University of Calabria

Department of Mathematics and Computer Science

Master's Degree in Computer Science

Theoretical Computer Science - 12 CFU

Theoretical Computer Science

Questions and Answers

Daniele Avolio

A.Y. 2022/2023

Contents

0.1 Introduzione

In questo PDF sono riportate le domande e le risposte per l'esame di Informatica Teorica.

Attenzione, le risposte non sono state validate dal professore e sono solamente frutto della mia interpretazione delle lezioni e del materiale di studio. Pertanto, non sono da considerarsi corrette al 100% e sopratutto non sono da considerarsi complete. In particolar modo, questo corso è molto vasto e non è stato possibile trattare tutti gli argomenti in maniera esaustiva, in particolar modo per il fatto che non e' stato possibile ignorare il resto dei corsi.

In ogni caso, spero che questo documento possa essere utile a qualcuno.

Grazie per la lettura. Da Daniele Avolio.

Figure 1: Yoko Taro

Chapter 1

Decidibilità e logica

1.1 Introduzione

In questo capitolo si parlerà di decidibilità e logica. In particolare, si parlerà di decidibilità e di come si possa dimostrare che un problema è decidibile o meno. Inoltre, si parlerà di logica proposizionale e predicativa, di come si possano dimostrare le formule e di come si possa dimostrare che una formula è valida o meno. Si parlera' di automi a stati finiti, macchine di Turing e di come si possano usare per dimostrare che un problema è decidibile o meno.

1.2 Stringhe e linguaggi

1.2.1 Domanda 1

Definizione di linguaggio Universale
Risposta
Un linguaggio universale $\sum^*$ è un insieme che contiene tutte le possibili stringhe di qualsiasi lunghezza da 0. Anche definito come $\bigcup_{i=0}^{\infty} \sum^i$ oppure $\{\epsilon\} \cup \sum^i$.

1.2.2 Domanda 2

Fornire una descrizione informale di x: $\exists y \in \{a, b\}$ **tale che** $x = ya$
Risposta

Si sta descrivendo l'insieme delle stringhe x tale che esiste una stringa y che appartiene all'insieme a,b di lunghezza maggiore o uguale a zero tale che x sia uguale alla concatenazione di y con a. In parole povere, tutte le stringhe che di lunghezza maggiore uguale a zero con a o b, che terminano con a.

1.2.3 Domanda 3

Fornire una procedura per enumerare le stringhe $\{a\}^*$ **oppure** $\{0, 1\}^*$ **Risposta**

Algorithm 1 Enumerazione di $\{a\}^*$

```
i ← 0
while true do
  i ← i + 1
  x ← a^i
  print(x)
end while
```

Algorithm 2 Enumerazione di $\{0,1\}^*$

```
i ← 0
while true do
  i ← i + 1
  x ← i_bin
  print(x)
end while
```

1.2.4 Domanda 4

Quali sono le possibili cardinalita' di un linguaggio?
Risposta
Le possibili cardinalita' di un linguaggio sono:

- *Finito*
- *Infinito numerabile*
- *Infinito non numerabile*

1.2.5 Domanda 5

Quanti linguaggi possono essere definiti su un alfabeto $\sum$?
Risposta
I linguaggi che possono essere definiti su un alfabeto sono infiniti, in quanto un linguaggio puo' essere definito come un insieme di stringhe, e un insieme puo' contenere un numero infinito di elementi. E questo infinito e' anche non numerabile, in quanto un linguaggio puo' essere definito come un insieme di stringhe, e un insieme puo' contenere un numero infinito di elementi.

- Un linguaggio è un insieme di stringhe.
- Queste stringhe possono essere infinite
- Se consideriamo l'insieme dei linguaggi che possono essere definiti su un alfabeto
- Ogni linguaggio è un insieme di stringhe
- Quindi l'insieme dei linguaggi è un insieme di insiemi di stringhe
- Siccome l'insieme delle stringhe è infinito, anche l'insieme dei linguaggi è infinito
- Inoltre, l'insieme dei linguaggi è non numerabile perché non abbiamo modo di creare una biezione tra i linguaggi e i numeri naturali

1.2.6 Domanda 6

Qual e' la relazione tra $|L1 \cdot L2|$ *e* $|L1| \cdot |L2|$?
Risposta
Certamente! Ecco la stessa spiegazione in formato LaTeX:

La relazione $|L_1 \cdot L_2| \leq |L_1| \cdot |L_2|$ può essere dimostrata considerando la definizione di concatenazione di due linguaggi L_1 e L_2. La concatenazione di L_1 e L_2, indicata con $L_1 \cdot L_2$, consiste di tutte le stringhe che possono essere ottenute concatenando una stringa in L_1 con una stringa in L_2. Formalmente, $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$.

Il prodotto cartesiano di due linguaggi L_1 e L_2, indicato con $L_1 \times L_2$, è invece definito come l'insieme di tutte le coppie ordinate (x, y) dove x appartiene a L_1 e y appartiene a L_2. Formalmente, $L_1 \times L_2 = \{(x, y) \mid x \in L_1, y \in L_2\}$.

Ogni stringa in $L_1 \cdot L_2$ corrisponde ad una coppia di stringhe (x, y) in $L_1 \times L_2$, dove x è una stringa in L_1 e y è una stringa in L_2. Quindi, ogni volta che si sceglie una stringa in $L_1 \cdot L_2$, si sta scegliendo una coppia di stringhe in $L_1 \times L_2$.

Poiché $L_1 \cdot L_2$ è un sottoinsieme di $L_1 \times L_2$, si ha che $|L_1 \cdot L_2| \leq |L_1 \times L_2|$. Inoltre, per la regola del prodotto, si ha che $|L_1 \times L_2| = |L_1| \cdot |L_2|$. Quindi, si conclude che $|L_1 \cdot L_2| \leq |L_1| \cdot |L_2|$.

La spiegazione logica è che semplicemente nella cardinalita della concatenazione abbiamo delle stringhe che si ripetono e quindi la cardinalità è minore.

1.2.7 Domanda 7

Dato un linguaggio L=$\{a, ab, bb\}$ **scrivere le prime 15 stringhe in** L^*
Risposta

- ϵ
- a
- ab
- bb
- aa
- aab
- abb
- bbb
- aaa
- $aaab$
- $aabb$
- $abbb$
- baa

- *baab*
- *babb*

1.2.8 Domanda 8

Definisci formalmente: $\{x\#y$: **x e y sono stringhe binarie** $\wedge$ **x e' sottostringa di y**$\}$.
Risposta
Stiamo definendo questo come l'insieme delle stringhe in formato x e y tale che la stringa sia formata da $x \cdot \# \cdot y$ dove x é una stringa binaria tale che $\exists a \in y \wedge \exists b \in y : a \cdot x \cdot b = y$

In modo più formale ancora:

$$\{x\#y : x, y \in \{0,1\}^* \wedge x \subseteq y\} \tag{1.1}$$

1.2.9 Domanda 9

Esibire una biezione tra $\{0,1\}^*$ *e* $\mathbb{N}$
Risposta
Se trovassimo una procedura per collegare ogni singola stringa dell'insieme $0, 1^*$ con un numero $\in \mathbb{N}$ potremmo mostrare una biezione tra i due insiemi. Possiamo scrivere una funzione che scrive in ordine tutte le stringhe dell'insieme in ordine.

Algorithm 3 Enumerazione di $\{0,1\}^*$

```
i ← 0
while true do
    x ← (i + 1)_bin
    n ← |x|
    print(x[2] ... x[n])
end while
```

In questo modo se stampiamo le stringhe in sequenza, stiamo associando ad ogni stringa un numero naturale. Possiamo anche stampare l'indice insieme.

1.3 Linguaggi regolari e teoria degli automi

1.3.1 Domanda 1

Spiega i componenti di un automa a stati finiti deterministico
Risposta

I componenti di un'automa a stati finiti sono:

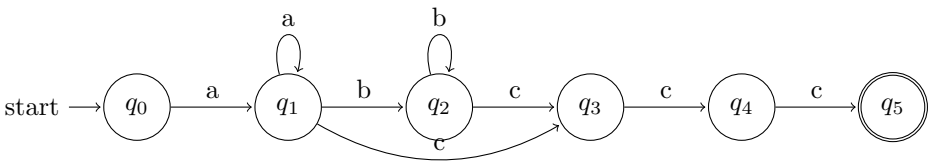
- Un'insieme finito di stati K
- Un'insieme finito di stati finali $A \subseteq K$
- Uno stato iniziale $s \in K$
- Una funzione di transizione $\delta : K \times \sigma \rightarrow$
- Un alfabeto Σ

1.3.2 Domanda 2

Definire graficamente un'automa a stati finiti per

$$L = \{a^n b^m c^3 : n > 1 \wedge m \geq 0\}$$

Risposta



1.3.3 Domanda 3

Definire matematicamente un'automa a stati finiti per

$$L = \{a^n b^m c^3 : n > 1 \wedge m \geq 0\}$$

Risposta
Per definire formalmente un'automa a stati finiti dobbiamo definire i componenti che lo compongono.

- Un'insieme finito di stati $K = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- Un'insieme finito di stati finali $A = \{q_5\}$
- Un'alfabeto $\Sigma = \{a, b, c\}$
- Uno stato iniziale $s = q_0$
- Una funzione di transizione δ che e' definita come: $\delta : K \times \Sigma \to K$ In particolare:
  - $\delta(q_0, a) = q_1$
  - $\delta(q_1, a) = q_1$
  - $\delta(q_1, b) = q_2$
  - $\delta(q_1, c) = q_3$
  - $\delta(q_2, b) = q_2$
  - $\delta(q_2, c) = q_3$
  - $\delta(q_3, c) = q_4$
  - $\delta(q_4, c) = q_5$

1.3.4 Domanda 4

Dato (q2,babbaa), (q2,a)→q1 e (q2,b) → q3, determinare lo yield in one step.
Risposta
Per determinare lo yield in one step dobbiamo capire come funziona la funzione di transizione $\delta(q_2, babbaa)$. In questo caso sappiamo che da $q2$ leggendo a saremo nello stato $q3$. Quindi, dato in input $babbaa$ lo yield in one step sara' ($q3$, abbaa).

1.3.5 Domanda 5

Determinare la chiusura transitiva di $R = \{(a,b),(b,c),(c,d),(b,e),(b,f)\}$
Risposta
Noi sappiamo che la chiusura transitiva di una relazione R e' definita come:

$$R^+ = R \cup \{(a,b) \in S \times S : (a,c) \in R^+ \wedge (c,b) \in R^+\}$$

Cioe, stiamo dicendo che viene definita come tutte le relazioni binarie su insiemi S tale che $(a,b) \in R$ e $(b,c) \in R$. Avendo la lista di transizioni binarie, possiamo definire la chiusura transitiva come:

$$R^+ = R \cup (a,c),(a,e),(a,f),(b,d)$$
$$R^+ = \{(a,b),(b,c),(c,d),(b,e),(b,f),(a,c),(a,e),(a,f),(b,d)\}$$

1.3.6 Domanda 6

Caratterizzare formalmente i linguaggio L(M) di un'automa a stati finiti M.
Risposta
Per caratterizzare formalmente il linguaggio di un'automa a stati finiti dobbiamo definire la funzione di transizione δ. Iniziamo dicendo che δ e' definita come:

$$\delta : K \times \Sigma \to K$$

dove K e' un insieme finito di stati e Σ e' un alfabeto. Quindi, possiamo dire che il linguaggio $L(M)$ e' definito come:

$$L(M) = \{w \in \Sigma^* : \exists q \in A t.c.\ (s,w) \vdash_M^* (q,\epsilon)\}$$

dove A e' un insieme finito di stati finali, s e' lo stato iniziale e $\vdash_M^*$ e' la chiusura riflessiva transitiva di M. Possiamo dire che quindi il linguaggio $L(M)$ e' definito come l'insieme di tutte le stringhe w che sono accettate da M.

1.3.7 Domanda 7

Qual e' la classe dei linguaggi regolari? Risposta
Noi diciamo che un linguaggio $L \subseteq \Sigma^*$ 'e regolare se e solo se esiste un'automa a stati finiti M che lo decide. Se vogliamo definire la classe, allora possiamo dire:

$$REG = \{L \subseteq \Sigma^* : L \text{ e' regolare}\}_{\forall \Sigma}$$

1.3.8 Domanda 8

Discuti la differenza tra una funzione di transizione parziale e una totale.
Risposta
Una funzione di transizione parziale e' una funzione che non contiene tutti gli stati dell'automa, mentre una funzione di transizione totale contiene al suo interno tutte gli stati dell'automa. Allora possiamo dire che se una funzione di transizione di un automa M, possiamo costituire un automa M' che e' definito come:

$$M' = (K \cup \{d\}, \Sigma, \delta', s', A')$$

dove d e' uno stato distinto, $s' = s$ e $A' = A$. E la funzione di transizione δ' e' definita come:

$$\delta' = delta \cup \{(q,a) \to d : \delta(q,a) \text{ non e' definita}\}$$

1.3.9 Domanda 9

Definire formalmente $\overline{M}$ di un'automa a stati finiti $M = (K, \Sigma, \delta, s, A)$.
Risposta
Sapendo che $\bar{M}$ e' il complemento di M, allora possiamo dire che $\bar{M}$ e' definito come:

$$\bar{M} = (K, \Sigma, \delta, s, K \setminus A)$$

Cio' questo automa ha la stessa funzione di transizione di M, ma siccome e' il complemento allora dovra' rifiutare tutte le stringhe che venivano accettate da M. Per questo motivo, l'insieme degli stati finali saranno tutti gli stati finali tranne quelli che erano definiti precedentemente. Quindi, possiamo definire M come:

- K e' un insieme finito di stati
- Σ e' un alfabeto
- $\delta : K \times \Sigma \to K$ e' la funzione di transizione
- s e' lo stato iniziale
- $K \setminus A$ e' l'insieme degli stati finali

1.3.10 Domanda 10

Scrivere le prime 20 stringhe di $L(r)$ dove $r = ((a^+b^*)c)^+$
Risposta

Indice	Stringa
1	ac
2	aac
3	abc
4	$aabc$
5	$abbc$
6	$acac$
7	$aacac$
8	$abcac$
9	$aabcac$
10	$abbcac$
11	$acabc$
12	$aacabc$
13	$abcabc$
14	$aabcabc$
15	$abbcabc$
16	$acabbc$
17	$aacabbc$
18	$abcabbc$
19	$aabcabbc$
20	$abbcabbc$

1.3.11 Domanda 11

Dimostrare formalmente che REG e' chiuso rispetto all'operazione di unione, concatenazione e complemento.
Risposta
Per dimostrare che REG e' chiuso rispetto all'operazione di unione, concatenazione e complemento, dobbiamo ricordare delle proprieta' delle espressioni regolari come:

- ϵ e' una espressione regolare
- se $a \in \Sigma$, allora a e' una espressione regolare
- se R_1 e R_2 sono espressioni regolari, allora $R_1 + R_2$ e' una espressione regolare
- se R_1 e R_2 sono espressioni regolari, allora $R_1 \cdot R_2$ e' una espressione regolare

Unione: Siano L_1 e L_2 due linguaggi regolari. Allora esistono due regular expression R_1 e R_2 che generano rispettivamente L_1 e L_2. Possiamo costruire una nuova regular expression R come $R = R_1 + R_2$, dove "+" rappresenta l'operazione di unione tra due espressioni regolari. L'espressione R genera tutte le stringhe che appartengono sia a L_1 che a L_2, dimostrando che REG è chiuso rispetto all'unione.

Concatenazione: Siano L_1 e L_2 due linguaggi regolari. Allora esistono due regular expression R_1 e R_2 che generano rispettivamente L_1 e L_2. Possiamo costruire una nuova regular expression R come $R = R_1R_2$, dove "R1R2" rappresenta l'operazione di concatenazione tra due espressioni regolari. L'espressione R genera tutte le stringhe che sono la concatenazione di una stringa in L_1 con una stringa in L_2, dimostrando che REG è chiuso rispetto alla concatenazione.

Complemento: Sappiamo che se un linguaggio e' regolare, allora ci sara' un'automata a stati finiti a deciderlo. Possiamo quindi costruirne uno definito come M = (K, Σ, δ, s, A), dove K e' un insieme finito di stati, Σ e' un alfabeto, δ e' la funzione di transizione, s e' lo stato iniziale e A e' l'insieme degli stati finali. Possiamo quindi definire il complemento di M come $\overline{M}$ = (K, Σ, δ, s, K $\setminus$ A), dove K e' un insieme finito di stati, Σ e' un alfabeto, δ e' la funzione di transizione, s e' lo stato iniziale e K $\setminus$ A e' l'insieme degli stati finali. Quindi, $\bar{M}$ e' un automa a stati finiti che accetta tutte le stringhe che non sono accettate da M, dimostrando che REG e' chiuso rispetto al complemento.

1.4 Macchine di Turing

1.4.1 Domanda 1

Definire formalmente una DTM.
Risposta

Una DTM o Macchina di Turing deterministica M è una tupla del tipo:

$$M = (K, \Sigma, r, \delta, s, H)$$

dove precisamente:

- K è un insieme finito di stati;
- Σ è un insieme finito di simboli di input;
- Γ è un insieme finito di simboli del nastro;
- δ è una funzione di transizione;
- $s \in K$ è lo stato iniziale;
- $H \subseteq K$ è l'insieme degli stati di arresto;

1.4.2 Domanda 2

Dato una DTM M_0 e una stringa w_0, eseguire M_0 su w_0 per ottenere l'output.
Risposta

$$M_0 = (K, \Sigma, r, \delta, s, H)$$
$$w_0 = \{a, b, c\}$$

1.4.3 Domanda 3

Definire formalmente una configurazione di una DTM.
Risposta Una configurazione di una turing machine è una tupla che ci indica lo stato della macchina in quel determinato momento.
In particolare:

$$C = (s, w, c, v)$$

con:

- $s \in K$ è lo stato corrente della macchina
- w è la stringa prima del cursore $\Gamma^* \cup \{\epsilon\}$
- c è il simbolo sotto al cursore $c \in \Gamma$
- v è la stringa dopo il cursore $\Gamma^* \cup \{\epsilon\}$

1.4.4 Domanda 4

Definire formalmente una computazione di una DTM.
Risposta Una computazione di una macchina di turing M è una sequenza di configurazioni $C_0, C_1, C_2, ...C_n$ tale che:

- La configurazione iniziale C_0 sia del tipo: $C_0 = (s, \epsilon, \square, w)$ con $w \in \Sigma^*$
- La configurazione finale C_n sia del tipo $C_n = (q, u, \alpha, v)$ con $q \in H$, $u, v \in \Gamma^*$ e $\alpha \in \Gamma$
- E deve esistere lo yield da $C_0 \vdash C_1 \vdash C_2 \vdash ... \vdash C_n$

1.4.5 Domanda 5

Fornire una DTM che accetti il linguaggio $L = \{allbnc3 : n > 1\}$ tramite la notazione di stato.
Risposta
Non mi va di farlo al pc.

1.4.6 Domanda 6

Fornire una DTM che accetti il linguaggio $L = \{allb11c3 : n > 1\}$ tramite la macro-linguaggio.
Risposta
Non mi va di farlo al pc.

1.4.7 Domanda 7

Fornire una DTM sull'alfabeto $\{a, b, c\}$ che calcoli w dall'input w^r.
Risposta
Non mi va di farlo al pc.

1.4.8 Domanda 8

Definire formalmente il linguaggio di una DTM che decide un linguaggio.
Risposta
Non mi va di farlo al pc.

Il linguaggio di una DTM $M = (K, \Sigma, \Gamma, \delta, s, H)$ che accetta è definito come $L(M) = \{w \in E^* : (s, \epsilon, \square, w) \vdash^* (y, u, \alpha, v)\}$ Cioè, data una stringa $w \in \Gamma$, la macchina M termina la sua esecuzione in uno stato di arresto y.

1.4.9 Domanda 9

Definire formalmente D e SD.
Risposta

- D è l'insieme di i linguaggi tale che una DTM M termina sempre la sua esecuzione su una stringa w $\in$ all'alfabeto del linguaggio Σ^*, che sia in uno stato di accettazione y oppure uno stato di rigetto n.
- SD è l'insieme di i linguaggi tale che se la stringa $w \in L$, allora una DTM M termina sempre la sua esecuzione in uno stato y.

1.4.10 Domanda 10

Da una DTM $M = (K, \Sigma, \Gamma, \delta, s, H)$, definire formalmente una DTM $\overline{M}$ per $L(\overline{M})$.
Risposta Sapendo che $M = (K, \Sigma, \Gamma, \delta', s, H)$, possiamo definire $\overline{M}$ come $\overline{M} = (K, \Sigma, \Gamma, \delta, s, H')$. Possiamo indicare come $\overline{M}$ la macchina di Turing che termina la sua esecuzione in uno stato di rigetto n per tutte le stringhe che terminano la loro esecuzione in uno stato di accettazione y e viceversa.

Una definizione formale di $\overline{M}$ è:

- K = l'insieme di tutti gli stati di M
- Σ = l'insieme di tutti i simboli di input di M
- Γ = l'insieme di tutti i simboli del nastro di M
- δ = la funzione di transizione di M che deve essere costruita in modo che $\delta(q, a) = (q', a', R)$ se e solo se $\delta'(q, a) = (q', a', L)$

1.5 Multitape turing machines

1.5.1 Domanda 1

Definire formalmente una multitape turing machine.
Risposta
Una turing machine a nastro multiplo e' una tupla del tipo:

$$M = (K, \Sigma, \Gamma, \delta, s, H)$$

dove in particolare abbiamo che:

- K e' un insieme finito di stati

- Σ e' un insieme finito di simboli di input anche detto alfabeto di input
- Γ e' un insieme finito di simboli del nastro
- δ e' una funzione di transizione che e' precisamente definita come:

$$\delta : (K - H) \times \Gamma_1 \times \cdots \times \Gamma_k \to K \times \Gamma_1 \times \{\rightarrow, \leftarrow, \uparrow\} \times \cdots \times \Gamma_k \times \{\rightarrow, \leftarrow, \uparrow\}$$

- $s \in K$ e' lo stato iniziale

1.5.2 Domanda 2

Considera una Macchina di turing 3-multinastro dove $|\Sigma| = 4$ e $|\Gamma| = 6$. Immagina ora una Macchina di Turing a nastro singolo. Definisci la carindalita' di $|\Gamma'|$
Risposta
Per rispondere a questa domanda dobbiamo capire il calcolo da effettuare ragionando su come viene costruita una macchina di turing a nastro singolo partendo da una multinastro. Abbiamo che la **la cardinalita'** di Γ' e' definita come:

- $\square$ e' sicuramente presente nell'alfabeto
- Γ' contiene tutti i simboli di Σ
- Avendo come numero di nastri 3, pensiamo ad un k = 3 come riferimento
- $|\Gamma|^3$ poiche' abbiamo 3 nastri
- 2^3 perche' abbiamo 3 nastri e per ogni nastro abbiamo 2 $\{0, 1\}$ per indicare se il curosore e' presente o meno

Quindi alla fine avremo: $|\Gamma'| = |\Sigma| + |\Gamma|^3 + 2^3 + 1 = 1 + 8 + 4 + 216 = 229$

1.5.3 Domanda 3

Considera una Macchina di turing 3-multinastro dove $|\Sigma| = 4$ e $|\Gamma| = 6$. Immagina ora una Macchina di Turing a nastro singolo. Quanti simboli di Γ' rappresentano almeno 1 cursore di M?
Risposta

Probabilmente sto sbagliando totalmente ragionamento, ma penso che il numero sia $\frac{3}{4}$. Perche' possiamo rappresentare lo stato di una colonna della macchina multitape in quella singole tape controllando se il cursore e' presente in uno dei 3 nastri. Possiamo avere 4 combinazioni di situazioni:

- Il cuirsore non e' presente in nessuno dei 3 nastri
- Il cursore e' presente in uno dei 3 nastri
- Il cursore e' presente in due dei 3 nastri

- Il cursore e' presente in tutti e 3 i nastri

Siccome a noi interessa sapere se il cursore e' presente in almeno uno dei 3 nastri, possiamo dire che il numero di simboli di Γ' che rappresentano almeno 1 cursore di M e' $|\Gamma'| \times \frac{3}{4}$.

Probabile correzione:
Dobbiamo pensare al fatto che la macchina quando ha lo 0 lo ha in base a quanti nastri stiamo identificando. Se stiamo identificando 3 nastri, allora dobbiamo pensare alla possibili combinazioni dove ci sono 3 zeri.

Le possibili configurazioni che contengono lo zero dovrebbero essere pari a $|\Gamma|^{numero\ di\ nastri}$

Basta fare totale delle configurazioni meno quelle solo con lo zero.

1.5.4 Domanda 4

Qual è il sovraccarico computazionale asintotico introdotto dal nastro multitraccia Risposta
Partiamo supponendo che n sia il numero di step che ci mette la macchina ad essere eseguita. Se si intende per la conversione di una multitape ad una single tape abbiamo che:

- $\mathcal{O}(|w|)$ per inizializzare la stringa
- Computazione:
  - $\mathcal{O}(n)$ per la spostarsi all'inizio della stringa
  - $\mathcal{O}(n)$ per applicare δ'

  In totale: $\mathcal{O}(n^2)$
- Pulizia della stringa, $\mathcal{O}(n)$

Assumendo che n sia $\geq |w|$ abbiamo che il sovraccarico computazionale asintotico introdotto dal nastro multitraccia e' $\mathcal{O}(n^2)$

1.6 Macchina di Turing Universale

1.6.1 Domanda 1

Fornisci una funzione che codifichi una macchina di turing a nastro singolo in una stringa Risposta
Per codificare una macchina di Turing M a nastro singolo in una stringa dobbiamo codificare:

- l'alfabeto del nastro Γ di M
- gli stati K di M
- la funzione di transizione δ di M

Codificare gli stati K Per codificare gli stati dobbiamo prima di tutto prendere un valore i che rappresenta la lunghezza della stringa binaria. Per fare cio' prendiamo $i = \lceil log_2(|K|) \rceil$. Ora dobbiamo codificare ogni stato $k \in K$ in una stringa binaria di lunghezza i. Per farlo dobbiamo avere un ordinamento f degli stati K.

Possiamo definire f come: $\{ : K \to \{0, \ldots, |K| - 1\}$. In questo modo stiamo trovando una biezione tra gli stati e i numeri naturali.

Ora dobbiamo convertire le stringhe in binario. Per fare cio' usiamo la funzione bin che sara' definita come: $bin : \{0, \ldots, |K| - 1\} \to \{0, 1\}^i$.

Abbiamo quindi questa funzione di conversione degli stati chiama ξ che preso uno stato $k \in K$ restituisce la stringa binaria che lo rappresenta: $\xi(q) = q \circ bin(f(q))$.

Codificarl'alfabeto Γ Per codificare l'alfabeto Γ dobbiamo prima di tutto prendere un valore j che rappresenta la lunghezza della stringa binaria. Per fare cio' prendiamo $j = \lceil log_2(|\Gamma|) \rceil$.

Ora dobbiamo codificare ogni simbolo $\gamma \in \Gamma$ in una stringa binaria di lunghezza j Per farlo dobbiamo avere un ordinamento g dei simboli Γ.

Possiamo definire g come: $\} : \Gamma \to \{0, \ldots, |\Gamma| - 1\}$. In questo modo stiamo trovando una biezione tra i simboli e i numeri naturali.

Ora dobbiamo convertire le stringhe in binario. Per fare cio' usiamo la funzione bin che sara' definita come: $bin : \{0, \ldots, |\Gamma| - 1\} \to \{0, 1\}^j$.

Abbiamo quindi questa funzione di conversione degli stati chiama ξ che preso un simbolo $\gamma \in \Gamma$ restituisce la stringa binaria che lo rappresenta: $\xi(\gamma) = \gamma \circ bin(g(\gamma))$.

Codifica di δ Data una funzione di transizione t nella forma $\Gamma(q, a) = (q', b, d)$, allora avremo $\xi(t) = (\xi(q), \xi(a)), (\xi(q'), \xi(b), \xi(d))$.

1.6.2 Domanda 2

Dimostra formalmente che il numero di TM e' infinito Risposta
Per dimostrare che il numero di TM e' infinito basta costruire una turing machine con uno stato extra che contiene uno stato in piu rispetto alla precedente. In questo modo possiamo costruire una turing machine con n stati, con $n \in \mathbb{N}$.

- Let $X = \{h\}$ if M is a transducer;
- Let $X = \{y\}$ if δ contains at least a transition of the form (q, a, y, a', d); otherwise let $X = \{n\}$.
- Let $K' = K \cup \{\hat{q}\}$, were $\hat{q}$ is an arbitrary state non in K;
- Let $\delta' = \{(q, a, q', a', d) \in \delta \mid q' \notin X\} \cup \{(q, a, \hat{q}, a', d) \mid (q, a, q', a', d) \in \delta \wedge q' \in X\} \cup \{(\hat{q}, a, x, a, \rightarrow) \mid a \in \Gamma \wedge x \in X\}$

Figure 1.1: Formally definition of infinite TM

Quello che la figura dice è che si aggiunge uno stato che non è compreso nel set degli stati della macchina. Stiamo collegando l'ultimo stato prima dello stato di halting / accettante o rifiutante al nuovo stato che non c'era prima. Si inseriscono due transizioni, uno che va dal penultimo di prima al nuovo stato aggiunto e uno che va dal nuovo stato aggiunto allo stato finale.

1.6.3 Domanda 3

Dimostra che il numero di TM e' numerabile Risposta
Per dimostrare che il numero di TM e' numerabile dobbiamo dimostrare che esiste una funzione biiettiva tra i numeri naturali e le TM. Per fare cio' dobbiamo prima di tutto codificare le TM in una stringa. Per fare cio':

- Fissiamo un alfabeto Σ_U che contiene $((,), a, q, y, n, h, 0, 1, ; , \rightarrow, \leftarrow)$.
- Generare le stringhe di Σ_U in ordine alfabetico
- Per ogni stringa w, se la codifica di una TM, allora la restituiamo altrimenti la ignoriamo.

In questo modo e' possibile definire la $i - esima$ TM.

1.6.4 Domanda 4

Descrivi i macro steps di una UTM Risposta
Gli steps di una UTM in generale sono:

1. Primo nastro contiene la stringa coppia di input $< M > e < w >$
2. Si sposta $< M >$ sul secondo nastro
3. Il terzo nastro conterra' la codifica di uno stato di M
4. Si scorre il secondo nastro fino a quando non si trova lo stato corrente
5. Si esegue l'operazione associata, cambiando i nastri 1 e 3 e se serve si estendono
6. Se non viene trovata la quintupla che rappresenta lo stato, HALT
7. Si ritorna lo stesso risultato che M riporterebbe

1.6.5 Domanda 5

Definire *tm2int* e *int2tm* assumendo che abbiamo le funzioni $int2str_{\Sigma_U}$ e $str2int_{\Sigma_U}$ Risposta

tm2int
La funzione deve prendere in input una codifica di una turing machine e restituire un numero intero.

Per fare cio' dobbiamo scrivere una procedura che scorre un indice partendo da 0 e utilizza la funzione $int2str_{\Sigma_U}$ per convertire l'indice ad una TM.

Dopo aver convertito l'indice in una TM dobbiamo controllare se la TM e' equivalente a quella di input. Se lo e' allora restituiamo l'indice, altrimenti incrementiamo l'indice e ripetiamo il processo.

```
def tm2int(M: Tm_encoding):
    i = 0
    while True:
        if M == int2str(i):
            return i
        i += 1
```

int2tm

La funzione deve prendere in input un numero intero e restituire una codifica di una turing machine.

Per fare cio' dobbiamo scrivere una procedura che scorre un indice partendo da 0 e utilizza la funzione $int2str\Sigma_U$ per convertire l'indice ad una TM.

Se l'indice corrisponde alla TM di input allora restituiamo la TM, altrimenti incrementiamo l'indice e ripetiamo il processo.

```
def int2tm(i: int):
    strCount = 0
    tmcount = 0
    while True:
        M = int2str(strCount)
        if (isaTmEncoding(M)):
            tmCount += 1
        if (tmCount == i):
            return M
        strCount++
```

1.6.6 Domanda 6

Descrivi in modo informale isaTmEncoding parlando di input, output e comportamento
Risposta

La funzione deve prendere in input una stringa e restituire un booleano **True** se la stringa e' una codifica di una turing machine, altrimenti restituisce **False**. Per fare cio' dobbiamo controllare che la stringa sia una codifica di una turing machine.

Il controllo viene fatto sugli elementi che caratterizzano la stringa, ovvero devono appartenere all'alfabeto Σ_U, devono essere tuple formate da 5 elementi e devono contenere uno stato, un simbolo, un altro stato, un altro simbolo e una direzione.

1.6.7 Domanda 7

Parla della differenza tra M(w) e di [M]([w])
Risposta

La differenza sta nel fatto che M(w) ci ritorna **lo stato in cui la macchina M si e' fermata sulla stringa w**, mentre [M]([w]) ci ritorna **l'output della macchina M sulla stringa w**.

1.6.8 Domanda 8

Possiamo manipolare in modo automatico le TMs in mod da avere nuove TMs? Se si, come?
Risposta

Mi verrebbe da dire SI. Il modo in cui possiamo manipolare le TMs e' quello di utilizzare aggiungere un nuovo stato e aggiungere una nuova quintupla che rappresenta la nuova operazione che vogliamo eseguire e inserirla prima della fine. Manna, a quanto dice **Davide**, parla delle reduction. Dice che praticamente puoi generarti infinite le macchine prendendo una macchina e aggiungendo una nuova operazione che esegue una nuova operazione. Questo e' un modo per generare infinite macchine.

1.7 La macchina RAM

1.7.1 Domanda 1

Descrivere i regitri della macchina RAM.
Risposta

La macchina RAM ha i seguenti registri:

1. PC: Program Counter — che contiene l'indice della prossima istruzione da eseguire.
2. ACC: Accumulator — che contiene il valore dell'ultima operazione eseguita e che mantiene i dati. E' un registro multi purpose.
3. IN: Registro di INPUT — che contiene il valore dell'input e che non e' limitato a destra
4. OUT: Registro di OUTPUT — che contiene il valore dell'output e che non e' limitato a destra
5. IR: Instruction Register — Contiene il codice della prossima istruzione da eseguire
6. AR: Address Register — che contiene l'indirizzo in memoria del parametro dell'istruzione da eseguire

1.7.2 Domanda 2

Parla della routine generale della macchina RAM.
Risposta

In generale la macchina RAM imposta il PC a 0 e legge dal registro di INPUT determinati BIT che rappresentano l'istruzione da eseguire e il parametro dell'istruzione da eseguire.
Dopodiche' la macchina RAM esegue l'istruzione e aumenta il PC di 1.
Se il valore dell'IR e' uguale all'operazione di HALTING, allora il processo termina.
Altrimenti viene eseguita l'istruzione specificata in IR in base al parametro in AR.

1.7.3 Domanda 3

Descrivi le istruzioni LOAD, ADD e STORE della macchina RAM Risposta

L'istruzione LOAD della MACCHINA RAM legge un determinato indirizzo di memoria M[xxx] e lo carica nel registro ACC.
L'istruzione ADD della MACCHINA RAM legge un determinato indirizzo di memoria M[xxx] e lo somma al valore del registro ACC. ACC=ACC+M[XXX]
L'istruzione STORE della MACCHINA RAM scrive il valore del registro ACC in un determinato indirizzo di memoria M[xxx]. M[XXX]=ACC

1.7.4 Domanda 4

Descrivi il formato dell'input di una TM simulante una macchina RAM
Risposta

Il contenuto del nastro di una TM che deve simulare una MACCHINA RAM e' una stringa separata da un simbolo specifico, ad esempio #, per ogni coppia di valori che rappresentano ISTRUZIONE e parametro separati da una virgola.
Un esempio e':

$$\#0, 00110111\#1, 00000000\#10, 00000001\#11, 00110101$$

1.7.5 Domanda 5

Descrivi il contenuto dei nastri di una TM che deve simulare una macchina RAM Risposta

I nastri saranno divisi in questo modo:

1. Nastro 1: la memoria del computer
2. Nastro 2: Il program counter PC
3. Nastro 3: L'address register AR
4. Nastro 4: l'Accumulator ACC
5. Nastro 5: Il codice dell'istruzione corrente
6. Nastro 6: Il file di input
7. Nastro 7: Il file di output

1.7.6 Domanda 6

Può una macchina RAM simulare una single tape?
Risposta

Per simulare una single tape, la macchina ram avrebbe bisogno di leggere la TM single tape come una singola stringa. Questo possiamo farlo perché possiamo rappresentare ogni macchina come una stringa, usando un alfabeto che contiene i simboli che utilizziamo per runnare la macchina e encodando ogni stato, simbolo e funzione di transizione.

1.8 Procedure Semi-decidibili

1.8.1 Domanda 1

Descrivi l'input, l'output e il comportamento della funzione utm()
Risposta

La funzione *utm()* prende in input una macchina di Turing M e una stringa w e restituisce q se M termina su w.
In particolare, la funzione controlla se eseguendo M su w si arriva ad uno stato finale. Se questo è corretto, allora la macchina ritorna lo stato in cui la macchina ha terminato la sua esecuzoine.

1.8.2 Domanda 2

Descrivi H e fornisci una procedura che lo decide
Risposta

H possiamo definirlo come $H = \{< M, w > | M$ è una macchina di Turing, $w \in \Sigma^* \wedge M(w) \in \{y, n, h\}\ \}$.

Una procedura che lo decide è la seguente:

```
def H(M: Tm-encoding, w: string):
    utm(M, w)
    return true
```

1.8.3 Domanda 3

Descrivi input, output e comportamento di $utms()$
Risposta

La funzione utms() è un'estensione della funzione utm(), che oltre a controllare se la macchina M termina su w, controlla anche che lo faccia in un determinato numero di steps. Quindi, la funzione prende in ipnut:

- una macchina di Turing M
- una stringa w
- un numero s di steps

Come output, invece la funzione restituisce:

- Uno stato q — se la macchina M termina su W oppure se ha raggiunto il numero di steps previsto

1.8.4 Domanda 4

Fornisci una procedura di enumerazione per le coppie in $\mathcal{N}^2$
Risposta
Penso sia proprio la fair enumeration.

Cioè si mappa ogni coppia di numeri naturali ad un numero naturale.

$$N \to N^2 \tag{1.2}$$

1.8.5 Domanda 5

Definire in pseudo-codice la funzione $int2Pair()$
Risposta

```
function int2Pair(m: int){
    posizione = 0
    diagonale = 0

    while(true){
        Per ogni val da 0 a diagonale{
            se (posizione == m)
                return <val, diagonale - val>
            posizione++
        }
        diagonale++
    }
}
```

Cioè ci scorriamo degli indici e per ogni indice ci calcoliamo un determinato valore.
Se l'indice è uguale al valore che stiamo cercando, allora restituiamo la coppia di valori che abbiamo calcolato.

1.8.6 Domanda 6

Definire H_{any} e fornire una procedure che lo decide. Risposta

Possiamo definire H_{any} come:

$$H_{any} = \{< M > | M \text{ è una TM}, \exists w \in \Sigma^* \wedge M(w) \in \{y, n, h\} \}$$

Cioè, stiamo dicendo che:

- M è una macchina di Turing
- Esista una stringa w tale che $M(w)$ termini in uno stato finale

Per definire una procedura, però, non possiamo utilizzare la funzione $utm()$, questo perché se la macchina dovesse andare in loop sulla stringa w, non lo sapremmo mai. Questo non ci permette di trovare **almeno una stringa** che ci permette di terminare.
Allora dobbiamo utilizzare la funzione $utms()$, che a differenza di utm(), termina sempre la sua esecuzione.

Una procedura che ci permette di definire H_{any} è la seguente:

```
def decideH_any(M: TM-encoding){
    indice = 0
    while(true){
        w, s = int2Pair(indice)
        w = int2String(w)
        if(utms(M, w, s) == q)
```

```
                return true
            indice++
        }
    }
```

In questa procedura stiamo dicendo che, se la macchina M dovesse terminare in una stringa generata scorrendo tutti i numeri, cioè infiniti, allora se la macchina si dovesse fermare su almeno 1 stringa, quella macchina appartiene a H_{any}.

1.9 Procedure Semi-Decidibili 2

1.9.1 Domanda 1

Definire formalmente i due linguaggi H_{all} e $\neg H_{all}$
Risposta

H_{all} possiamo definirlo come segue:

$$H_all = \{< M > | M \in \Sigma - TM, \forall w \in \Sigma^* : M(w) \in \{y, n, h\}\}$$

Cioè stiamo definendo come l'insieme dell'encoding delle macchina su Σ tale che *per ogni stringa* $w \in \Sigma^*$, M(w) termina la sua esecuzione.

Al contrario, $\neg H_{all}$ possiamo definirlo come:

$$\neg H_{all} = \{< M > | M \in \Sigma - TM, \exists w \in \Sigma^* : M(w) \notin y, n, h\}$$

Cioe, stiamo dicendo che $\neg H_{all}$ è l'insieme dell'encoding di tutte le macchine tali che **esiste una** stringa in Σ^* tale che la macchina M su w non termini mai la sua esecuzione.

1.9.2 Domanda 2

Definire formarlmente H_{any} eH_{none} e discutere della loro relazione
Risposta

Possiamo dire che H_{any} è l'insieme di tutte le macchine di Turing che accettano almeno una stringa, mentre H_{none} è l'insieme di tutte le macchine di Turing che non accettano nessuna stringa.
Più formalmente:

$$H_{any} = < M > | M \in \Sigma - TM, \exists w \in \Sigma^* : M(w) \in \{y, n, h\}$$
$$H_{none} = < M > | M \in \Sigma - TM, \forall w \in \Sigma^* : M(w) \notin \{y, n, h\}$$

Cioè, per precisare H_{none} sono tutte le macchine tali per cui andranno in loop in qualsiasi stringa in Σ^*.

1.9.3 Domanda 3

Definire il complemento sintattico e semantico di:
$L = \{< M > \in < \Sigma - TM > | steps(M, ab) \neq steps(M, ba)\}_{\forall\Sigma}$
Risposta

- Complemento sintattico:

$$\bar{L} = \{< M > \notin < \Sigma - TM > | steps(M, ab) = steps(M, ba)\}_{\forall\Sigma}$$

- Complemento semantico:

$$\neg L = \{< M > \in < \Sigma - TM > | steps(M, ab) = steps(M, ba)\}_{\forall\Sigma}$$

1.9.4 Domanda 4

Fornire una procedura semi decidibile per:
$L = \{< M > \in < \Sigma - TM > | \exists w \in \Sigma^* : [|w| \leq | < M > | \wedge steps(M, w) > 0]\}_{\forall\Sigma}$ **Risposta**

Una procedura che può semi decidere questo linguaggio è la seguente:

```
bool semidecide_L (TM-Encoding M){
    for i in N{
        (w, steps) = int2Pair(i)
        w = int2String(w)
        q = utms(M,w, steps)
        if (|w| <= |M| && q in {y,n,h} )
            return true
    }
}
```

1.9.5 Domanda 5

L'altra domanda era praticamente uguale a quello che c'era nelle slides, per questo evito di scriverla.

1.10 Proprietà di D e SD

1.10.1 Domanda 1

Parla delle difference tra enumerable e turing enumerable
Risposta

Non so la differenza tra le due. Ma la **Turing enumeration** consiste nel poter stampare in un ordine, non per forza lessicografico, le stringhe di un linguaggio L.

In più, diciamo che un linguaggio ammette un'enumerazione di turing P se esiste una procedura get-L() che passato un indice i ci permette di ottenere la i-esima stringa che outputterebbe P.

1.10.2 Domanda 2

Qual è la cardinalità di SD? Perchè? Risposta
La cardinalità di SD è $infinito\ numerabile$.

Il motivo è perché siccome sappiamo che se un linguaggio appartiene ad SD, allora sappiamo che ci sarà una TM M che lo semidecide, tale che $\mathcal{L}(M) = L$. Ora, siccome sappiamo che il numero di TMs è infinito numerabile, poichè per un singolo linguaggio possiamo definire diverse macchine di Turing che lo riconoscono, sappiamo anche che $TMs \geq SDs$.

Allora, possiamo dire che SD è anch'esso infinito numerabile.

1.10.3 Domanda 3

Qual è la cardinalità di D? Perchè? Risposta

La risposta è molto semplice. Poichè D è un sottoinsieme di SD, che è *infinito numerabile*, allora anch'esso sarà infinito numerabile.

1.10.4 Domanda 4

Qual è la cardinalità di $\neg SD$? Risposta

Per dire che $\neg SD$ è infinito non numerabile, dobbiamo dire a cosa è uguale $\neg SD$

$$\neg SD = \mathcal{P}(\Sigma^*) - SD \tag{1.3}$$

Siccome sappiamo che il powerset $\mathcal{P}$ di un qualcosa infinito numerabile è infinito non numerabile, e sapendo che Σ^* è infinito numerabile, allora se togliamo un qualcosa di infinito numerabile ad un qualcosa di infinito non numerabile, il restante rimane comunque infinito non numerabile.

1.10.5 Domanda 5

Provare che se L ammette una numerazione di turing, allora L $\in SD$
Risposta
Se un linguaggio L ammette una numerazione di Turing significa che possiamo stampare l'i-esima stringa del linguaggio L utilizzando una procedura. Se un lnguaggio L apparteiene ad SD allora significa che possiamo scrivere una procedura che stampa termina in un determinato numero di passi se la stringa appartiene al linguaggio L, altrimenti potrebbe non terminare.
Dimostrazione L ha una Turing Enumeration

```
def LinSD(M: TM_encoding):
    for i in N:
        (x,y) = int2pair()
        w = x
        if (utms(M,w,y) == y AND utms(M,w,y-1) != y):
        print(w)
```

In questo modo stiamo stampando tutte le stringhe che si fermano esattamente in y passi.
Dimostrazione L $\in SD$
Per dimostrare ciò utilizziamo il metodo get-L() che ci ritornerà l'i-esima stringa del linguaggio.

```
def semidecideL(w: str):
    for i in N:
        if (get_L(i) == w):
            return True
```

1.10.6 Domanda 6

Prova che se L $\in SD$ allora L ammette una numerazione di Turing
Risposta
Se un linguaggio $L \in SD$ questo significa che il linguaggio è infinito numerabile. Allora, per questo motivo possiamo trovare una procedura che ci permetta di ricavare l'i-esima stringa del linguaggio. Una procedura può essere questa:

$$\forall w \in L : L \in SD$$

```
def turingEnum(i: integer):
    return get_L(i)
```

Non sono sicuro di questa cosa.

1.10.7 Domanda 7

Provare che se L ammette una numerazione lessicografica canonica, allora $L \in D$ Risposta
Se un linguaggio ammette una numerazione lessicografica canonica, allora possiamo scrivere una procedura che ci permette di stampare tutte le stringhe del linguaggio in ordine lessicografico. Ora, sapendo questo, come dimostriamo che il linguaggio $\in D$ per questo motivo?

Sappiamo che $D = < M > | M \in \Sigma - TM, \forall w \in \Sigma^*, M(w) \in y, n, h.$ Quindi, se per provare che $L \in D$ dobbiamo dimostrare che per ogni stringa $w \in L$ la macchina di Turing M si ferma in un numero finito di passi, e che se $w \notin L$ allora la macchina di Turing M rifiuta la stringa

```
def decideL(w:str):
    for i in N:
        ith = int2str(i)
        if (|ith| >= |w|):
            return False
        if (ith == w):
            return True
```

1.10.8 Domanda 8

Provare che se L $\in D$ allora ammette una numerazione lessicografica canonia
Riposta
Allora, per spiegare questo dobbiamo capire che se L $\in D$, possiamo sempre dire se una stringa appartiene al linguaggio perché ci sarà una macchina di turing M tale che essa andrà in uno stato di halting h, y, n. Se cosi, allora possiamo stampare tutte le stringhe che appartengono al linguaggio in ordine lessicografico.
Sia L un linguaggio tale che $L \in D$:

```
def Lenum(M: TM-encoding):
    for i in N:
        w = int2str(i)
        if (utm(M,w) == {y,n,h}):
            print(w)
```

In questo modo stiamo scorrendo tutte le stringhe e se la macchina di Turing si ferma in uno stato di halting, allora stampiamo la stringa.

1.10.9 Domanda 9

Provare che D è chiuso sotto unione-concatenazione-complemento-intersezione
Risposta

Unione
Iniziamo provando che D è chiuso sotto l'unione.
Siano $L_1, L_2 \in D$, allora sappiamo che $L_1 \bigcup L_2 \in D$. Per dimostrarlo, prendiamo 2 macchine che decidono i due linguaggio L_1 eL_2. Siano esse TM_1, TM_2.
Una procedura che decide $L_1 \bigcup L_2$ può essere la seguente:

```
def decideUnion(w: str, TM_1: TM-encoding, TM_2: TM-encoding):
    if (utm(TM_1,w) == y || utm(TM_2,w) == y):
        return True
    else:
        return False
```

Stiamo dicendo che se questa stringa appartiene a uno dei due linguaggi, allora una delle due macchine di turing dei due linguaggi accetterà la stringa w.

Intersezione Proviamo ora che se D è chiuso sotto $\cap$.

Siano $L_1, L_2 \in D$, allora sappiamo che $L_1 \cap L_2 \in D$. Per dimostrarlo, prendiamo 2 macchine che decidono i due linguaggio L_1 eL_2. Siano esse TM_1, TM_2.
Una procedura che decide $L_1 \cap L_2$ può essere la seguente:

```
def decideIntersect(w:str, TM_1: TM-Encoding, TM_2:T TM-Encoding):
    if(utm(TM_1,w) == y && utm(TM_2,w) == y)
        return True
    else:
        return False
```

In questo modo stiamo dicendo che se entrambe le macchine riconoscono la stringa, allora essa appartiene all'intersezione dei due linguaggi. Possiamo sempre deciderlo poiché entrambi i linguaggi sono in D. Se la stringa non dovesse appartenere al linguaggio, essa verrà rifiutata e non andrà mai in loop.

Concatenazione
Proviamo ora che D è chiuso sotto la concatenazione $\circ$.
Siano $L_1, L_2 \in D$, allora sappiamo che $L_1 \circ L_2 \in D$. Per dimostrarlo, prendiamo 2 macchine che decidono i due linguaggio L_1, L_2. Siano esse TM_1, TM_2.
Una procedura che decide $L_1 \circ L_2$ può essere la seguente:

```
decideConcat(w: str, TM_1: TM-encoding, TM_2: TM-encoding):
    for i in |w|:
        if (utm(TM_1, w[0:i]) == y && utm(TM_2, w[i:|w|]) == y):
            return True
    return False
```

In questo modo stiamo dicendo che, data una stringa, se le due macchine accettano entrambe *la stringa dall'inizio fino ad un indice i* e l'altra accetta *dall'indice i fino alla fine della stringa*, allora la stringa appartiene alla concatenazione dei due linguaggi.
Possiamo dire che la procedura non andrà mai in loop poiché i due linguaggi sono in D e quindi possiamo sempre dire se una stringa appartiene o meno al linguaggio.

Complemento

Proviamo ora che D è chiuso sotto il complemento.
Per provarlo, basta prendere un linguaggio $L \in D$ e dimostrare che $\overline{L} \in D$.
Una procedura per farlo è la seguente:

```
def decideComplement(w: str, TM: TM_encoding):
    if (utm(TM, w) == y):
        return False
    else:
        return True
```

In questo modo stiamo semplicemente negando il valore di ritorno della macchina di turing. Se essa accetta la stringa, allora il complemento la rifiuterà e viceversa.
Sappiamo che non andremo mai in loop poiché $L \in D$ e quindi possiamo sempre dire se una stringa appartiene o meno al linguaggio.

1.10.10 Domanda 10

Provare che SD è chiuso sotto unione-intersezione-concatenazione
Risposta
Attenzione, in queste procedure **NON RIUSCIREMO SEMPRE A TORNARE FALSE, POICHE' NON SAPPIAMO QUANDO FERMARCI.**
Unione
Proviamo che SD è chiuso sotto l'unione.
Per provare che SD è chiuso sotto l'unione $\bigcup$, prendiamo due linguaggi $L_1, L_2 \in SD$.
Questo significa che se una stringa $w \in L1$ *oppure* $w \in L2$, allora esiste una macchina di Turing M che accetta w, e se non appartiene al linguaggio andrà in loop.
Possiamo costruire una procedura che ci permette di sfruttare questa proprietà di L_1 e L_2.

```
semidecideUnion(w:str,TM1:TM_encoding,TM2:TM_encoding):
    for i in N:
        if (utms(TM1,w,i) == y || utms(TM2,w,i) == y):
            return True
```

Cioè, stiamo dicendo che se una delle due macchine accetta la stringa in un numero fissato di passi i, allora la stringa appartiene all'unione dei due linguaggi.
Non possiamo sapere se la procedura terminerà mai, quindi possiamo dire che la prendiamo come **False** quando questa andrà in loop.

Intersezione

Proviamo che SD è chiuso sotto l'intersezione.

Per provare che SD è chiuso sotto l'intersezione $\bigcap$, prendiamo due linguaggi $L_1, L_2 \in SD$.
Questo significa che se una stringa $w \in L1$ *e* $w \in L2$, allora entrambe le macchine che semidecidono L_1 e L_2 accetteranno w in un numero preciso di passi, e se non appartiene al linguaggio andrà in loop.
Possiamo costruire una procedura che ci permette di sfruttare questa proprietà di L_1 e L_2.

```
semidecideIntersect(w:str,TM1:TM-encoding,TM2:TM-encoding):
    for i in N:
        if (utms(TM1,w,i) == y && utms(TM2,w,i) == y):
            return True
```

Cioè, stiamo dicendo che se entrambe le macchine accettano la stringa in un numero fissato di passi i, allora la stringa appartiene all'intersezione dei due linguaggi.

Concatenazione

Proviamo che SD è chiuso sotto la concatenazione.
Per provare che SD è chiuso sotto la concatenazione $\circ$, prendiamo due linguaggi $L_1, L_2 \in SD$.
Questo significa che se una stringa $w \in L1 \circ L2$, allora significa che possiamo dividerla in due parti $w_1 \in L_1$ e $w_2 \in L_2$.
Possiamo costruire una procedura che ci permette di sfruttare questa proprietà di L_1 e L_2.

```
semidecideconcat(w:str,TM1:TM-encoding,TM2:TM-encoding):
    for steps in N:
        for i in |w|:
            w1 = w[0:i]
            w2 = w[i-|w|]
            if (utms(TM1, w, steps) == y && utms(TM2, w, steps) == y):
                return True
```

In questo modo stiamo scorrendo gli steps da 1 per tutti i numeri reali e stiamo dividendo la stringa in tutte le possibili parti, per ogni steps.
Se entrambe le macchine accettano le due stringhe che vengono passate, allora la stringa apparteiene a $L_1 \circ L_2$.
Non sappiamo se la procedura terminerà mai, quindi possiamo dire che la prendiamo come **False** quando questa andrà in loop.

1.10.11 Domanda 11

Dimostra che SD NON è chiuso sotto il complemento
Risposta
Per quale motivo SD non è chiuso sotto il complemento? Vediamo.

Prendiamo un linguaggio $L \in SD$.

Se L fosse chiuso sotto complemento, vorrebbe dire che anche il suo complemento sarebbe in SD.
Ma se il complemento fosse in SD, significherebbe che potremmo costruire una macchina di turing che accetta le stringhe in L e una macchina di turing che le rifiuta tutte. In particolare:

$\mathcal{L}(M1) = L$ e $\mathcal{L}(M2) = \overline{L}$

Una procedura che ci mostra che questo non è possibile è la seguente:

```
def semidecideComplement(w: str, TM1: TM–encoding, TM2: TM–encoding):
    for steps in N:
        if (utms(M1,w,steps) == y)
            return True
        if (utms(M2,w,steps) == y)
            return False
```

Questa procedura funzione ma ha un problema: **Non va mai in loop!**. Abbiamo quindi costruito una procedura decidibile, poiché stiamo rifiutando e accettando tutte le stringhe, ma non va mai in loop.
Quindi questa procedura dimostra che non SD non è chiuso sotto il complemento.

1.10.12 Domanda 12

Dimostra che $SD \bigcap CO - SD = D$
Risposta
Partiamo definendo cosa è SD, cioè l'insieme dei Linguaggi tale che essi siano semidecidibili.

Se un linguaggio è semi-decidibile, vuol dire che $\forall w \in \Sigma^*$, se $w \in L$ allora la macchina di Turing M accetta w in un numero finito di passi, altrimenti va in loop.
Quindi se $L \in SD$, allora $\exists M$ tale che $\mathcal{L}(M) = L$.

Definiamo ora $CO - SD$, cioè l'insieme dei linguaggi che sono complementari di un linguaggio semidecidibile.
Se u linguaggio è co-semi-decidibile, vuol dire che $\forall w \in \Sigma^*$, se $w \notin L$ allora la macchina di Turing M rifiuta w in un numero finito di passi, altrimenti va in loop.
Quindi se $L \in CO - SD$, allora $\exists M$ tale che $\mathcal{L}(M) = \neg L$.

Per provare che $SD \bigcap CO - SD = D$, dobbiamo provare una doppia inclusione stretta.
Inclusione 1
Proviamo che $D \subseteq SD \bigcap CO - DS$.
Questo vale per definizione, poichè stiamo dicendo che in SD stiamo prendendo la terminazione delle stringhe che appartengono al linguaggio, mentre per CO-SD stiamo prendendo la terminazione delle stringhe che non appartengono al linguaggio.
Quindi se prendiamo l'intersezione di questi due insiemi, stiamo prendendo la terminazione di tutte le stringhe, cioè D.

Inclusione 2
Per provare l'inclusione $SD \bigcap CO - SD \subseteq D$, possiamo usare una procedura che ci permetta di mostrare che l'intersezione delle di due macchine appartenenti rispettivamente a SD e $CO - SD$ è decidibile.

```
bool decideL(w:str, M1:TM, M2:TM){
    for steps in N{
            if (utms(M1,w,steps) == y) return True
            if (utms(M2,w,steps) == y) return False
        }
```

}

Abbiamo creato una procedura che data una stringa,utilizzando una macchina in SD e una in CO-SD, ci permette di decidere se la stringa appartiene o meno all'intersezione dei due linguaggi.
Questa procedura è decidibile, poichè stiamo accettando e rifiutando tutte le stringhe, ma non va mai in loop.
Quindi abbiamo dimostrato che $SD \bigcap CO - SD \subseteq D$.

1.11 L'indecibilita' dell'Halting Problem

1.11.1 Domanada 1

Dimostrare che $H \notin D$
Risposta

Innanzitutto dobbiamo descrivere cosa e' H. Con H indichiamo tutte le coppie din TM e W tale che TM si arresta su W.
Per essere precisi:

$$H = \{< M, w > \in \Sigma - TM \times \Sigma^* | steps(M, w) > 0\}_{\forall \Sigma} \tag{1.4}$$

Precisamente, tutte le coppie di TM in $\Sigma - TM$ e le stringhe in Σ tali che il numero di steps di M su w sia finito, e non vada in loop.

Spiegazione del perché $H \notin D$
Ipotizziamo per assurdo che $H \in D$. Questo significa che esiste una macchina M_H tale che H = $L(M_H)$.
Questo vorrebbe dire che possiamo costruire una procedura che ci permette di dire sempre se una macchina termina su una stringa oppure no.

Tale procedura sarebbe di questo tipo:

```
def decide_H(TM M, string w){
    if (w ∈ Σ*_M and utm(M_H , <M,w>) == y)
        return true
    else
        return false
}
```

Cioe stiamo controllando utilizzando la macchina M_H che esiste per la nostra ipotesi iniziale, che prendendo in input una coppia di macchina e stringa ci dice se se la macchina M riconosce la stringa w.

Possiamo costruire utilizzando questa procedura, una nuova procedura chiamata **trouble()** che sara' definita cosi':

```
def trouble(string w){
    if (w ∈ Σ − TMS AND decide_H(w,w) == true)
        while(true)
}
```

In questo modo abbiamo costruito una procedura che sfrutta la procedura decide_H() per se la stringa e' una codifica di una macchina e se macchina termina sulla codifica di se stessa, allora andra' in loop.

Consideriamo ora quindi di avere una macchina M_T che e' equivalente alla procedura trouble.

Immaginiamo ora di dare in input alla macchina trouble, proprio la stessa macchina trouble.

Possiamo avere due casi:

1. **Trouble termina**: Se trouble termina significa che la procedura termination ha tornato false, quindi tecnicamente dovremmo essere in un loop. Ma se trouble ha terminato allora non è possibile che termination non termini. CONTRADDIZIONE

2. **Trouble va in loop**: Se trouble va in loop, allora significa che termination ha ritornato true. Ma se termination ha ritornato true, questo significa che la macchina termina su se stessa, e non sarebbe possibile poiché trouble è andat ain loop. CONTRADDIZIONE

1.11.2 Domanda 2

Provare che se $H \in D$, allora ogni linguaggio SD sarebbe in D
Risposta

Per provare questa cosa, dobbiamo quindi supporre che $H \in D$. Questo significa che esiste una macchina M_H tale che passato in input un encoding di una macchina e una stringa, essa ci dice se quella macchina termina la propria esecuzione in numero finito di passi su quella stringa.

Ora, sapendo questo, tutte le procedure che semidecidono un linguaggio diventano decidibili. Prendiamo quindi una procedura che semidecide un linguaggio. Sia $L \in SD$, allora esiste una M tale che $L = L(M)$. Allora scriviamo la procedura che semidecide L:

```
def semidecide_L(string w){
    for steps ∈ℕ{
        if (utms(M,w,steps)==y)
            return true
    }
}
```

Questa procedura potrebbe andare in loop poiche' potremmo raggiungere un numero di steps infinito.

Ora, noi possiamo utilizzare la macchina M_H per controllare se la macchina M termina su w. Una procedura che dimostra questo e' la seguente:

```
def decide_L(string w){
    if (decide_H(M,w) == true and utm(M,w)==y)
        return true
    else
        return false
}
```

In questo modo stiamo controllando e stiamo decidendo per ogni stringa se essa appartiene al linguaggio della macchina, rifiutando e accettando ogni possibilita'.

Allora, per questo motivo, ogni linguaggio semidecidibile diventa decidibile, e quindi ogni linguaggio semidecidibile e' in D.

1.11.3 Domanda 3

Definire H_ϵ
Risposta
H_ϵ e' definito come l'insieme di tutte le macchine di turing tali che esse terminino sulla stringa vuota.

$$H_\epsilon = \{< M > \in < \Sigma - TM > | M(\epsilon) \neq loop\}_{\forall \Sigma} \tag{1.5}$$

Tra l'altro, possiamo anche dire che H_ϵ non e' decidibile. Questo perche' come per H noi non conosciamo tutte le macchine che terminano sulla stringa vuota, e quindi siamo obbligati ad utilizzare una procedura che andrebbe in loop.

1.11.4 Domanda 4

Definire $H_\epsilon^{\leq 3245}$
Risposta
$H_\epsilon^{\leq 3245}$ e' definito come l'insieme di tutte le macchine di turing tali che esse terminino sulla stringa vuota e che la cardinalita' —M— sia minore o uguale a 3245.

$$H_\epsilon^{\leq 3245} = \{< M > \in < \Sigma - TM > | M(\epsilon) \neq loop \wedge |M| \leq 3245\} \tag{1.6}$$

E tra l'altro, $H_\epsilon^{\leq 3245}$ e' decidibile. Questo perche' questo insieme e' ben definito e limitato superiormente. **Non possiamo costruire infinite macchine che non vallo in loop sulla stringa vuota e che hanno cardinalita' minore di 3245**, questo appunto perche' l'insieme e' limitato.
In questo caso il problema e' diventato regolare ($\in REG$) ed e' possibile costruire un automa che risolva tale problema

1.12 Turing e Mapping Reductions

1.12.1 Domanda 1

Definizione di Turing Reduction
Risposta

Dati due linguaggi L_1 ed L_2. Assumiamo di avere una procedura che risolve L_2 chiamata Π_2 e una procedura che risolve L_1 chiamata Π_1.

Una Turing Reduction da L_1 a L_2 anche scritta come $L_1 \leq L_2$ e' un algoritmo che risolve L_1 utilizzando Π_2 come una blackbox.

Un esempio e' una macchina di Turing M1 che risolve L_1 e che utilizza una macchina di Turing M2 che risolve L_2 come una subroutine.

1.12.2 Domanda 2

Definizione di Mapping Reduction
Risposta

Dati due linguaggi L_1 su un alfabeto Σ_1 ed L_2 su un alfabeto Σ_2. Una Mapping Reduction da L_1 a L_2, anche scritta come $L_1 \leq_M L_2$ ρ e' una funzione computabile definita come: $\rho : \Sigma^1 \to \Sigma^2$, ovvero un algoritmo tale che:

$$\forall x \in \Sigma_1, x \in L_1 \iff \rho(x) \in L_2 \tag{1.7}$$

Cioe' e' una funzione che mappa ogni valore di L_1 ad un valore in L_2.

1.12.3 Domanda 3

Dimostrare che se $L_2 \in D$ e $L_2 \leq_M L_1$, allora $L_1 \leq L_2$
Risposta

Dobbiamo dimostare che se L_2 e' decidibile e la Mapping Reduction da L_2 a L_1 esiste, esiste una Turing Reduction da L_1 a L_2. Se L_2 e' decidibile, allora esiste un algoritmo Π_2 che risolve L_2. Sia ρ una funzione computabile tale che $\rho : \Sigma^1 \to \Sigma^2$ tale che $\forall x \in \Sigma^1, x \in L_1 \iff \rho(x) \in L_2$.

Da questo sappiamo che $\Pi_1 = \Pi_2 \circ \rho$ e' una turing reduction da L_1 a L_2.

$$\forall w \in \Sigma^1, \Pi_1(w) = \Pi_2(\rho(w)) \tag{1.8}$$

Spiegazione per i posteri Se L2 e' decidibile esiste questa procedura Π_2 che risolve L_2. In piu', siccome esiste una Mapping Reduction da L_2 a L_1, esiste una funzione ρ che mappa ogni valore di L_1 ad un valore in L_2.

Se concateniamo il nostro algoritmo Π_2 con la funzione ρ otteniamo un algoritmo che risolve L_1.

1.12.4 Domanda 4

Dimostrare che se esiste una Mapping Reduction $L_1 \leq_M L_2$ e che $L_2 \in D$, allora $L_1 \in D$
Risposta

Se esiste una mapping reduction $L_1 \leq_M L_2$ e $L_2 \in D$, allora esiste una funzione computabile $\rho : \Sigma^1 \to \Sigma^2$ tale che $\forall x \in \Sigma^1, x \in L_1 \iff \rho(x) \in L_2$.

Se $L_2 \in D$, allora esiste un algoritmo Π_2 che risolve L_2. Da questo cosa possiamo dedurre? Per dimostrare che L_1 appartiene a D:

1. Se esiste un algoritmo che risolve L_2
2. Se esite una mapping reduction da L_1 a L_2
3. Allora esiste un algoritmo che risolve L_1 che viene definito come $\Pi_1 = \Pi_2 \circ \rho$

$$\forall w \in \Sigma^1, \Pi_1(w) = \Pi_2(\rho(w)) \tag{1.9}$$

1.12.5 Domanda 5

Dimostrare che se $L_1 \leq L_2$ e $L_1 \notin D$, allora $L_2 \notin D$
Risposta
Per dimostrare questo supponiamo che L_1 sia indecidibile e che L_2 sia decidibile. Se L_1 e' indecidibile, allora non esiste un algoritmo Π_1 che risolve L_1. Ma noi sappiamo che se esiste una Turing Reduction da L_1 a L_2, allora la funzione ρ esiste ed definita come $\rho : \Sigma^1 \to \Sigma^2$ tale che $\forall x \in \Sigma^1, x \in L_1 \iff \rho(x) \in L_2$.

Ora, sapendo che L_2 e' decidibile, allora esiste un algoritmo Π_2 che risolve L_2.

Come sappiamo, siccome esiste la mapping reduction da L_1 a L_2, allora dovrebbe esistere un algoritmo Π_1 che risolve L_1 che viene definito come $\Pi_1 = \Pi_2 \circ \rho$. Ma noi sappiamo che L_1 e' indecidibile, quindi e' una **contraddizione**.

1.12.6 Domanda 6

Dimostra che se $L_2 \in SD$ e $L_1 \leq L_2$ allora $L_1 \in SD$
Risposta

Se $L_2 \in SD$ allora esiste un algoritmo Π_2 che semi-decide L_2.

Se $L_1 \leq L_2$ allora esiste una Turing Reduction da L_1 a L_2, ovvero esiste una funzione $\rho : \Sigma^1 \to \Sigma^2$ tale che $\forall x \in \Sigma^1, x \in L_1 \iff \rho(x) \in L_2$.

Da questo possiamo dire che esiste un algoritmo Π_1 che semi-decide L_1 che viene definito come $\Pi_1 = \Pi_2 \circ \rho$, ovvero:

$$\forall w \in \Sigma_1^*, \Pi_1(w) = \Pi_2(\rho(w)) \tag{1.10}$$

1.12.7 Domanda 7

Dimostare che se $L_2 \in CO - SD$ e $L_1 \leq L_2$ allora $L_1 \in CO - SD$
Risposta

Se $L_2 \in CO - SD$ allora esiste un algoritmo Π_2 che semi-decide $\neg L_2$. Se $L_1 \leq L_2$ allora esiste una Turing Reduction da L_1 a L_2, ovvero esiste una funzione $\rho : \Sigma^1 \to \Sigma^2$ tale che $\forall x \in \Sigma^1, x \in L_1 \iff \rho(x) \in L_2$. Se Π_2 semi-decide $\neg L_2$, allora $\Pi_1 = \Pi_2 \circ \rho$ semi-decide $\neg L_1$. Precisamente, e' definita come:

$$\forall w \in \Sigma_1^*, \Pi_1(w) = \Pi_2(\rho(w)) \tag{1.11}$$

1.12.8 Domanda 8

Dimostrare che $H_\epsilon \notin D$ Risposta

Per questa dimostrazione utilizziamo una reduction da $H \leq_M H_\epsilon$, Cioe' da H a H_ϵ.

Abbiamo quindi $\rho :< \Sigma - TM \times \Sigma^* > \to \Sigma' - TMs$.

Cioe' avremo un qualcosa del tipo: $< M, W > \to \rho < M' >$.

Definiamo la nuova macchina M' come una macchina che all'interno chiama la macchina M. In particolare, la macchina esegue le seguenti operazioni:

- scrive la stringa w sul nastro.

- Va a sinistra fino al primo blank
- Chiama la macchina M
- Si ferma

Il comportamento dipende quindi da come si comporta M su un determinato input.
Dobbiamo dimostrare che ρ sia una reduction, dimostrando la doppia implicazione $\iff$

- $\implies$ Se $< M, W > \in H$ allora sappiamo che M(w) termina in uno stato di halt (y,n,h). Questo allora comporta che anche M' termina in uno stato di halt (y,n,h). Quindi $< M' > \in H_\epsilon$.
- $\impliedby$ Se $< M, W > \notin H$ allora sappiamo che M(w) non termina. Questo allora comporta che anche M' non termina. Quindi $< M' > \notin H_\epsilon$.

1.12.9 Domanda 9

Dimostrare che $H_{any} \notin D$ Risposta

Per dimostrare questo utilizziamo una reduction da $H_\epsilon \leq_M H_{any}$, cioe' da H_ϵ a H_{any}. Una reduction ρ da H_ϵ a H_{any} e' una funzione $\rho :< \Sigma - TMs > \to \Sigma' - TMs$. Dobbiamo quindi dimostrare che:

$$< M > \in H_\epsilon \iff \rho(< M >) \in H_{any} \tag{1.12}$$

La reduction sara' del tipo: $< M > \to \rho \to < M' >$, con M' che e' una macchina che esegue le seguenti operazioni:

- Va a destra
- Se legge qualcosa $\neq \square$, allora scrive $\square$ e va a destra
- Se legge $\square$, chiama la macchina M
- Si ferma

Cioe' e' una macchina che svuoa il nastro e chiama la macchina M, che appartiene a H_ϵ.
Dobbiamo dimostrare che ρ sia una reduction, dimostrando la doppia implicazione $\impliedby \implies$

- $\implies$ Se $< M > \in H_\epsilon$ allora sappiamo che M(w) termina in uno stato di halt (y,n,h). Questo allora comporta che anche M' termina in uno stato di halt (y,n,h). Quindi $< M' > \in H_{any}$.
- $\impliedby$ Se $< M > \notin H_\epsilon$ allora sappiamo che M(w) non termina. Questo allora comporta che anche M' non termina. Quindi $< M' > \notin H_{any}$.

1.12.10 Domanda 10

Dimostrare che $H_{all} \notin D$ Risposta

Per dimostrare questo utilizziamo una reduction da $H_\epsilon \leq_M H_{all}$, cioe' da H_ϵ a H_{all}. Una reduction ρ da H_ϵ a H_{all} e' una funzione $\rho :< \Sigma - TMs >\rightarrow \Sigma' - TMs$. Dobbiamo quindi dimostrare che:

$$< M >\in H_\epsilon \iff \rho(< M >) \in H_{all} \tag{1.13}$$

La reduction sara' del tipo: $< M >\rightarrow \rho \rightarrow < M' >$, con M' che e' una macchina che esegue le seguenti operazioni:

- Va a destra
- Se legge qualcosa $\neq \square$, allora scrive $\square$ e va a destra
- Se legge $\square$, chiama la macchina M
- Si ferma

Dobbiamo dimostrare che ρ sia una reduction, dimostrando la doppia implicazione $\Longleftarrow \Longrightarrow$

- $\Longrightarrow$ Se $< M >\in H_\epsilon$ allora sappiamo che M(w) termina in uno stato di halt (y,n,h). Questo allora comporta che anche M' termina in uno stato di halt (y,n,h). Quindi $< M' >\in H_{all}$.
- $\Longleftarrow$ Se $< M >\notin H_\epsilon$ allora sappiamo che M(w) non termina. Questo allora comporta che anche M' non termina. Quindi $< M' >\notin H_{all}$.

1.12.11 Domanda 11

Dimostrare che $\neg H_{all} \notin D$ Risposta

Utilizziamo una reduction da $H_\epsilon \leq_M \neg H_{all}$, cioe' da H_ϵ a $\neg H_{all}$.
Una reduction ρ da H_ϵ a $\neg H_{all}$ e' una funzione $\rho :< \Sigma - TMs >\rightarrow \Sigma' - TMs$. Dobbiamo quindi dimostrare che:

$$< M >\in H_\epsilon \iff \rho(< M >) \in \neg H_{all} \tag{1.14}$$

La reduction sara' del tipo: $< M >\rightarrow \rho \rightarrow < M' >$, con M' che e' una macchina che esegue le seguenti operazioni:

- Chiama il metodo $utms(M, \epsilon, |w|)$
- Salva lo stato della funzione
- Se lo stato e' uno stato di halt, va in loop
- Altrimenti, ritorna lo stato della funzione

Dobbiamo dimostrare che ρ sia una reduction, dimostrando la doppia implicazione $\Longleftarrow \Longrightarrow$

- $\Longrightarrow$ Se $< M >\in H_\epsilon$ allora sappiamo che M(w) termina in uno stato di halt (y,n,h). Questo significa che, per come funziona M', la macchina andra' in loop. Per questo motivo, $M' \in \neg H_{all}$.
- $\Longleftarrow$ Se $< M >\notin H_\epsilon$ allora sappiamo che M(w) non termina. Questo significa che, per come funziona M', la macchina non andra' in loop. Per questo motivo, $M' \notin \neg H_{all}$.

1.12.12 Domanda 12

Dimostrare che $H_{all} \wedge \neg H_{all} \notin SD \wedge \notin CO - SD$
Risposta
Per dimostrare questo teorema, supponiamo di avere un linguaggio $L_1 \in SD \setminus D$ e un linguaggio L2. Supponiamo di avere una reduction $L_1 \leq_M L_2$ e $L_1 \leq_M \neg L_2$.

Ora, assumiamo che $L_2 \in SD$ e che ovviamente $\neg L_2 \in CO - SD$. In questa situazione, siccome per ipotesi abbiamo che $L_1 \leq_M \neg L_2$, significherebbe che L_1 e' in CO-SD, ma questa sarebbe una contraddizione perche' abbiamo ipotizzato che L_1 appartenesse a $SD \setminus D$.

Per lo stesso motivo, se $\neg L_2 \in SD$ e $L_2 \in CO - SD$, allora siccome esiste una mapping reduction da L_1 a L_2, allora $L_1 \in CO - SD$ e $L_1 \in SD$, che e' una contraddizione poiche' $L_1 \in SD \setminus D$.

1.13 Boolean Logic

1.13.1 Domanda 1

Descrivi in modo informale una procedura che decide SAT,UNSAT E VALID
Risposta

Una procedura che permette di decidere SAT, UNSAT E VALID, che sono:

1. SAT — L'insieme dell'encoding delle wff ϕ che sono soddisfacibili

2. UNSAT — L'insieme dell'encoding delle wff ϕ che non sono soddisfacibili

3. VALID — L'insieme dell'encoding delle wff ϕ che sono valide

Avremo in modo preciso che:

1. ϕ e' SAT se esiste almeno assegnamento di verita' che soddisfa ϕ

2. ϕ e' UNSAT se non esiste alcun assegnamento di verita' che soddisfa ϕ

3. ϕ e' VALID se per ogni assegnamento di verita' ϕ e' soddisfatta

Per provare che sono in D, dobbiamo pensare al concetto di assegnamento di verita' minimale.

Un assegnamento di verita e' una mappatura $v : \{x_1, x_2, ..., x_n\} \to \{0, 1\}$ che associa ad ogni variabile un valore di verita'.
Un assegnamento di verita' e' minimale se le variabili della formula ϕ sono uguali alle variabili dell'assegnamento di verita'.

Per provare che sono in D, dobbiamo pensare al concetto di assegnamento di verita' minimale.

Se il numero di variabili di una formula ϕ e' n, allora il numero di assegnamenti di verita' minimali e' 2^n.
Per questo motivo possiamo enumerare ogni possibile assegnamento di verita'.

1.13.2 Domanda 2

Definire entailment e ENTAIL
Risposta

Entailment:

Dati un insieme logico $A = \phi_1, \phi_2, ..., \phi_n$ e una formula ϕ, diciamo che ϕ e' entailment di A se e solo se ogni assegnamento di verita' che soddisfa A soddisfa anche ϕ.

$$A \models \phi \iff \forall v : v \models A \implies v \models \phi \tag{1.15}$$

ENTAIL:

Entail lo definiamo come l'insieme delle coppie $< A, \phi$ tali che ϕ e' una boolean wff e A e' un insieme di assiomi, e $A \models \phi$.

$$ENTAIL = \{< A, \phi > | \phi \in WFF, A \; insieme \; di \; assiomi, A \models \phi\} \tag{1.16}$$

1.13.3 Domanda 3

Definire provability e PROVABLE
Risposta

Quando parliamo di provability intendiamo che una formula ϕ è provabile se esiste una prova tale che, utilizzando un'insieme di assiomi A, si arriva a ϕ, ovvero $A \vdash \phi$.
Noi diciamo che esiste una proof se, dato questo insieme di assiomi A, esiste una sequenza di formule $\phi_1, \phi_2, ..., \phi_n$ tale che $\phi_n = \phi$ e ϕ_i e' o un assioma o una formula che si ottiene da una regola di inferenza applicata alle formule precedenti.

PROVABLE:

Provability lo definiamo come l'insieme delle coppie $< A, \phi$ tali che ϕ e' una boolean wff e A e' un insieme di assiomi, e $A \vdash \phi$.

1.13.4 Domanda 4

Fornire una reduction da ENTAIL a VALID
Risposta

Una reduction da $ENTAIL \leq_M VALID$ significa provare che $VALID \in D \implies ENTAILS \in D$.

Questa reduction e' formata nel seguente modo:

1. INPUT: Un insieme A di assiomi booleani e una Bwff ϕ
2. OUTPUT: ϕ' che e' uguale a $((\phi_1 \wedge ... \wedge \phi_n) \implies \phi)$

1.13.5 Domanda 5

Fornire una procedura che decide ENTAIL Risposta
Penso che una procedura sarebbe quello di scorrere tutti i possibili assegnamenti di verità minimali e devo controllare che soddisfa ogni singola formula dell'insieme di assiomi A e che soddisfa anche la formula ϕ.

1.13.6 Domanda 6

Parlare di un set di regole di inferenza sound e complete
Risposta

Un insieme di regole di inferenza si dice **sound** quando l'insieme contiene solamente regole di inferenza che producono formule vere a partire da formule vere. Quindi praticamente riesco a generare usando regola di inferenza regole vere e basta.
Un insieme di regole di inferenza si dice **complete** quando tutti gli elementi veri al suo interno hanno una proof che li conferma.

1.13.7 Domanda 7

Domanda con formule
Risposta

1.13.8 Domanda 8

Descrivere il concetto di Proof System
Risposta

Un proof system e' un insieme di regole di inferenza che permettono di derivare nuove formule da formule gia' esistenti. Possiamo dirlo anche se un proof system è un insieme di passi in sequenza che portano ad una conclusione desiderata.
Ad esempio, prendendo un insieme di assiomi A e una Bwff ϕ, esiste una **proof** se esiste una sequenza di formule $\phi_1, \phi_2, ..., \phi_n$ tale che $\phi_n = \phi$ e ϕ_i e' o un assioma o una formula che si ottiene da una regola di inferenza applicata alle formule precedenti.

1.13.9 Domanda 9

Descrivere il concetto regole di inferenza
Risposta

Una regola di inferenza è un meccanismo che permette di ricavare delle formula partendo dagli assiomi. Cioè, avendo un insieme di assiomi A e una formula ϕ, una regola di inferenza permette di ricavare ϕ a partire da A.

Quidi, $A \models \phi$.
Alcuni esempi sono:

- Modus ponens: $A \wedge (A \implies B) \implies B$
- Modus tollens: $(A \implies B) \wedge \neg B \implies \neg A$

1.14 First Order Logic

1.14.1 Domanda 1

Definire il concetto di Interpretazione in FOL
Risposta
Considerando un **vocabolario V** = $\{\phi, \Pi, r\}$, dove:

1. ϕ E' un insieme di simboli di funzione
2. Π E' un insieme di simobli di relazione
3. r E' una funzione che associa ad ogni simbolo di funzione un numero naturale che rappresenta l'arità del simbolo di funzione

Un'interpretazione in First Order Logic è una coppia del tipo $I = (U, \mu)$, con U che è un insieme non vuoto, è l'universo dell'interpretazione I che contiene gli elementi che lo compongono, e μ è la funzione che mappa gli elementi di $Y \cup \Phi \cup \Pi$ agli elmenti di U

1.14.2 Domanda 2

Definire il concetto di soddisfacibilità e validità in FOL
Risposta

Soddisfacibilità
Data una FOE ϕ definita su Y e V, questa si dice **soddisfacibile** se $\exists$ un'interpretazione I = (U,μ) tale che $\mu \models \phi$
Cioè una ϕ è soddisfatta da un'interpretazione I se si riesce a creare un mapping tra gli elementi del vocabolario e gli elementi dell'universo U e la formula in quel contesto avrà un valore di verità vero.

Validità Data una FOE ϕ definita su Y e V, questa si dice **valida** se $\forall$ I = (U,μ): $\mu \models \phi$, ovvero se tutte le interpretazioni I sono modelli di ϕ

1.14.3 Domanda 3

Definire il concetto di Entailment in FOE Risposta

Data un vocabolario $V = \{\phi, \Pi, r\}$, un'insieme $A = \{\phi_1, \ldots, \phi_n\}$ su Y e V e una FOE ϕ anch'essa su V e Y, diciamo che l'insieme A implica logicamento o entails ϕ, anche scritto $A \models \phi$ se, per ogni interpretazione I, I soddisfa ϕ se I soddisfa anche ogni formula ϕ_i in A.

1.14.4 Domanda 4

Definire FOL-VALID, FOL-SAT, FOL-UNSAT e FOL-ENTAIL Risposta

FOL-VALID

E' l'insieme di tutte le FOE valide, cioè:

$$FOL - VALID = \{< \phi >: \phi\ FOE \wedge \phi\ VALIDA\} \tag{1.17}$$

FOL-SAT

E' l'insieme di tutte le FOE soddisfacibili, cioè:

$$FOL - SAT = \{< \phi >: \phi\ FOE \wedge \phi\ SODDISFACIBILE\} \tag{1.18}$$

FOL-UNSAT

E' l'insieme di tutte le FOE insoddisfacibili, cioè:

$$FOL - UNSAT = \{< \phi >: \phi\ FOE \wedge \phi\ INSODDISFACIBILE\} \tag{1.19}$$

FOL-ENTAIL

E' l'insieme delle coppia A, ϕ tale che A sia un'insieme di assiomi non logici, ϕ è una FOE e $A \models \phi$.

$$FOL - ENTAIL = \{< A, \phi >: A \in Assiomi\ Logici \wedge \phi\ FOE \wedge A \models \phi\} \tag{1.20}$$

1.14.5 Domanda 5

Definire il vocabolario $V_N = \{\Phi_N, \Pi_N, r_N\}$ e l'interpretazione $N = (\mathcal{N}, \mu)$ Risposta

Vocabolario

Gli elementi del vocabolario. Inizimao da Φ_N che è un insieme di simboli di funzione, Π_N che è un insieme di simboli di relazione e r_N che è una funzione che associa ad ogni simbolo di funzione un numero naturale che rappresenta l'arità del simbolo di funzione.

Φ_N

1. zero
2. succ
3. add
4. mult
5. exp

Π_N

1. $<$
2. $=$

r_N

1. $r_N(zero) = 0$ poichè ha arità 0
2. $r_N(succ) = 1$ poichè ha arità 1
3. $r_N(add) = 2$ poichè ha arità 2
4. $r_N(mult) = 2$ poichè ha arità 2
5. $r_N(exp) = 2$ poichè ha arità 2

Interpretazione
L'interpretazione è una coppia del tipo $I = (U, \mu)$, con U che è un insieme non vuoto, è l'universo dell'interpretazione I che contiene gli elementi che lo compongono, e μ è la funzione che mappa gli elementi di $Y \cup \Phi \cup \Pi$ agli elmenti di U

Praticamente, avremo un mapping tra i simboli di funzione e i simboli di relazione e gli elementi di U.

- $\mu(zero) = 0$
- $\mu(succ) = g : N \to N \mid g(x) = x + 1$
- $\mu(add) = g : N^2 \to N \mid g(x, y) = x + y$
- $\mu(mult) = g : N^2 \to N \mid g(x, y) = x * y$
- ...
- $\mu(\asymp) = \{(x, x) : x \in N\}$
- $\mu(<) = \{(x, y) : x \in N, y \in N, x < y\}$

1.14.6 Domanda 6

Definire il concetto di assioma non logico in FOE
Risposta

Un assioma non logico è un enunciato che consideriamo vero a priori nel mondo reale anche se non è valido . Cioè, sono quelle formule che non sono per forza valide ma che sono soddisfatte da una qualche interpretazione. Quindi sono quelle formule che sono considerate vere sotto una qualche interpretazione.

1.14.7 Domanda 7

Mostrare che $N \models\!\prec (succ(zero), add(succ(zero), succ(zero)))$ **Risposta**

Per mostrare che $N \models\!\prec (succ(zero), add(succ(zero), succ(zero)))$ dobbiamo mostrare che per ogni interpretazione $I = (U, \mu)$, $I \models\!\prec (succ(zero), add(succ(zero), succ(zero)))$.
Utilizzando l'interpretazione che abbiamo nella teori dei numeri, abbiamo che:

1. $N \models \phi if(\mu(succ(zero)), add(\mu(succ(zero)), \mu(succ(zero)))) \in \mu(\prec)$
2. $N \models \phi if(\mu(succ(0)), \mu(add(succ(0)), \mu(succ(0)))) \in \mu(\prec)$
3. $N \models \phi if(0+1, (0+1+0+1)) \in \mu(\prec)$
4. $N \models \phi if(1, (1+1)) \in \mu(\prec)$
5. $N \models \phi if(1,2) \in \mu(\prec)$

Abbiamo quindi provato $N \models \prec (succ(zero), add(succ(zero), succ(zero)))$.

1.14.8 Domanda 8

Mostrare che $N \not\models \asymp (succ(succ(zero)), mult(succ(zero), succ(zero)))$ **Risposta**

1. $N \not\models \asymp (succ(succ(zero)), mult(succ(zero), succ(zero)))$
2. $N \models \phi if(\mu(succ(succ(zero))), \mu(mult(succ(zero), succ(zero)))) \in \asymp$
3. $N \models \phi if(\mu(succ(succ(zero))), \mu(succ(zero) * succ(zero))) \in \asymp$
4. $N \models \phi if(\mu(succ(zero)+1), \mu(zero+1*zero+1)) \in \asymp$
5. $N \models \phi if(0+1+1, 1*1) \in \asymp$
6. $N \models \phi if(2,1) \in \asymp$

Vediamo che questa formula ϕ non è soddisfatta, poiché $2 \neq 1$.

1.14.9 Domanda 9

Mostrare che $N \models \asymp (succ(zero)), mult(succ(zero), succ(zero))$ **Risposta**

1. $N \models \phi if(\mu(succ(zero)), \mu(mult(succ(zero), succ(zero)))) \in \asymp$
2. $N \models \phi if(\mu(succ(zero)), \mu(succ(zero) * succ(zero))) \in \asymp$
3. $N \models \phi if(zero+1, zero+1*zero+1) \in \asymp$
4. $N \models \phi if(0+1, 0+1*0+1) \in \asymp$
5. $N \models \phi if(1, 1*1) \in \asymp$
6. $N \models \phi if(1,1) \in \asymp$

Abbiamo quindi provato che $N \models \asymp (succ(zero)), mult(succ(zero), succ(zero))$, poichè $1 = 1$.

1.15 Indecidibilita' in First Order Logic

1.15.1 Domanda 1

Definire il concetto di assioma logico
Risposta
Chiamiamo assiomi logici l'insieme di (Λ) FOE (First Order Expressions) che sono valide per i 3 seguenti motivi

1. Proprieta' dei quantificatori: $\phi(y1, \ldots, yn, t) \implies \exists y_{n+1} tc : \phi(y, \ldots, y_{n+1})$
   Questa vale poiche' per rendere t vera, deve esisere y che soddisfa la formula.
   Spiegazione di Simone: è una formula in questa forma e sai sicuro che questa formula è valida.
2. Proprieta' dell'uguaglianza: $\phi(t1, \ldots, tn) \asymp \phi(t1, \ldots, tn)$ e $y \asymp y$ sono entrambe valide
   La seconda vale poiche' per qualsiasi valutazione dei termini e' sempre valida
3. Validita' booleana: $(\forall y1 \ldots \forall yn\phi(y1, \ldots, yn)) \vee \neg(\forall y1 \ldots \forall yn\phi(y1, \ldots, yn)) e' valida$
   La terza vale poiche' e' come se stessimo dicendo $x \vee \neg x$ che e' sempre vera

Praticamente, sono espressioni che possiamo creare partendo dal nostro vocabolario, e sono valide perche' la validita viene rispettata dalle 3 regole.

1.15.2 Domanda 2

Definire FOL-PROVABLE Risposta

Per definire FOL provable, dobbiamo definire il concetto di prova.

Dato un insieme A di assiomi non logici, un insieme Λ di assiomi logici e una FOE ϕ, noi consideriamo FOL-PROVABLE come l'insieme delle FOE tale che esse sono provabili, ovvero che $A \vdash \phi$.

Per definire questo, pero', dobbiamo prima dire che cosa significa che una formula e' dimostrabile. Utilizziamo un proof system. Diciamo che una **prova** per ϕ dal'insieme A e' una sequenza di FOE $\phi_1, \ldots, \phi_n$ tale che:

- $\phi_n = \phi$
- Per ogni $i \in \{1, \ldots, n\}$:
  - $\phi_i \in \Lambda \cup A$ oppure
  - $\exists j < i \ t.c : (\phi_j \implies \phi_i) \in \{\phi_1, \ldots, \phi_n\}$

Quindi, una formula e' dimostrabile se esiste una prova per essa e una prova e' stata definita sopra.

$$FOL - PROVABLE = \{< A, \phi >: A \text{ e' un insieme di assiomi non logici }, \phi \text{ e' una FOE e} A \vdash \phi\} \quad (1.21)$$

1.15.3 Domanda 3

Enuncia e parla del teorema della soundess
Risposta

Con **Soundess** intendiamo che tutto cio' che viene generato a partire dagli assiomi sono delle espressioni valide.

Il teorema della soundess dice che se $A \vdash \phi$ allora $A \models \phi$. Questo e' vero perche' se $A \vdash \phi$ allora esiste una prova per ϕ a partire da A e quindi tutte le formule che sono state generate sono valide. Quindi, se $A \vdash \phi$ allora $A \models \phi$. Con questo possiamo dire che FOL-PROVABLE $\subseteq$ FOL-VALID, perche' se una formula e' provabile, allora e' valida.

1.15.4 Domanda 4

Enuncia e parla del teorema della completezza di Godel
Risposta

Con **Completezza** intendiamo che tutte le espressioni entailed da A sono generate anche dal proof system.

Il teorema della completezza di Godel dice che se $A \models \phi$ allora $A \vdash \phi$. Questo e' vero perche' se $A \models \phi$ allora ϕ e' valida e quindi $A \vdash \phi$. Cioe' dimostra che il nostro proof system e' completo, perche' se una qualsiasi formula e' ENTAILED da A, allora puo' essere derivata partendo dal nostro proof sytems. Quindi, FOL-ENTAIL $\subseteq$ FOL-PROVABLE.

1.15.5 Domanda 5

Spiegare perche' FOL-ENTAIL e FOL-VALID sono in SD Risposta

Ricordiamoci cosa significano FOL-ENTAIL e FOL-VALID.

- FOL-ENTAIL = $\{< A, \phi >$: A e' un insieme di assiomi non logici , ϕ e' una FOE e$A \models \phi\}$
- FOL-VALID = $\{< \phi >$: ϕ e' una FOE e $\models \phi\}$

Ora, consideriamo che $\Lambda \in D$ perche ammette un'enumerazione lessicografica, cioe' possiamo enumerare tutte le possibili stringhe sull'alfabeto di FOL. Per ogni stringa possiamo vedere se e' una FOE e se soddisfa le 3 condizioni.

Ricordiamo che da Godel abbiamo dimostrato che FOL-ENTAIL = FOL-PROVABLE, quindi quando $A \models \phi$ c'e' una proof per ϕ a partire da A che puo' essere trovata in tempo finito. Enumerando Λ e applicando il modus ponens si possono derivare tutte le possibili conseguenze. Se $A \vdash \phi$ allora questa procedura ad un certo punto trovera' una prova per ϕ

Ora, per provare che anche FOR-VALID $\in SD$, ci basti pensare che FOL-VALID e' un caso particolare di FOR-ENTAIL. Infatti, se li mettiamo a confronto:

- FOL-ENTAIL = $\{< A, \phi >$: A e' un insieme di assiomi non logici , ϕ e' una FOE e$A \models \phi\}$
- FOL-VALID = $\{< \phi >$: ϕ e' una FOE e $\models \phi\}$

Notiamo che FOL-VALID e' un caso particolare di FOL-ENTAIL prendendo come insieme di assiomi non logici A l'insieme vuoto $\emptyset$. Quindi, FOL-VALID $\subseteq$ FOL-ENTAIL. Per questo possiamo utilizzare la stessa procedura anche per FOL-VALID, e quindi esiste una mapping reduction $FOL-VALID \leq_M FOL-ENTAIL$

1.15.6 Domanda 6

Spiegare perche' FOL-ENTAIL e FOL-VALID NON sono in D Risposta

Per spiegarlo c'e' una dimostrazione di TURING. Seguiamo i seguenti passaggi:

1. Sia P_0 l'insieme delle macchine che stampano sempre il simbolo 0
2. Dimostriamo che $H \leq_M P_0$ in modo semplice dicendo che se M_H termina, allora M' cancella l'output e stampa 0, altrimenti se non termina andiamo in loop. Abbiamo dimostrato che P_0 non e' in D
3. Dimostriamo che $P_0 \leq_M FOL - VALID$: Data una TM M che stampa sempre 0, costruiamo una formula ϕ_M tale che ϕ_M e' valida se e solo se M stampa sempre 0. Quindi, abbiamo dimostrato che FOL_VALID non e' in D
4. Dimostriamo che $FOL-VALID \leq_M FOL-ENTAIL$: Data una formula ϕ costruiamo una formula ϕ' tale che ϕ' e' valida se e solo se ϕ e' valida. Questo lo facciamo prendendo come A l'insieme vuoto $\emptyset$. Quindi, abbiamo dimostrato che $FOL - ENTAIL$ non e' in D
5. A catena: $H \leq_M P_0 \leq_M FOL - VALID \leq_M FOL - ENTAIL$

Siccome sappiamo che FOL-VALID $\subseteq$ FOL-ENTAIL, allora possiamo dire che $FOL - VALID \leq_M FOL - ENTAIL$. Quindi, se $FOL - ENTAIL \notin D$ allora anche $FOL - VALID \notin D$.

1.15.7 Domanda 7

Spiegare perche' FOL-UNSAT $\in SD \setminus D$
Risposta

Per dimostrare questo, partiamo dal concetto che per ogni FOE ϕ vale che essa e' valida $\iff$ $\neg\phi$ e' insoddisfacibile. Possiamo dimostrare le seguenti riduzioni:

- $FOL-UNSAT \leq_M FOL-VALID$: Siccome sappiamo che FOL-VALID e' in SD, anche FOL-UNSAT sara' in SD. Per dimostrare la reduction, crediamo una ρ tale che essa neghi l'output prodotto dalla macchina che risolve FOL-UNSAT. In questo modo avremo una formula valida.
- $FOL - VALID \leq_M FOL - UNSAT$: Siccome sappiamo che FOL-VALID non e' in D, anche FOL-UNSAT sara' non in D. Per dimostrare la reduction, crediamo una ρ tale che essa neghi l'output prodotto dalla macchina che risolve FOL-VALID. In questo modo avremo una formula insoddisfacibile.

1.15.8 Domanda 8

Spiegare perche' $FOL - SAT \in coSD \setminus D$
Risposta

Noi sappiamo che FOL-UNSAT $= \neg FOL - SAT$. Quindi, se FOL-UNSAT e' in SD, allora FOL-SAT e' in coSD. Inoltre, sappiamo che FOL-UNSAT e' in SD \ D, quindi FOL-SAT e' in coSD \ D.

1.16 Indecibilita' e incompletezza nella teoria dei numeri

1.16.1 Domanda 1

Qual e' la correlazione tra $FOL-ENTAIL_{NT}$ *e* $FOL-TRUE-PROPERTIES_N$
Risposta

Iniziamo definendo i due insiemi:

- $FOL-ENTAIL_{NT} = \{<\phi> | \phi\ FOE, NT \models \phi\}$ cioe' tutte le FOL che possono essere provate con gli assiomi di Peano
- $FOL-TRUE-PROPERTIES_N = \{<\phi> | \phi\ FOE, NT \vdash \phi\}$ con N che rappresenta gli ingredienti principali della Teoria dei Numeri

Ora, dalla Teoria dei Numeri noi consideriamo che gli assiomi di peano siano veri. Questo implica che N soddisfa tutti gli assimoi in NT. Per questo motivo, per definizione di entailment, N soddisfa anche ϕ. Quindi, $FOL-ENTAIL_{NT} \subseteq FOL-TRUE-PROPERTIES_N$.
Per questo motivo, quando $N \models \phi$, allora $NT \models \phi$

1.16.2 Domanda 2

Qual e' la definizione di separabilita' ricorsiva?
Risposta

In generale, i linguaggi che sono definiti DECIDABLE, vengono anche chiamati RICORSIVI, mentre dei linguaggi SEMI-DECIDIBILI vengono chiamati RICORSIVAMENTE ENUMERABILI.

Per definizione, diciamo che due linguaggi sono definiti *ricorsivamente separabili* se dati due linguaggi $L1$ e $L2$ tali che la loro intersezione $= \emptyset$, esiste un altro linguaggio R *ricorsivo* tale che:

1. $L1 \cap R = \emptyset$
2. $L2 \subseteq R$

Per capire, immaginiamo di avere una TM M_R che riconosce il linguaggio R. Se la macchina riconosce una determinata stringa, allora sappiamo che appartiene a L2. Se la stringa non viene riconosciuta, allora sappiamo che appartiene a L1. Questo perche' abbiamo definito R come sopra.

1.16.3 Domanda 3

Qual e' la definizione di inseparabilita' ricorsiva?
Risposta

Il teorema dice che dati due linguaggi disgiunti $L1$ e $L2$, se $L1$ e $L2$ sono entrambi ricorsivamente inseparabili, allora essi sono entrambi *indecidibili*.

1.16.4 Domanda 4

Spiega perche $Self_y$ e $Self_n$ sono ricorsivamente inseparabili
Risposta

Definiamo inizialmente $Self_y$ e $Self_n$:

- $Self_y = \{< M > | M\ e'una\ TML : M(< M >) = y\}$
- $Self_n = \{< M > | M\ e'una\ TML : M(< M >) = n\}$

Ora, assumiamo che esista un linguaggio $R \subset \Sigma^*$ tale che renda i due linguaggi ricorsivamente separabili. Immaginiamo di separare i due insiemi in questo modo:

- $Self_y \cap R = \emptyset$
- $Self_n \subseteq R$

1. Caso 1: Se $< M_R > \in R$ allora vuol dire che la macchina R si fermerebbe su $< M_R >$ e quindi $< M_R > \in R$ e anche $< M_R > \in Self_y$. Questo pero' e' un assurdo perche' abbiamo detto che $Self_y \cap R = \emptyset$
2. Caso 2: Se $< M_R > \notin R$ allora vuol dire che la macchina R non si fermerebbe su $< M_R >$ e quindi $< M_R > \notin R$ e anche $< M_R > \in Self_n$. Questo pero' e' un assurdo perche' abbiamo detto che $Self_n \subseteq R$ e non sarebbe possibile.

Abbiamo provato che i due linguaggi sono ricorsivamente inseparabili. Quindi, per il teorema di inseparabilita' ricorsiva, i due linguaggi sono indecidibili.

1.16.5 Domanda 5

Mostra che $FOL - TRUE - PROPERTIES_N$ e' fuori da SD e $co - SD$
Risposta

Per spiegare questo, bisogna spiegare il teorema dell'incompletezza di Godel, che ci dice che gli assiomi di Peano non ci permettono di provare tutte le verita' aritmetiche.
Quindi, per il teorema di Godel, esistono delle formule ϕ che sono vere in N ma che non possono essere provate con gli assiomi di Peano. Questo pero' vale anche se avessimo un superset A di NT, tale che esso sia costruito ricorsivamente.
Quindi, se esiste un superset A di NT, tale che esso sia costruito ricorsivamente, allora esistono delle formule ϕ che sono vere in N ma che non possono essere provate con gli assiomi di A.

1.16.6 Domanda 6

Discuti il teorema dell'incompletezza di Godel
Risposta

Per il suo teorema, per ogni insieme semi-decidibile di assiomi non logici A su Vn, cioe' il vocabolario della teoria dei numeri, c'e' almeno una FOE su Vn tale che: $N \models \phi \wedge A \nvdash \phi$
Per provarlo assumiamo di avere un superset di assiomi non logici di NT, chiamiamolo A.

1. Assumiamo per contraddizione che con questo A allora $\forall \phi, N \models \phi \iff A \vdash \phi$.

$$FOL - ENTAIL_A = FOL - TRUE - PROPERTIES_N \tag{1.22}$$

2. Siccome $FOL - ENTAIL_A \in SD$ allora esiste una procedura per enumerarlo, ma siccome stiamo dicendo nel punto (1) che sono uguali allora avremmo anche una procedura per enumerare $FOR - TRUE - PROPERTIES_N$

3. Per definizione, $\neg FOL - TRUE - PROPERTIES_N = \{< \phi >: N \models \neg\phi\}$, allora abbiamo anche una procedura P' per enumerarlo. Infatti per farlo basta fare in modo che quando la procedura P stampa per FOL-TRUE-PROPERTIES$_N$ ϕ, allora deve stampare $\neg\phi$

4. CONTRADDIZIONE: Sia FOL-TRUE-PROPERTIES$_N$ che il suo complemento sarebbero in SD, ma questo non e' possibile perche' se entrambi fossero in SD, allora vorrebbe dire che FOL-TRUE-PROPERTIES$_N$ sarebbe in D, ma questo non e' possibile perche' sappiamo che $FOL - TRUE - PROPERTIES_N \notin D$.

Per questo motivo, ne FOL-TRUE-PROPERTIES$_N$ che il suo complemento non possono essere in SD. Infatti, sono fuori sia da SD che co-SD.

Chapter 2

Complessità computazionale

2.1 Introduzione

In questo capitolo si parlerà di complessità computazionale. In particolare, si parlerà di come si possa dimostrare che un problema è NP-completo o meno.

2.2 Misurare la complessità computazionale

2.2.1 Domanda 1

Data una funzione f, definire $\mathcal{O}(g)$, Ω(g) e Θ(g).
Risposta

Data una funzione f, definiamo $\mathcal{O}(g)$, $\Omega(g)$ e $\Theta(g)$ come segue:
$\mathcal{O}(g)$:
$\mathcal{O}(g)$ è l'insieme di tutte le funzioni f che crescono al più come g (in modo asintotico). Formalmente, $\mathcal{O}(g)$ è definito come segue:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{2.1}$$

Cioè stiamo dicendo che f ha una crescita al massimo come g.

$\Omega(g)$:
$\Omega(g)$ è l'insieme di tutte le funzioni f che crescono almeno come g (in modo asintotico). Formalmente, $\Omega(g)$ è definito come segue:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{2.2}$$

$\Theta(g)$:
$\Theta(g)$ è l'insieme di tutte le funzioni f che crescono esattamente come g (in modo asintotico). Formalmente, $\Theta(g)$ è definito come segue:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \tag{2.3}$$

con $c > 0$.

2.2.2 Domanda 2

Dire se $2n^2 + 4n + 1 \in \Omega(n^3 + 1)$ regge oppure no
Risposta

Quando parliamo di Ω stiamo dicendo che la funzione f cresce almeno come g. Allora in questo caso dobbiamo controllare che il limite:

$$\lim_{n\to\infty} \frac{2n^2 + 4n + 1}{n^3 + 1} > 0 \tag{2.4}$$

Guardando banalmente l'equazione, notiamo che il denomiatore cresce più velocemente rispetto al numeratore, quindi il limite tende a 0. Quindi la funzione f non cresce almeno come g e quindi $2n^2 + 4n + 1 \notin \Omega(n^3 + 1)$.

2.2.3 Domanda 3

Data una TM M, definire $timereq_M(n)$
Risposta

Quando parliamo di tempo richiesto da una TM per eseguire un certo input, stiamo parlando del numero di passi che la TM deve fare per eseguire l'input. Quindi il tempo richiesto da una TM M per eseguire un input w è definito come segue:

$$timereq_M(n) = \forall n \in N,\ max_{w\in\Sigma^n} steps(M, w) \tag{2.5}$$

Stiamo letteralmente definendo il numero massimo di steps che la macchina M esegue sulle stringhe di lunghezza n, per qualsiasi n che appartiene ad N.

2.2.4 Domanda 4

Data una TM M, definire $spaceeq_M(n)$
Risposta

Quando parliamo di spazio richiesto per eseguire un certo input, stiamo parlando del numero di celle che la TM deve usare per eseguire l'input. Dobbiamo quindi innanzitutto definire come calcolare quante celle usa una TM M. Per farlo, pensiamo alla chiusura transitiva riflessiva partendo da una configurazione iniziale ad una finale. Possiamo definire la funzione *squares* defintia come $|u| + |v| + 1$ che non è altro che il numero di celle toccate fino alla configurazione finale. Quindi lo spazio richiesto da una TM M per eseguire un input w è definito come segue:

$$spacereq_M(n) = \forall n \in N,\ max_{w\in\Sigma^n} squares(M, w) \tag{2.6}$$

Stiamo letteralmente definendo il numero massimo di celle che la macchina M usa sulle stringhe di lunghezza n, per qualsiasi n che appartiene ad N.

2.2.5 Domanda 5

Discutere della complessità temporale di un problema P
Risposta

Prendiamo un problema copmutabile P (un linguaggio decidibili o una funzione computabile) e una funzione $f : N \to N$.
Ora possiamo analizzare la complessità temporale di P, parlando di $\mathcal{O}(f(n))$ e $\Omega(f(n))$ e $\Theta(f(n))$.

$\mathcal{O}(f(n))$:
Diciamo che un problema P ha come upper bound $mathcalO(f(n))$ se esiste una TM M che risolve P tale che $timereq_M(n) \in \Theta(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con un tempo richiesto che cresce al più come $f(n)$.

$\Omega(f(n))$:
Diciamo che un problema P ha come lower bound $Omega(f(n))$ se esiste una **proof** che mostra che per ogni TM M che risolve P, $timereq_M(n) \in \Omega(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con un tempo richiesto che cresce almeno come $f(n)$. Stiamo praticamente dicendo quanto può essere veloce l'algoritmo. Non possiamo fare un algoritmo più veloce.
$\Theta(f(n))$:
Diciamo che un problema P ha come tight bound $\Theta(f(n))$ se esiste una TM M che risolve P tale che $timereq_M(n) \in \Theta(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con un tempo richiesto che cresce esattamente come $f(n)$. Ha cioè:

- Un lower bound temporale $\Omega(f(n))$
- Un upper bound temporale $\mathcal{O}(f(n))$

2.2.6 Domanda 6

Discutere della complessità spaziale di un problema P
Risposta

Prendiamo un problema copmutabile P (un linguaggio decidibili o una funzione computabile) e una funzione $f : N \to N$.
Ora possiamo analizzare la complessità spaziale di P, parlando di $\mathcal{O}(f(n))$ e $\Omega(f(n))$ e $\Theta(f(n))$.

$\mathcal{O}(f(n))$:
Diciamo che un problema P ha come upper bound $\mathcal{O}(f(n))$ se esiste una TM M che risolve P tale che $spacereq_M(n) \in \Theta(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con uno spazio richiesto che cresce al più come $f(n)$.

$\Omega(f(n))$:
Diciamo che un problema P ha come lower bound $Omega(f(n))$ se esiste una **proof** che mostra che per ogni TM M che risolve P, $spacereq_M(n) \in \Theta(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con uno spazio richiesto che cresce almeno come $f(n)$. Stiamo praticamente dicendo quanto può essere veloce l'algoritmo. Non possiamo fare un algoritmo più veloce.

$\Theta(f(n))$:
Diciamo che un problema P ha come tight bound $\Theta(f(n))$ se esiste una TM M che risolve P tale che $spacereq_M(n) \in \Theta(f(n))$.
Stiamo praticamente dicendo che il problema P può essere risolto con una macchina M con uno spazio richiesto che cresce esattamente come $f(n)$. Ha cioè:

- Un lower bound spazile $\Omega(f(n))$
- Un upper bound spaziale $\mathcal{O}(f(n))$

2.2.7 Domanda 7

Discutere di problemi aperti e problemi chiusi
Risposta

Quando si discute di problemi si possono distinguere due tipologie di problemi:

- Problemi APERTI
- Problemi CHIUSI

Problemi APERTI:
I problemi sono considerati APERTI se non si riesce a definire un algoritmo tale che funzioni sia da **lower bound** che da **upper bound**.
Questo significa che esiste ancora margine di miglioramento. Possiamo essere più formali e dire:
Un problema viene considerato aperto se la funzione di complessità che determina il lower bound **NON COINCIDE** con la funzione di complessità che determina l'upper bound.

Problemi CHIUSI:
I problemi sono considerati CHIUSI se si riesce a definire un algoritmo tale che funzioni sia da **lower bound** che da **upper bound**.
Questo significa che non esiste margine di miglioramento. Possiamo essere più formali e dire:
Un problema viene considerato chiuso se la funzione di complessità che determina il lower bound **COINCIDE** con la funzione di complessità che determina l'upper bound.

2.3 Classi di complessità deterministiche

2.3.1 Domanda 1

Spiega perché f(n) = 2n è una proper complexity function
Risposta
Consideriamo una proper complexity function una funzione che:

- f non decresce
- C'è una macchina di turing multitape k tale che:
  - Nel primo nastro viene contenuto l'input e non cambia
  - Nell'ultimo k tape viene contenuto l'output ed è read only (append only)
  - timereq(n) $\in$ O(n+f(n))
  - spacereq(n) $\in$ O(f(n))
  - $[M(w)] = \square^{f(|w|)}$

Ora, per dire che f(n) = 2n è una proper complexity function dobbiamo dimostrare che soddisfa i punti sopra elencati:

- f non decresce: f(n) = 2n $\geq$ n $\forall$ n $\in$ N
- C'è una macchina di turing multitape k tale che:
  - Nel primo nastro viene contenuto l'input e non cambia
  - Nell'ultimo k tape viene contenuto l'output ed è read only (append only)
  - timereq(n) $\in$ O(n+f(n)) = O(n+2n) = O(3n) = O(n)
  - spacereq(n) $\in$ O(f(n)) = O(2n) = O(n)
  - $[M(w)] = \square^{f(|w|)} = \square^{2|w|}$

2.3.2 Domanda 2

Spiega perché f(n) = n^2 è una proper complexity function
Risposta

Come per la precedente, dobbiamo controllare i punti precedenti:

- f non decresce: f(n) = n^2 $\geq$ n $\forall$ n $\in$ N
- C'è una macchina di turing multitape k tale che:
  - timereq(n) $\in$ O(n+f(n)) = O(n+n^2) = O(n^2)
  - spacereq(n) $\in$ O(f(n)) = O(n^2)
  - $[M(w)] = \square^{f(|w|)} = \square^{n^2}$

2.3.3 Domanda 3

Spiega perché f(n) = 7 è una proper complexity function
Risposta

Come per la precedente, dobbiamo controllare i punti precedenti:

- f non decresce: f(n) = 7 $\forall$ n $\in$ N

- C'è una macchina di turing multitape k tale che:
  - timereq(n) $\in$ O(n+f(n)) = O(n+7) = O(n)
  - spacereq(n) $\in$ O(f(n)) = O(7) = O(1)
  - $[M(w)] = \square^{f(|w|)} = \square^7$

2.3.4 Domanda 4

Definire PTIME EXPTIME, PSPACE, EXPSPACE, ..., ELEMENTARY
Risposta

Dobbiamo innanzitutto definire DTIME e DSPACE.

Con **DTIME(f(n))** definiamo l'insieme dei linguaggi tali per cui esiste una Turing Machine M tale che $\forall n \in N$, il tempo di esecuzione è per decidere stringhe di lunghezza n è al massimo f(n). Un esempio è DTIME(n^2) che per stringhe di lunghezza n il tempo di esecuzione massimo è quadratico, al massmo.

Con **DSPACE(f(n))** definiamo l'insieme dei linguaggi tali per cui esiste una Turing Machine M tale che $\forall n \in N$, lo spazio di memoria occupato per decidere stringhe di lunghezza n è al massimo f(n). Un esempio è DSPACE(n^2) che per stringhe di lunghezza n lo spazio di memoria occupato è quadratico, al massmo.

Ora, parlando di PTIME, stiamo parlando dell'insieme dei linguaggi tali per cui esiste una macchina di turing M, tale che per cui per qualsiasi stringa di lunghezza n, il tempo di esecuzione massimo è polinomiale. Cioè n^k dove k è una costante.

$$PTIME = P = DTIME(n^k) = \bigcup_{i>0 \wedge f \in \mathcal{O}(n^i)} DTIME(f(n)) \tag{2.7}$$

Qui dentro ci sono tutti i linguaggi che hanno come tempo di esecuzione massimo un tempo polinomiale:

- n^2
- n^3
- n^4
- ...

Parlando di EXPTIME, stiamo parlando invece dell'insieme dei linguaggi tali per cui esiste una TM che su stringhe di lunghezza n, ha come tempo di esecuzione massimo un tempo esponenziale.

$$EXPTIME = EXP = DTIME(2^{n^k}) \bigcup_{i>0 \wedge f \in 2^{\mathcal{O}(n^i)}} DTIME(f(n)) \tag{2.8}$$

Qui dentro ci sono tutti i linguaggi che hanno come tempo di esecuzione massimo un tempo esponenziale: 2^{n^k}

Parlando di 2EXPTIME, 3EXPTIME, I-exptime, finiamo per parlare di ELEMENTARY.

Quando parliamo di elementary stiamo praticamente parlando di classi di complessità superiori a EXPTIME, per cui vale $2^{2^{n^k}}$.

Questo è un esempio di una classe di complessità che è superiore a EXPTIME, ma non è la più grande. La più exp è ELEMENTARY, che viene definita come:

$$ELEMENTARY = \bigcup_{i>0} i - EXPTIME \tag{2.9}$$

Qui dentro sono definite tutte le classi di EXPTIME come 2EXPTIME, 3EXPTIME, 4EXPTIME, ...

Per PSPACE vale lo stesso ragionamento, solo che non si parla di tempo di esecuzione ma di numero di celle toccate e di spazio utilizzato.

PSPACE ad esempio parla dell'insieme dei linguaggi tali per cui utilizzano al massimo uno spazio polinomiale:

$$PSPACE = DSPACE(n^k) = \bigcup_{i>0 \wedge f \in \mathcal{O}(f(n^i))} DSPACE(f(n)) \tag{2.10}$$

In questo insieme avremo i linguaggi che utilizzadno al massimo spazio pari a:

- n^2
- n^3
- n^4
- ...

2.3.5 Domanda 5

Parlare delle relazioni tra le principali classi di complessità
Risposta

Le principali classi di complessità sono PTIME, PSPACE, EXPTIME, EXPSPACE.

La relazione che lega le principali classi e' la seguente:

$$PTIME \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE \tag{2.11}$$

Ora, per spiegare la motivazione di questa relazione, dobbiamo capire il significato di questo sott'insieme.

Ad esempio, sappiamo per certo che se un'algoritmo utilizza un tempo polinomiale, siamo certi che utilizza uno spazio polinomiale. Questo perché se un algoritmo utilizza un tempo polinomiale, vuol dire che il numero di passi che fa è polinomiale, e quindi il numero di celle che tocca è polinomiale. Questo basta per dimostrare che $PTIME \subseteq PSPACE$.

Per dimostrare invece che $PSPACE \subseteq EXPTIME$ dobbiamo pensare al concetto che se un algoritmo utilizza uno spazio polinomiale, il tempo potrebbe essere esponenziale.

Fissandoci in mente che lo spazio che utilizziamo è polinomiale, nel caso di una macchina di turing deterministica, il numero massimo di configurazioni che possiamo avere e' $\Sigma^{|w|}$, parlando di alfabeto binario avremo $2^{|w|}$.

Quindi siccome il numero di configurazioni è esponenziale, il tempo di esecuzione è esponenziale poiché il tempo che ci mettiamo per esplorarle tutte è esponenziale.

2.3.6 Domanda 6

Dimostrare che PTIME e' chiuso sotto l'unione
Risposta

Per dimostraer che PTIME e' chiuso sotto l'unione ci serve una procedura che ci mostra che utilizzando un tempo polinomiale, possiamo unire due linguaggi e ottenere un linguaggio che utilizza un tempo polinomiale. Prendiamo due linguaggi decidibili L1 e L2 appartengano entrambi a PTIME, con le rispettive TM M1 e M2, tali che $L(M1) = L1$ e $L(M2) = L2$. Ci serve una procedura che ci aiuta a dimostrare che e' possibile decidere L1 e L2 in un tempo polinomiale.

```
decideUnioneL1L2(w:str , M1:TM-Encoding , M2:TM-encoding){
    if (utm(M1,w) == y || utm(M2,w) == y){
        return y
    }
    else{
        return false
    }
}
```

Immaginiamo che le due macchine riescano a decidere in un tempo polinomiale, allora la procedura che abbiamo scritto sopra, ci permette di decidere l'unione dei due linguaggi anch'essa in un tempo polinomiale. Anche se il tempo di esecuzione di una delle due macchine fosse maggiore rispetto all'altra, siccome noi le valutiamo asintoticamente, avremo sempre un tempo polinomiale. Lo scaling costante che avremo sarà di 2, poiché chiamiamo la procedura *utm* 2 volte.

2.3.7 Domanda 7

Dimostrare che PSPACE è chiuso sotto l'intersezione
Risposta

Prendiamo due linguaggi L1 e L2, tali che essi siano entrambi $\in PSPACE$. Date due TM M1, e M2 con $L(M1) = L1$ e $L(M2) = L2$, possiamo dimostrare che l'intersezione dei due linguaggi è anche essa in PSPACE.

Per farlo, scriviamo una procedura che ci permette di dimostrare che se entrambi i linguaggi sono in PSPACE, allora la loro intersezione sarà anche in PSPACE.

```
decideIntersection(M1:TM-encoding , M2:TM-encoding , w:str){
    if (utm(M1,w) == y && utm(M2,w) == y){
        return y
    }
    else{
        return false
    }
}
```

La spiegazione è semplice. Se la macchina M1 utilizza uno spazio polinomiale per risolvere l'algoritmo e la macchina M2 utilizza uno spazio polinomiale per risolvere l'algoritmo, allora la procedura che abbiamo scritto sopra, ci permette di dimostrare che l'intersezione dei due linguaggi utilizza anche uno spazio polinomiale.

Questo perché la procedura *utm* viene chiamata due volte, e questo comprende uno scaling di 2 sullo spazio totale utilizzato. Se lo spazio utilizzato da M1 fosse O(n) e quello utilizzato da M2 fosse O(2n), allora spazio totale sarebbe di O(n)+(2n) = O(3n) = O(n), quindi uno spazio polinomiale.

2.4 Problemi in tempo polinomiale

In questa sezione non c'erano domande, quindi ne faccio una io.

2.4.1 Domanda 1

Definire perché L è chiuso sotto l'operatore +
Risposta
Per definire questa procedura, dobbiamo sfruttra REACH.

Il nostro problema è dire se una stringa w appartiene a L^+.

Ora, per farlo, dobbiamo costruire un grafo basandoci sulla stringa. Il nostro grafo lo definiamo utilizzando w come base. Quindi, avremo:

- N: i nodi
- A: gli archi

Consideriamo n che è la lunghezza della stringa.

I nodi che appartengono ad N sono del tipo $u_{i,j,s}$, dove i è minore uguale a j e j è minore uguale a n. s è la stringa che è w[i..j]. Oltre a questi nodi ci sono due nodi u_{start}, u_{end} Ora, per ogni nodo:

- u_{start} è collegato al nodo iniziale $u_{0,j,s}$
- u_{end} è collegato al nodo che $u_{j,n,s}$
- $u_{i,j,s}$ è collegato al nodo $u_{j,k,s'}$

Se ora abbiamo questo grafo, allora diciamoc che $w \in L^+$ se e solo se $< G_w, u_{start}, u_{end} > \in REACH$.

2.5 Problemi completi in tempo polinomiale

2.5.1 Domanda 1

Definire il problema F-REACH
Risposta
Il problema F-REACH e' un problema decisionale che ci chiede, dato un grafo orientato $G = (N, A)$, con N rappresentante i nodi del grafo, A gli archi, e dati due nodi $u, v \in N$, $RITORNARE$ se esiste un simple path, ovvero una sequenza di nodi senza ripetizione.
Quindi:

- INPUT: Un grafo orientato $G = (N, A)$, due nodi $u, v \in N$
- OUTPUT: *Simple Path* da u a v, se esiste

2.5.2 Domanda 2

Spiegare la differenza tra REACH e F-REACH
Risposta

La sostanziale differenza è che REACH viene definito come un problema che, dato un grafo orientto G e due nodi u,v, ci chiede se esiste un path da u a v, mentre F-REACH ci chiede di ritornare il path stesso.

- REACH:
  - INPUT: Un grafo orientato $G = (N, A)$, due nodi $u, v \ \in N$
  - OUTPUT: $True$ se esiste un path da u a v, $False$ altrimenti
- F-REACH:
  - INPUT: Un grafo orientato $G = (N, A)$, due nodi $u, v \ \in N$
  - OUTPUT: $Simple\ Path$ da u a v, se esiste

Quindi uno richiede effettivamente la soluzione, l'altro invece ritornare solamente se la soluzione esiste o meno.

2.5.3 Domanda 3

Cosa sono le classi di complessità delle funzioni?
Risposta
Le classi di complessità delle funzioni non sono altro che un insieme di funzioni che risolvono un problema in un tempo comune. Ad esempio, l'insieme **FP-TIME** è l'insieme delle funzioni che possono essere risolte in tempo polinomiale.

2.5.4 Domanda 4

Definire la classe di complessità FP-TIME
Risposta

L'ho letteralmente appena fatto nella domanda precedente, ma lo ripeto volentieri.
FP-TIME è l'insieme delle funzioni che possono essere risolte in tempo polinomiale.

2.5.5 Domanda 5

Definire una $\mathcal{F}$ mapping reduction
Risposta
Prendiamo una classe di complessità di funzioni $\mathcal{F}$ e due linguaggi L_1 ed L_2.
Una $\mathcal{F}$ mapping reduction da L_1 a L_2, è una funzione ρ: $\Sigma_1^* \to \Sigma_2^*$ tale che:

- $\rho \in \mathcal{F}$
- $\forall w \in \Sigma_1^* \ , \ w \in L_1 \iff \rho(w) \in L_2$

$\mathcal{F}$ limita la potenza del transducer. Ci serve controllare la potenza delle riduzioni altrimenti si rischierebbe che la funzione fornita possa risolvere essa stessa il problema.

2.5.6 Domanda 6

Definire PTIME-HARD Risposta

Prendiamo per esempio una classe di complessità di funzioni $\mathcal{F} \subset FPTIME$ e un linguaggio $L \in FPTIME$. Quindi, la nostra classe $\mathcal{F}$ sarà strettamente più piccola di FPTIME.

Noi diciamo che un linguaggio L appartiene a $PTIME - HARD$ se per ogni $L' \in PTIME$, L' $\leq_F$ L.

2.5.7 Domanda 7

Definire PTIME-COMPLETE
Risposta

Dato un linguaggio L, diciamo che $L \in PTIME - COMPLETE$ se il linguaggio $L \in PTIME \cap PTIME - HARD$.

2.5.8 Domanda 8

Descrivere la nozione di circuito booleano senza variabile
Risposta

Un circuito booleano variable free è una tripla formata da $C = (N, A, \lambda)$ dove:

- La coppia $< N, A >$ è un grafo orientato aciclico (DAG)
- La funzione $\lambda : N \to \{\wedge, \vee, \neg, \bot, \top\}$ è una funzione che associa ad ogni nodo un operatore logico
- I nodi etichettati con $\neg$ hanno grado interno 1
- Esiste per forza un nodo radice
- $\lambda(p) \in \{\top, \bot\}$ solamente se p è un nodo foglia

2.5.9 Domanda 9

Definire il problema CIRCUIT VALUE e definire la complessità
Risposta

Il problema CIRCUIT VALUE è un problema decisionale che, dato un circuito C, ci chiede se il circuito è vero.
Possiamo definirlo come:

- INPUT: Un circuito booleano variable free C = (N,A,λ)
- OUTPUT: val(C) = $\top$

$$CIRCUIT - VALUE = \{< C > | val(C) = \top\} \tag{2.12}$$

La complessità del problema CIRCUIT-VALUE è PTIME-COMPLETE.

2.5.10 Domanda 10

Discutere perché CIRCUIT VALUE è in PTIME
Risposta

Se sappiamo che CIRCUIT VALUE è in PTIME-COMPLETE, sappiamo sicuramente che è in PTIME. Oltre questo, però, possiamo dimostrare la *membership*, cioè dire che $CIRCUIT - VALUE \in PTIME$ dicendo che è sufficiente navigare il circuito livello per livello, dalle foglie alla radice, e calcolare il valore di ogni nodo. Se arrivati al nodo foglia abbiamo che $val(C) = \top$, allora ritorna TRUE. False altrimenti.

2.6 Turing Machine non deterministiche

2.6.1 Domanda 1

Definire una macchina di turing non deterministica
Risposta

Una macchina di turing non deterministica è una tupla del tipo $M = (K, \Sigma, \Gamma, s, \Delta, H)$, dove :

- K è un insieme finito di stati
- Σ è l'alfabeto di input
- Γ è l'alfabeto del nastro
- $s \in K$ è lo stato iniziale
- $H \subseteq K$ è l'insieme degli stati di arresto
- $\Delta : (K - H) \times \Gamma \to \times K \times \Gamma \times \{\to, \leftarrow\})$ è la relazione di transizione

In particolare, a differenza delle macchine di turing deterministiche, ora da una configurazione C_1 è possibile arrivare a più configurazioni con gli stessi simboli sul puntatore e simoblo da leggere, per questo motivo viene definito non deterministico.

2.6.2 Domanda 2

Discutere il concetto di albero di computazione
Risposta

Un albero di computazione è un albero in cui ogni nodo rappresenta una configurazione. Una configurazione non è altro che una tupla del tipo (stato, stringa prima del puntatore, simbolo sotto il puntatore, stringa rimanente). Possiamo pensare come un albero di computazione come un albero di ricerca, in cui ogni nodo rappresenta una configurazione e ogni arco rappresenta una transizione.

Ora, nel caso deterministico, la profondità dell'albero veniva definito dal numero di passi per raggiungere la configurazione con y oppure n.

Nel caso non deterministico, la profondità dell'albero dipende dalla configurazione con lunghezza maggiore. Questo perché nel caso non deterministico possiamo avere delle configurazione più lunghe rispetto ad altre.

2.6.3 Domanda 3

Exhibit a single-tape NTM that takes in input a string 1 n and gives in output all possible strings in one for each path of the computation tree.
Risposta

Non penso di falra ora.

2.6.4 Domanda 4

Definire il linguaggio di una NTM M
Risposta

Dobbiamo partire col dire che una NTM M accetta una stringa w se esista una computazione tale che la stringa w sia accettata. Invece la rifiuta se tutte le computazioni non accettano la stringa w.

Il linguaggio di una macchina di turing non deterministica, NTM M è $\mathcal{M}$ è definito come l'insieme delle stringhe accettate dalla macchina.

Diciamo che una NTM M:

- semidecide L se L = $\mathcal{L}(M)$
- decide L se, $\forall w \in Sigma^*$, M ammette solamente un numero finito di computazioni differenti a partire da w, tutti di lunghezza finita.

2.6.5 Domanda 5

Nella dimostrazione che una DTM M e una NTM NM hanno la stessa capacità espressiva si è usato una BFS, cioè visita in ampiezza. Si potrebbe usare una visita in profondità?
Risposta

Il concetto di visita in profondità può essere fatto solamente se il numero totale di computazioni per la macchina non deterministica è FINITO. Perché questo?

Perché se dovessimo avere un numero infinito di computazioni, ovvero se in un branch andiamo in loop, non usciremmo mai. Questo perché con la visita in profondità DFS non ci visitiamo l'albero in modo orizzontale, e se becchiamo un branch che looppa, non arriveremmo mai alla configurazione di accettazione.

2.6.6 Domanda 6

Qual è l'overhead computazionale da (*semi*) **decide_L?**
Risposta

L'overhead computazionale portato dalla procedura (*semi*)decide_L non è polinomial time, ma exponential time. Noi stiamo guardando tutti i nodi nel grafo delle computazioni.

Siccome siamo nel caso non deterministico, data una stringa in input, i nodi delle configurazioni potrebbero essere esponenziali rispetto alla stringa in input. Quello che viene introdotto è un overhead perché il numero dei nodi è esponenziale e il tempo per l'esecuzione diventa esponenziale.

Al contrario, una DTM crea un singolo path e quindi il numero di nodi è polinomiale rispetto alla stringa in input, perché la computazione è unica e polinomiale rispetto all'input.

2.7 La classe di complessità NP

2.7.1 Domanda 1

Definire la funzione $timereq_M$ di una NTM
Risposta
Dobbiamo prima di tutto definire la funzione $steps(M, w)$ per una NTM. Questa funzione ci ritorna il numero di passi che la macchina esegue per raggiungere uno stato di halting, cioè la lunghezza della computazione finita più unga.
Definiamo una macchina di turing non deterministica M. Diciamo che la funzione $timereq_M(n)$ è definito come il massimo numero di passi che la macchina esegue su un input di lunghezza n.

$$\forall n \in \mathcal{N}, timereq_M(n) = max_{w \in \Sigma^n} steps(M, w) \tag{2.13}$$

Letteralmente stiamo dicendo che *per ogni n in N, il tempo richiesto da una macchina M per eseguire un input di lunghezza n è definito come il massimo numero di steps che la macchina esegue su una stringa di lunghezza n*

2.7.2 Domanda 2

Definire la classe NTIME(f) e NP
Risposta

Definiamo una funzione $f : N \to R$.
$NTIME(f)$ lo definiamo come l'insieme dei linguaggi tali per cui esista una macchina di turing non deterministica che risolve il linguaggio e ha come upper bound la funzione f.

$$NTIME = \{L \mid \exists NTMM \ tc \ \mathcal{L}(M) = L \wedge \ \forall n \in N \ timereq_M(n) \leq f(n)\} \tag{2.14}$$

Allora definiamo anche NP. NP lo definiamo come l'insieme dei linguaggi NTIME che lavorano in tempo polinomiale.

$$NP = NTIME(n^k) = \bigcup_{i>0 \ f \in O(n^i)} NTIME(f) \tag{2.15}$$

2.7.3 Domanda 3

Discutere i concetti di guess, certificato e check
Risposta

Quando parliamo di guess & check stiamo parlando di una parte di un algoritmo che si occupa di risolvere in tempo polinomiale un problema NP.
La parte di *guess* è come se un oracolo ci fornisse un oggetto che potrebbe essere la soluzione al nostro problema e che noi dobbiamo verificare.

L'oggetto ritornato dal guess è il certificato. Questa procedura è detta non deterministica poiché non viene esplicitato come viene fatto, ma è come se fosse una scelta casuale tra le possibili computazioni che la macchina può fare.
La parte deterministica riguarda, invece, il controllo del certificato. Un algoritmo esegue dei passaggi deterministici per controllare che quella computazione sia effettivamente valida. I passaggi in questa parte sono chiari e deterministici.

2.7.4 Domanda 4

Fornire una procedura non determinitica per SAT
Risposta

Ricordiamo come era definito il problema SAT:

$$SAT = \{< \phi > | \phi \text{ è una Boolean wff} \wedge \phi \text{ è soddisfacibile}\} \tag{2.16}$$

Ora, cosa significa risolvere il problema con una procedura non deterministica? Significa risolvere il problema utilizzando una procedura di guess & check.
Cosa potremmo guessare? La cosa più banale da guessare sarebbe quello di guessare il truth assignment per ϕ tale per cui ϕ sia soddisfacibile.

```
nondeterministicSAT(φ){
    guess T : vars(φ) → {⊤, ⊥} Guessato truth assignment T

    Ora serve la parte deterministica

    if (T(φ) is satisfied)
        accept computation

    reject computation
}
```

2.7.5 Domanda 5

Descrivere il problema 3COL e fornire una procedura non deterministica per risolverlo
Risposta

Il problema 3COL fa riferimento al problema della **3 Colorabilità** di un grafo.

Definiamo il problema quindi avendo un grafo non orientato G = (V,E). Il nostro obiettivo è trovare una 3 Colorabilità ovvero assegnare ad ogni nodo $v \in V$ un colore tra rosso, verde, blu tale che se esiste un arco tra due nodi u e v allora u e v hanno colori diversi.
Possiamo definire una funzione $\mathcal{C}$ che assegna ad ogni nodo di V un colore tra rosso, verde, blu.

$$\mathcal{C} : V \to \{rosso, verde, blu\} \tag{2.17}$$

Allora il problema può essere definito come:

$$3COL = \{< G > \mid \text{G è un grafo indiretto} \wedge \forall u, v \in V \ se \ \exists edge(u, v), \mathcal{C}(u) \neq \mathcal{C}(v)\} \tag{2.18}$$

Avendo definito il problema, possiamo definire una procedura guess and check che risolve il problema in tempo polinomiale.

```
solve3COL(UG: G){
    guess C: V → {rosso, verde, blu}

    for each edge(u,v) ∈ E
        if C(u) == C(v)
            reject computation

    accept computation
}
```

2.7.6 Domanda 6

Qual è la relazione tra P, NP e coNP?
Risposta

La relazione tra P, NP e coNP è la seguente:

$$P \subseteq NP \cap coNP \tag{2.19}$$

Questo è facilmente dimostrabile sapendo che P è chiuso sotto il complemento. Quindi, preso un linguaggio L $\in P$, il complemento di questo linguaggio sarà comunque in P.

Sapendo questo, prendendo un linguaggio L $\in$ NP, il suo complemento $\neg$ L $\in$ coNP.

2.8 Cock Levin no

2.9 Ancora problemi in NP-Complete

2.9.1 Domanda 1

Dimostra in modo informale che $SAT \leq_F CNF - SAT$
Risposta

Dobbiamo dimostrare che è possibile costruire una reduction da SAT a $CNF - SAT$, tale che se una forula è soddisfacibile allora anche la sua forma normale congiuntiva è soddisfacibile e viceversa.

Dobbiamo definire cosa è una CNF. Una CNF è una formula ϕ tale che le sue clausole sono congiunzioni di letterali. Un esempio:

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \tag{2.20}$$

In questo caso, vediamo che ci sono 3 letterali per ogni clausola e ogni clausola è formata da una disgiunzoine di letterali. Inoltre, ogni clausola è messa in congiunzione con le altre. Indichiamo come CNF-SAT l'insieme delle Bwff che sono soddisfacibili e che sono in forma normale congiuntiva.

$$CNF - SAT = \{\phi \in BWFF \mid \phi \text{ è soddisfacibile e } \phi \text{ è in CNF}\} \tag{2.21}$$

Per dimostrare che $SAT \leq_F CNF - SAT$ dobbiamo costruire una funzione ρ tale che ρ è calcolabile in tempo polinomiale e tale che $\phi \in SAT \iff \rho(\phi) \in CNF - SAT$.

In poche parole, partendo da una ϕ dobbiamo essere in grado di costruire una CNF $\rho(\phi)$ tale che se ϕ è soddisfacibile allora anche $\rho(\phi)$ è soddisfacibile e viceversa.

Per costruire questa CNF dobbiamo costruirci un circuito in cui i nodi foglia saranno i letterali della formula ϕ e i nodi interni saranno gli operatori logici.
Per ogni OR che incontriamo, dobbiamo trasformarlo in un AND $(y \vee z)$, dobbiamo costruirci una formula in AND che sarà del tipo $(v \vee y) \wedge (\not{v} \vee z)$. In questo modo se l'OR iniziale è soddisfacibile, anche la formula in AND sarà soddisfacibile.
In questo modo stiamo rappresentando allo stesso modo la ϕ ma in forma di CNF.

2.9.2 Domanda 2

Trasforma $((x \wedge y) \vee (\neg x \wedge z)) \wedge (y \vee \neg z)$ in una CNF con un algoritmo in un tepmo polinomiale.
Risposta

Per farlo, possiamo usare l'algoritmo precedente.

$$\begin{aligned} ((x \wedge y) \vee (\neg x \wedge z)) \wedge (y \vee \neg z) = \\ ((x \wedge y) \vee (\neg x \wedge z)) \wedge ((v \vee y) \wedge (\neg v \vee \neg z)) = \\ ((u \vee (x \wedge y)) \wedge (\neg u \vee (\neg x \wedge z))) \wedge ((v \vee y) \wedge (\neg v \vee \neg z)) \end{aligned} \tag{2.22}$$

2.9.3 Domanda 3

Fare un confronto tra il concetto di equivalenza ed equisoddisfacibilità delle Bwffs
Risposta

Equivalenza: due formule ϕ_1 e ϕ_2 sono equivalenti se e solo se hanno lo stesso insieme di variabili. I.E hanno gli stessi assegnamenti di verità.

Equisoddisfacibilità: data una formula, è possibile mimare lo stesso comportamento di essa con l'aggiunta di una variable extra tale che se una formula $x \vee y$ è vera, allora è possibile aggiungere v e $\neg v$ tale che $(x \vee v) \wedge (y \neg v)$ è vera.

2.9.4 Domanda 4

Qual è la proprietà chiave che ci aiuta a dimostrare che $SAT \leq_F CNF - SAT$
Risposta

La proprietà è quelal dell'equisoddisfacibilità, cioe:

$$\phi_1 \vee \phi_2 \in SAT \iff (\phi_1 \vee x) \wedge (\phi_2 \vee \neg x) \in SAT \tag{2.23}$$

2.9.5 Domanda 5

Spiega 3SAT-EQ e dimostra che CNF-SAT $\leq_F$ 3SAT-EQ
Risposta

Per spiegare 3SAT-EQ, dobbiamo prima spiegare cosa $3CNF_=$.
Noi sappiamo che una 3CNF è l'insieme delle formule in CNF che hanno al massimo 3 letterali per clausola. Invece $3CNF_=$ fa riferimento alle CNF che hanno esattamente 3 letterali per clausola.
Ora, sapendo questo, possiamo definire 3SAT-EQ:

$$3SAT - EQ = \{< \phi > \mid \phi \text{ è una Bwff in } 3CNF_= \wedge \phi \text{ è soddisfacibile}\} \tag{2.24}$$

Ora, noi vogliamo dimostrare che la reductoin da CNF-SAT a 3SAT-EQ è possibile. Per farlo, definiamo anche cos'è CNF-SAT.

$$CNF - SAT = \{< \phi > \mid \phi \text{ è una Bwff in CNF} \wedge \phi \text{ è soddisfacibile}\} \tag{2.25}$$

Ora, per dimostrare che CNF-SAT $\leq_F$ 3SAT-EQ dobbiamo costruire una funzione ρ tale che ρ è calcolabile in tempo polinomiale e tale che $\phi \in CNF - SAT \iff \rho(\phi) \in 3SAT - EQ$.

L'idea è quella di considerare C_i le clausole formula ϕ. Per ogni clausola, abbiamo dei letterali.

- $\phi = C_1 \wedge \cdots \wedge C_n$
- $C_i = l_i \vee \cdots \vee l_k$, per $i = 1, \ldots, k$, con k ¿ 0

L'idea chiave ora è fare in modo che ogni formula che ha un numero diverso di letterali per clausola, venga trasformata in una formula che ha esattamente 3 letterali per clausola. Questo viene fatto utilizzando una funzione *norm* che fa esattamente quando indicato sopra. Quindi, per ogni clausola, dipendentemente dal numero di letterali, viene trasformata in una clausola che ha esattamente 3 letterali.

Esempio:

- k = 3: $C_i = l_1 \vee l_2 \vee l_3$

- k = 2: $C_i = (l_1 \vee l_2) = (l_1 \vee l_2 \vee v) \wedge (l_1 \vee l_2 \vee \neg v)$
- k = 1: $C_i = l_1 = (l_1 \vee v \vee u) \wedge (l_1 \vee \neg v \vee u) \wedge (l_1 \vee v \vee \neg u) \wedge (l_1 \vee \neg v \vee \neg u)$ tutte le possibili combinazioni per avere i 3 letterali
- k ¿ 3: Spezzetta i letterali in più in più clausole legate e negando alcune variabili per mantenere la formual equisoddisfacibile

In pratica, definiamo ρ come segue:

$$\rho(\phi) = \bigwedge_{i \in 1 \ldots n} norm(C_i) \tag{2.26}$$

Abbiamo quindi dimostrato che CNF-SAT $\leq_F$ 3SAT-EQ.

2.9.6 Domanda 6

Trasformare la formula $((x \wedge y) \vee (\neg x \wedge z)) \wedge (y \vee \neg z)$ **in una 3CNF$_=$.**
Risposta
Per eseguire questa operazione, ci basta applicare per ogni clausola la funzione *norm*, ovvero di trasformare ogni clausola in una clausola che ha esattamente 3 letterali.

$$\begin{aligned}
((x \wedge y) \vee (\neg x \wedge z)) \wedge (y \vee \neg z) = \\
((x \wedge y) \vee (\neg x \wedge z)) \wedge (y \vee \neg z \vee v) \wedge (y \vee \neg z \vee \neg v) = \\
(\neg(x \wedge y) \wedge \neg(\neg x \wedge z)) \wedge (y \vee \neg z \vee v) \wedge (y \vee \neg z \vee \neg v) = \\
(\neg x \vee \neg y) \wedge (x \vee \neg z) \wedge (y \vee \neg z \vee v) \wedge (y \vee \neg z \vee \neg v) = \\
(\neg x \vee \neg y \vee u) \wedge (\neg x \vee \neg y \vee \neg u) \wedge (x \vee \neg z \vee j) \wedge (x \vee \neg z \vee \neg j) \wedge (y \vee \neg z \vee v) \wedge (y \vee \neg z \vee \neg v)
\end{aligned} \tag{2.27}$$

2.10 La gerarchia polinomiale

2.10.1 Domanda 1

Definisci UNIQUE-SAT e classificalo
Risposta

Con UNIQUE-SAT intediamo l'insieme delle Bwff tale che esiste una sola assegnazione di verità che soddisfa la formula.

$$UNIQUE - SAT = \{< \phi > | \phi \in BWFF \text{ e } \exists \text{ una sola assegnazione di verità che soddisfa } \phi\} \tag{2.28}$$

UNIQUE-SAT $\in coNP - hard$

2.10.2 Domanda 2

Dimostra che UNIQUE-SAT ∈ coNP-hard
Risposta

Per dimostrare che UNIQUE-SAT ∈ coNP-hard, dobbiamo dimostrare che $\forall L \in coNP, L \leq_p UNIQUE-SAT$.
Ma, invece di fare cosi, potremmo utilizzare un linguaggio coNP-hard e dimostrate che è riducibile a UNIQUE-SAT.
In particolare, possiamo utilizzare UNSAT. Infatti UNSAT ∈ coNP-hard.

Ricordiamoci com'è definito UNSAT:

$$UNSAT = \{< \phi > | \phi \in BWFF \text{ e } \phi \text{ non è soddisfacibile}\} \tag{2.29}$$

Sapendo ciò, dobbiamo costruire una reduction ρ tale che $\phi \in UNSAT \iff \rho(\phi) \in UNIQUE - SAT$. Per fare ciò, dobbiamo costruire una formula ϕ' tale che ϕ' è soddisfacibile da un solo truth assignment $\iff$ ϕ non è soddisfacibile.

Una formula $\rho(\phi) = \phi'$ può essere definita in questo modo:

$$\rho(\phi) = \phi' = (x_0 \wedge \cdots \wedge x_n) \vee (\neg x_0 \wedge \phi) \tag{2.30}$$

$[\implies]$
Se $\phi \in UNSAT$ vuol dire che non ci sono assegnamenti di verità che rendono ϕ vera e, costruendo la nostra nuova ϕ' in questo modo, sappiamo che l'unico truth assignment $T : vars(\phi) \to \{\top, \bot\}$ è singolo. Questo assegnamento è il seguente:

$$T_1 = \{x_i \to T : 0 \leq i \leq n\} \tag{2.31}$$

In questo modo, siamo sicuri al 100% che $\phi' \in UNIQUE - SAT$. Questo perché se tutte le variabili sono TRUE, allora la prima parte della formula è TRUE, e quindi la formula è TRUE.
$[\impliedby]$
Se $\phi \notin UNSAT$ significa che esiste almeno un truth assignment per ϕ. Questo vorrebbe dire che nella nostra nuova formula ϕ', esistono almeno due truth assignment.
Queto rende $\phi' \notin UNIQUE - SAT$.
In particolare, i truth assignment sono:

1. $T_1 = \{x_i \to T : 0 \leq i \leq n\}$
2. $T_2 = \{x_0 \to \bot\} \cup T$ – Mettendolo a false abbiamo la seconda parte della formula true, per un determinato Truth Assignment per ϕ

2.10.3 Domanda 3

Discutere il concetto di macchina oracolo
Risposta

La macchina oracolo è una macchina che lavora in tempo costante $\mathcal{O}(1)$. Questa tipologia di macchina è come se stesse simulando un forall all'interno dell'algoritmo di guess and check.

Ad esempio, se ci fosse bisogno di controllare se esiste un truth assignment per una determinata formula, si potrebbe usare una macchina oracolo O_{SAT} che controlla se la formula è soddisfacibile. Se non avessimo la possibilità di utilizzare una macchina del genere, non potremmo controllare in un tempo polinomiale se la formula è soddisfacibile.
Da notare che un oracolo può essere chiamato più volte durante l'esecuzione di un algoritmo.

2.10.4 Domanda 4

Definire formalmente la gerarchia polinomiale
Risposta
La gerarchia polinomiale è un insieme di classi che identificato delle classi di difficoltà degli algoritmi che lavorano in tempo polinomiale e che utilizzano degli oracoli per risolvere dei problemi.
In modo formale, la gerarchia polinomiale è definita in questo modo:

- $\Sigma_0^P = \Pi_0^P = \Delta_0^P = P$
- per $k > 0$:
  - $\Delta_{k+1}^p = P^{\Sigma_k^p}$
  - $\Sigma_{k+1}^p = NP^{\Sigma_k^p} = NP^{\Pi_k^p}$
  - $\Pi_{k+1}^p = co\Sigma^{p_k+1} = coNP^{\Sigma_k^p} = coNP^{\Pi_k^p}$

In particolare, possiamo scrivere:

$$PH = \cup_{k\geq 0}\Sigma_k^p = \cup_{k\geq 0}\Pi_k^p = \cup_{k\geq 0}\Delta_k^p \tag{2.32}$$

2.10.5 Domanda 5

Dimostrare con una procedura che UNIQUE-SAT $\in \Sigma_2^p$
Risposta

Se diciamo che UNIQUE-SAT $\in \Sigma_2^P$, vuol dire che esiste una macchina di Turing non deterministica M che lavora in tempo polinomiale tale che utilizza un oracolo in Σ_1^p

La procedura è la seguente:

Ricordiamoci come funziona la funzione wff: Prendendo un truth assignment $T : V \to \{\top, \bot\}$, dove V è praticamente un insieme di variabili booleano $\{x_1, \ldots, x_n\}$

Diciamo che la Boolean wff **wff(T)**:

$$wff(T) = l_1 \wedge \cdots \wedge l_n \tag{2.33}$$

dove:

- $l_i = x_i$ se $\mathrm{T}(x_i) = \top$

- $l_i = \neg x_i$ se $\mathrm{T}(x_i) = \bot$

```
decide–UNIQUE–SAT(Boolean wff φ){
    guess T: vars(φ) → {⊤, ⊥};

    if (T ⊭ φ)
        reject computation
    else{
        φ′ = wff(T)
        φ″ = φ ∧ ¬φ′
        if (O_SAT(φ″) == y)
            reject computation

        accept computation
    }
}
```

Usando l'oracolo stiamo controllando che effettivamente la formula ϕ'' sia non soddisfacibile, rendendo cosi la formula ϕ unica soddisfacibile.

Sappiamo che la classe di SAT e' **NP**. Siccome noi stiamo lavorando in tempo polinomiale con il guess and check e l'oracolo lavora in tempo polinomiale, cioè Σ_1^p, allora UNIQUE-SAT $\in \Sigma_2^p$.

2.11 Dentro la gerarchia polinomiale

2.11.1 Domanda 1

Dimostra che UNIQUE-SAT è Δ_2^P usando una procedura con un oracolo
Risposta

Stiamo dicendo che quindi UNIQUE-SAT è P^{NP}, cioè utilizziamo una procedura che lavora in tempo polinomiale senza utilizzare un guess and check, ma utilizza un oracolo che lavora in NP.

Il concetto è semplice. Non dobbiamo guessare nessun truth assignment per la formula, ma un metodo è quello di controllare per ogni variabile che compare nella formula se la formula sarebbe soddisfacibile anche con quella variabile negata.

```
UNIQUE–SAT–P^NP(Bwff φ){
    if (O_SAT(φ) == n)
        return false
    for each variable x_i in vars(φ)
        if (O_SAT(φ) ∧ x_i == y) ∧ (O_{SAT}(φ) ∧ ¬x_i == y)
            return false

    return true
```

}

Cioè, stiamo controllando in tempo polinomiale con un oracolo che lavora in NP se la formula è soddisfatta sia dalla variabile che dalla sua negazoine. Se questo fosse vero, allora vuol dire che esisterebbero 2 assegnamenti di verità e la formula non sarebbe più in UNIQUE-SAT.

2.11.2 Domanda 2

Dimostra che UNIQUE-SAT $\in$ DP
Risposta

Dobbiamo definire innanzitutto cos'è DP.

Prendiamo due linguaggi L1 e L2 tali che $L_1 \in NP$ e $L_2 \in coNP$. Indichiamo DP come l'insieme delle intersezioni tra L1 e L2.

$$DP = \{L1 \cap L2 : L1 \in NP \wedge L2 \in coNP\} \tag{2.34}$$

Ora, come facciamo a dire che UNIQUE-SAT $\in DP$?
Per farlo, dobbiamo trovare due linguaggi, uno in NP e uno in coNP, tali che l'intersezione tra i due sia UNIQUE-SAT.
Se ci pensiamo, UNIQUE-SAT è definito come:

$$UNIQUE - SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{ è soddisfacibile} \wedge \text{ ha un solo assegnamento di verità}\} \tag{2.35}$$

Ora, se pensiamo alla definizione, noi conosciamo altri due insiemi tali per cui l'intersezione è proprio UNIQUE-SAT.

- SAT
- 1-SAT

Perché:

$$SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{ è soddisfacibile}\} \in NP \tag{2.36}$$

$$1 - SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{ è soddisfacibile} \wedge \text{ ha al massimo un solo assegnamento di verità}\} \in coNP \tag{2.37}$$

Sapendo che SAT $\in NP$ e che 1-SAT $\in coNP$, e sapendo che UNIQUE-SAT è l'intersezione tra i due, possiamo dire che UNIQUE-SAT $\in DP$. Per farlo, dobbiamo dimostrarlo.

$[\implies]$
Se $\phi \in UNIQUE - SAT$ significa che la formula è soddisfacibile e che ha al massimo 1 assegnamento di verità. Da questo possiamo dire che $< \phi > \in SAT \cap 1 - SAT$

$[\impliedby]$
Se $< \phi > \in SAT \cap 1 - SAT$ significa che la formula è soddisfacibile e che ha al massimo 1 assegnamento di verità. Da questo possiamo dire che $\phi \in UNIQUE - SAT$

2.11.3 Domanda 3

Definire 1-SAT e dimostrare che è coNP-hard
Risposta

Come detto prima, abbiamo che 1-SAT è definoto come:

$$1 - SAT = \{< \phi >: \phi \text{è una Bwff} \wedge \text{ ha al massimo 1 assegnamento di verità}\} \tag{2.38}$$

Per dimostrare che appartiene a coNP, dobbiamo dimostrare che il suo complemento $\neg 1 - SAT$ appartiene a NP.
Per farlo, possiamo dire che $\neg 1 - SAT$ è definito come:

$$\neg 1 - SAT = \{< \phi >: \phi \text{è una Bwff} \wedge \text{ ha più di 1 assegnamento di verità}\} \tag{2.39}$$

Questo è dimostrabile con una procedura non deterministica che lavora in tempo polinomiale di questo tipo:

- guess: un assegnamento di verità T1
- guess: un assegnamento di verità T2
- check: se ϕ è soddisfacibile con T1 e T2 allora accetta
- altrimenti, rifiuta

In questo modo abbiamo dimostrato che $\neg 1 - SAT \in NP$ e quindi che $1 - SAT \in coNP$.
Ora, dobbiamo dimostrare l'hardness. Questo significa che dobbiamo provare che un qualsiasi linguaggio in coNP è riducibile a 1-SAT.
Oppure, possiamo usare un linguaggio in coNP-hard che già conosciamo e ridurlo a 1-SAT.

Un linguaggio che sappiamo essere coNP-hard è UNSAT:

$$UNSAT = \{< \phi >: \phi \text{è una Bwff} \wedge \text{ non è soddisfacibile}\} \tag{2.40}$$

Per dimostrare che 1-SAT è coNP-hard, dobbiamo dimostrare che UNSAT $\leq_p$ 1-SAT.
Cioè, dobbiamo trovare una funzione $\rho \in \mathcal{F} \subset FP$ tale che:

$$< \phi > \in UNSAT \iff \rho(< \phi >) \in 1 - SAT \tag{2.41}$$

Per farlo, dobbiamo trovare una funzione che trasforma una formula non soddisfacibile in una formula che ha al massimo un assegnamento di verità.
Pensiamocoi. Se la formula è insoddisfacibile, qualsiasi assegnamento di verità che facciamo, la formula non sarà mai soddisfacibile.
L'unico modo per generare un assegnamento di verità unico è quello di generare una formula sempre vera.

$\rho(\phi) = \phi' = (x_1 \wedge \cdots \wedge x_n) \vee (\neg x_1 \vee \phi)$
La nostra nuova formula funzionerà in questo modo:

- $\Longrightarrow$ Se $\phi \in UNSAT$, l'unico assegnamento di verità valido è quello che rende vera la parte $(x_1 \wedge \cdots \wedge x_n)$ della formula. $T = x_i \to \top : 0 \leq i \leq n$. Allora $\phi' \in 1 - SAT$

- $\Longleftarrow$ Se $\phi \notin UNSAT$, allora esisteranno almeno 2 assegnamenti di verità che rendono vera ϕ', sia un altro assegnamento che rende x_1 falsa e $\neg x_1$ vera. Allora $\phi' \notin 1 - SAT$

Abbiamo dimostrato quindi che UNSAT $\leq_p$ 1-SAT e quindi che 1-SAT è coNP-hard.

2.11.4 Domanda 4

Definire 1-SAT e dimostrare che è in coNP
Risposta

Ricopio la risposta di prima:
Come detto prima, abbiamo che 1-SAT è definoto come:

$$1 - SAT = \{< \phi >: \phi \text{è una Bwff} \wedge \text{ ha al massimo 1 assegnamento di verità}\} \tag{2.42}$$

Per dimostrare che appartiene a coNP, dobbiamo dimostrare che il suo complemento $\neg 1 - SAT$ appartiene a NP.
Per farlo, possiamo dire che $\neg 1 - SAT$ è definito come:

$$\neg 1 - SAT = \{< \phi >: \phi \text{è una Bwff} \wedge \text{ ha più di 1 assegnamento di verità}\} \tag{2.43}$$

Questo è dimostrabile con una procedura non deterministica che lavora in tempo polinomiale di questo tipo:

- guess: un assegnamento di verità T1
- guess: un assegnamento di verità T2
- check: se ϕ è soddisfacibile con T1 e T2 allora accetta
- altrimenti, rifiuta

In questo modo abbiamo dimostrato che $\neg 1 - SAT \in NP$ e quindi che $1 - SAT \in coNP$.

2.11.5 Domanda 5

Definire UNIQUE-SAT, 1SAT, SAT. Discutere la loro relazione
Risposta

Ricopio le definizioni date prima:

$$UNIQUE - SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{è soddisfacibile} \wedge \text{ ha esattamente 1 assegnamento di verità}\} \quad (2.44)$$

$$1 - SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{ è soddisfacibile} \wedge \text{ha al massimo 1 assegnamento di verità}\} \quad (2.45)$$

$$SAT = \{< \phi >: \phi \text{ è una Bwff} \wedge \text{ è soddisfacibile}\} \quad (2.46)$$

Ora, la loro relazione è che UNIQUE-SAT è esattamente uguale all'intersezione tra SAT E 1-SAT.

$$UNIQUE - SAT = SAT \cap 1 - SAT \quad (2.47)$$

2.11.6 Domanda 6

Definire $k - QBF_{\exists}$ e $k - QBF_{\forall}$
Risposta

Prendiamo degl insiemi $X_1, \ldots, X_k$ di coppie di variabili booleane disgiunte, cioè la loro intersezione è uguale a .
Ora, definiamo $k-QBF_{\exists}$ come una formula Φ tale che $\Phi = \exists X_1 \forall X_2 \ldots \exists X_k \phi$ dove ϕ è una formula booleana del tipo $X_1 \cup \cdots \cup X_k$, quando k è pari.
Altrimenti, se k è dispari, allora $\Phi = \exists X_1 \forall X_2 \ldots \forall X_k \ \phi$

Invece, per $k - QBF_{\forall}$ è il contrario, cioè $\Phi = \forall X_1 \exists X_2 \ldots \forall X_k \phi$ dove ϕ è una formula booleana del tipo $X_1 \cup \cdots \cup X_k$, quando k è pari.
Altrimenti, se k è dispari, allora $\Phi = \forall X_1 \exists X_2 \ldots \exists X_k \ \phi$

Ora, noi diciamo che una formula k-QBF Φ è soddisfacibile se:

- Φ è una formula $1 - QBF_{\exists}$ e wff(Φ) è soddisfacibile
- Φ è una formula $1 - QBF_{\forall}$ e wff(Φ) è valida
- Φ è una formula $k-QBF_{\exists}$ con k > 1 della forma $\exists X_1 \forall X_2 \ldots \exists X_k \phi$ e ϕ ed **esiste un truth assignment** $T : X_1 \to \{\top, \bot\}$ tale che la formula (k-1)-$QBF_{\forall}$ $\Phi = \forall X_2 \ldots \exists X_k T(\phi)$ è soddisfacibile
- Φ è una formula $k-QBF_{\forall}$ con k > 1 della forma $\forall X_1 \exists X_2 \ldots \forall X_k \phi$ e ϕ e **per ogni truth assignment** $T : X_1 \to \{\top, \bot\}$ tale che la formula (k-1)-$QBF_{\exists}$ $\Phi = \exists X_2 \ldots \forall X_k T(\phi)$ è soddisfacibile

2.11.7 Domanda 7

Dimostrare che $QSAT_{2,\exists} \in \Sigma_2^P$
Risposta

Iniziamo a definire $QSAT_{k,\exists}$. Definiamo l'insieme:

$$QSAT_{k,\exists} = \{< \Phi >: \Phi \text{ è una } k - QBF_{\exists} \wedge \Phi \text{ è soddisfacibile}\} \quad (2.48)$$

Indichiamo quindi tutte quelle formule che iniziano col quantificatore $\exists$ e che hanno un'alternanza di quantificatori pari a k. Per ogni sotto formula (k-1) deve esistere un assegnamento di verità per le variabili sotto lo scopo del quantificatore.

Ora, dobbiamo dimostrare peché $QSAT_{2,\exists} \in \Sigma_2^P$. Per farlo scriviamo una procedura che ci permette di definirlo, che lavora in tempo polinomiale, utilizzando una procedura guess and check (quindi non determinsitica) e che chiama un oracolo in Σ_1^P.

N.B: con wff(Φ) indichiamo la formula senza quantificatori.

```
decide¬QSAT_{2,∃}(2QBFΦ){
    guess un truth assignment: T : vars_∃(Φ) → {⊤, ⊥}
    φ' = wff(T(Φ))
    φ'' = ¬φ'

    if (O_SAT(φ'') ==' n')
        accept

    reject
}
```

Spiegazione: questo funziona perché stiamo guessando un assegnamento di verità per le variabili sotto lo scopo del quantificatore $\exists$. Ora, stiamo prendendo la formula senza quantificatori con ϕ'. Successivamente neghiamo questa formula, e la passiamo ad un oracolo che controlla se la formula è soddisfacibile.

Cosa succede se la formula non è soddisfacibile? Significa che, per qualsiasi assegnamento di verità alle variabili, la formula è falsa. Se questo è vero, allora vuol dire che la formula originale, che ha dato origine al complemento, sarà **valida**. Quindi, se la formula è valida, allora vuol dire che la formula originale è soddisfacibile.

2.11.8 Domanda 8

Definire SAT-UNSAT e dimostrare che è DP-hard
Risposta

2.11.9 Domanda 9

Dimostrare che UNIQUE-SAT è DP-hard
Risposta

2.12 La complessità dei circuiti

2.12.1 Domanda 1

Definire formalmente il concetto di circuito
Risposta

Chiamiamo un circuito un grafo orientato aciclico $C = (N, A, \lambda)$, con:

- N insieme dei nodi
- A l'insieme degli archi
- λ è una funzione di etichettatura che associa ad ogni nodo un etichetta

In particolare, parlando di λ, dobbiamo fare attenzione alle varie etichette:

- I nodi che hanno zero archi entranti sono chiamati nodi di input e sono etichettati con le variabili
- I nodi che hanno zero archi uscenti sono chiamati nodi di output
- I nodi che hanno sia archi entranti che uscenti sono nodi che rappresentano degli operatori logici

N.B: Se il circuito ha un solo nodo output viene chiamato circuito booleano.

2.12.2 Domanda 2

Classifica e discuti le varie tipologie di circuiti
Risposta

Abbiamo suddiviso i circuiti in 3 tipi:

- Circuiti booleani
- Circuiti
- Circuiti booleani senza variabili

Circuiti booleani
I circuiti booleani sono circuiti che hanno come nodi di input delle variabili $x_1, \ldots, x_n$ con n che rappresenta il numero di variabili.
I nodi di output sono un solo nodo che rappresenta il risultato della formula, per questo motivo si chiama circuito booleano.

Circuiti
Un circuito è semplicemente un grafo che ha come nodi di input delle variabili $x_1, \ldots, x_n$ con n che rappresenta il numero di variabili, e come nodi di output dei nodi che rappresentano degli operatori logici.

Circuiti booleani senza variabili
I circuiti booleani senza variabili sono dei circuiti che hanno come nodi di input dei valori booleani $\top, \bot$, e come nodi di output un solo nodo che rappresenta il risultato della formula.

2.12.3 Domanda 3

Definire l'output di un circuito generale e quello di un circuito variable free
Risposta
Non so cosa intende la domanda.

Un circuito generale ha un input determinato dalle variabili $x_1, \ldots, x_n$ e un output determinato dal nodo finale del circuito che, se non booleano, rappresenta un operatore logico.

Un circuito variable free è della stessa tipologia di un circuito genreale, ma l'unica cosa che cambia è solamente il fatto che in input non ci sono delle variabili ma dei valori booleano $\top, \bot$.

Non so cosa voglia.

2.12.4 Domanda 4

Definire una famiglia di circuiti booleani e il linguaggio che definisce
Risposta

Dobbiamo partire con capire cosa significa famiglia di circuiti. Indichiamo con famiglia di circuiti una **sequenza** di circuiti $C_1, \ldots, C_n$ dove ogni circuito C_i ha n_i variabili di input.
Ora, una famiglia di circuiti booleani è una famiglia di circuiti dove ogni circuito C_i ha n_i variabili di input e un solo nodo di output.

Il linguaggio di una famiglia di circuiti booleani è l'insieme di tutte le stringhe tali per cui un circuito che ha un numero di nodi di input, cioè di variabili, pari alla cardinalità della stringa da riconoscere, restituisce 1.

$$\mathcal{L}(\{C_n\}) = \{x \in \{0,1\}^+ : C_{|x|}(x) = 1\} \tag{2.49}$$

Stiamo letteralmente dicendo che tutte le stringhe del tipo 1,0 tali per cui il circuito C_n che ha n variabili di input, restituisce 1, appartengono al linguaggio della famiglia di circuiti booleani.

2.12.5 Domanda 5

Definisci il concetto di funzione di una famiglia di circuiti
Risposta

Una funzione di una famiglia di circuiti è una coppia del tipo input, output.
Essa ci indica, dando un certo input x ad una un circuito con numero di nodi pari alla cardinalità della stringa, appartenente ad una fmaiglia di circuiti, quale sarà l'output del circuito.

$$f(\{C_n\}) = \{x \implies y : C_{|x|}(x) = y\} \tag{2.50}$$

2.12.6 Domanda 6

Definire la classe AC^i e FAC^i
Risposta

Io davver non so cosa dire, davvero.

AC^i

La classe AC^i è la classe delle funzioni calcolabili da una famiglia di circuiti booleani con profondità DEPTH $O(\log^i n)$, quindi logaritmicha rispetto all'input, e SIZE $O(n^k)$, quindi polinomiale rispetto all'input.
Definiamo al volo SIZE e DEPTH anche se proprio guarda non riesco a capire: Data una funzione di complessità $g : N \to N$

FAC^i

- $\textbf{SIZE}(g(n)) = \{L : \exists \{C_n\} \text{ s.t. } \mathcal{L}(\{C_n\}) = L \wedge \sigma_{\{C_n\}}(n) \leq g(n)\}$
- $\textbf{SIZE}(n^k) = \textbf{SIZE}(n^{\mathcal{O}(1)}) = \bigcup_{i>0} \textbf{SIZE}\left(\mathcal{O}(n^i)\right)$
- $\textbf{DEPTH}(g(n)) = \{L : \exists \{C_n\} \text{ s.t. } \mathcal{L}(\{C_n\}) = L \wedge \delta_{\{C_n\}}(n) \leq g(n)\}$

Figure 2.1: AC0

- $\textbf{FSIZE}(g(n)) = \{\mu : \exists \{C_n\} \text{ s.t. } f(\{C_n\}) = \mu \wedge \sigma_{\{C_n\}}(n) \leq g(n)\}$
- $\textbf{FSIZE}(n^k) = \textbf{FSIZE}(n^{\mathcal{O}(1)}) = \bigcup_{i>0} \textbf{FSIZE}\left(\mathcal{O}(n^i)\right)$
- $\textbf{FDEPTH}(g(n)) = \{\mu : \exists \{C_n\} \text{ s.t. } f(\{C_n\}) = \mu \wedge \delta_{\{C_n\}}(n) \leq g(n)\}$

Figure 2.2: FAC0

Con FAC^i che è la classe delle funzioni calcolabili da una famiglia di circuiti booleani senza variabili con profondità FDEPTH $O(\log^i n)$, quindi logaritmicha rispetto all'input, e FSIZE $O(n^k)$, quindi polinomiale rispetto all'input.

2.12.7 Domanda 7

Discuti la relazione tra AC^i e FAC^i con le classi di complessità basate sulle TMs
Risposta

Non so cosa dire, davvero ancora di più.
Se definiamo $AC^i = \bigcup_{\{} i \geq 0\} AC^i$ e $FAC^i = \bigcup_{\{} i \geq 0\} FAC^i$.
Per un teorema, regge che:

- $AC^0 \subseteq AC^1 \subseteq AC^2 \subseteq \cdots \subseteq AC \subseteq P \subseteq NP$
- $FAC^0 \subseteq FAC^1 \subseteq FAC^2 \subseteq \cdots \subseteq FAC \subseteq FP$
- $AC^0 \subset P \subseteq NP$
- $FAC^0 \subset FP$

Copia incolla da Matteo Perfidio: La classe AC0 'e sottoinsieme stretto di P, e anche la sua corrispettiva funzione 'e in FP. Dunque, linguaggi riconosciuti da famiglie di circuiti di questo tipo sono di complessit'a inferiore a PTIME. Essa rappresenta dunque la classe a cui devono appartenere le ρ delle nostre reduction dei linguaggi PTIME. Ovviamente, dopo aver costruito la reduction, 'e necessario dimostrare l'appartenenza ad AC0. Anche la reduction di Cook-levin 'e dimostrabile tramite un circuito AC0. Ovviamente i linguaggi AC0 operano su alfabeto binario quindi va trasformata la formula costruita tramite la matrice delle computazioni in codice binario.

2.12.8 Domanda 8

Costruisci in FAC0 un circuito per il complementare di un grafo

2.12.9 Domanda 9

Dimostra che AC0 è chiuso sotto unione, intersezione e complemento

2.13 Oltre la gerarchia polinomiale

2.13.1 Domanda 1

Fornisci una procedura deterministica in PSPACE per SAT, UNSAT e VALID
Risposta

2.13.2 Domanda 2

Parla della relazione che c'è tra $PSPACE$ **E** $PSPACE^{PSPACE}$
Risposta

Il concetto di relazione che c'è tra problemi che lavorano in $PSPACE$ e altri che lavorano $PSPACE^{PSPACE}$ sono considerabili ugualmente in spazio polinomiale.

Per essere precisi, un algoritmo $PSPACE$ è un algoritmo deterministico che lavora in spazio polinomiale, e un algoritmo in $PSPACE^{PSPACE}$ è un algoritmo anche deterministico che lavora in spazio polinomiale che chiama un oracolo che lavora in spazio polinomiale.

Ora, lo spazio totale alla fine sarà considerato sempre polinomiale, poiché lo spazio utilizzato non si moltiplica ma si somma. Quindi, se un algoritmo lavora in spazio polinomiale e chiama un oracolo che lavora in spazio polinomiale, lo spazio totale sarà sempre polinomiale. Quindi, possiamo dire che $PSPACE = PSPACE^{PSPACE}$.

Accorgimenti di Simone:

Abbiamo questo tape che lavora in PSPACE. Chiamando l'oracolo è come se mettessimo un separatore su dove stiamo lavorando. La destra di questo separatore è riservato all'oracolo. A sinitra abbiamo i calcoli fatti senza l'oracolo e sono in PSPACE, a destra abbiamo l'oracolo che lavora in PSPACE.

Siamo sicuri di essere in PSPACE? Si, perché per definizione l'oracolo viene chiamato un numero polinomiale di volte. Un polinomio per spazio polinomiale è sempre spazio polinomiale.

2.13.3 Domanda 3

Parla della relazione che c'è tra $PSPACE$ **E** $NPSPACE$
Risposta

E' lo stesso concetto che abbiamo utilizzato per la risposta precedente.

Se prendiamo un algoritmo deterministico che lavora in spazio polinoiale e un algoritmo non deterministico che lavora in spazio polinomiale, essi sono considerabili ugualmente in spazio polinomiale. Non cambia assolutamente nulla.

Possiamo quindi dire anche in questo caso che $PSPACE = NPSPACE$.

Accorgimenti di Simone:

I motivi sono diversi. una NTM in spazio polinomiale, ogni singolo nodo dell'albero sarà sempre spazio polinomiale. Noi possiamo simulare una macchina che usa spazio polinomiale con una macchina deterministica facendo una sorta di backtracking o reachability. Noi esploriamo un ramo e una volta che viene esplorato, ogni configurazione siamo sicuri che è in spazio polinomiale e quando torni indietro (cioè applichi il backtracking), cancelli quello che hai avuto sul tape e riscrivi sulla stessa parte. Quindi rimani su spazio polinomiale

2.13.4 Domanda 4

Parla della rappresentazione succinta di un grafo diretto
Risposta

Parliamo prima di cosa intendiamo con rappresentazione succinta di un grafo.

Una rappresentazione succinta di un grafo non è altro che la conversione di un grafo in un circuito. Questo circuito è un tipo di circuito che ha come numero di nodi $2 \circ \lceil \log n \rceil$ e come spazio totale $\mathcal{O}(log\ n)$. Lo scopo del circuito è quello di dire se nel grafo iniziale G, dati due nodi u, v, esiste un arco tra i due nodi. Questo viene fatto tramite una codifica binaria dei nodi u, v che vengono rappresentati dai nodi in input.

2.13.5 Domanda 5

Parla della rappresentazione succinta di un circuito booleano variable free
Risposta

Un circuito booleano variable free è un circuito che non ha variabili in input e che ha come nodo di output un unico nodo.

Ora, per rappresentare un circuito in un altro circuito succinto, non ho idea di come fare. Quello che so è che un circuito succinto è un circuito che ha come numero di nodi $2 \circ \lceil \log n \rceil$ e come spazio totale $\mathcal{O}(log\ n)$.

QUESTA DOMANDA E' DA CONTROLLARE

2.13.6 Domanda 6

Definire SUC-CIRCUIT-VALUE e SUC-3SAT
Risposta

SUC-CIRCUIT-VALUE
L'insieme dei circuiti C tale che C è un circuito e l'output della rappresentazione succinta sia uguale a 1.

$$SUC - CIRCUIT - VALUE = \{C : C \text{ è un circuito} \wedge vfbc(C) = 1\} \tag{2.51}$$

SUC-3SAT
L'insieme dei circuiti tali per cui C è un circuito e Bwff(C) contiene 3 letterali e C è soddisfacibile.

$$SUC - 3SAT = \{C : C \text{ è un circuito} \wedge Bwff(C) \text{ contiene 3 letterali} \wedge C \text{ è soddisfacibile}\} \tag{2.52}$$

Tutti e due sono in EXPTIME-complete.

2.13.7 Domanda 7

Parla delle chiusure per PSPACE, EXPTIME, NEXPTIME
Risposta

PSPACE e EXPTIME sono chiusi rispetto a:

- Unione
- Intersezione
- Concatenazione
- Complemento
- Kleen star
- Operator+

NEXPTIME è chiuso rispetto a:

- Unione
- Intersezione
- Concatenazione
- Kleen star
- Operator+

Non è chiuso sotto complemento

2.14 Spazio Logaritmico Deterministico

2.14.1 Domanda 1

Descrivi l'architettura di una macchina che lavora in spazio Logaritmico
Risposta

Iniziamo a dire per quale motivo una macchina di turnig normale non potrebbe lavorare in spazio logaritmico.
Le macchine di turing che abbiamo sempre utilizzato inserivano l'input nel nastro iniziale. Per lavorarci, quindi, si occupava ugualmente uno spazio lineare rispetto all'input. Occuparne uno logaritmico in questo modo non è possibile.

Le macchine di turing che riescono a risolvere i problemi utilizzando uno spazio logaritmico vengono divise in tape differenti, che hanno proprio funzioni diverse: **INPUT TAPE e WORK TAPE**
Nell'input tape si tiene conto solamente dell'input, mentre nel work tape si eseguono le operazioni che permettono di risolvere il problema, utilizzando però uno spazio logaritmico rispetto all'input.

Una configurazione quindi terrà conto di entrambi i nastri, e sarà composta dallo stato, dalla stringa nell'input prima, stringa nell'input dopo, stringa del work prima e stringa del work dopo.

2.14.2 Domanda 2

Descrivi in modo informale una macchina di turing che lavora in logspace per controllare che una stringa sia palindroma Risposta

Da **Simone**:

Salvo la posizione iniziale della stringa e quella finale su un tape. Poi, ti salvi il carattere in prima posizione e scorri la stringa con un contatore e controlli l'ultimo carattere. Se l'ultimo è uguale al primo, allora continui. Aumenti di 1 il contatore e diminuisci di 1 la posizione dell'ultimo carattere. Si va avanti cosi. Posizione stringa, carattere su un tape, incrementi di 1 il contatore fino a quando non arrivi alla fine della stringa e lo controlli. Tutto questo fino a quando inizio e fine non coincidono

2.14.3 Domanda 3

Fornisci un nuovo task che potrebbe essere risolto da una macchina che lavora in logSpace Risposta

Un altro task banale potrebbe essere quello di controllare il massimo di una lista di elementi.

Salvandoti solamente il valore massimo ogni volta, e confrontandolo con il valore successivo, puoi risolvere il problema utilizzando uno spazio logaritmico.

2.14.4 Domanda 4

Definisci LOGSPACE e dici perche LOGSPACE $\subseteq$ PTIME
Risposta

La definizione di LOGSPACE intende l'insieme dei linguaggi tali per cui lo spazio utilizzato è logaritmico rispetto all'input, utilizzando il work tape.

$$LOGSPACE = SPACE(\mathcal{O}(log_2 n)) \tag{2.53}$$

Un linguaggio per appartenere a LOGSPACE deve rispettare 2 proprietà

- Deve poter essere computato da una macchina di turing che utilizza un input tape solamente per scorrere la stringa
- Deve utilizzare uno o più work tape per eseguire le operazioni, utilizzando uno spazio logaritmico rispetto all'input. Si possono utilizzare un numero maggiore di 1 di work tape, poiché considerando un numero fissati di nastri lo spazio rimane comunque $\mathcal{O}(log_2 n)$

Per dimostrare invece che LOGSPACE $\subseteq$ PTIME, dobbiamo dimostrare che una macchina che lavora in spazio logaritmico può eseguire un numero di passi che è al massimo polinomiale, senza considerare i loop.

Ora, per farlo, dobbiamo pensare a cosa significa il numero di passi massimo di una TM. Dobbiamo contare il numero di configurazioni massimo che la macchina può avere.

Il calcolo da fare è complesso, ma comprende:

- q = il numero di possibili stati
- w_{in}, u_{in} cioè il numero di possibili configurazioni nell'input tape, che sono n

- w_{work}, u_{work}, prendendo k come il numero di tape, il numero di combinazioni di stringhe sui work tape sono $|\Sigma|^{klog_2 n}$, cioè il numero di simboli elevato alla grandezza massima del tape. Dobbiamo poi moltiplicare questo per il numero massimo di posizioni, cioè la grandezza del tape che è $klog_2 n$

Cioè che esce furoi da questo è un calcolo che mostra il numero massimo di configurazioni raggiungibili è polinomiale. Con questo dimostriamo che il numero massimo di passi che possiamo fare è polinomiale, settando cosi un upper bound sul tempo. Per questo motivo LOGSPACE $\subseteq$ PTIME

2.14.5 Domanda 5

Spiega perché decide_DREACH può essere risolto con una TM che lavora in logpsace
Risposta

Dobbiamo prima di tutto definire DREACH. E' un problema che fornisce in input:

- Un grafo deterministico diretto
- due nodi u e v

Come viene definito il grafo deterministico diretto? Come un grafo in cui ogni nodo ha al massimo un arco uscente per ogni nodo. In questo modo il grafo viene considerato deterministico, poiché ha solamente una possibile uscita. Il problema è quello di dire se esiste un simple path, quindi un cammino senza cicli, tra u e v.

Possiamo risolvere questo problema in spazio logaritmico con una TM poiché possiamo segnare l'input come una stringa che contiene il numero totale di nodi e siccome è deterministico i nodi che sono collegati tra loro sono separati a 2 a 2 da un cancelleto.

In questo modo, siccome a noi interessa se esiste un cammino, possiamo esplorare il grafo tenendo traccia solamente dell'ultimo nodo visitato, senza dover tenere traccia di tutti i nodi visitati precedentemente. In questo modo lo spazio occuapto sarà logaritmico rispetto al numero di nodi. Facciamo attenzione a non andare in loop sapenod che se esiste un simple path, la lunghezza massima sarà $n-1$ cioè il numero di nodi meno il primo.

2.14.6 Domanda 6

Definire DREACH e dire perché $\in LOGSPACE$
Risposta

DREACH è un problema che fornisce in input:

- Un grafo deterministico diretto
- due nodi u e v

Possiamo definire quindi il problema come:

$$DREACH = \{< G, u, v > | G \text{ è un grafo deterministico diretto} \wedge \text{ esiste un simple path tra u e v in G}\} \tag{2.54}$$

Ora, per dimostrare che $DREACH \in LOGSPACE$ dobbiamo fornire una procedura che lavora in spazio logaritmico per risolvere il problema.

```
decide_DREACH(DDG G, node u, node v){
    curr = u
    n = |nodes(G)|
    for i ∈ {1, ..., −1}{
        for each w in nodes(G){
            if (curr, w) ∈ E(G){
                curr = w
                break
            }
            if (curr == v)
                return true
        }
    }
    return false
}
```

Questa procedura funzione nel seguente modo:

1. Parte dal nodo u
2. Prende la cardinalità dei nodi
3. Comincia a scorrere un indice fino a n-1, cioè il numero massimo di nodi
4. Poi, per ogni nodo che c'è in G controlla se esiste un arco tra curr e il nodo
   - Se esiste, si salva curr come w e si esce dal loop
   - Se non esiste, va avanti
5. Se curr è uguale a v, cioè il nodo di arrivo, allora ritorna true
6. Se non è uguale, va avanti
7. Quando arriva alla fine del loop, ritorna false

Questa procedura funziona poiché il grafo è deterministico e quindi non ci sono nodi che vengono visitati più volte. In questo modo, possiamo tenere traccia solamente dell'ultimo nodo visitato, senza dover tenere traccia di tutti i nodi visitati precedentemente. In questo modo lo spazio occuapto sarà logaritmico rispetto al numero di nodi.

2.14.7 Domanda 7

Spiegare perché DREACH $\in LOGSPACE - hard$
Risposta

Per dimostrare che DREACH $\in LOGSPACE-hard$ dobbiamo dimostrare che ogni problema in LOGSPACE può essere ridotto a DREACH.

Per farlo, possiamo costruire una reduction ρ tale per cui ogni linguaggio $L \in LOGSPACE$ può essere ridotto a DREACH. Prendendo un linguaggio $L \in LOGSPACE$, possiamo costruire un grafo in FAC^0

contenente tutte le configurazioni della macchina. Cioè, questo grafo contiene le configurazioni tali per cui se una stringa appartiene al linguaggio, il circuito avrà valore positivo. Negativo altrimenti. A questo punto basta che inseriamo due nodi **u,v** all'inizio e alla fine e controlliamo, il primo con un arco uscente verso la configurazione iniziale e il secondo avente un arco entrante dalla configurazione finale se esiste un simple path tra i due.

- Se esiste, cioè se $w \in L$, allora $\rho(w) \in DREACH$
- Se non esiste, cioè se $w \notin L$, allora $\rho(w) \notin DREACH$

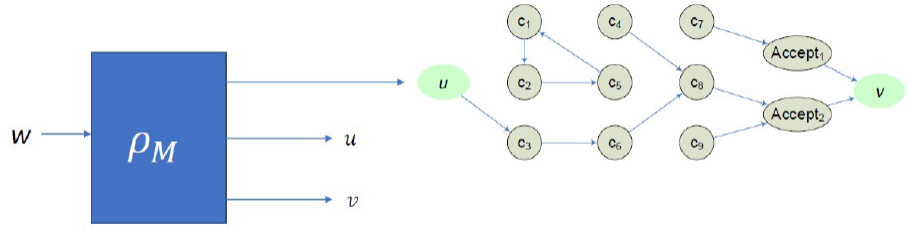


Figure 2.3: Esempio di dreach reduction

2.15 Spazio logaritmico non deterministico

2.15.1 Domanda 1

Definisci NLOGSPACE e parla della relazione con LOGSPACE
Risposta

Con NLOGSPACE indichiamo l'insieme dei linguaggi che possono essere riconosciuti da una macchina di turing non deterministica e che lavorano in spacio logaritmico: NSPACE $\mathcal{O}(log_2(n))$.

Anche in questo caso per appartenere a NLOGSPACE, la macchina di turing non deterministica deve:

- avere un nastro di input di dimensione n che è read only
- k nastri di lavoro chiamati work tape, con un numero di celle usate pari a $log_2(n)$

Cioè che lega NLOGSPACE a LOGSPACE è che $LOGSPACE \subseteq NLOGSPACE$. Questo è banale perché ogni macchina di turing deterministica è intrinsicamente una macchina di turing non deterministica. E' lo stesso motivo per cui $P \subseteq NP$.

2.15.2 Domanda 2

Dimostrare che NLOGSPACE $\subseteq$ PTIME
Risposta

Dobbiamo quindi dimostrare che una macchina di turing non deterministica che lavora in spazio logaritmico rispetto all'input può avere come tempi di esecuzione massimo un tempo polinomiale. Cioè stiamo dicendo che $timereq(M) \in \mathcal{O}(n^i)$. Questo lo dimostriamo nello stesso modo in cui abbiamo dimostrato che $LOGSPACE \subseteq PTIME$.

Il calcolo sarà sempre quello di avere: $c \circ n \circ |\Sigma|^{klog_2(n)} \circ klog_2(n)$, che sarebbe il numero diverso di configurazione possibili. Abbiamo che questo valore è $\leq n^p$ quindi abbiamo che $NLOGSPACE \subseteq PTIME$.

2.15.3 Domanda 3

Dimostrare che REACH $\in NLOGSPACE$
Risposta

Ricordiamo la definizione di REACH:

$$REACH = \{< G, u, v > | G \text{è un grafo orientato} \wedge u, v \in N(G) \wedge \text{ esiste un simple path da u a v in G}\} \tag{2.55}$$

Ora, in questo caso a differenza di DREACH non abbiamo un grafo deterministico, ma abbiamo un grafo che può avere più archi che escono da un nodo. Abbiamo perso il determinismo dai nodi.

Quindi, per risolvere il problema, dobbiamo usare una macchina di turing non deterministica che usando il guess and check lavora in spazio polinomiale per risolvere il problema.

Per farlo, dobbiamo guessare di volta in volta il nodo n-1 volte massimo, cioè —N(G)—-1. Perché se esiste un path, la lunghezza massima sarà proprio questa.

```
decide_REACH(DG G, node u, node v){
    curr = u
    n = |nodes(G)|

    for i ∈{1,...,n−1}{
        guessa un nodo w ∈ N(G)
        if (curr, w) ∈ E(G){
            curr = w
        }
        else{
            reject
        }
        if (curr == v){
            accept
        }
    }
    reject
}
```

2.15.4 Domanda 4

Discuti di come si potrebbe dimostrare che REACH $\in$ NLOGSPACE-hard
Risposta

Per dimostrare che REACH $\in$ NLOGSPACE-hard, dobbiamo dimostrare che ogni linguaggio in NLOGSPACE è riducibile a REACH. Quindi dobbiamo dimostrare che $\forall L \in NLOGSPACE, L \leq_F REACH$.

Pensiamola cosi. Abbiamo un linguaggio L che è in NLOGSPACE, quindi abbiamo una macchina di turing non deterministica che lavora in spazio logaritmico che lo decide. Possiamo costruire un circutio in FAC^0 che contiene tutte le configurazioni possibili della macchina tale per cui se w appartiene ad L, allora il circuito accetta, altrimenti no. A questo punto basta avere come per DREACH due nodi u,v tali per cui u ha un arco uscnete verso le configurazioni iniziali e v ha un arco entrante dalle configurazioni finali.

Se vediamo che esiste un path da u a v vuol dire che la stringa w apparteneva al linguaggio, e se fosse vero esisterebbe anche il path. Viceversa, se non fosse cosi allora non esisterebbe.

- se $w \in L, \exists$ path da u a v, allora $< G, u, v > \in REACH$
- se $w \notin L, \nexists$ path da u a v, allora $< G, u, v > \notin REACH$

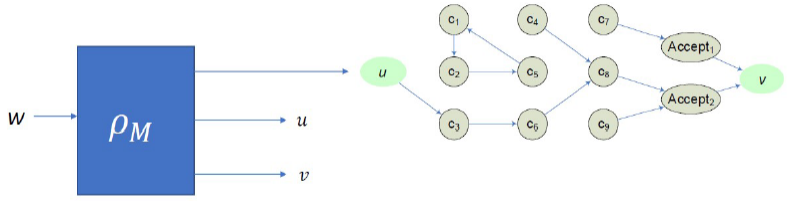


Figure 2.4: REACH

2.15.5 Domanda 5

Discuti di $\{< M, w > | M$ **è una NFSM** $\wedge M(w) = y\} \in LOGSPACE$
Risposta

Da fare

2.15.6 Domanda 6

$\{a^n b^n | n > 0\} \in LOGSPACE$
Risposta

2.15.7 Domanda 7

Discuti di $\{u\#v | \exists w1 \exists w2 : w1 \circ u \circ w2 = v\} \in LOGSPACE$

2.16 Valutazione di query congiunte

2.16.1 Domanda 1

Definire un database relazione
Risposta

Un database relazione è definito come un insieme finito di atomi ground.

Indichiamo con atomo ground un predicato p applicato ad un insieme di costanti $c_1, \ldots, c_n$.

Un atomo α è una formula $p(t_1, \ldots, t_n)$, dove p è un predicato e $t_i, 0 \leq i \leq n$ sono termini. Se i termini del predicato sono costanti, quindi valori che non cambiano, allora l'atomo è detto ground.

Alcuni esempi:

- predicato: $motherOf(x, y)$ x è madre di y
- atomo ground: $motherOf(Anna, Maria)$ Anna è madre di Maria, Anna e Maria sono costanti
- atomo non ground: $motherOf(x, Maria)$ Maria è figlia di x, Maria è una costante, x è una variabile
- costanti: $Anna, Maria$

2.16.2 Domanda 2

Definire una query congiunta
Risposta

Una query congiunta CQ è una query che contiene in output una variabile e ha all'interno dei quantificatori esistenziali per le variabili non in output. La variabile che sarà in output viene messa in congiunzione all'interno del predicato con le variabili che sono quantificate esistenzialmente. Un esempio:

$$q(x) = \exists y motherOf(x, y) \tag{2.56}$$

In queto caso la variabile x viene messa in congiunzione con la variabile y. Facciamo un esempio:

$$q(Anna) = \exists y motherOf(Anna, y) \tag{2.57}$$

In questo caso la query ci restituirà tutte le persone che sono figlie di Anna.

2.16.3 Domanda 3

Definire una query congiunta booleana
Risposta

Una query congiunta è booleana quando non ha nessuna varaibile specificata in output. Ad esempio:

$$q = \exists x eat(x, pizza) \tag{2.58}$$

In questo caso la query ci restituirà tutte le persone che mangiano la pizza.

A quanto pare bisogna parlare anche di homomorphismo. Un homomorphismo è una funzione cha mappa gli atomi di un insieme in un altro insieme. In questo caso, da una query ad un database.

Diciamo che q è una query congiunta e D è un database relazionale. Un homomorphismo da q a D è una funzione h che mappa le variabili di q in termini di D in modo che $h(q)$ sia un sottoinsieme di D.

Per dire che una query congiunta booleana è soddisfacibile, dobbiamo trovare un homomorphismo da q a D tale per cui $D \models q$, cioè D soddisfa q.

2.16.4 Domanda 4

Spiegare la semantica delle risposte delle query congiunte
Risposta

Praticamente dobbiamo considerare un insieme una query q(x) e una tupla t tale che $|t| = |x|.t \in D$. Otteniamo una BCQ q(t) che otteniamo andando a cambiare le variabili con le costanti della tupla t, rispettando l'ordine. Un esempio:

$$q(x) = \exists y motherOf(x, y) \tag{2.59}$$

$$t = \langle Anna, Maria \rangle \tag{2.60}$$

$$q(t) = \exists y motherOf(Anna, Maria) \tag{2.61}$$

La risposta alla query sarà un insieme del tipo:

$$ans(q, D) = \{t \in (terms(D)^{|x|}) | D \models q(t)\} \tag{2.62}$$

Cioè la risposte contiene un insieme di tuple di lunghezza t tali per cui il database soddisfa la query ottenuta cambiando il valore delle variabili con la tupla t.

2.16.5 Domanda 5

Definire la valutazione di una query congiunta booleana
Risposta

Non ho ben capito. Penso uguale a quello sopra.

2.16.6 Domanda 6

Spiega perché BCQ-EVAL $\in NP$
Risposta

Intanto diciamo che BCQ-EVAL è definito come:

$$BCQ - EVAL = \{< D, q >: D \text{ è un database} \wedge q \text{ è una query} \wedge D \models q\} \tag{2.63}$$

Per dimostrare che BCQ-EVAL $\in NP$ dobbiamo dimostrare che esiste un algoritmo che verifica se un input è accettato in tempo polinomiale. Per farlo, è abbastanza banale. Dobbiamo fornire una procedura che lavora in tempo polinomiale in modo non deterministico prima utilizza un guess per guessare un possibile omomorfismo e verifica se esso soddisfa la query. Ad esempio:

- guessiamo un omomorfismo h
- verifichiamo se $h(q) \subseteq D$
- se $h(q) \subseteq D$ allora accettiamo
- altrimenti rifiutiamo

Il numero totale di omomorfismi dipende dal numero di costanti e dal numero di variabili. Quindi il numero di omomorfismi è $O(|D|^{|x|})$. Essendo che il numero di omomorfismi è polinomiale, allora BCQ-EVAL $\in NP$.

2.16.7 Domanda 7

Spiega perché $BCQ - EVAL_q \in LOGSPACE$
Risposta

Per dimostrare che BCQ-EVAL $\in LOGSPACE$ dobbiamo dimostrare che esiste un algoritmo che verifica se un input è accettato in spazio logaritmico. In questo caso, la query è fissata. Siccome dobbiamo dimostrare che appartiene a LOGSPACE, dobbiamo fornire un algoritmo deterministico che utilizza spazio logaritmico.

Chapter 3

Domandone dell'esame

3.0.1 Domande persona 1

Parte A

1. Definizione di mapping reduction e mostra come costruire una Turing Reduction
2. Mostra formalmente perché la classe SD non è chiusa sotto complemento

Parte B

1. Definisci unique-sat e la classe DP. mostra formalmente perché unique-sat è in DP
2. Mostra formalmente che PTIME è chiuso sotto l'operatore +

3.0.2 Domande persona 2

Parte A

1. Classifica H_{any} e fornisci una procedura decidibile o semidecidibile per H_{any}, se esiste.
2. Dimostra perché $H \notin D$

Parte B

1. Definisci $REACH$ e fornisci una procedura in $NLOGSPACE$ per $REACH$
2. Definisci $3SAT - EQ$ e dimostra perché $NP - hard$

3.0.3 Domande persona 3

Parte A

1. Definisci i registri della macchina RAM
2. Parla dei linguaggi ricorsivamente separabili

Parte B

1. Definire com'è fatto un linguaggio NP-Complete e fare un esempio
2. Perché $LOGSPACE \subset PTIME$