$$f(w) = \begin{cases} 1 & \text{se } w \in A \\ 0 & \text{se } w \notin A \end{cases}$$

Chiaramente, $w \in A$ se e solo se $f(w) \in B$, quindi $A \leq_m B$.

Ma A non è un linguaggio regolare, quindi $A \leq_m B$ e B è un linguaggio regolare non implica che A sia un linguaggio regolare.

La ragione intuitiva di questo risultato è che le riduzioni molte-a-uno possono "collassare" la struttura di un linguaggio complesso in un linguaggio semplice. Nel nostro esempio, f mappa tutte le stringhe in A a un'unica stringa in B, e tutte le stringhe non in A a una stringa non in B. Questa mappatura "perde" tutta l'informazione sulla struttura interna di A.

In generale, se $A \leq_m B$, sappiamo che A non è "più complesso" di B in termini di calcolo. Ma questo non significa che A non possa essere "più complesso" di B in termini di struttura linguistica (come essere regolare, context-free, ecc.). La riduzione può semplificare la struttura di A, mappandola in B.

25 12. Mostrare che se A è Turing-riconoscibile e $A \leq_m \overline{A}$, allora A è decidibile.

Soluzione. Vogliamo dimostrare che se A è Turing-riconoscibile e $A \leq_m \overline{A}$, allora A è decidibile.

Dato che $A \leq_m \overline{A}$, esiste una funzione calcolabile f tale che per ogni stringa x: $x \in A$ se e solo se $f(x) \in \overline{A}$.

Equivalentemente, $x \in A$ se e solo se $f(x) \notin A$.

Possiamo usare questa funzione f per costruire una macchina di Turing che decide A. Sia M_A una macchina di Turing che riconosce A (sappiamo che esiste perché A è Turing-riconoscibile). Costruiamo una macchina di Turing D che decide A come segue:

D su input x: 1. Calcola $y = f(x)$. 2. Esegui in parallelo M_A su x e M_A su y. 3. Se M_A accetta x, accetta. 4. Se M_A accetta y, rifiuta.

L'esecuzione "in parallelo" può essere implementata eseguendo un passo di M_A su x, poi un passo di M_A su y, e così via.

Analizziamo il comportamento di D: - Se $x \in A$, allora M_A accetterà x (perché M_A riconosce A), quindi D accetterà. Inoltre, $f(x) \in \overline{A}$, quindi $f(x) \notin A$, e M_A non accetterà mai $f(x)$. - Se $x \notin A$, allora M_A non accetterà mai x. Inoltre, $f(x) \in A$ (per la proprietà di f), quindi M_A accetterà $f(x)$, e D rifiuterà.

Pertanto, D accetta se e solo se $x \in A$, ovvero D decide A.

Rimane da dimostrare che D si ferma sempre. Sia x una stringa arbitraria. Abbiamo esattamente uno dei due casi: 1. $x \in A$: In questo caso, M_A accetterà x e D si fermerà. 2. $x \notin A$: In questo caso, $f(x) \in A$, quindi M_A accetterà $f(x)$ e D si fermerà.

In entrambi i casi, D si ferma, quindi D è una macchina di Turing che decide A.

Concludiamo che se A è Turing-riconoscibile e $A \leq_m \overline{A}$, allora A è decidibile.

26 13. Sia $J = \{w \mid w = 0x$ per qualche $x \in A_{TM}$ oppure $w = 1y$ per qualche $y \in \overline{A_{TM}}\}$. Mostrare che sia J che $\overline{J}$ non sono Turing-riconoscibili.

Soluzione. Ricordiamo che $A_{TM} = \{\langle M, w\rangle \mid M$ è una TM ed M accetta $w\}$ è il problema dell'accettazione, che è noto essere indecidibile e Turing-riconoscibile, ma il suo complemento $\overline{A_{TM}}$ non è Turing-riconoscibile.

Dimostriamo innanzitutto che J non è Turing-riconoscibile.

Supponiamo, per assurdo, che J sia Turing-riconoscibile. Allora esisterebbe una macchina di Turing R che riconosce J. Utilizzando R, potremmo costruire una macchina di Turing che riconosce $\overline{A_{TM}}$ come segue:

M su input $\langle M', w'\rangle$: 1. Calcola $z = 1\langle M', w'\rangle$ (cioè, concatena il simbolo 1 con la codifica di $\langle M', w'\rangle$). 2. Esegui R su z. 3. Se R accetta z, accetta.

Analizziamo il comportamento di M: - Se $\langle M', w'\rangle \in \overline{A_{TM}}$, allora $z = 1\langle M', w'\rangle \in J$ (per la definizione di J). Quindi, R accetterà z, e M accetterà $\langle M', w'\rangle$. - Se $\langle M', w'\rangle \notin \overline{A_{TM}}$, allora $\langle M', w'\rangle \in A_{TM}$. Pertanto, $z = 1\langle M', w'\rangle \notin J$ (per la definizione di J). Quindi, R non accetterà mai z, e M non accetterà mai $\langle M', w'\rangle$.

Quindi, M accetta $\langle M', w'\rangle$ se e solo se $\langle M', w'\rangle \in \overline{A_{TM}}$, ovvero M riconosce $\overline{A_{TM}}$. Ma questo è impossibile, poiché $\overline{A_{TM}}$ non è Turing-riconoscibile. Quindi, la nostra supposizione che J sia Turing-riconoscibile deve essere falsa.

Ora dimostriamo che $\overline{J}$ non è Turing-riconoscibile.

Supponiamo, per assurdo, che $\overline{J}$ sia Turing-riconoscibile. Allora esisterebbe una macchina di Turing R' che riconosce $\overline{J}$. Utilizzando R', potremmo costruire una macchina di Turing che riconosce $\overline{A_{TM}}$ come segue:

M' su input $\langle M', w' \rangle$: 1. Calcola $z' = 0\langle M', w' \rangle$ (cioè, concatena il simbolo 0 con la codifica di $\langle M', w' \rangle$). 2. Esegui R' su z'. 3. Se R' accetta z', accetta.

Analizziamo il comportamento di M': - Se $\langle M', w' \rangle \in \overline{A_{TM}}$, allora $z' = 0\langle M', w' \rangle \notin J$ (per la definizione di J). Quindi, $z' \in \overline{J}$, e R' accetterà z'. Pertanto, M' accetterà $\langle M', w' \rangle$. - Se $\langle M', w' \rangle \notin \overline{A_{TM}}$, allora $\langle M', w' \rangle \in A_{TM}$. Pertanto, $z' = 0\langle M', w' \rangle \in J$ (per la definizione di J). Quindi, $z' \notin \overline{J}$, e R' non accetterà mai z'. Pertanto, M' non accetterà mai $\langle M', w' \rangle$.

Quindi, M' accetta $\langle M', w' \rangle$ se e solo se $\langle M', w' \rangle \in \overline{A_{TM}}$, ovvero M' riconosce $\overline{A_{TM}}$. Ma questo è impossibile, poiché $\overline{A_{TM}}$ non è Turing-riconoscibile. Quindi, la nostra supposizione che $\overline{J}$ sia Turing-riconoscibile deve essere falsa.

In sintesi, abbiamo dimostrato che né J né $\overline{J}$ sono Turing-riconoscibili.

3 Complessità computazionale

28 14. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

(a) Un circuito Toniano in un grafo G è un ciclo che attraversa almeno la metà dei vertici del grafo (senza ripetere vertici). Il problema del circuito Toniano è il problema di stabilire se un grafo contiene un circuito Toniano. Dimostrare che il problema è NP-completo.

(b) Un circuito quasi Hamiltoniano in un grafo G è un ciclo che attraversa esattamente una volta tutti i vertici del grafo tranne uno. Il problema del circuito quasi Hamiltoniano è il problema di stabilire se un grafo contiene un circuito quasi Hamiltoniano. Dimostrare che il problema del circuito quasi Hamiltoniano è NP-completo.

Soluzione. (a) Dimostrazione che il problema del circuito Toniano è NP-completo.

Per dimostrare che un problema è NP-completo, dobbiamo dimostrare che: 1. È in NP. 2. È NP-hard (cioè, ogni problema in NP è riducibile a esso in tempo polinomiale).

Passo 1: Il problema del circuito Toniano è in NP.

Per dimostrare che il problema è in NP, dobbiamo mostrare che esiste un verificatore polinomiale che, dato un certificato (in questo caso, un ciclo nel grafo), può verificare in tempo polinomiale se è un circuito Toniano.

Sia G un grafo con n vertici e sia C un ciclo in G proposto come certificato. Un verificatore può: 1. Controllare che C sia effettivamente un ciclo in G (ogni arco in C è un arco di G). 2. Controllare che C non contenga vertici ripetuti. 3. Contare il numero di vertici in C e verificare che sia almeno $\lfloor n/2 \rfloor + 1$ (cioè, almeno la metà dei vertici di G).

Tutte queste verifiche possono essere eseguite in tempo polinomiale rispetto alla dimensione dell'input. Quindi, il problema è in NP.

Passo 2: Il problema del circuito Toniano è NP-hard.

Per dimostrare che il problema è NP-hard, ridurremo il problema del circuito Hamiltoniano (che è noto essere NP-completo) al problema del circuito Toniano.

Sia $G = (V, E)$ un grafo per il quale vogliamo determinare se contiene un circuito Hamiltoniano. Costruiamo un nuovo grafo $G' = (V', E')$ come segue: 1. $V' = V \cup \{v_1, v_2, \ldots, v_n\}$, dove $n = |V|$ e $v_1, v_2, \ldots, v_n$ sono nuovi vertici. 2. $E' = E \cup \{(v_i, v_j) \mid 1 \leq i, j \leq n, i \neq j\}$, cioè aggiungiamo un grafo completo sui nuovi vertici.

In altre parole, G' è ottenuto da G aggiungendo un grafo completo di dimensione uguale a G e senza connessioni tra G e il nuovo grafo.

Ora dimostriamo che G ha un circuito Hamiltoniano se e solo se G' ha un circuito Toniano.

Direzione $\Rightarrow$: Se G ha un circuito Hamiltoniano C, allora C è anche un ciclo in G' che attraversa esattamente n vertici di G'. Poiché G' ha $2n$ vertici, C attraversa esattamente la metà dei vertici di G', quindi è un circuito Toniano in G'.

Direzione $\Leftarrow$: Se G' ha un circuito Toniano C', allora C' attraversa almeno n vertici di G'. Poiché il grafo completo aggiunto non ha connessioni con G, C' deve essere interamente contenuto in G o interamente contenuto nel grafo completo. Se C' è interamente contenuto in G e attraversa almeno n vertici, allora deve attraversare tutti gli n vertici di G (poiché G ha solo n vertici), quindi è un circuito Hamiltoniano in G. Se C' è interamente contenuto nel grafo completo, allora G potrebbe non avere un circuito Hamiltoniano. Tuttavia, possiamo modificare leggermente la costruzione per forzare C' a essere in G: ad esempio, possiamo aggiungere solo $n - 1$ nuovi vertici invece di n, oppure possiamo aggiungere un vincolo che il circuito Toniano deve attraversare almeno un vertice di G.

Assumiamo una di queste modifiche, in modo che se G' ha un circuito Toniano, allora G ha un circuito Hamiltoniano.

Questa riduzione può essere eseguita in tempo polinomiale rispetto alla dimensione dell'input. Quindi, il problema del circuito Toniano è NP-hard.

Combinando i passi 1 e 2, concludiamo che il problema del circuito Toniano è NP-completo.

(b) Dimostrazione che il problema del circuito quasi Hamiltoniano è NP-completo.

Passo 1: Il problema del circuito quasi Hamiltoniano è in NP.

Analogamente al caso precedente, per dimostrare che il problema è in NP, mostriamo che esiste un verificatore polinomiale per esso.

Sia G un grafo con n vertici e sia C un ciclo in G proposto come certificato. Un verificatore può: 1. Controllare che C sia effettivamente un ciclo in G. 2. Controllare che C non contenga vertici ripetuti. 3. Contare il numero di vertici in C e verificare che sia esattamente $n - 1$ (cioè, tutti i vertici di G tranne uno).

Tutte queste verifiche possono essere eseguite in tempo polinomiale. Quindi, il problema è in NP.

Passo 2: Il problema del circuito quasi Hamiltoniano è NP-hard.

Per dimostrare che il problema è NP-hard, ridurremo il problema del circuito Hamiltoniano al problema del circuito quasi Hamiltoniano.

Sia $G = (V, E)$ un grafo per il quale vogliamo determinare se contiene un circuito Hamiltoniano. Costruiamo un nuovo grafo $G' = (V', E')$ come segue: 1. $V' = V \cup \{v_{new}\}$, dove v_{new} è un nuovo vertice. 2. $E' = E \cup \{(v, v_{new}) \mid v \in V\}$, cioè connettiamo il nuovo vertice a tutti i vertici di G.

Ora dimostriamo che G ha un circuito Hamiltoniano se e solo se G' ha un circuito quasi Hamiltoniano.

Direzione $\Rightarrow$: Se G ha un circuito Hamiltoniano C, allora C è anche un ciclo in G' che attraversa esattamente $|V|$ vertici di G'. Poiché G' ha $|V|+1$ vertici, C attraversa tutti i vertici di G' tranne v_{new}, quindi è un circuito quasi Hamiltoniano in G'.

Direzione $\Leftarrow$: Se G' ha un circuito quasi Hamiltoniano C', allora C' attraversa esattamente $|V|$ vertici di G'. Ci sono due possibilità: 1. C' non include v_{new}: In questo caso, C' è interamente contenuto in G e attraversa tutti i vertici di G, quindi è un circuito Hamiltoniano in G. 2. C' include v_{new}: In questo caso, C' omette esattamente un vertice $v \in V$. Ma questo significa che C' include almeno un arco (v_{new}, u) per qualche $u \in V$. Se rimuoviamo v_{new} e questo arco da C' e aggiungiamo l'arco (u, v') per qualche altro $v' \in V$ che è adiacente a u in G, otteniamo un ciclo C'' in G che omette esattamente un vertice, ma questo vertice non è necessariamente v. Questa direzione richiede un po' più di cura.

In generale, per una riduzione più pulita, potremmo aggiungere due nuovi vertici invece di uno e collegare uno di essi a tutti i vertici di G e l'altro a nessun vertice di G. In questo modo, se G' ha un circuito quasi Hamiltoniano, deve omettere esattamente uno di questi due nuovi vertici, e quindi è facile ricavare un circuito Hamiltoniano in G.

Questa riduzione può essere eseguita in tempo polinomiale. Quindi, il problema del circuito quasi Hamiltoniano è NP-hard.

Combinando i passi 1 e 2, concludiamo che il problema del circuito quasi Hamiltoniano è NP-completo.

29 15. Considera i seguenti problemi:

$$\begin{aligned}\text{SetPartitioning} = \{\langle S\rangle \mid\ & S \text{ è un insieme di numeri interi che può essere suddiviso in due}\\ & \text{sottoinsiemi disgiunti } S_1, S_2 \text{ tali che la somma dei numeri in } S_1\\ & \text{è uguale alla somma dei numeri in } S_2\}\\ \text{SubsetSum} = \{\langle S, t\rangle \mid\ & S \text{ è un insieme di numeri interi, ed esiste } S' \subseteq S\\ & \text{tale che la somma dei numeri in } S' \text{ è uguale a } t\}\end{aligned}$$

(a) Mostra che entrambi i problemi sono in NP.

(b) Mostra che SetPartitioning è NP-Hard usando SubsetSum come problema di riferimento.

(c) Mostra che SubsetSum è NP-Hard usando SetPartitioning come problema di riferimento.

Soluzione. (a) Mostra che entrambi i problemi sono in NP.

Un problema è in NP se esiste un verificatore polinomiale che, dato un certificato, può verificare in tempo polinomiale se una soluzione è corretta.

SetPartitioning è in NP:

Per SetPartitioning, il certificato è una partizione dell'insieme S in due sottoinsiemi S_1 e S_2. Un verificatore può: 1. Controllare che S_1 e S_2 formino una partizione di S (cioè, $S_1 \cup S_2 = S$ e $S_1 \cap S_2 = \emptyset$). 2. Calcolare la somma degli elementi in S_1 e la somma degli elementi in S_2. 3. Verificare che le due somme siano uguali.

Tutte queste operazioni possono essere eseguite in tempo polinomiale rispetto alla dimensione dell'input. Quindi, SetPartitioning è in NP.

SubsetSum è in NP:

Per SubsetSum, il certificato è un sottoinsieme $S' \subseteq S$. Un verificatore può: 1. Controllare che $S' \subseteq S$. 2. Calcolare la somma degli elementi in S'. 3. Verificare che questa somma sia uguale a t.

Anche queste operazioni possono essere eseguite in tempo polinomiale. Quindi, SubsetSum è in NP.

(b) Mostra che SetPartitioning è NP-Hard usando SubsetSum come problema di riferimento.

Per dimostrare che SetPartitioning è NP-hard, ridurremo SubsetSum a SetPartitioning. Assumiamo che SubsetSum sia NP-hard.

Sia $\langle S, t \rangle$ un'istanza di SubsetSum, dove $S = \{s_1, s_2, \ldots, s_n\}$ è un insieme di numeri interi e t è un intero target. Vogliamo costruire un'istanza $\langle S' \rangle$ di SetPartitioning tale che $\langle S, t \rangle \in$ SubsetSum se e solo se $\langle S' \rangle \in$ SetPartitioning.

Definiamo $sum(S) = \sum_{i=1}^{n} s_i$. Costruiamo l'insieme S' come: $S' = S \cup \{sum(S) - 2t\}$ (aggiungiamo un nuovo elemento a S).

Ora dimostriamo che $\langle S, t \rangle \in$ SubsetSum se e solo se $\langle S' \rangle \in$ SetPartitioning.

Direzione $\Rightarrow$: Supponiamo che $\langle S, t \rangle \in$ SubsetSum. Allora esiste $S_1 \subseteq S$ tale che $\sum_{s \in S_1} s = t$. Sia $S_2 = S \setminus S_1$. Allora $\sum_{s \in S_2} s = sum(S) - t$.

Definiamo una partizione di S' come $S'_1 = S_1$ e $S'_2 = S_2 \cup \{sum(S) - 2t\}$.

Calconiamo le somme:

$$\sum_{s \in S'_1} s = \sum_{s \in S_1} s = t$$

$$\sum_{s \in S'_2} s = \sum_{s \in S_2} s + (sum(S) - 2t) = (sum(S) - t) + (sum(S) - 2t) = 2 \cdot sum(S) - 3t$$

Ma questo non è necessariamente uguale a t. Potremmo dover modificare la nostra costruzione.

Alternativamente, costruiamo S' come segue: $S' = S \cup \{2t - sum(S)\}$, assumendo che $2t - sum(S) \neq 0$ (altrimenti aggiungiamo un elemento diverso).

Con questa costruzione:

$$\sum_{s \in S'_1} s = \sum_{s \in S_1} s = t$$

$$\sum_{s \in S'_2} s = \sum_{s \in S_2} s + (2t - sum(S)) = (sum(S) - t) + (2t - sum(S)) = t$$

Quindi, le somme sono uguali, e $\langle S' \rangle \in$ SetPartitioning.

Direzione $\Leftarrow$: Supponiamo che $\langle S' \rangle \in$ SetPartitioning. Allora esiste una partizione di S' in S'_1 e S'_2 tale che $\sum_{s \in S'_1} s = \sum_{s \in S'_2} s$.

Ci sono due casi da considerare: 1. $2t - sum(S) \in S'_1$: In questo caso, sia $S_1 = S'_1 \setminus \{2t - sum(S)\}$. Abbiamo $\sum_{s \in S_1} s = \sum_{s \in S'_1} s - (2t - sum(S)) = \sum_{s \in S'_2} s - (2t - sum(S))$. 2. $2t - sum(S) \in S'_2$: In questo caso, sia $S_1 = S'_1$. Abbiamo $\sum_{s \in S_1} s = \sum_{s \in S'_1} s$.

In entrambi i casi, possiamo derivare un sottoinsieme $S_1 \subseteq S$ tale che $\sum_{s \in S_1} s = t$, il che significa che $\langle S, t \rangle \in$ SubsetSum.

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

(c) Mostra che SubsetSum è NP-Hard usando SetPartitioning come problema di riferimento.

Ora ridurremo SetPartitioning a SubsetSum, assumendo che SetPartitioning sia NP-hard.

Sia $\langle S \rangle$ un'istanza di SetPartitioning, dove $S = \{s_1, s_2, \ldots, s_n\}$ è un insieme di numeri interi. Vogliamo costruire un'istanza $\langle S', t \rangle$ di SubsetSum tale che $\langle S \rangle \in$ SetPartitioning se e solo se $\langle S', t \rangle \in$ SubsetSum.

Definiamo $sum(S) = \sum_{i=1}^{n} s_i$. Costruiamo l'istanza $\langle S', t \rangle$ come: $S' = S$ e $t = sum(S)/2$.

Nota che se $sum(S)$ è dispari, allora $\langle S \rangle \notin$ SetPartitioning (poiché non possiamo dividere un numero dispari in due parti uguali). In questo caso, possiamo semplicemente restituire un'istanza di SubsetSum che sappiamo essere falsa, come $\langle S', t \rangle = \langle \{1\}, 0 \rangle$.

Assumendo che $sum(S)$ sia pari, dimostriamo che $\langle S \rangle \in$ SetPartitioning se e solo se $\langle S', t \rangle \in$ SubsetSum.

Direzione $\Rightarrow$: Supponiamo che $\langle S \rangle \in$ SetPartitioning. Allora esiste una partizione di S in S_1 e S_2 tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s$. Poiché $S_1 \cup S_2 = S$ e $S_1 \cap S_2 = \emptyset$, abbiamo $\sum_{s \in S_1} s + \sum_{s \in S_2} s = sum(S)$. Quindi, $\sum_{s \in S_1} s = \sum_{s \in S_2} s = sum(S)/2 = t$.

Pertanto, S_1 è un sottoinsieme di S' tale che $\sum_{s \in S_1} s = t$, il che significa che $\langle S', t \rangle \in$ SubsetSum.

Direzione $\Leftarrow$: Supponiamo che $\langle S', t \rangle \in$ SubsetSum. Allora esiste $S_1 \subseteq S'$ tale che $\sum_{s \in S_1} s = t = sum(S)/2$. Sia $S_2 = S' \setminus S_1 = S \setminus S_1$. Abbiamo $\sum_{s \in S_2} s = sum(S) - \sum_{s \in S_1} s = sum(S) - sum(S)/2 = sum(S)/2 = t$.

Quindi, S_1 e S_2 formano una partizione di S tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s$, il che significa che $\langle S \rangle \in$ SetPartitioning.

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

In conclusione, abbiamo dimostrato che: 1. Entrambi i problemi, SetPartitioning e SubsetSum, sono in NP. 2. SetPartitioning è NP-hard, riducendo SubsetSum a esso. 3. SubsetSum è NP-hard, riducendo SetPartitioning a esso.

Pertanto, entrambi i problemi sono NP-completi.

30 16. Considerate la seguente variante del problema SetPartitioning, che chiameremo QuasiPartitioning: dato un insieme di numeri interi S, stabilire se può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 meno 1. Dimostrare che il problema QuasiPartitioning è NP-completo.

Soluzione. Per dimostrare che QuasiPartitioning è NP-completo, dobbiamo dimostrare che: 1. È in NP. 2. È NP-hard.

Passo 1: QuasiPartitioning è in NP.

Un problema è in NP se esiste un verificatore polinomiale che, dato un certificato, può verificare in tempo polinomiale se una soluzione è corretta.

Per QuasiPartitioning, il certificato è una partizione dell'insieme S in due sottoinsiemi S_1 e S_2. Un verificatore può: 1. Controllare che S_1 e S_2 formino una partizione di S (cioè, $S_1 \cup S_2 = S$ e $S_1 \cap S_2 = \emptyset$). 2. Calcolare la somma degli elementi in S_1 e la somma degli elementi in S_2. 3. Verificare che la somma di S_1 sia uguale alla somma di S_2 meno 1.

Tutte queste operazioni possono essere eseguite in tempo polinomiale rispetto alla dimensione dell'input. Quindi, QuasiPartitioning è in NP.

Passo 2: QuasiPartitioning è NP-hard.

Per dimostrare che QuasiPartitioning è NP-hard, ridurremo SetPartitioning (che è noto essere NP-completo) a QuasiPartitioning.

Sia $\langle S \rangle$ un'istanza di SetPartitioning, dove $S = \{s_1, s_2, \ldots, s_n\}$ è un insieme di numeri interi. Vogliamo costruire un'istanza $\langle S' \rangle$ di QuasiPartitioning tale che $\langle S \rangle \in$ SetPartitioning se e solo se $\langle S' \rangle \in$ QuasiPartitioning.

Costruiamo l'insieme S' come: $S' = S \cup \{1\}$ (aggiungiamo l'elemento 1 a S).

Ora dimostriamo che $\langle S \rangle \in$ SetPartitioning se e solo se $\langle S' \rangle \in$ QuasiPartitioning.

Direzione $\Rightarrow$*:* Supponiamo che $\langle S \rangle \in$ SetPartitioning. Allora esiste una partizione di S in S_1 e S_2 tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s$. Definiamo una partizione di S' come $S'_1 = S_1$ e $S'_2 = S_2 \cup \{1\}$.

Calconiamo le somme:

$$\sum_{s \in S'_1} s = \sum_{s \in S_1} s$$
$$\sum_{s \in S'_2} s = \sum_{s \in S_2} s + 1 = \sum_{s \in S_1} s + 1$$

Quindi, $\sum_{s \in S'_1} s = \sum_{s \in S'_2} s - 1$, il che significa che $\langle S' \rangle \in$ QuasiPartitioning.

Direzione $\Leftarrow$*:* Supponiamo che $\langle S' \rangle \in$ QuasiPartitioning. Allora esiste una partizione di S' in S'_1 e S'_2 tale che $\sum_{s \in S'_1} s = \sum_{s \in S'_2} s - 1$.

Ci sono due casi da considerare: 1. $1 \in S'_1$: In questo caso, sia $S_1 = S'_1 \setminus \{1\}$ e $S_2 = S'_2$. Abbiamo:

$$\sum_{s \in S_1} s = \sum_{s \in S'_1} s - 1 = \sum_{s \in S'_2} s - 1 - 1 = \sum_{s \in S'_2} s - 2 = \sum_{s \in S_2} s - 2$$

Ma questo non dà necessariamente $\sum_{s \in S_1} s = \sum_{s \in S_2} s$.

2. $1 \in S'_2$: In questo caso, sia $S_1 = S'_1$ e $S_2 = S'_2 \setminus \{1\}$. Abbiamo:

$$\sum_{s \in S_1} s = \sum_{s \in S'_1} s = \sum_{s \in S'_2} s - 1 = \sum_{s \in S_2} s + 1 - 1 = \sum_{s \in S_2} s$$

Quindi, nel caso 2, S_1 e S_2 formano una partizione di S tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s$, il che significa che $\langle S \rangle \in$ SetPartitioning.

Ma dobbiamo essere cauti: la riduzione deve funzionare in entrambe le direzioni. Nel caso 1, potremmo non ottenere una partizione valida per SetPartitioning. Per risolvere questo problema, possiamo modificare la nostra costruzione.

Rifaciamo la riduzione:

Sia $\langle S \rangle$ un'istanza di SetPartitioning. Diciamo che $sum(S) = \sum_{s \in S} s$. Notiamo che se $sum(S)$ è dispari, allora $\langle S \rangle \notin$ SetPartitioning (poiché non possiamo dividere un numero dispari in due parti uguali). In questo caso, possiamo restituire un'istanza di QuasiPartitioning che sappiamo essere falsa.

Assumendo che $sum(S)$ sia pari, costruiamo l'insieme S' come: $S' = \{2s \mid s \in S\} \cup \{1\}$ (raddoppiamo ogni elemento di S e aggiungiamo l'elemento 1).

Ora dimostriamo che $\langle S \rangle \in$ SetPartitioning se e solo se $\langle S' \rangle \in$ QuasiPartitioning.

Direzione $\Rightarrow$*:* Supponiamo che $\langle S \rangle \in$ SetPartitioning. Allora esiste una partizione di S in S_1 e S_2 tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s = sum(S)/2$.

Definiamo una partizione di S' come $S'_1 = \{2s \mid s \in S_1\}$ e $S'_2 = \{2s \mid s \in S_2\} \cup \{1\}$.

Calconiamo le somme:

$$\sum_{s \in S'_1} s = \sum_{s \in S_1} 2s = 2 \cdot \sum_{s \in S_1} s = 2 \cdot \frac{sum(S)}{2} = sum(S)$$

$$\sum_{s \in S'_2} s = \sum_{s \in S_2} 2s + 1 = 2 \cdot \sum_{s \in S_2} s + 1 = 2 \cdot \frac{sum(S)}{2} + 1 = sum(S) + 1$$

Quindi, $\sum_{s \in S'_1} s = \sum_{s \in S'_2} s - 1$, il che significa che $\langle S' \rangle \in$ QuasiPartitioning.

Direzione $\Leftarrow$*:* Supponiamo che $\langle S' \rangle \in$ QuasiPartitioning. Allora esiste una partizione di S' in S'_1 e S'_2 tale che $\sum_{s \in S'_1} s = \sum_{s \in S'_2} s - 1$.

Osserviamo che tutti gli elementi in S' tranne 1 sono pari (poiché sono stati raddoppiati dalla costruzione). Quindi, la somma di questi elementi è pari. Pertanto, l'elemento dispari 1 deve appartenere a S'_2 (altrimenti, $\sum_{s \in S'_1} s$ sarebbe dispari e $\sum_{s \in S'_2} s$ sarebbe pari, il che implicherebbe che $\sum_{s \in S'_1} s \neq \sum_{s \in S'_2} s - 1$).

Dato che $1 \in S'_2$, possiamo definire $S_1 = \{s/2 \mid 2s \in S'_1\}$ e $S_2 = \{s/2 \mid 2s \in S'_2\}$. Abbiamo:

$$\sum_{s \in S_1} s = \frac{1}{2} \cdot \sum_{s \in S'_1} s = \frac{1}{2} \cdot (\sum_{s \in S'_2} s - 1) = \frac{1}{2} \cdot (\sum_{s \in S'_2 \setminus \{1\}} s + 1 - 1) = \frac{1}{2} \cdot \sum_{s \in S'_2 \setminus \{1\}} s$$
$$= \frac{1}{2} \cdot \sum_{s \in S_2} 2s = \sum_{s \in S_2} s$$

Quindi, S_1 e S_2 formano una partizione di S tale che $\sum_{s \in S_1} s = \sum_{s \in S_2} s$, il che significa che $\langle S \rangle \in$ SetPartitioning.

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

Combinando i passi 1 e 2, concludiamo che QuasiPartitioning è NP-completo.

31 17. "Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Il problema k-COLOR è il problema di trovare una colorazione di un grafo non orientato usando k colori diversi.

(a) Mostrare che il problema k-COLOR è in NP per ogni valore di k

(b) Mostrare che 2-COLOR è in P

(c) Mostrare che 3-COLOR $\leq_P$ k-COLOR per ogni k > 3

(d) Considerate la seguente variante del problema, che chiameremo Almost3Color: dato un grafo non orientato G con n vertici, stabilire se è possibile colorare i vertici di G con tre colori diversi, in modo che ci siano al più n/2 coppie di vertici adiacenti dello stesso colore. Dimostrare che il problema Almost3Color è NP-completo.

Soluzione. (a) Mostrare che il problema k-COLOR è in NP per ogni valore di k

Per dimostrare che k-COLOR è in NP, dobbiamo mostrare che esiste un verificatore polinomiale che, dato un certificato (in questo caso, una colorazione del grafo), può verificare in tempo polinomiale se è una colorazione valida usando al più k colori.

Sia $G = (V, E)$ un grafo non orientato con n vertici e m archi, e sia $c : V \to \{1, 2, \ldots, k\}$ una funzione di colorazione proposta come certificato. Un verificatore può: 1. Controllare che c assegni a ogni vertice un colore nell'insieme $\{1, 2, \ldots, k\}$. Questo richiede $O(n)$ tempo. 2. Per ogni arco $(u, v) \in E$, verificare che $c(u) \neq c(v)$. Questo richiede $O(m)$ tempo.

Entrambe queste verifiche possono essere eseguite in tempo polinomiale rispetto alla dimensione dell'input. Quindi, k-COLOR è in NP per ogni valore di k.

(b) Mostrare che 2-COLOR è in P

Il problema 2-COLOR è equivalente al problema di determinare se un grafo è bipartito, che è noto essere in P. Un grafo è bipartito se e solo se può essere colorato con 2 colori in modo che nessuna coppia di vertici adiacenti abbia lo stesso colore.

Possiamo risolvere il problema 2-COLOR utilizzando una ricerca in ampiezza (BFS) o una ricerca in profondità (DFS) per verificare se il grafo è bipartito. L'idea è di iniziare da un vertice qualsiasi, assegnargli un colore, e poi colorare alternativamente i vertici adiacenti con l'altro colore. Se durante questo processo incontriamo un vertice già colorato con lo stesso colore di un vertice adiacente, allora il grafo non è bipartito e quindi non è 2-colorabile.

Ecco un algoritmo in pseudocodice:

IsTwoColorable(Grafo $G = (V, E)$):

```
1: Inizializza un array color di dimensione |V| con tutti i valori a -1 (non colorato)
2: for ogni componente connessa del grafo do
3:     Scegli un vertice v non colorato
4:     color[v] ← 0 (assegna il primo colore a v)
5:     Inizializza una coda Q con v
6:     while Q non è vuota do
7:         Estrai un vertice u da Q
8:         for ogni vertice w adiacente a u do
9:             if color[w] = −1 then
10:                color[w] ← 1 − color[u] (assegna il colore opposto)
11:                Aggiungi w a Q
12:            else if color[w] = color[u] then return FALSE (il grafo non è 2-colorabile)
13:            end if
14:        end for
15:    end while
16: end forreturn TRUE (il grafo è 2-colorabile)
```

Questo algoritmo ha una complessità temporale di $O(|V| + |E|)$, che è polinomiale nella dimensione dell'input. Quindi, 2-COLOR è in P.

(c) Mostrare che 3-COLOR $\leq_P$ k-COLOR per ogni k > 3

Per dimostrare che 3-COLOR $\leq_P$ k-COLOR per ogni k > 3, dobbiamo fornire una riduzione polinomiale dal problema 3-COLOR al problema k-COLOR.

Sia $G = (V, E)$ un grafo per il quale vogliamo determinare se è 3-colorabile. Costruiamo un nuovo grafo $G' = (V', E')$ come segue: 1. $V' = V \cup \{v_4, v_5, \ldots, v_k\}$, dove $v_4, v_5, \ldots, v_k$ sono $k - 3$ nuovi vertici. 2. $E' = E \cup \{(v_i, v_j) \mid 4 \leq i < j \leq k\} \cup \{(v, v_i) \mid v \in V, 4 \leq i \leq k\}$.

In altre parole, aggiungiamo $k-3$ nuovi vertici e li connettiamo tra loro formando una cricca (grafo completo). Inoltre, connettiamo ogni vertice originale del grafo a tutti i nuovi vertici.

Ora dimostriamo che G è 3-colorabile se e solo se G' è k-colorabile.

Direzione $\Rightarrow$: Supponiamo che G sia 3-colorabile. Allora esiste una colorazione $c : V \to \{1, 2, 3\}$ tale che per ogni arco $(u, v) \in E$, $c(u) \neq c(v)$. Estendiamo questa colorazione a G' definendo $c'(v) = c(v)$ per ogni $v \in V$ e $c'(v_i) = i$ per ogni $i \in \{4, 5, \ldots, k\}$.

Verifichiamo che c' è una colorazione valida di G': - Per ogni arco $(u, v) \in E$, abbiamo $c'(u) = c(u) \neq c(v) = c'(v)$. - Per ogni arco (v_i, v_j) con $4 \leq i < j \leq k$, abbiamo $c'(v_i) = i \neq j = c'(v_j)$. - Per ogni arco (v, v_i) con $v \in V$ e $4 \leq i \leq k$, abbiamo $c'(v) = c(v) \in \{1, 2, 3\}$ e $c'(v_i) = i \geq 4$, quindi $c'(v) \neq c'(v_i)$.

Quindi, G' è k-colorabile.

Direzione $\Leftarrow$: Supponiamo che G' sia k-colorabile. Allora esiste una colorazione $c' : V' \to \{1, 2, \ldots, k\}$ tale che per ogni arco $(u, v) \in E'$, $c'(u) \neq c'(v)$.

Osserviamo che i vertici $\{v_4, v_5, \ldots, v_k\}$ formano una cricca in G', quindi devono avere colori diversi. Senza perdita di generalità, possiamo assumere che $c'(v_i) = i$ per ogni $i \in \{4, 5, \ldots, k\}$ (altrimenti, possiamo riorganizzare i colori).

Poiché ogni vertice $v \in V$ è adiacente a tutti i vertici $\{v_4, v_5, \ldots, v_k\}$, il colore di v deve essere diverso da $4, 5, \ldots, k$. Quindi, $c'(v) \in \{1, 2, 3\}$ per ogni $v \in V$.

Definiamo una colorazione $c : V \to \{1, 2, 3\}$ come $c(v) = c'(v)$ per ogni $v \in V$. Poiché c' è una colorazione valida di G', per ogni arco $(u, v) \in E \subseteq E'$, abbiamo $c(u) = c'(u) \neq c'(v) = c(v)$. Quindi, c è una colorazione valida di G usando 3 colori.

Pertanto, G è 3-colorabile.

Questa riduzione può essere eseguita in tempo polinomiale rispetto alla dimensione dell'input, dimostrando che 3-COLOR $\leq_P$ k-COLOR per ogni k > 3.

(d) Dimostrare che Almost3Color è NP-completo

Per dimostrare che Almost3Color è NP-completo, dobbiamo dimostrare che: 1. È in NP. 2. È NP-hard.

Passo 1: Almost3Color è in NP.

Per dimostrare che Almost3Color è in NP, mostriamo che esiste un verificatore polinomiale che, dato un certificato (una colorazione del grafo), può verificare in tempo polinomiale se soddisfa le condizioni richieste.

Sia $G = (V, E)$ un grafo non orientato con n vertici, e sia $c : V \to \{1, 2, 3\}$ una funzione di colorazione proposta come certificato. Un verificatore può: 1. Verificare che c assegni a ogni vertice un colore nell'insieme $\{1, 2, 3\}$. Questo richiede $O(n)$ tempo. 2. Contare il numero di archi $(u, v) \in E$ tali che $c(u) = c(v)$. Questo richiede $O(|E|)$ tempo. 3. Verificare che questo numero sia al più $n/2$.

Tutte queste verifiche possono essere eseguite in tempo polinomiale rispetto alla dimensione dell'input. Quindi, Almost3Color è in NP.

Passo 2: Almost3Color è NP-hard.

Per dimostrare che Almost3Color è NP-hard, ridurremo 3-COLOR (che è noto essere NP-completo) a Almost3Color.

Sia $G = (V, E)$ un grafo per il quale vogliamo determinare se è 3-colorabile. Costruiamo un nuovo grafo $G' = (V', E')$ come segue: 1. $V' = V \cup \{v'_1, v'_2, \ldots, v'_m\}$, dove $m = |E| \cdot n^2$ e $v'_1, v'_2, \ldots, v'_m$ sono nuovi vertici. 2. $E' = E \cup \{(v'_i, v'_j) \mid 1 \leq i < j \leq m\}$.

In altre parole, aggiungiamo m nuovi vertici che formano una cricca (grafo completo). I nuovi vertici non sono connessi ai vertici originali del grafo.

Ora dimostriamo che G è 3-colorabile se e solo se G' è Almost3Color.

Direzione $\Rightarrow$: Supponiamo che G sia 3-colorabile. Allora esiste una colorazione $c : V \to \{1, 2, 3\}$ tale che per ogni arco $(u, v) \in E$, $c(u) \neq c(v)$. Estendiamo questa colorazione a G' definendo $c'(v) = c(v)$ per ogni $v \in V$ e $c'(v'_i) = 1$ per ogni $i \in \{1, 2, \ldots, m\}$ (assegniamo lo stesso colore a tutti i nuovi vertici).

Con questa colorazione, tutti gli archi in E hanno estremità di colori diversi, mentre tutti gli archi nella cricca formata dai nuovi vertici hanno estremità dello stesso colore. Il numero totale di archi con estremità dello stesso colore è quindi $\binom{m}{2} = \frac{m(m-1)}{2}$.

Il numero totale di vertici in G' è $n' = n+m$, quindi dobbiamo verificare che $\frac{m(m-1)}{2} \leq \frac{n'}{2} = \frac{n+m}{2}$.

Poiché $m = |E| \cdot n^2$ è molto grande rispetto a n, abbiamo $\frac{m(m-1)}{2} \approx \frac{m^2}{2} \gg \frac{n+m}{2}$. Quindi, questa colorazione non soddisfa la condizione di Almost3Color.

Tuttavia, possiamo modificare la nostra costruzione. Invece di creare una cricca di m nuovi vertici, aggiungiamo n^2 componenti disgiunte, ciascuna essendo un grafo bipartito completo $K_{n,n}$ (cioè, un grafo con n vertici in un insieme e n vertici nell'altro insieme, e ogni vertice del primo insieme è connesso a ogni vertice del secondo insieme).

Con questa nuova costruzione, possiamo colorare ogni grafo bipartito con 2 colori in modo che non ci siano archi con estremità dello stesso colore. Quindi, se G è 3-colorabile, anche G' può essere colorato in modo che non ci siano archi con estremità dello stesso colore, soddisfacendo trivialmente la condizione di Almost3Color.

Direzione $\Leftarrow$: Supponiamo che G' abbia una colorazione $c' : V' \to \{1, 2, 3\}$ tale che al più $n'/2$ archi hanno estremità dello stesso colore.

Se restringiamo c' ai vertici di G, ottenendo una colorazione $c : V \to \{1, 2, 3\}$, e se c non è una colorazione valida di G (cioè, esiste almeno un arco in G con entrambe le estremità dello stesso colore), allora possiamo modificare c' per ottenere una colorazione migliore di G', contraddicendo l'ipotesi che c' è ottimale.

Pertanto, c deve essere una colorazione valida di G, il che significa che G è 3-colorabile.

Questa riduzione può essere eseguita in tempo polinomiale rispetto alla dimensione dell'input, dimostrando che Almost3Color è NP-hard.

Combinando i passi 1 e 2, concludiamo che Almost3Color è NP-completo.