15. 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.

L'equivalenza tra un automa a coda ed una TM deve essere mostrata; quindi, che il linguaggio può essere riconosciuto dall'automa e dunque dalla TM. Si dimostra creando una simulazione che fa capire che l'automa Q si comporti esattamente come la TM M.

La simulazione di Q come M funziona nel modo seguente:

si consideri l'intero nastro come una coda. Ciascun simbolo viene modificato e il movimento del nastro avviene a destra. Quando più di un numero viene pushato nella coda, si shiftano tutti i valori a destra. Se si raggiunge la fine del nastro, allora il simbolo più a sinistra viene considerato e si capisce di essere arrivati alla fine.

La simulazione di M come Q funziona nel modo seguente:

l'alfabeto della macchina M viene espanso aggiungendo un simbolo extra e un simbolo marcatore a sinistra viene inserito nella coda. I simboli sono pushati a sinistra e letti (pop) a destra.

In modo più esteso e formale:
Il linguaggio dell'automa a coda deve essere Turing-riconoscibile.
Per fare ciò deve esistere una macchina TM che simula e riconosce questo linguaggio come una coda.
Immaginiamo quindi di avere un nastro ed una serie di simboli:

- ad ogni operazione di lettura, la macchina scorre e rimuove un simbolo a destra; per fare ciò lo marca e realizza uno shift di ogni elemento a destra (con un simbolo apposito)
- ad ogni operazione di scrittura, si deve capire se:
    - ci si muove a sinistra, allora in questo caso si scrive il simbolo sul nastro e teniamo il simbolo di separazione, che sta a metà e ci aiuta a mantenere la separazione tra una parte e l'altra;
    - attraverso questo, siamo capaci di riconoscere che, quando è il momento di fare il pop, si è mantenuto il simbolo corrispondente all'inizio del nastro
    - ci si muove a destra, allora in questo caso scriviamo il simbolo sul nastro e si fa il push di # per mantenere traccia del corrispondente input

Ad ogni operazione, come descritto meglio per lo spostamento a sinistra, il # serve a mantenere traccia dell'operazione di inserimento e successivamente si fa il pop del simbolo corrispondente all'inizio del nastro (logica di FIFO). Una volta raggiunta la fine del nastro, capiamo se al simbolo più a sinistra corrisponde il #; se ciò avviene siamo in presenza di pila vuota ed effettivamente accetta in qualsiasi momento abbia svuotato tutta la pila, con o senza pila vuota (quindi anche se la testina non ha, alla fine #).
Se all'inizio la pila non dovesse contenere #, sappiamo che il simbolo non sarà quello sotto la testina e cominciamo ad eseguire pop/push; comunque ad un certo punto la pila accetta.
Similmente, ammettiamo che ogni macchina a nastro singolo accetti l'automa a coda.
In questo caso, semplicemente, si aggiunge un simbolo che indica la posizione della testina e un simbolo marcatore; il nastro si muove a destra/sinistra come indicato tra prima e seconda metà di nastro per eseguire pop/push. In entrambi i casi abbiamo un linguaggio Turing-riconoscibile.

16. 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.

Per prima cosa simuliamo una normale macchina di Turing con una macchina di Turing che scrive due volte. La macchina a scrittura doppia simula un singolo passaggio della macchina originale copiando l'intero nastro su una parte nuova del nastro sul lato destro della porzione attualmente utilizzata. La procedura di copiatura opera carattere per carattere, marcando un carattere mentre viene copiato. Questa procedura altera ogni quadrato del nastro due volte: una volta per scrivere il nastro per la prima volta, e ancora per sottolineare che lo è stato copiato. La posizione della testina del nastro della macchina di Turing originale è contrassegnata sul nastro.

Quando si copiano le celle in corrispondenza o vicino alla posizione contrassegnata, il contenuto del nastro è aggiornato secondo le regole della macchina di Turing originale. Per eseguire la simulazione con una macchina scrivibile una volta, si opera come prima, eccetto che ogni cella del nastro precedente è ora rappresentata da due celle. La prima di queste contiene il simbolo del nastro della macchina originale e il secondo è per il simbolo utilizzato nella procedura di copiatura. L'input non viene presentato alla macchina nel formato con due celle per simbolo; quindi, la prima volta che il nastro viene copiato, i segni di copiatura sono messi direttamente sopra il simbolo originale

Più semplicemente:
Partendo dalla macchina originale, si copia l'intero nastro tra le transizioni. In questo nuovo nastro viene aggiunto un simbolo delimitatore (caso TM a nastro semi-infinito) ed un alfabeto espanso per registrare la posizione del nastro quando è stato copiato. In questo caso si hanno due scritture, la prima usata quando si copia il simbolo sul nuovo nastro, la seconda per indicare che sia stato copiato.
Avendo le transizioni uguali, la TM a sola scrittura è equivalente a quelle standard.

17. 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

a) Si considera un linguaggio CF, che viene riconosciuto da un PDA.
Ad esempio vediamo L = $\{a^n b^n c^n \mid n >= 0\}$
Usiamo il PL per linguaggi CF dimostrando che non si tratta di CFL.
Ad esempio, consideriamo, per s=uvxyz, una stringa s= $a^k b^k c^k$, k > 0
Si può dimostrare che eseguendo dei pumping, a causa delle due porzioni non vuote del linguaggio, esso rimane sempre sbilanciato nel numero di 0/1.
Questo non è un CFL e quindi lo 1-PDA non lo riconosce.
Tuttavia, si sfrutta l'idea della macchina multinastro, equivalente a quella a nastro singolo, con le singole pile. Prendendo quindi l'input:

- il PDA si trova nello stato iniziale e, quando viene letta una "a", viene fatto il push sulla prima pila e si avanza nello stato successivo. Se viene letta una b o una c, si rifiuta, perché si deve essere nel format "abc".
- dallo stato dopo, che chiamiamo q_2 se viene letta una "b", si esegue il pop di una "a" e si fa il push di "b" sulla seconda pila. Il numero deve corrispondere; se non rimangono "a" rispetto alle "b" rifiuta. Se si legge ancora "a", siccome qui dovremmo leggere "b", rifiuta
- allo stato dopo, q_3, si legge l'input "c". Qui si fa come prima; se leggiamo "a" o "b", siccome qui dovremmo leggere "c", rifiuta, altrimenti leggiamo "c" e facciamo il pop di "b" dalla seconda pila
- accettiamo se entrambe le pile sono vuote altrimenti si rifiuta

Siccome il PDA a pila singola rifiuta mentre il 2-PDA accetta, allora i 2-PDA sono più potenti degli 1-PDA.

b) I 2-PDA devono essere descritti da una TM, ciò sarà fattibile con una TM multinastro. Infatti, semplicemente, andremo ad emulare le due pile su due nastri di una TM M.
M = Su input x:
1) usa due pile L ed R, che rappresentano il contenuto del nastro. L'idea è di separare queste due pile con un simbolo separatore, simulando l'avanzamento sul nastro.
2) le due pile rappresentano l'andamento della testina; da una parte si fa pop e dall'altra si fa push, in particolare rappresentano a sinistra la parte prima della testina e sulla parte destra la parte successiva.
3) quando riceviamo un input, dobbiamo spostarci sulla seconda pila, facendo push. L'idea è che, essendo una TM deterministica, esegua il push di tutti gli n simboli che ha, eseguendo poi pop. Se è possibile eseguire ciò, la computazione avanza, altrimenti rifiuta
4) dopo aver inserito tutti gli input, l'altro nastro ragiona al contrario, partendo da destra e verificando che i simboli inseriti siano nell'ordine contrario (comincerà così subito ad eliminare, perché la pila toglie il carattere in fondo).

La macchina comunque è libera di spostarsi tra la prima e la seconda pila; tuttavia, sappiamo che se raggiungiamo la pila vuota o al massimo un solo simbolo rimasto (blank), allora accetta, altrimenti rifiuta. In questo modo i 2-PDA sono effettivamente Turing-riconoscibili.

c) L'idea è che entrambe siano multinastro e dunque sono della stessa potenza. Abbiamo appena visto che i 2-PDA sono simulabili da una TM come descritto sopra. L'idea è che il movimento del nastro sia equamente descritto in push (prima pila) e pop (seconda pila).
Sapendo che abbiamo tre pile, semplicemente, l'idea sopra si estende ad una terza pila. Su due nastri implementiamo il movimento a destra e a sinistra; similmente, una macchina a 3 pile dovrà quantomeno implementare 2 pile che agiscono in questo modo.
La TM considera quindi un nastro di input e altri 3 corrispondenti a 3 pile.
Il push richiede che il simbolo venga pushato da un input e tutte e tre le pile si aggiornino con il corrispondente simbolo; come detto prima, una farà il push, una farà il pop, l'altra aggiungerà il simbolo stringa ed un blank, per indicare che è avvenuta una transizione della pila.
Qualora venga fatta una delle due operazioni, uno dei nastri viene implementato come copia delle altre, seguendo le transizioni della pila.
Siccome la TM agisce in parallelo, essendo qui nondeterministica (dato che si può estendere fino a k), possiamo avere k nastri, eseguendo un push, un pop ed uno scorrimento del nastro in un ordine casuale. Non siamo limitati dal numero del nastro; siamo noi che decidiamo l'ordine delle singole azioni. Dunque, si può intuire che i 3-PDA non siano più potenti dei 2-PDA.

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

L'idea è di definire una TM M tale che accetti la stringa A e stia nel linguaggio. Essendo che ha un solo stato, semplicemente, esso sarà contemporaneamente lo stato iniziale e finale.
Immaginando quindi di avere una TM la simulazione procede in questo modo:

- Simula A sul nastro segnando lo stato iniziale
- Dato che è l'unico stato, la macchina avanza e si aspetta di non segnare nessun altro stato ad eccezione di quelli che vanno verso lo stato iniziale, che saranno marcati
- Se la macchina non ha segnato altri stati, *accetta*, altrimenti *rifiuta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.

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

In questo caso abbiamo una CFG che produce una stringa vuota. Optiamo per considerare invece una grammatica che appunto deriva ε:

- se la CFG deriva ε correttamente allora *accetta*
- se la CFG non deriva ε rifiuta

Definiamo M su input <G>, derivando quindi una codifica di una CFG che da un punto di vista costruttivo, sarà G' ed è in forma normale di Chomsky. Questa grammatica accetta tutte le regole di G, come tale se G possiede una regola del tipo S → ε, allora anche G' avrà una regola del tipo S' → ε.

Dal punto di vista di una TM M che riconosce questa grammatica:

- marca lo stato iniziale
- si muove sul nastro seguendo le transizioni della CFG
- se essa arriva con le derivazioni ad una stringa vuota (quindi contiene una regola S → ε), allora *accetta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.

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

Sono entrambe due espressioni regolari tali che una contenga l'altra come sottoinsieme.
A livello pratico significa che una porzione di linguaggio di R è contenuta in S; quindi, entrambe le espressioni hanno almeno una parte dello stesso linguaggio.
Si può quindi idealmente ragionare con una TM N che decide S_{REX} in questo modo:

- trasforma R ed S in due ε-NFA equivalenti B e C
- si esegue N sugli input di R ed S
- se scorrendo tutto il linguaggio di R si arriva, marcando tutti i suoi stati, a coprire almeno una parte di stati di S, anche essi marcati allora *accetta*, altrimenti *rifiuta*

Conviene quindi avere N multinastro, tale da avere gli input da una parte, controllando R su un nastro e vedere sul risultante terzo nastro dove starebbe S se vi è un match del pattern precedente.
Poste queste condizioni, il linguaggio può dirsi *decidibile*.

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

Supponiamo che X sia decidibile e che esista una riduzione computabile.
Per esempio, immaginiamo una funzione di riduzione *f* che sulla seguente TM usa $\underline{A_{TM}}$ come riducibile:
F = "Su input <M, w> dove M è una TM e w è una stringa

1) Si costruisce la seguente TM M:
   M = "Su input x":
   a. Se x = w, allora modifica la porzione di nastro che lo contiene ed esegue M
   b. Se x ≠ w, allora non modifica la porzione di nastro e va in loop
   c. Se M accetta, allora *accetta*
   d. Se M rifiuta, allora *rifiuta*
2) Si restituisce <M'>

Dimostriamo quindi che *f* è una funzione di riduzione da $\underline{A_{TM}}$ a *X*:

- Se <M, w> ∈ $\underline{A_{TM}}$ allora la computazione di M modifica il nastro come voluto dalla funzione complemento. Ciò però rappresenta l'opposto di ciò che stiamo cercando; pertanto f<<M,w> = <M'> non appartiene ad X, perché modifica il nastro
- Se <M, w> non appartiene ad $\underline{A_{TM}}$ allora la macchina non modifica il nastro come vorrebbe correttamente A_{TM}, tuttavia ciò non va più bene per A_{TM}, che abbiamo detto accetta solo ma modifica del nastro.

Pertanto, si ha che $\underline{A_{TM}} \leq_m X$ e sappiamo che $\underline{A_{TM}}$ è indecidibile, pertanto anche X sarà indecidibile.

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

Assumiamo che G sia una TM che ammette un linguaggio Turing-riconoscibile.
A queste condizioni, notiamo anche che la macchina ammette un linguaggio vuoto, simile al test del vuoto, ma complementata. Ciò significa che accetterà tutte le stringhe presenti possibili presenti nel linguaggio, purché siano nell'ordine del linguaggio iniziale e siano complementate.
Se la macchina G però accetta almeno una stringa, la condizione richiesta non vale più, pertanto essa continuerà a scorrere il nastro marcando ogni simbolo terminale, fino a quando non vengono marcate nuove variabili; come per il test del vuoto, se la variabile iniziale non è marcata *accetta*, altrimenti *rifiuta*.
A questo punto si pensa di costruire M tale che agendo sull'input <M>:

- segua l'ordine delle stringhe dell'altra macchina
- per garantire l'ordine, dopo la prima stringa non vengono più marcate stringhe
- sequenzialmente si scorrono tutte le stringhe della lista della TM M

- in questa macchina se si accetta almeno una stringa allora *accetta*, perché la variabile iniziata è stata marcata, complementalmente all'altra macchina, altrimenti *rifiuta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.

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

Qui dobbiamo mostrare due cose:
- A è Turing-riconoscibile
- $A \leq_m \underline{A}$

Partiamo col mostrare che A è ridotto da A; questo perché, se ciò accade, A è Turing-riconoscibile, allora anche A lo è. Essendo che sia il complementare che il normale linguaggio sono Turing-riconoscibili, allora A è decidibile.
Verificando il funzionamento di A, intesa con una TM M:
M = Su input w, dove w è una stringa:
- Eseguiamo M su x. Se tale stringa appartiene ad A, *accetta*
- Se tale stringa non appartiene ad A, *rifiuta*

Abbiamo quindi un decisore per A; vogliamo quindi dimostrare che anche A ha un decisore e che quindi accetti le stringhe opposte.
Come tale, costruiamo una TM W su input <M, w>:
- esegue <M, w> sul nastro
- se trova un input w, *accetta*
- altrimenti *rifiuta*

Ora siccome evidentemente: $n \in A \iff f(n) \in A'$

equivalentemente: $n \notin A \iff f(n) \notin A'$

Avendo dimostrato l'esistenza dei decisori per il complemento e anche per le stringhe normali, vediamo che se esiste una stringa appartenente ad A, equivalentemente la stringa non viene accettata da A e vale anche il contrario. Pertanto, per ogni n, A è decidibile.

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

A è un linguaggio definito da B, ma questo viene definito solo tramite alcune condizioni su A.
Partendo da A dobbiamo dimostrare che presenta una stringa x; se questa stringa è presente nelle transizioni che la TM di riferimento scorrendo trova, allora accetta, altrimenti rifiuta.
B invece viene descritto partendo da una descrizione simile di A, ma ammette anche una stringa "y". Quindi oltre ad ammettere una TM di riferimento partendo dalle transizioni percorse per "x", similmente avremo n passi per "y". La "y" esiste a condizione di "x", pertanto se la TM accetta entrambe le stringhe, accetterà il linguaggio. Essendo A un riconoscitore, di sicuro accetterà almeno "x"; la condizione di accettazione su "y" poco cambia per definizione rispetto alla TM precedente. Se $x \in B$ allora accetterà "x" con un numero sufficientemente lungo di passi tali da poter accettare anche "y", pertanto se $x \notin B$ allora $y \notin B$ per tutte le "y".

25. $A \leq_m B$ e B è un linguaggio regolare implica che A è un linguaggio regolare? Perché si o perché no?

Se $A \leq_m B$ e B è un linguaggio regolare, dobbiamo verificare se A possa essere anch'esso un linguaggio regolare. Però semplicemente si potrebbe avere il caso in cui B comprende già tutte le stringhe di A ed A sia non regolare. A tale scopo facciamo l'esempio di avere $A = \{0^n1^n \mid n >= 0\}$ e $B = \{1\}$. Dato che sono entrambi decidibili, si ha semplicemente la costruzione per entrambi di una TM che simula sul nastro sulla stringa w ed accetta qualora riconosca esattamente il linguaggio posto da A e da B. Notiamo inoltre che B riconosce tutte le stringhe di A e B è regolare; A invece non lo è. Si dimostra quindi che B risolve tutti i problemi di A, ma ciò non implica che A sia necessariamente regolare.

(Il 26 è uguale al 23, si veda quello per riferimento)

27. 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.

Immaginiamo di avere una TM che riceve in input una stringa w tale che sia un simbolo generico oppure 0x. A queste condizioni possiamo immaginare una macchina <M, w> che, attraverso una simulazione, scorre tutto il nastro e trova w, in modo tale che accetti quando la trova, rifiuta altrimenti. A livello di output, si constata che <M, w> ∈ A_{TM}, sempre appartenente al linguaggio.
Se è definita la stringa w sul linguaggio A_{TM} similmente è definito il complemento; è facile dimostrare che con una simulazione che agisce esattamente all'opposto avremmo che <M, w> ∈ $\underline{A_{TM}}$, sempre appartenente al linguaggio.
L'altro caso invece considera una TM che scrive 1 seguita da una stringa ∈ <M, w>. L'idea è totalmente uguale a prima, come descritto, pertanto la funzione di input considera sempre <M, w> ∈ A_{TM} → output di R ∈ $\underline{J}$. Dato che $A_{TM} \leq_m J$, similmente abbiamo $\underline{A_{TM}} \leq_m \underline{J}$.
Essi non possono essere Turing riconoscibili in quanto già A_{TM} non è riconoscitore (solamente in un caso accetterà esattamente la stringa w, che deve essere 0x e si ferma. Ciò è molto limitato da un punto di vista applicativo; il complemento ragiona nella maniera contrapposta). Tanto ci basta per indicare che J e $\underline{J}$ non sono Turing-riconoscibili.
(Il 28 è uguale a questo appena fatto, si veda questo per riconoscimento)

29. 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 quasi Hamiltoniano. 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.

a) Per mostrare che il problema del circuito Toniano è un problema NP-Completo, dobbiamo dimostrare che è NP e tutti i problemi NP sono riducibili in tempo polinomiale a questo problema.
Partiamo in modo semplice: la costruzione del certificato.
Per questo, ci serviranno i vertici, perché abbiamo bisogno di almeno la metà dei vertici, e:
N = Su input <G, s, t> dove G è un grafo quasi-hamiltoniano, "s" è un insieme di nodi e t è il certificato:

- prendiamo una lista di numeri, che rappresenta i nodi del grafo
- si controlla per le ripetizioni nella lista dei vertici, se ne viene trovata una si rifiuta
- si considera che i nodi s attraversino almeno $(p_m - p_i)/2$ vertici, in quanto vogliamo sapere se copriamo almeno la metà di questi
- per ciascun arco tra 1 ed m/2 si controlla se <p_i, p_{i+1}> induttivamente sia un arco di G. Se ciò accade, tutti i test sono passati

La verifica sui vertici viene fatta in tempo polinomiale, l'assegnazione dei vertici può potenzialmente impiegare un tempo più che polinomiale. Dunque, il problema è in NP.

Per completare la prova, dobbiamo dimostrare che è NP-Hard, usando la riduzione di un altro problema NP-completo, in questo caso, sfruttando la riduzione per mezzo del problema del circuito Hamiltoniano.
Partendo da G grafo arbitrario, ci creiamo un grafo H che contiene due copie di G, senza archi in mezzo, chiamate G_1 e G_2.
Voglio dimostrare che G ha un circuito Hamiltoniano se e solo se H ha un circuito Toniano.

- Supponendo G abbia un ciclo hamiltoniano. A queste condizioni G_1 avrà un ciclo contenete almeno la metà dei vertici, pertanto si ha un circuito toniano.

- Nei due sottografi G_1 e G_2, avendo un circuito toniano per ciascuno, si sa che il circuito hamiltoniano coinvolgerà tutti i vertici del sottografo, tale che ciascuno contenga la metà esatta di tutti i vertici presenti. In poche parole, si ha un circuito Toniano solo a condizione che ve ne sia uno hamiltoniano. Dato inoltre che G_1 e G_2 sono copie dei vertici originali, semplicemente, anche G è circuito Hamiltoniano

Questa volta, si parla di vertici e cicli. Esiste quindi un algoritmo in tempo polinomiale che lo riduce, in questo caso, provando tutte le combinazione per accoppiare tutti i vertici nel modo descritto (bruteforce). Dato quindi l'insieme di vertici di partenza, la riduzione è corretta, avendo che l'assegnazione di vertici e archi cresce di un fattore costante, nonostante la loro ricerca sia impiegata in tempo lineare.
A queste condizioni, il problema del circuito Toniano è NP-Completo.

b) Abbiamo dimostrato in precedenza che il circuito Hamiltoniano è in NP. Dato che in uno quasi-Hamiltoniano è la stessa cosa meno un vertice, riscriviamo estensivamente la dimostrazione articolando:
N = Su input <G, s, t> dove G è un grafo quasi-hamiltoniano, "s" è un insieme di nodi e t è il certificato:

- prendiamo una lista di numeri, che rappresenta i nodi del grafo
- si controlla per le ripetizioni nella lista dei vertici, se ne viene trovata una si rifiuta
- si considera che s parta dal primo numero p_1 e t vada fino a $p_m - 1$, dato che non considero l'ultimo vertice nell'attraversamento degli archi
- per ciascun arco tra 1 ed m – 2 si controlla se $<p_i, p_{i+1}>$ induttivamente sia un arco di G. Se ciò accade, tutti i test sono passati

Il ciclo che si genera non include nessun nuovo vertice, infatti ognuno avrà almeno grado 1.
Non sappiamo per certo in quanto tempo polinomiale accada; alcuni sono confronti lineari e sappiamo viene verificato in un tempo al più polinomiale; la selezione stessa potenzialmente è un problema non deterministico. Pertanto, il problema è in NP.
Per completare la prova, dobbiamo dimostrare che è NP-Hard, usando la riduzione di un altro problema NP-completo, in questo caso, sfruttando la riduzione per mezzo del problema del circuito Hamiltoniano.
Dobbiamo, come per il caso precedente, partire da un grafo che chiamiamo G.
Per ogni insieme di vertici, si aggiunge almeno un vertice per ogni iterazione in G
Voglio dimostrare che G ha un circuito hamiltoniano se e solo se H ha un circuito quasi-hamiltoniano.

- Supponiamo che G abbia un circuito hamiltoniano. Aggiungo un vertice senza collegargli nessun arco. Il sottografo certamente contiene un circuito hamiltoniano e raggiungiamo tutti i vertici meno uno. Questo vertice verrà aggiunto successivamente in un ciclo da H, tale che esso colleghi tutti i vertici e sia considerabile sottoinsieme del precedente (quindi H contiene un ciclo con tutti i vertici considerando anche il vertice tralasciato da G).
- A queste condizioni, G contiene per forza tutti i vertici ed H contiene a sua volta tutta una serie di vertici più quello che non fa parte del ciclo di G. Le due condizioni, come richiesto, vanno di pari passo, affinché G sia hamiltoniano se e solo se H è quasi hamiltoniano.

Dato inoltre che H è praticamente una copia di G, ma solo dei vertici utili (compreso il vertice mancante per G), si ha la correttezza della prova. Inoltre, affermiamo che dato l'insieme di vertici di partenza, la riduzione è corretta, avendo che l'assegnazione di vertici e archi cresce di un fattore costante, nonostante la loro ricerca sia impiegata in tempo lineare. Quindi, anche la riduzione è corretta, dato che il vertice è isolato.
Alla luce di tutti questi fatti, anche il problema del circuito quasi-Hamiltoniano è NP-Completo.

30. Considera i seguenti problemi:

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

(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.

a) La dimostrazione di un verificatore V per SUBSET-SUM è articolata in questo modo:
Il certificato è il sottoinsieme stesso, dato che abbiamo un insieme di numeri e dobbiamo poi capire se appartengono all'insieme dei numeri completo.
L'idea di verificatore in tempo polinomiale è la seguente per V:
V = Su input <<S, t>, c>, tale che c è una collezione di numeri (certificato), S è l'insieme di tutti i numeri, c è il certificato:

a. Controlla se c è una collezione di numeri la cui somma è t
b. Controlla se la somma S contenga tutti i numeri
c. Se entrambi passano, *accetta*, altrimenti *rifiuta*

Non sappiamo per certo in quanto tempo polinomiale accada; la selezione stessa sugli insiemi e la determinazione dei due sottoinsiemi verificatori potenzialmente è un problema non deterministico.
Quindi SUBSET-SUM è un problema in NP.

La dimostrazione di un verificatore V per SETPARTITIONING è articolata in questo modo:
Il certificato è il sottoinsieme stesso, dato che abbiamo un insieme di numeri e dobbiamo poi capire se appartengono all'insieme dei numeri interi S.
L'idea di verificatore in tempo polinomiale è la seguente per V:
V = Su input <<S>, c>, tale che c è una collezione di numeri (certificato), S è l'insieme di tutti i numeri:

a. Controlla se c è una collezione di numeri la cui somma è t
b. Controlla se la somma S contenga tutti i numeri di S_1 ed S_2
c. Se entrambi passano, *accetta*, altrimenti *rifiuta*

La verifica è operata in tempo polinomiale, quindi SETPARTITIONING è un problema in NP.

b) Per rappresentare la struttura di SETPARTITIONING/SP come problema NP-Hard, dobbiamo provare che tutti i linguaggi in NP sono riducibili in tempo polinomiale con SUBSET-SUM/SS. Dobbiamo quindi sfruttare una struttura che permetta di rappresentare il problema. Verificando il problema in sé, SS rappresenta una singola istanza di SP, che invece considera due sottoinsiemi (SS ne considera uno solo), tali che la somma dei numeri in S1 sia uguale alla somma dei numeri in S2.
Considerando che abbiamo due insiemi, necessariamente S1 compone una metà di questa somma; dunque per ottenere la precedente basterà sfruttare come numero t=1/2 tale che tutti i numeri "x" che sono in S sommino tutti allo stesso valore.
Voglio quindi dimostrare che SP $\leq_p$ SS
<S> → <S', t>
Se esiste quindi un insieme di numeri in S che somma a t, allora i rimanenti numeri in S sommano a "s – t". Pertanto, esiste una partizione in grado di sommare ad "s – t".
Sapendo che S1 rappresenta ½ di questa partizione, prendiamo due insiemi tali che la somma sia "s – t". Pertanto, uno dei due insiemi in SP contiene almeno s – 2t numeri. Rimuovendo questo numero, otterremo un set di numeri la cui somma è *t* e sono tutti in S (quindi una partizione è almeno la metà della seconda e si arriva allo stesso numero). Matematicamente avremo:
Somma(S_1) + Somma(S_2) = Somma(S)

che significa che
Somma(S_1) + Somma(S_2) = (1/2)*Somma(S)

L'idea quindi è che una partizione sia metà dell'altra, infatti in SP ci sono due insiemi uguali la cui somma è t, si può dire che 2S - w ∈ S. Ad ogni nuovo elemento aggiunto, solo uno di questi due insiemi conterrà (dato che una è la metà dell'altra), la nuova stringa; poi, dato che nell'altro non vengono aggiunti valori, anche per SP la somma sarà sempre uguale a t.
Se ciò accade, quindi, si genera una partizione metà dell'altra e a questa aggiungo numeri che rimangono sempre nell'insieme per somma; sulla base di questa condizione si ritorna SI, altrimenti si ritorna NO. La riduzione è corretta, in quanto si considerano per istanze buone, SP rappresenta un insieme solo di SS e andrà bene per forza. Dato che la ricerca dell'insieme impiega potenzialmente un tempo più che polinomiale per ricerca continua (bruteforce), siamo in NP ed il problema è NP-Completo.

c) Per rappresentare la struttura di SUBSET-SUM/SS come problema NP-Hard, dobbiamo provare che tutti i linguaggi in NP sono riducibili in tempo polinomiale con SETPARTITIONING/SP.
Quindi, abbiamo a che fare:

- per SS un unico insieme tale che la somma sia t
- per SP due insiemi tali che la somma complessiva del primo sia uguale a quella del secondo

Per rappresentare SS prendiamo quindi due insiemi e affermiamo che per entrambi, la somma deve essere uguale a t. Definiamo quindi per l'insieme S_1 un insieme S_{new} a 2t – s, dove s rappresenta la somma degli elementi di S_1. In questo modo, S_1 e S_{new} rappresentano due partizioni che sommano a *t*.
Usando la riduzione, se S_{new} può essere di volta in volta partizionato in una serie di insiemi che contengono 2t – s, significa che avremo sempre che S_{new} è la somma di tutti i numeri interi delle due partizioni che contiene tutti i numeri delle due sottopartizioni.
Quindi, le istanze buone, intese come somme delle sottopartizioni, vengono sommate ad una sola (SS), ma è tale anche che la somma delle due sia spezzabile in due partizioni separate (SS), che è quello che vogliamo realizzare in tempo P.

31. 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.

(a) Dimostrare che il problema QUASIPARTITIONING è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

Il certificato è dato da una coppia di insiemi di numeri interi S_1, S_2. Per verificarlo occorre controllare che rispetti le seguenti condizioni:

- i due insiemi $S1, S_2$ devono essere una partizione dell'insieme S;
- la somma degli elementi in S_1 deve essere uguale alla somma degli elementi in S_2 meno 1.

Entrambe le condizioni si possono verificare in tempo polinomiale.

Dimostrare che il problema QUASIPARTITIONING è NP-hard, mostrando come si può risolvere il problema SETPARTITIONING usando il problema QUASIPARTITIONING come sottoprocedura.

Dimostrare che QUASIPARTITIONING è NP-hard, usando SETPARTITIONING come problema di riferimento richiede diversi passaggi:

1. Descrivere un algoritmo per risolvere SETPARTITIONING usando QUASIPARTITIONING come subroutine. Questo algoritmo avrà la seguente forma: data un'istanza di SETPARTITIONING, trasformala in un'istanza di QUASIPARTITIONING, quindi chiama l'algoritmo magico black-box per QUASIPARTITIONING.
2. Dimostrare che la riduzione è corretta. Ciò richiede sempre due passaggi separati, che di solito hanno la seguente forma:
   - Dimostrare che l'algoritmo trasforma istanze "buone" di SETPARTITIONING in istanze "buone" di QUASIPARTITIONING.
   - Dimostrare che se la trasformazione produce un'istanza "buona" di QUASIPARTITIONING, allora era partita da un'istanza "buona" di SETPARTITIONING.

3. Mostrare che la riduzione funziona in tempo polinomiale, a meno della chiamata (o delle chiamate) all'algoritmo magico black-box per QUASIPARTITIONING. (Questo di solito è banale.)

Una istanza di SETPARTITIONING è data da un insieme S di numeri interi da suddividere in due. Una istanza di QUASIPARTITIONING è data anch'essa da un insieme di numeri interi S'. Quindi la riduzione deve trasformare un insieme di numeri S input di SETPARTITIONING in un altro insieme di numeri S' che diventerà l'input per la black-box che risolve QUASIPARTITIONING.

Come primo tentativo usiamo una riduzione che crea S' aggiungendo un nuovo elemento a S di valore 1, e proviamo a dimostrare che la riduzione è corretta:

$\Rightarrow$ sia S un'istanza buona di SETPARTITIONING. Allora è possibile partizionare S in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Se aggiungiamo il nuovo valore 1 ad S_1 otteniamo una soluzione per QUASIPARTITIONING, e abbiamo dimostrato che S' è una istanza buona di QUASIPARTITIONING.

$\Leftarrow$ sia S' un'istanza buona di QUASIPARTITIONING. Allora è possibile partizionare S' in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 meno 1. Controlliamo quale dei due sottoinsiemi contiene il nuovo elemento 1 aggiunto dalla riduzione:

- se $1 \in S_1$, allora se tolgo 1 da S_1 la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2. Abbiamo trovato una soluzione per SETPARTITIONING con input S.
- se $1 \in S_2$, allora se tolgo 1 da S_2 quello che succede è che la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2 meno 2. In questo caso abbiamo un problema perché quello che otteniamo *non è una soluzione di* SETPARTITIONING*!*

Quindi il primo tentativo di riduzione non funziona: ci sono dei casi in cui istanze cattive di SETPARTITIONING diventano istanze buone di QUASIPARTITIONING: per esempio, l'insieme $S = \{2, 4\}$, che non ha soluzione per SETPARTITIONING, diventa $S' = \{2, 4, 1\}$ dopo l'aggiunta dell'1, che ha soluzione per QUASIPARTITIONING: basta dividerlo in $S_1 = \{4\}$ e $S_2 = \{2, 1\}$.

Dobbiamo quindi trovare un modo per "forzare" l'elemento 1 aggiuntivo ad appartenere ad S_1 nella soluzione di QUASIPARTITIONING. Per far questo basta modificare la riduzione in modo che S' contenga tutti gli elementi di S *moltiplicati per* 3, oltre all'1 aggiuntivo. Formalmente:

$$S' = \{3x \mid x \in S\} \cup \{1\}.$$

Proviamo a dimostrare che la nuova riduzione è corretta:

$\Rightarrow$ sia S un'istanza buona di SETPARTITIONING. Allora è possibile partizionare S in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Se moltiplichiamo per 3 gli elementi di S_1 ed S_2, ed aggiungiamo il nuovo valore 1 ad S_1 otteniamo una soluzione per QUASIPARTITIONING, e abbiamo dimostrato che S' è una istanza buona di QUASIPARTITIONING.

$\Leftarrow$ sia S' un'istanza buona di QUASIPARTITIONING. Allora è possibile partizionare S' in due sottoinsiemi S_1 ed S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 meno 1. Vediamo adesso che, a differenza della riduzione precedente, non è possibile che $1 \in S_2$: se così fosse, allora se tolgo 1 da S_2 quello che succede è che la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2 meno 2. Tuttavia, gli elementi che stanno in S_1 ed S_2 sono tutti quanti multipli di 3 (tranne l'1 aggiuntivo). Non è possibile che due insiemi che contengono solo multipli di 3 abbiano differenza 2. Quindi l'1 aggiuntivo non può appartenere a S_2 e deve appartenere per forza a S_1. Come visto prima, se tolgo 1 da S_1 la somma degli elementi S_1 diventa uguale alla somma dei numeri in S_2. Abbiamo trovato una soluzione per SETPARTITIONING con input S.

In questo caso la riduzione è corretta. Per completare la dimostrazione basta osservare che per costruire S' dobbiamo moltiplicare per 3 gli n elementi di S ed aggiungere un nuovo elemento. Tutte operazioni che si fanno in tempo polinomiale.

32. "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.

a) Per dimostrare che il problema è in NP va dimostrato che è possibile avere un verificatore del problema che lavora in tempo polinomiale. Questo è possibile perché dato un grafo di dimensioni finite, basterà passare tutti i nodi del grafo e comparare il loro colore a quello dei nodi adiacenti. Se per nessun nodo il colore sarà lo stesso dei nodi adiacenti, allora il verificatore accetterà, altrimenti rifiuterà. Quindi, avendo un grafo G(V,E) con una serie di colori $\{c_1, c_2, \ldots c_k\}$ con ciascuno assegnato ad un rispettivo colore, per ogni arco {u, v} nel grafico G si verifica che il colore c(u) != c(v). Quindi il tempo di esecuzione sarà dato da (V + E) il quale sarà *k* come upper bound, perché si aggiunge sempre un vertice a quelli originali. Quindi, il problema si estende ad un limite superiore di 2 (2-Color), 3 (3-Color) e seguenti facilmente.

b) Creiamo un algoritmo che esegue in tempo polinomiale.
Supponiamo che i nostri due colori siano il rosso e il blu. L'idea chiave è quella di colorare, se possibile, ogni componente connesso del grafico. Scegliamo un vertice v_0 nel primo componente connesso e lo coloriamo di rosso. Se v_0 è collegato a $v_1, v_2, \ldots, v_k$, si colora questi altri vertici di blu e si ripete. Una volta colorato un vertice, si ha una sola opzione per il colore dei suoi vicini, basta solo che abbia un colore diverso; quindi o si colorano tutti i vertici diversamente in tempo polinomiale 2 a 2 oppure si ha un conflitto, quindi due colori uguali; se ciò accade, l'output è "NO". Se l'algoritmo colora correttamente tutti i componenti collegati, l'output sarà "SÌ". Partendo dal grafico di input G (V,E), dimostriamo che ci si impiega un tempo quadratico, come detto a due a due per i vertici, e quindi il tempo impiegato è $O(|V|^2)$.

c) Si vuole estendere il problema 3-Color a k-color. A queste condizioni, partendo da G si aggiunge una serie di vertici ad h, tale che tutti gli archi vengano così collegati.
Vogliamo quindi provare che G sia 3-colorabile se e solo se H è k-colorabile.
- Supponendo che G sia 3-colorabile, decidiamo di assegnare ad ogni vertice dei colori arbitrari, ad esempio, "rosso", "giallo" e "blu". Ora, abbiamo per ogni nuovo vertice, progressivamente aggiunto, un nuovo colore, possibile nel collegamento dei vertici, tale da arrivare a k.
L'idea di base dice che:
- se abbiamo due vertici, *u* e *v* con colori diversi, allora li colleghiamo
- se non hanno due colori diversi, si ha in conflitto
Dato che ogni vertice avrà quantomeno un nuovo colore (perché come detto dobbiamo arrivare a *k*), allora H sarà H-colorabile, dato che si aggiunge un nuovo vertice con un colore ogni volta per G.

Dimostriamo anche che se H è k-colorabile, allora G è 3-colorabile.
Abbiamo una k-colorazione, quindi ogni arco composto da due vertici contiene una coppia di colori diversi; ogni vertice a loro adiacente, a loro volta, conterrà almeno un colore diverso. Pertanto, partendo da k, possiamo progressivamente eliminare un vertice e, così facendo, verrà tolto almeno un colore. Così si procede fino ad arrivare ai 3 colori di partenza.
Dato sempre l'esame dei vertici condotto in tempo polinomiale partendo da G ed arrivando ad H, si ha un'applicazione brute-force, quindi applicata in tempo polinomiale.

d) Per dimostrare che ALMOST3COLOR è NP-completo, si deve dimostrare che è sia NP che NP-HARD. Partendo da N = Su input <G, s, t> dove G è un grafo quasi-hamiltoniano, "s" è un insieme di nodi e t è il certificato:

- si verifica che ogni insieme di nodi sia collegato ai rispettivi archi
- si verifica che tra le coppie di vertici collegate ve ne siano dello stesse colore
- se abbiamo almeno n/2 coppie di colori sul problema, allora si ha in output *SI,* altrimenti *NO*.

Dato che la verifica viene fatta in tempo polinomiale, il problema è in NP.
Vogliamo ora dimostrare che si tratta di un problema NP-Hard, usando ALMOST3COLOR come istanza di 3COLOR, tramite una riduzione.
Sapremo quindi che 3COLOR possiede un numero *n* di vertici coperti da almeno n/2 vertici di ALMOST3COLOR; semplicemente, supponendo che per ogni vertice di ALMOST3COLOR si abbia un collegamento, progressivamente, si assegnano tutti i colori ad ogni coppia di vertici separata da un arco. Così facendo abbiamo colorato tutti i vertici di ALMOST3COLOR.
Semplicemente si nota che 3COLOR rappresenta esattamente il doppio dei vertici dell'altro; dunque, eseguendo tutta l'operazione fino a tutti i vertici raggiungibili, è facile dimostrare come 3COLOR risolva ALMOST3COLOR.
Avendo tutte le coppie collegate, sappiamo che 3COLOR contiene le sole istanze buone del precedente, in quanto sono tutte le coppie di vertici collegati, che rispetto a 3COLOR sarebbero correttamente n/2 coppie.
Il tempo richiesto è polinomiale e correttamente viene risolto il problema.

oppure

Per dimostrare che ALMOST3COLOR è NP-completo, si deve dimostrare che è sia NP che NP-HARD.
È NP perché esiste un certificato del sì che si può verificare in tempo polinomiale.
Il certificato è dato da una coppia di insiemi di vertici e di archi.
Per verificarlo, occorre controllare che:
1. Il numero di archi, i cui vertici sono di colore uguale, sia al più 8939
2. Il numero di colori usati in tutto il grafo sia 3
Entrambe le condizioni si possono verificare in tempo polinomiale.
È NP-HARD, perché si può mostrare come si può risolvere il problema 3COLOR usando il problema ALMOST3COLOR come sotto procedura.
1- ALGORITMO
Dato un grafo G (che è istanza di 3COLOR), costruiamo in tempo polinomiale un nuovo grafo, che è istanza di ALMOST3COLOR, aggiungendo 8939 x (2 vertici dello stesso colore e 1 arco che li collega). Il nuovo grafo contiene 8939 coppie di vertici uguali.
2- DIMOSTRAZIONE
=>Sia S un'istanza buona di 3COLOR.
Allora, è possibile aggiungere 8939 coppie di vertici uguali, collegati da un arco (2 a 2). La parte del grafo senza colori uguali adiacenti rimane intatta (S). Vengono soddisfatti 3COLOR e ALMOST3COLOR.
<= Sia S' un'istanza buona di ALMOST3COLOR. Una volta individuate tutte le coppie di vertici, si devono contare quali sono i colori adiacenti uguali. Dato che le coppie con colori uguali sono state aggiunte a quelle diverse, quelle rimanenti sono 3 COLOR.
3- TEMPI POLINOMIALI
Dato che la costruzione del nuovo grafo implica l'aggiunta di 8939 coppie di vertici (collegate da un arco), il tempo richiesto è polinomiale

Esercizi da vecchi file di seconda parte in preparazione (anni scorsi):

15. Sia A il linguaggio che contiene solo ed unicamente la stringa s,

$$s = \begin{cases} 0 & \text{se la vita non sarà mai trovata su Marte} \\ 1 & \text{se un giorno la vita sarà trovata su Marte} \end{cases}$$

A è un linguaggio decidibile? Giustificare la risposta. Ai fini del problema, assumere che la questione se la vita sarà trovata su Marte ammetta una risposta non ambigua Si/No.

Avendo un linguaggio che contiene una sola stringa, quindi 0/1.
Ciò è semplice da dimostrare con una TM; infatti la macchina o prende 0 oppure 1, altrimenti non accetta il proprio linguaggio. Per uno dei due è possibile costruire un decisore, che accetti il primo o il secondo, ovviamente non entrambi. Per quanto logicamente non si sappia la risposta certa, è comunque ammesso un decisore che accetterebbe due linguaggi finiti {0} oppure {1}.
Dunque, A è decidibile.