2. (12 punti) Una variabile A in una grammatica context-free G è *persistente* se compare in ogni derivazione di ogni stringa w in $L(G)$. Data una grammatica context-free G e una variabile A, considera il problema di verificare se A è persistente.

(a) Formula questo problema come un linguaggio $PERSISTENT_{\text{CFG}}$.

(b) Dimostra che $PERSISTENT_{\text{CFG}}$ è decidibile.

PERSISTENT → CFG che produce sempre una variabile (TM che lo fa)

Conclusioni (discorsive) con Se e solo se (decidibile)

TM che dimostra che il linguaggio è decidibile (quindi = esiste un decisore) ⇒

PERSISTENT produce una CFG con A che è persistente (quindi = proprietà del linguaggio è decidibile)

(⇒) Se esiste una TM che produce PERSISTENT, significa che ogni passo prendendo in input $\langle G, A \rangle$, per ogni regola di G, produce una variabile appartenente alla regola A (entro un certo numero di passi) – come dire, sicuramente ad ogni passo produci A, perché senno la TM non termina e rifiuta. Se produce tutte le regole accetta e funziona SOLO SE la G è persistente.

(⇐) Se CFG è persistente, allora esiste una TM in grado di processarne la grammatica G, in grado di produrre regole persistenti. Essendo che le regole sono in numero finito, tutte le mosse della TM calcolano, essendo un oggetto decidibile, ad ogni passaggio le variabili a appartenenti ad A, comparendo in ogni mossa della TM, altrimenti rifiuta. Questo avviene SOLO SE esiste la TM, altrimenti non accetta finché PERSISTENT non ha A che è persistente.

Problemi decidibili (logica: https://stem.elearning.unipd.it/pluginfile.php/783471/mod_resource/content/3/14-decidibili.pdf)

$$A_{DFA}, A_{REX}, A_{NFA}, EQ_{DFA}, A_{CFG}, E_{CFG}$$

DFA/NFA = oggetto decidibile

REX = espressione regolare (regex)

EQ_DFA = decidibile perché due DFA sotto sono decidibili

E_CFG = problema del vuoto per CFG, decidibile perché lo sono le CFG

Fine esercizi (discorsiva) - Se e solo se (indecidibile)

Se l'input appartiene ad una TM che è indecidibile (quindi = A è indecidibile) ⇒

Variante proposta dall'esercizio è indecidibile (quindi = B è indecidibile)

$$A_{TM} \leq_m PERSISTENT_{CFG}$$

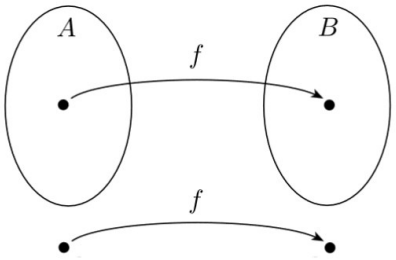
$$\langle M, w \rangle \in A_{TM} \Rightarrow M' \in PERSISTENT_{CFG}$$

- Se la TM si ferma sull'input w, allora esiste una TM M' che processa tutte le regole e verifica l'appartenenza di a ad ogni regola. Se così non fosse, M' (quella che fa PERSISTENT) non è in grado di elaborare tutte le regole, perché rifiuta o va in loop. Se w è un input persistente per M, allora M' accetta, altrimenti no.
- Se M' accetta PERSISTENT, allora esiste una TM in grado di elaborare questo input, verificando che sia persistente. M' processa infatti un input x che ha la condizione di essere persistente dato che M si ferma accettando w e ciò avviene se ogni regola è persistente. Se M' accetta, allora ogni regola di M è persistente.

Il se e solo se deriva dalla definizione di riduzione:

Un linguaggio A è riducibile mediante funzione al linguaggio B ($A \leq_m B$), se esiste una funzione calcolabile $f : \Sigma^* \mapsto \Sigma^*$ tale che

per ogni w : $w \in A$ se e solo se $f(w) \in B$



f è la riduzione da A a B

Problemi indecidibili (usabili con riduzione):

- A_{TM} (A per Arresto = Problema dell'arresto = Halting problem = Il mio programma si ferma SEMPRE su qualsiasi input nonostante le sue condizioni)

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM che accetta la stringa } w\}$$

Senso di utilizzo: $A_{TM} \leq Problema$

- $\overline{A_{TM}}$

Senso di utilizzo: $\overline{A_{TM}} \leq Problema$

In questo caso, quando la macchina non accetta, si esegue il complemento e quando vale la condizione del problema, rifiuta o va in loop

- E_{TM} (E per Empty = Problema del vuoto = "Non c'è mai la stringa nel linguaggio)

$$E_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$$

- $\overline{E_{TM}}$

Senso di utilizzo: $\overline{E_{TM}} \leq Problema$

In questo caso, quando la macchina vede che esiste almeno un $w \in L$ (cioè, fa l'opposto del vuoto), si esegue il complemento e quando vale la condizione del problema, rifiuta o va in loop.

$$EQ_{TM} = \{\langle M_1, M_2 \mid M_1, M_2 \text{ TM tali che } L(M_1) = L(M_2)\}$$

Aka, usare due TM che portano alla stessa cosa → Ma sono entrambe indecidibili!

Senso di utilizzo: $EQ_{TM} \leq Problema$

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che si ferma su input } w\}$$

Senso di utilizzo: $HALT_{TM} \leq Problema$

Se e solo se (NP-Hard)

Se l'input appartiene ad un problem che è NP-Hard (quindi = A è NP-Hard) ⇒

Variante proposta dall'esercizio è NP-Hard (quindi = B è NP-Hard)

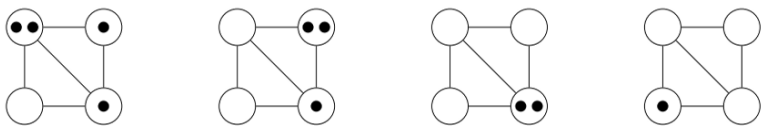
Classi di problemi NP:

- SAT
  - Clausole booleane che valutano tutte ad 1
- CircuitSAT
  - Clausole booleane in circuito che valutano tutte ad 1
- 3SAT
  - 3 Clausole booleane che valutano tutte ad 1
  - Variante K-SAT = K clausole booleane
- Independent Set
  - Massimo insieme di archi senza vertici in comune
- SetPartitioning
  - Due sottoinsiemi diversi tali che il primo ha la stessa somma del secondo
- Vertex Cover
  - Minimo numero di vertici che copre tutto il grafo
- Hamilton (Ciclo Hamiltoniano)
  - Ciclo che attraversa tutto il circuito almeno una volta per tutti i vertici
- Colorare il grafo
  - Attraversare tutto il grafo garantendo che tutti i vertici abbiano colore diverso
  - 3-COLOR
    - Un vertice è collegato ad altri, ciascuno con colore diverso
    - Per un massimo di 3 colori
  - K-COLOR
    - Per un massimo di K colori

Senso:

$$SAT \leq_m PROBLEMA$$

3. Pebbling è un solitario giocato su un grafo non orientato G, in cui ogni vertice ha zero o più ciottoli. Una mossa del gioco consiste nel rimuovere due ciottoli da un vertice v e aggiungere un ciottolo ad un vertice u adiacente a v (il vertice v deve avere almeno due ciottoli all'inizio della mossa). Il problema PEBBLEDESTRUCTION chiede, dato un grafo $G = (V, E)$ ed un numero di ciottoli $p(v)$ per ogni vertice v, di determinare se esiste una sequenza di mosse che rimuove tutti i sassolini tranne uno.

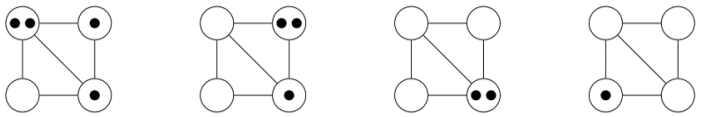


Una soluzione in 3 mosse di PEBBLEDESTRUCTION.

Dimostra che PEBBLEDESTRUCTION è NP-hard usando il problema del Circuito Hamiltoniano come problema NP-hard noto (un circuito Hamiltoniano è un ciclo che attraversa ogni vertice di G esattamente una volta).

$$HAM \leq_m PebbleDestruction$$

3. Pebbling è un solitario giocato su un grafo non orientato G, in cui ogni vertice ha zero o più ciottoli. Una mossa del gioco consiste nel rimuovere due ciottoli da un vertice v e aggiungere un ciottolo ad un vertice u adiacente a v (il vertice v deve avere almeno due ciottoli all'inizio della mossa). Il problema PebbleDestruction chiede, dato un grafo $G = (V, E)$ ed un numero di ciottoli $p(v)$ per ogni vertice v, di determinare se esiste una sequenza di mosse che rimuove tutti i sassolini tranne uno.



Una soluzione in 3 mosse di PebbleDestruction.

Dimostra che PebbleDestruction è NP-hard usando il problema del Circuito Hamiltoniano come problema NP-hard noto (un circuito Hamiltoniano è un ciclo che attraversa ogni vertice di G esattamente una volta).

NP:

1) Verificatore/Certificato
2) Usa una riduzione

Definizioni generali:

1. Certificato: Un certificato è una stringa (o struttura dati) che serve come "prova" o "testimone" per dimostrare che una certa istanza di un problema appartiene a una classe di complessità. Per problemi in NP, un certificato è una prova che può essere verificata in tempo polinomiale.
2. Verificatore: Un verificatore è un algoritmo che prende in input un'istanza del problema e un certificato, e controlla in tempo polinomiale se il certificato dimostra correttamente che l'istanza è una soluzione valida del problema.

Per il problema PebbleDestruction specifico:

Certificato: Una sequenza di mosse che rimuove tutti i ciottoli dal grafo.

Verificatore: Un algoritmo che prende in input:

1. Il grafo G con la configurazione iniziale dei ciottoli
2. Il numero massimo di mosse k
3. La sequenza di mosse proposta come certificato

L'algoritmo verificherebbe che:

1. Ogni mossa è legale (rimuove 2 ciottoli da un vertice e ne aggiunge 1 a un vertice adiacente)
2. Il numero di mosse non supera k
3. Alla fine della sequenza, tutti i ciottoli sono stati rimossi

Il verificatore restituirebbe "accetta" se tutte queste condizioni sono soddisfatte, "rifiuta" altrimenti.

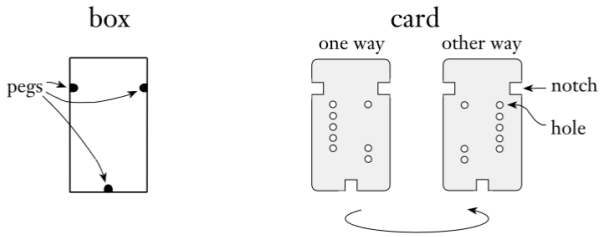
Per dimostrare che PebbleDestruction è NP-hard, dobbiamo effettuare una riduzione da un problema noto come NP-hard (in questo caso, il Circuito Hamiltoniano) a PebbleDestruction. Ecco un'idea di come potrebbe funzionare questa dimostrazione:

1. Partiamo da un'istanza del problema del Circuito Hamiltoniano su un grafo G.
2. Costruiamo un'istanza equivalente del problema PebbleDestruction come segue: a) Creiamo un nuovo grafo G' identico a G. b) Per ogni vertice in G', posizioniamo 2 ciottoli. c) Aggiungiamo un ciottolo extra su un vertice arbitrario.
3. Dimostriamo che G ha un circuito hamiltoniano se e solo se possiamo rimuovere tutti i ciottoli da G' in esattamente |V| mosse (dove |V| è il numero di vertici).

Idea della dimostrazione:

- Se G ha un circuito hamiltoniano:
    - Seguiamo il circuito in G'.
    - Ad ogni passo, rimuoviamo 2 ciottoli dal vertice corrente e ne aggiungiamo 1 al prossimo.
    - Dopo |V| passi, avremo rimosso tutti i ciottoli tranne l'ultimo.
- Se possiamo rimuovere tutti i ciottoli da G' in |V| mosse:
    - Ogni mossa deve rimuovere 2 ciottoli da un vertice e aggiungerne 1 a un adiacente.
    - Per rimuovere tutti i ciottoli, dobbiamo visitare ogni vertice esattamente una volta.
    - La sequenza di vertici visitati forma un circuito hamiltoniano in G.
4. Questa riduzione può essere effettuata in tempo polinomiale.
5. Quindi, se potessimo risolvere PebbleDestruction in tempo polinomiale, potremmo anche risolvere il problema del Circuito Hamiltoniano in tempo polinomiale.

7.28 You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box (i.e., every hole position is blocked by at least one card that has no hole there). Let $PUZZLE = \{\langle c_1, \ldots, c_k \rangle \mid$ each c_i represents a card and this collection of cards has a solution$\}$. Show that *PUZZLE* is NP-complete.



Definizioni generali:

1. Certificato: Un certificato è una stringa (o struttura dati) che serve come "prova" o "testimone" per dimostrare che una certa istanza di un problema appartiene a una classe di complessità. Per problemi in NP, un certificato è una prova che può essere verificata in tempo polinomiale.
2. Verificatore: Un verificatore è un algoritmo che prende in input un'istanza del problema e un certificato, e controlla in tempo polinomiale se il certificato dimostra correttamente che l'istanza è una soluzione valida del problema.

Certificato - Insieme delle carte

Verificatore - Controlla che:

1. Tutte le carte nel certificato siano valide.
2. I buchi coperti da ogni carta non siano già stati coperti.
3. I pioli della scatola corrispondano alle tacche delle carte nell'orientamento specificato.
4. Alla fine, tutti i buchi siano coperti.

Il verificatore restituisce True se il certificato rappresenta una soluzione valida, False altrimenti.

Per dimostrare che il problema PUZZLE è NP-hard, dobbiamo effettuare una riduzione da un problema noto essere NP-hard a PUZZLE. In questo caso, possiamo utilizzare il problema 3-SAT, che è un problema classico NP-completo.

Ecco un'idea di come potrebbe funzionare questa riduzione:

1. Partiamo da un'istanza di 3-SAT con n variabili e m clausole.
2. Costruiamo un'istanza del problema PUZZLE come segue: a) Creiamo una scatola con 2n + m buchi. b) I primi 2n buchi corrispondono alle variabili e loro negazioni. c) Gli ultimi m buchi corrispondono alle clausole.
3. Creiamo le carte: a) Per ogni variabile x_i, creiamo una carta con due configurazioni:
    - Una configurazione copre il buco x_i e lascia scoperto il buco not x_i
    - L'altra configurazione fa l'opposto

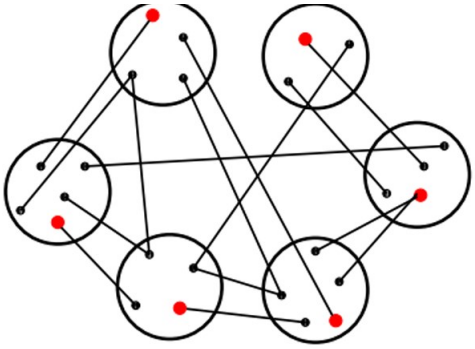
b) Per ogni clausola, creiamo una carta che può coprire il buco della clausola e i buchi corrispondenti ai letterali che la soddisfano.

4. Aggiungiamo pioli e tacche in modo che:
    - Le carte delle variabili possano essere posizionate solo nei loro buchi designati.
    - Le carte delle clausole possano essere posizionate solo se almeno uno dei letterali che la soddisfano è "vero" (cioè, il buco corrispondente è scoperto).

5. Dimostriamo che esiste una soluzione al PUZZLE se e solo se la formula 3-SAT è soddisfacibile:
    - Se la formula è soddisfacibile, possiamo posizionare le carte delle variabili secondo l'assegnamento che soddisfa la formula, e poi posizionare le carte delle clausole per coprire i buchi rimanenti.
    - Se il PUZZLE ha una soluzione, l'arrangiamento delle carte delle variabili corrisponde a un assegnamento che soddisfa la formula 3-SAT.

Questa riduzione può essere effettuata in tempo polinomiale rispetto alla dimensione dell'istanza 3-SAT.

Poiché 3-SAT è NP-completo, questa riduzione dimostra che PUZZLE è almeno altrettanto difficile, e quindi è NP-hard. Inoltre, poiché abbiamo già un verificatore in tempo polinomiale (come mostrato nella risposta precedente), possiamo concludere che PUZZLE è in realtà NP-completo.



Definiamo il linguaggio

$$COMMITTEE = \{\langle D_1, \ldots, D_m, I\rangle \mid \text{esiste una buona commissione } C\}.$$

(a) Dimostra che *COMMITTEE* è un problema NP.
(b) Dimostra che *COMMITTEE* è NP-hard, usando *3SAT* come problema NP-hard di riferimento.

Per dimostrare che COMMITTEE è NP-hard e NP-completo, procediamo in due passi come richiesto dal problema:

(a) Dimostrare che COMMITTEE è in NP:

Un problema è in NP se esiste un verificatore in tempo polinomiale che può controllare una soluzione candidata.

Per COMMITTEE, un certificato sarebbe una lista di membri della commissione. Il verificatore dovrebbe controllare in tempo polinomiale che:

1. Ogni membro della commissione appartiene a uno dei dipartimenti D1, ..., Dm.
2. La commissione contiene almeno un membro da ogni dipartimento.
3. Non ci sono conflitti di interesse (nessun collegamento diretto tra membri della commissione nel grafo).
4. La dimensione della commissione è al più I.

Questo verificatore può essere implementato in tempo polinomiale, quindi COMMITTEE è in NP.

(b) Dimostrare che COMMITTEE è NP-hard usando 3SAT:

Per dimostrare che COMMITTEE è NP-hard, dobbiamo ridurre 3SAT a COMMITTEE in tempo polinomiale. Ecco come:

1. Data un'istanza di 3SAT con n variabili e m clausole:
    - Creiamo 2n + m dipartimenti: due per ogni variabile (xi e not-xi) e uno per ogni clausola.
    - Ogni dipartimento avrà un solo membro.
2. Costruiamo il grafo di conflitti:
    - Colleghiamo ogni coppia di dipartimenti xi e not-xi.
    - Per ogni clausola, colleghiamo il suo dipartimento ai dipartimenti delle variabili non presenti nella clausola.
3. Impostiamo l = n + m (dimensione massima della commissione).
4. La riduzione è completa. Ora dimostriamo l'equivalenza:
    - Se 3SAT ha una soluzione, possiamo formare una commissione valida scegliendo:
        - Per ogni variabile, il membro del dipartimento xi se la variabile è vera, altrimenti quello di not-xi.
        - Un membro da ogni dipartimento delle clausole.
    - Se COMMITTEE ha una soluzione, possiamo costruire un'assegnazione di verità per 3SAT:
        - Se il membro di xi è nella commissione, assegniamo vero alla variabile, altrimenti falso.
        - Questa assegnazione soddisfa tutte le clausole, altrimenti non potremmo aver incluso tutti i membri dei dipartimenti delle clausole senza conflitti.

Questa riduzione può essere fatta in tempo polinomiale e preserva la soluzione del problema originale.

Conclusione: Poiché COMMITTEE è in NP (passo a) e è NP-hard (passo b), possiamo concludere che COMMITTEE è NP-completo.

1. **(12 punti)** Se L è un linguaggio sull'alfabeto $\{0,1\}$, la *rotazione a sinistra* di L è l'insieme delle stringhe

$$\mathrm{ROL}(L) = \{wa \mid aw \in L, w \in \{0,1\}^*, a \in \{0,1\}\}.$$

Per esempio, se $L = \{0, 01, 010, 10100\}$, allora $\mathrm{ROL}(L) = \{0, 10, 100, 01001\}$. Dimostra che se L è regolare allora anche $\mathrm{ROL}(L)$ è regolare.

Esempio “per i mona come me”:

[1]010{0}

[] = stato iniziale!

{} = stato finale!

{0}100[1]

E poi scorro tutti gli altri caratteri fino a beccare di nuovo il mio vecchio stato iniziale, in quanto tutti i caratteri sono stati shiftati a destra.

Si può fare “tranquillamente” con un DFA che prende in input la stringa, la scorre fino alla fine. Una volta fatto questo, mette l’ultimo input come nuovo stato iniziale, scorre tutti gli altri caratteri a destra e poi mette l’inizio come nuovo stato finale.

Essendo che il DFA è regolare, allora L è regolare.

Soluzione di Claudio:

1. Costruzione del DFA: Supponiamo di avere un DFA M che riconosce il linguaggio L. Costruiremo un nuovo DFA M' per ROL(L).
2. Stati di M':
    - Per ogni stato q in M, creiamo due copie in M':
        - q0: rappresenta "non abbiamo ancora letto il primo bit"
        - q1: rappresenta "abbiamo letto e memorizzato il primo bit"
    - Lo stato iniziale di M' sarà q0, dove q è lo stato iniziale di M.
3. Transizioni in M': Per ogni transizione δ(p, a) = q in M: a) Da p0 a q1 leggendo a (memorizziamo il primo bit) b) Da p1 a q1 leggendo qualsiasi bit (continuiamo la computazione)
4. Stati finali di M': Uno stato q1 è finale in M' se: a) q è finale in M e la transizione su 0 da q porta a uno stato finale in M, o b) q è finale in M e la transizione su 1 da q porta a uno stato finale in M
5. Funzionamento:
    - M' inizia in q0 (stato iniziale)
    - Legge il primo bit e passa a q1 (memorizzazione)
    - Continua a leggere i bit successivi, rimanendo negli stati q1
    - Alla fine, controlla se l'ultimo stato raggiunto è finale secondo le regole sopra
6. Esempio pratico: Se M accetta "0101", M' accetterà "1010" (la rotazione a sinistra)
7. Correttezza: Questo DFA M' accetterà esattamente le stringhe che sono rotazioni a sinistra delle stringhe accettate da M, implementando così ROL(L).

Questa costruzione dimostra che se L è regolare (riconosciuto da un DFA), anche ROL(L) è regolare, poiché abbiamo costruito un DFA che lo riconosce.

L'idea chiave è che il nuovo DFA "simula" il comportamento del DFA originale, ma tiene traccia del primo bit letto e lo "ricorda" fino alla fine della computazione, dove decide se accettare o meno basandosi su questo bit memorizzato e sullo stato finale raggiunto.