Automi e Linguaggi Formali
Esame del 19 Giugno 2019 – Parte II

Tempo a disposizione: 1 h 30 min

1. Descrivere la relazione tra i linguaggi liberi da contesto, quelli riconosciuti da DPDA che accettano per stato finale e quelli riconosciuti da DPDA che accettano per pila vuota. Spiegare anche la relazione tra i linguaggi non ambigui e quelli riconosciuti dai DPDA per entrambe le modalità di accettazione.

I DPDA accettano linguaggi diversi a seconda del modo di accettazione. I DPDA che accettano per stato finale accettano tutti i linguaggi regolari e una parte dei linguaggi CF. Per esempio il linguaggio dei palindromi pari costituito dalle stringhe ww r non può essere accettato in modo deterministico perché è necessario "indovinare" la metà dell'input. I DPDA che accettano per stack vuoto sono meno espressivi in quanto riconoscono i linguaggi accettati dai DPDA per stato finale che soddisfano la proprietà del prefisso. Un linguaggio L ha la proprietà del prefisso quando non esistono 2 stringhe diverse di L tali che una sia un prefisso dell'altra. E' possibile dimostrare che i linguaggi accettati da DPDA che accettano per stack vuoto sono non inerentemente ambigui. E' possibile dimostrare che anche i linguaggi riconosciuti dai DPDA che accettano per stato finale sono non inerentemente ambigui. Questo risultato lo si ottiene con un "trucco" che permette di dare la proprietà del prefisso a tutti i linguaggi accettati dai DPDA che accettano per stato finale. Il trucco è di aggiungere un simbolo speciale (che si assume non appaia nelle stringhe del linguaggio originale) alla fine di ciascuna stringa del linguaggio. Questo garantisce che il nuovo linguaggio abbia la proprietà del prefisso e che quindi possa essere riconosciuto da un DPDA che accetta per stack vuoto.

2. Dato l'automa a pila $P = (\{q\}, \{a, b\}, \{a, Z\}, \delta, q, Z, \{q\})$ dove δ è come segue:
$\delta(q, a, Z) = \{(q, aZ)\}, \delta(q, a, a) = \{(q, aa)\}, \delta(q, b, a) = \{(q, \epsilon)\}$.
P accetta per stato finale (q).

- Descrivere precisamente il linguaggio riconosciuto da P.
- Trasformare P in un PDA P' che accetta per pila vuota lo stesso linguaggio accettato da P (per stato finale).
- Rispetto al determinismo-nondeterminismo, ci sono differenze tra P e P'? Spiegate i motivi di uguaglianza-differenza.

L(P) consiste delle stringhe w tali che ogni prefisso di w abbia un numero di a maggiore o uguale al numero dei b. ϵ è in L(P). Per ottenere P' basta aggiungere uno stato e le transizioni $\delta(q, \epsilon, ?) = \{(p, \epsilon)\}$ e $\delta(p, \epsilon, ?) = \{(p, \epsilon)\}$, dove ? sta per qualsiasi simbolo dello stack. Lo stato p serve a svuotare la pila senza consumare input. In questo modo P' accetta un input w per stack vuoto sse P accetta w per stato finale. P è deterministico, mentre P' è non deterministico. Non c'è modo di avere un DPDA che accetti L(P) per stack vuoto visto che L(P) non ha la proprietà del prefisso.

3. Dare la definizione del linguaggio L_U e spiegare in dettaglio come si dimostra che L_U è un linguaggio RE e non ricorsivo.

$L_u = \{(M, w) | w \in L(M)\}$. Per dimostrare che è RE si deve produrre una TM che riconosca L_u. Questa TM è la TM Universale che è indicata con U. U è capace di simulare il calcolo di una qualsiasi TM M su un qualsiasi input w. Ovviamente sia la TM M che w sono stringhe binarie. U si basa sul fatto che le TM si possono codificare in un modo semplice e tale che U possa riconsocere e simulare le transizioni di una TM qualsiasi. U è come un computer capace si eseguire tutti i programmi che noi gli diamo. Per dimostrare che L_u non è ricorsivo, ragioniamo per assurdo. Se L_u fosse ricorsivo, sarebbe ricorsivo anche comp(L_u) e con un algoritmo per comp(L_u) potremmo facilmente costruire un algoritmo per L_d. Infatti un'istanza di L_d è una TM M, basterà trasformare M in (M,M) per avere un'istanza di comp(L_u) e l'algoritmo che abbiamo ipotizzato risponderebbe SI/NO a (M,M) e le risposte SI individuano le TM che sono in L_d. Questo è un assurdo visto che L_d è non RE e quindi L_u non è ricorsivo.

4. Data la seguente CFG: $S \to ASB \mid \epsilon,\ A \to aBAS \mid a \mid \epsilon,\ B \to SbAb \mid AB \mid b$, descrivere, possibilmente seguendo l'algoritmo visto in classe, come si eliminano da essa le $\epsilon-$produzioni, ottenendo una grammatica che genera $L(S) \setminus \{\epsilon\}$.

Le variabili che generano ϵ sono S ed A. Dobbiamo quindi eliminare le $\epsilon-$produzioni e per le produzioni che contengono S e/o A nella parte destra dobbiamo produrre tante produzioni quante sono le combinazioni di presenza/assenza di queste 2 variabili. Quindi se consideriamo la produzione $B \to SbAb$, avremo le produzioni, $B \to SbAb \mid bAb \mid Sbb \mid bb$.

5. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S 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. Sappiamo che questo problema è NP-completo.

Considerate la seguente variante del problema, 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*.

(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.

(b) 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.