Università di Perugia
Dipartimento di Matematica e Informatica

LAUREA MAGISTRALE IN INFORMATICA

Computability and Complexity

Prof. Arturo Carpi

Anno Accademico 2021-2022

Indice

1 Introduzione al corso **9**

1.1 Il decimo problema di Hilbert 9

1.1.1 Teoria della Computabilità 10

2 Cosa è un algoritmo **11**

2.1 Le regole di un algoritmo 11

2.2 Linguaggi Formali 12

2.2.1 Parole 12

2.2.2 Lunghezza 13

2.2.3 Concatenzione, fattori 13

2.2.4 Linguaggio Formale 14

2.2.5 Ordinamento 14

3 La Macchina di Turing **15**

3.0.1 Funzionamento 16

3.0.2 Programma 17

3.0.3 Definizione Formale 17

3.0.4 Configurazione istantanea 18

3.0.5 Configurazione consecutiva 19

3.0.6 Computazione 19

3.0.7 Algoritmi e macchina di Turing 21

3.0.8 L'impiegato diligente 21

3.0.9 Macchina di turing e linguaggi 22

3.0.10 Problemi decidibili 22

3.0.11 Aritmetizzazione della macchina di turing 23

3.0.12 Numerazione di i Gödel 24
3.0.13 Numerazione della macchina di turing 26
3.0.14 Coppie di interi 28
3.0.15 Descrizione delle macchine di turing 29
3.0.16 Tesi di Church-Turing 30
3.0.17 Funzioni e problemi 31

4 Problemi insolubili **32**
4.0.1 La macchina di turing Universale 32
4.0.2 La diagonalizzazione 33
4.0.3 Il problema dell'arresto 34
4.0.4 Il problema dell'arresto2 35
4.0.5 Riduzioni 35
4.0.6 Riduzioni-2 36
4.0.7 Equivalenza di macchine di Turing 37
4.0.8 Funzioni Totali 38
4.0.9 Funzione Nulla 39
4.0.10 Equivalenza di macchine di Turing 40

5 Problemi semi-decidibili **41**
5.0.1 Insiemi semi-decidibili 41
5.0.2 Enumerabilità effettiva 42
5.0.3 Insiemi semidecidibili e decidibili 43
5.0.4 Il 10 problema di Hilbert 44
5.0.5 Insiemi diofanteei 44
5.0.6 Insiemi Diofantei e insiemi semidecidibili 45
5.0.7 Prova della congettura di Davis 46

6 Auto-riferimento **47**
6.1 Autoriferimento 47
6.1.1 Teorema di ricursione di Turing 47
6.1.2 Teorema s-m-n 48
6.1.3 Dimostrazione del Teorema s-m-n 49
6.1.4 Dimostrazione del Teorema di Ricursione 50

6.1.5 Costruzione 51
6.1.6 Esempio 52
6.1.7 Teorema del punto fisso 53
6.1.8 Teorema di Radice 54
6.1.9 Macchina di Turing minimali 55

7 Il problema di corrispondenza di post **56**
7.0.1 PCP 56
7.0.2 Problema corrispondenza di post 57
7.0.3 Ridurre l'arresto MPCP 58
7.0.4 Costruzione dell'istanza 58
7.0.5 Ridurre MPCP a PCP 60
7.0.6 Matrici con angolo zero in alto a destra 61
7.0.7 Riduzione di PCP a Zero nell'angolo 62
7.0.8 CFG ambigue 63

8 Funzioni ricorsive **64**
8.0.1 Approcci alternativi alla nozione di calcolabilità 64
8.0.2 Funzioni ricorsive di base 65
8.0.3 Composizione e recursione 65
8.0.4 Funzioni ricorsive primitive 66
8.0.5 La funzione di Ackermann 67
8.0.6 Minimalizzaione 68
8.0.7 Composizione e recursione 68
8.0.8 Funzioni ricorsive parziali 69
8.0.9 Funzioni ricorsive parziali e calcolabilità 69
8.0.10 Dimostrazione 70
8.0.11 Dimostrazione-2 71

9 Teoria della complessità **72**
9.0.1 Costo della computazione 72
9.0.2 Teoria della complessità 75
9.0.3 Teoria della complessità2 75
9.0.4 Come misurare la complessità 76

9.0.5 Complessità temporale 78
9.0.6 La classe P 78
9.0.7 Tesi di Edmonds-Cook-Karp 79

10 La classe P **80**
10.0.1 La classe P 80
10.0.2 2Col 80
10.0.3 Algoritmo 81
10.0.4 SAT 81
10.0.5 2SAT 82
10.0.6 Ridurre 2COL a 2SAT 83
10.0.7 Equazioni lineari 84
10.0.8 Numeri primi 85
10.0.9 coP 85

11 La classe NP **86**
11.0.1 Macchine di Turing non deterministiche 86
11.0.2 Configurazioni consecutive 86
11.0.3 Computazioni 87
11.0.4 Complessità temporale per macchine di Turing non deterministiche 87
11.0.5 Macchine e linguaggi 88
11.0.6 La classe NP 88
11.0.7 SAT 89
11.0.8 3COL 89
11.0.9 Grafi 90
11.0.10 Grafi-2 90
11.0.11 Grafi-3 91
11.0.12 coNP 92
11.0.13 Un'altra definizione di Np 92
11.0.14 Il testimone 93
11.0.15 Dimostrazione 94

12 NP-completezza **96**
12.0.1 P = NP? . . . 96
12.0.2 Riduzioni polinomiali . . . 97
12.0.3 Problemi NP-ardui . . . 98
12.0.4 Un problema NP-completo . . . 98
12.0.5 Il teorema di Cook-Levin . . . 99
12.1 Il Teorema di Cook-Levin - Approfondito . . . 101
12.1.1 Riduzione di S e SAT . . . 106
12.1.2 Variabili . . . 106
12.1.3 Clausole . . . 106
12.2 Altre clausole . . . 108
12.2.1 Ancora clausole . . . 109
12.2.2 Conclusione . . . 109

13 Problemi NP-completi **110**
13.0.1 3SAT . . . 110
13.0.2 Insieme indipendente . . . 111
13.0.3 Ridurre 3SAT a IS . . . 112
13.0.4 CLIQUE e VC . . . 112
13.0.5 3COL . . . 112
13.0.6 Ridurre 3SAT a 3COL . . . 113
13.0.7 Ridurre 3SAT a 3COL . . . 114
13.0.8 Problemi NP-intermedi . . . 115
13.0.9 Problemi NP-intermedi . . . 115
13.0.10 Congettura di Berman-Hartmanis . . . 116

14 Il criptosistema Merkle-Hellman **117**
14.1 Il problema dello zaino (KNAPSACK . . . 117
14.1.1 Crittografia . . . 119
14.1.2 Successioni supercrescenti . . . 120
14.1.3 Creazione delle chavi . . . 121
14.1.4 Decodifica . . . 122
14.1.5 La classe UP . . . 123

15 La gerarchia polinomiale **124**
15.0.1 NP e coNP 124
15.0.2 $\sum_2^p \prod_2^P$ 127
15.0.3 $\sum_k^p \prod_k^P$ 128
15.0.4 Osservazioni 128
15.0.5 Gerarchia Polinomiale 129
15.0.6 Tempi Esponenziali 130

16 La complessità spaziale **131**
16.0.1 Macchina di turing a piu' strati 131
16.0.2 Equivalenza tra modelli 133
16.0.3 Dimostrazione 134
16.0.4 Complessità spaziale 134
16.0.5 Il modello 134
16.0.6 Il modello non deterministico 135
16.0.7 Dimostrazione 135
16.0.8 Classi di complessità spaziale 136
16.0.9 Classi non deterministiche 137
16.0.10 Il grafo delle computazioni 137

17 Problemi NL-completi **138**
17.0.1 Il grafo delle computazioni 138
17.0.2 Un'applicazione 139
17.0.3 L=NL? 139
17.0.4 Riduzioni in spazio logaritmico 140
17.0.5 Dimostrazione 140
17.0.6 La relazione <= log 140
17.0.7 Dimostrazione 141
17.0.8 NL-completezza 142
17.0.9 Gap 142
17.0.10 GAP è NL-arduo 143

18 Teorema di Savitch **144**
18.0.1 Teorema di Savitch 144

18.0.2 Gap 144
18.0.3 Analisi 146
18.0.4 Conclusione 146
18.0.5 Il caso NL 146
18.0.6 Il caso generale 146
18.0.7 Il complemento di GAP 147
18.0.8 Vertici j-accessibili 147
18.0.9 Procedura Accesso 148
18.0.10 Calcolo di r 148
18.0.11 Accettare il complemento di GAP 149
18.0.12 NL = coNL 149
18.1 NL = coNL Approfondito 150

19 La somma di sottoinsiemi è NP-completa **154**
19.1 La somma di sottoinsiemi è NP-completa 154

Capitolo 1

Introduzione al corso

1.1 Il decimo problema di Hilbert

Data un'equazione *Diofantea* (polinomio con coefficienti interi posta uguale a 0) con qualsiasi numero di quantità incognite e con coefficienti numerici razionali interi, individuare un procedimento mediante il quale si possa determinare in un numero finito di operazioni se l'equazione è risolubile in razionali interi (Hilbert, Int. Congress of Mathematicians, Sorbona (Parigi), 8/8/1900).

In termini moderni: determinare un algoritmo per sapere se un'equazione Diofantea è risolubile. Non ci sono limiti al numero delle variabili ecc... , si possono quindi scrivere infinite equazioni Diofantee.

$$\begin{array}{ll} 2x + 4y - 3 = 0 & z = x + y \\ 2z + 2y - 3 = 0 & w = z + y \\ 2w - 3 = 0 & \end{array}$$

nessuna soluzione

$$\begin{array}{ll} 2x + 3y - 8 = 0 & z = x + y \\ 2z + y - 8 = 0 & w = z + y \\ w + z - 8 = 0 & t = w + z \\ t - 8 = 0 & \end{array}$$

ci sono soluzioni

e.g.: $t = 8,\ w = 5,\ z = 3,\ y = 2,\ x = -1$

1.1.1 Teoria della Computabilità

E se il 10th problema di Hilbert non avesse soluzione?
Dobbiamo rispondere alle domande:

1. quali sono le funzioni per cui esiste un procedimento di calcolo effettivo?
2. e quali quelle per cui un tale procedimento non esiste?

Ma per rispondere alla domanda 2, è necessaria una **definizione formale di procedimento di calcolo effettivo**.

Millennium prize problems (Millennium meeting Collège de France (Parigi), 24/5/2000)
Il Clay Math. Institute (Boston) mette in palio 1.000.000$ per chi risolve uno dei 7 problemi matematici più difficili.
Tra questi: **P vs. NP**

Capitolo 2

Cosa è un algoritmo

2.1 Le regole di un algoritmo

1. Un algoritmo è di lunghezza finita.
2. Esiste un agente di calcolo (la macchina calcolatrice, appunto) che sviluppa la computazione eseguendo le istruzioni dell' algoritmo.
3. L'agente di calcolo ha a disposizione una memoria, dove vengono immagazzinati i risultati intermedi del calcolo.
4. Il calcolo avviene per passi discreti.
5. Il calcolo non è probabilistico.
6. Non deve esserci alcun limite finito alla lunghezza dei dati di ingresso.
7. Non deve esserci alcun limite alla quantità di memoria disponibile.
8. Deve esserci un limite finito alla complessità delle istruzioni eseguibili dal dispositivo.
9. Il numero di passi della computazione è finito ma non limitato.
10. Sono ammesse computazioni senza fine.

2.2 Linguaggi Formali

Ogni informazione può essere rappresentata da una sequenza finita di simboli su un opportuno alfabeto finito.

Esempio.
La Divina Commedia, il genoma umano, i numeri interi e quelli razionali (per es. in base 10), grafi, polinomi, ecc... Possiamo quindi assumere che i processi di calcolo riguardino stringhe di simboli scelti in un insieme finito e non vuoto.

2.2.1 Parole

Definizione
Chiameremo alfabeto un insieme finito non vuoto A di simboli. I suoi elementi sono detti lettere.

Esempi:
$\{a, b\}, \{0, 1\}, \{a, b, c\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.

Definizione
Ogni sequenza finita di lettere di A si dice parola sull'alfabeto A. L'insieme delle parole sull'alfabeto A sarà denotato con A*.

Esempi
a, abb, ababbabb sono parole sull'alfabeto {a, b}.
01001010, 0110, 0000 sono parole sull'alfabeto {0, 1}.

Possiamo anche considerare la sequenza priva di lettere, che si denota con $\wedge$ (oppure ε) e si dice parola vuota.

2.2.2 Lunghezza

Una parola sull'alfabeto A è una sequenza u = a1a2...ak con k >= 0, a1, a2, . . . , ak ∈ A.

Definizione
L'intero k si dice lunghezza della parola u e si denota con l(u).

Esempi
l(a) = 1, l(aab) = 3, l(ababbabb) = 8, l(∧) = 0.

2.2.3 Concatenzione, fattori

Definizione
Si considerino le parole u = a1a2...ak e v = b1b2...bh con k, h >= 0
a1, a2, . . . , ak , b1, b2, ... ,bh ∈ A.
La concatenazione di u e v è **la parola uv**:

uv = a1a2...ak b1b2 ... bh.

Esempi
La concatenazione delle parole abb e aaab è la parola abbaaab.
La concatenazione delle parole aaab e abb è la parola aaababb.
La concatenazione delle parole baa e ∧ è la parola baa.

Definizione
Diremo che una parola v è un **fattore** di una parola w se risulta w = xvy.
Per opportune parole x , y. Nel caso in cui x = ε (risp., y = ∧) il fattore v si dice prefisso (**risp.**, **suffisso**) di w.
Diremo che v è un fattore proprio se v ≠ w.

Esempio
I fattori di abb sono ∧, a, b, ab, bb e abb. Dove i prefissi sono ∧, a, ab e abb ed i suffissi sono ∧, b, bb e abb.

2.2.4 Linguaggio Formale

Definizione
Ogni sottoinsieme di A* si dice linguaggio formale o linguaggio o problema sull'alfabeto A.

Esempio
Sono linguaggi formali sull'alfabeto A = {a, b}:
L0 = {a, b} , L1 = {a, ab, abb} , L2 = {abna | n >= 0} , L3 = $\emptyset$, L4 = A*,
L5 = {a^p | p primo}
Le espansioni binarie dei numeri primi costituiscono un linguaggio sull'alfabeto {0,1}.
L'insieme dei polinomi Diofantei nelle variabili x , x', x'', x''', . . . che ammettono una radice intera possono essere rappresentati come un linguaggio sull'alfabeto

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \hat{}, x, "\}$$

2.2.5 Ordinamento

Definizione
Sia A un alfabeto totalmente ordinato. L'ordine radicale (o militare) su A* è definito come segue:
siano u, v $\in$ A*.
Si ha u < v se è soddisfatta una delle due condizioni seguenti:

- l(u) < l(v)
- l(u) = l(v) e u precede v nell'ordine lessicografico

  (cioè u = ras, v = rbt con r,s,t $\in$ A* a, b $\in$ A e a < b nell'ordinamento dell'alfabeto A)

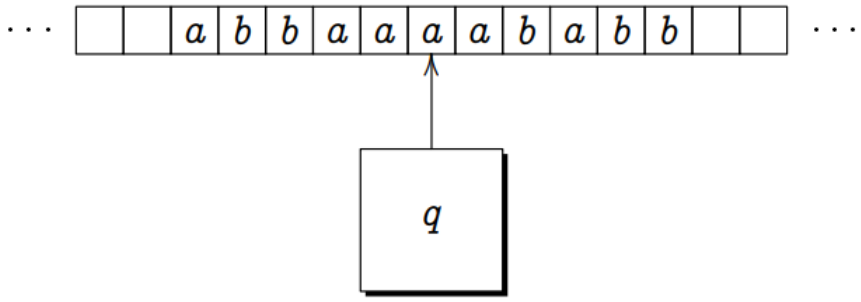
Esempio
I numeri naturali sono rappresentati in base 10 da parole sull'alfabeto:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \text{ prive di 0 iniziali.}$$

L'ordinamento usuale dei numeri naturali coincide con ordine radicale delle corrispondenti espansioni decimali.

Capitolo 3

La Macchina di Turing



E' composta da:

- un unità di controllo a stati finiti che
  - può spostare il nastro a destra o a sinistra (una cella alla volta)
  - contiene il programma secondo cui verrà eseguito il calcolo
  - registra lo stato della macchina
  - L'insieme di possibili stati della macchina è un insieme finito Q = {q0, q1 ,. . . qn}

- un nastro di lunghezza infinita che
  - è suddiviso in celle
  - ogni cella può contenere un solo simbolo di un fissato alfabeto A, oppure essere bianca.
  - solo un numero finito di celle contiene una lettera di A.
- Testina di lettura e scrittura
  - che permette all'unità di controllo di leggere e scrivere un simbolo per volta dal nastro

3.0.1 Funzionamento

All'avvio di ogni computazione, la macchina si trova in uno stato iniziale prefissato. A ogni passo l'unità di controllo in funzione dello stato in cui si trova e del simbolo contenuto nella cella che la testina indica:

- Rivede il suo stato.
- Scrive un simbolo nella cella indicata dalla testina, sostituendo il simbolo esistente (eventualmente bianco);
- Sposta la testina di una posizione a sinistra o a destra.

Il nuovo stato assunto dall'unità di controllo, il simbolo da scrivere sulla cella indicata dalla testina e lo spostamento della testina a sinistra o a destra sono determinati dal programma della macchina.

3.0.2 Programma

Il programma di una macchina di Turing si può rappresentare come una lista di 5-ple (istruzioni):

$$(q, a, q', a', x)$$

- q è lo stato dell'unità di controllo
- a l simbolo nella cella indicata dalla testina
- q' l nuovo stato che la macchina deve assumere
- a' la lettera da scrivere nella cella esaminata
- x è la testina.

  Se $x = -1$ spostamento della testina a sx, $x = +1$ spostamento della testina a dx

Per ogni coppia (q, a) ci deve essere al più una 5-pla (q, a, q', a', x) nel programma della macchina **(determinismo)**

3.0.3 Definizione Formale

Definizione
Una macchina di Turing M è una quadrupla:

$$M =< Q, A, \delta, q_0 >$$

- Q è un insieme finito di stati
- A è un alfabeto a cui si aggiunge il simbolo *bianco* #
- δ è una funzione **parziale** da $Q \times (A \cup \{\#\})$ a
  $Q \times (A \cup \{\#\}) \times \{-1, +1\}$
- $q_0 \in Q$ è lo stato iniziale

Le 5-ple (q, a, q', a', x) tali che δ(q,a) = (q', a', x) sono dette **istruzioni** di M.

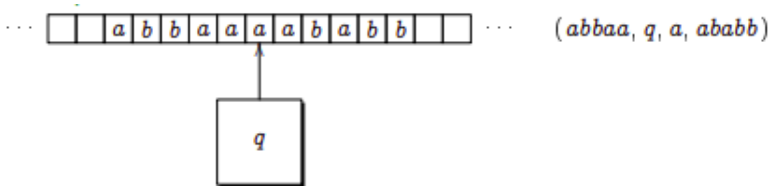
3.0.4 Configurazione istantanea

Lo stato di una Macchina di Turing a un dato istante può essere descritta da una 4-pla:

$$C_i = (\xi, q, a, \eta)$$

- ξ (xi) è il contenuto del nastro a sinistra della cella indicata dalla testina, privato della sequenza infinita di celle bianche che precedono l'ultimo simbolo non bianco. : ξ contiene tutte le lettere a sinistra della testina fino alle infinite celle bianche (il nastro è infinito sia a destra che a sinistra)
- q è lo stato della Macchina di Turing nell'istante considerato
- a è la lettera nella cella indicata dalla testina
- η (eta) è il contenuto del nastro a destra della cella indicata dalla testina, privato della sequenza infinita di celle bianche che seguono l'ultimo simbolo non bianco. : è come ξ ma a destra

Esempio



3.0.5 Configurazione consecutiva

Una configurazione istantanea di una Macchina di Turing

$$M =< Q, A, \delta, q_0 >$$

è un elemento dell'insieme (A U {#})* × Q × (A U {#}) × (A U {#})*.
Nell'insieme delle configurazioni istantanee di M introduciamo la relazione binaria $\vdash_M$ che associa alla configurazione C quella che la segue nella computazione di M. Qualora tale configurazione non esistesse, scriveremo $\vdash /_M$.

Esempio
Se M ha l'istruzione (q, a, q, b, -1), allora:

$$(\text{abbaa, q, a, ababb}) \vdash_M (\text{abba, } q', \text{ a, bababb})$$

3.0.6 Computazione

- Al passo iniziale
  - M esamina una parola w scritta sul nastro di input.
  - La testina indica la prima lettera di w.
  - Lo stato è lo stato iniziale q0.
  - la configurazione istantanea iniziale è ($\wedge$, q0, a, w'), ove w = aw' con a lettera
- Se all'istante i, M si trova nella configurazione C_i e $C_i \vdash_M C_i{+}1$, allora all'istante i+1 M sarà nella configurazione $C_i{+}1$.
- Se invece $C_i \vdash /_M$, allora la macchina si arresta.
- Se M si arresta dopo un numero finito di passi diremo che M converge sull'input w e scriveremo M $\downarrow$ w, in caso contrario diremo che M diverge sull'input w e scriveremo M $\uparrow$ w.

Esempio: bitwise not

	0	1	#
q_0	$(q_0, 1, +1)$	$(q_0, 0, +1)$	

Esempio: palindrome

	a	b	#	
q_0	$(q_a, \#, +1)$	$(q_b, \#, +1)$	$(q_2, Y, +1)$	memorizza e cancella la 1. lettera
q_a	$(q_a, a, +1)$	$(q_a, b, +1)$	$(q_a', \#, -1)$	va alla fine della parola
q_b	$(q_b, a, +1)$	$(q_b, b, +1)$	$(q_b', \#, -1)$	
q_a'	$(q_1, \#, -1)$	$(q_2, N, -1)$	$(q_2, Y, +1)$	cancella ultima lettera se uguale,
q_b'	$(q_2, N, -1)$	$(q_1, \#, -1)$	$(q_2, Y, +1)$	altrimenti scrive N (fail)
q_1	$(q_1, a, -1)$	$(q_1, b, -1)$	$(q_0, \#, +1)$	torna a inizio parola
q_2	$(q_2, \#, -1)$	$(q_2, \#, -1)$		cancella il nastro e si arresta

3.0.7 Algoritmi e macchina di Turing

Un algoritmo è di lunghezza finita	n. finito di istruzioni
Esiste un agente di calcolo che sviluppa la computazione eseguendo le istruzioni dell' algoritmo	la macchina di Turing
L'agente di calcolo ha a disposizione una memoria, dove vengono immagazzinati i risultati intermedi del calcolo	il nastro
Il calcolo avviene per passi discreti	✓
Il calcolo non è probabilistico	✓
Nessun limite finito alla lunghezza dei dati di ingresso	nastro infinito
Nessun limite alla quantità di memoria	nastro infinito
Limite finito alla complessità delle istruzioni eseguibili dal dispositivo	solo 5-uple
Numero di passi della computazione finito ma non limitato	✓
Sono ammesse computazioni senza fine	✓

3.0.8 L'impiegato diligente

La macchina di turing è un Meccanismo che simuli le potenzialità computazionali dell'uomo comune quindi:

- Esegue ordinatamente le sue istruzioni, purché non siano tra loro in concorrenza.
- In assenza di istruzioni si arresta.

3.0.9 Macchina di turing e linguaggi

Come detto in precendeza:

- La configurazione istantanea iniziale è C0 = ($\wedge$, q0, a, w'), ove w = aw' con a lettera.
- Se $C_0 \vdash_M C_1 \vdash_M ... \vdash_M C_n \vdash /_M$ M allora M **converge** sull'input w.
- Se $C_0 \vdash_M C_1 \vdash_M ... \vdash_M C_n \vdash_M$ M allora M **diverge** sull'input w.

Definizioni
Sia L un linguaggio sull'alfabeto A.

1. Una macchina di Turing M accetta il linguaggio L se M **converge su tutti gli input** w $\in$ L e **diverge** su tutti gli input w $\notin$ L.
2. Una macchina di Turing M decide il linguaggio L se M **converge su tutti gli input** w $\in$ A* con output SI se w $\in$ L e NO se w $\notin$ L.

3.0.10 Problemi decidibili

Le nozioni di linguaggio accettato e deciso sono profondamente diverse.

- Entrambe definiscono correttamente un linguaggio.
- Solo la nozione di linguaggio deciso ci fornisce una procedura effettiva che, in un numero finito di passi, ci permette di dire se una parola appartiene o meno al linguaggio.

Definizione
Un linguaggio L si dice decidibile secondo Turing se esiste una macchina di Turing M che decide L, indecidibile (secondo Turing) altrimenti.
Un linguaggio L si dice semidecidibile secondo Turing se esiste una macchina di Turing M che accetta L.

Proposizione
Un linguaggio decidibile è anche semidecidibile

Dimostrazione

Trasformare la macchina che decide L in una che accetta L, aggiungendo un loop infinito se la macchina originale si arresta sul NO.

Esempio

Rappresentiamo ogni numero intero n con la parola $\underbrace{111\cdots1}_{n+1}$

Costruiamo una macchina di Turing che decide (le codifiche dei) numeri pari

Codifichiamo SI con la parola 11 (cioè l'intero 1) e NO con la parola 1 (cioè l'intero 0)

	1	#	
q_0	$(q_1, \#, +1)$	$(q_2, 1, +1)$	cancella gli 1, alternando gli stati
q_1	$(q_0, \#, +1)$	$(q_0, 1, +1)$	al termine scrive uno o due volte 1
q_2			

Per accettare il medesimo linguaggio modificchiamo la macchina:

	1	#
q_0	$(q_1, \#, +1)$	$(q_0, \#, +1)$
q_1	$(q_0, \#, +1)$	

3.0.11 Aritmetizzazione della macchina di turing

Obbietivo

- Associare un numero a ogni input e output di una macchina di Turing.
- Associare un numero (indice) a ogni macchina di Turing.
- In maniera effettiva:
  - Data una macchina di Turing, ne posso calcolare l'indice.
  - Dato un indice, posso calcolare le istruzioni della macchina corrispondente.

Una prima strategia

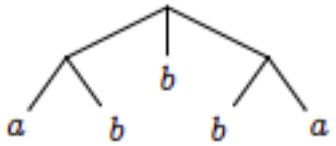
- Ordinare le parole nell'ordine militare.
- Associare a ogni parola la sua posizione nell'elenco ordinato.

  Oppure

- Interpretare ogni lettera come una cifra $\neq 0$ di un sistema numerico.
- Associare a ogni parola il numero corrispondente.

3.0.12 Numerazione di i Gödel

Metodo per numerare gli alberi (con foglie etichettate).



- A ogni etichetta x associo un numero dispari $G_0(x) >= 3$
- A un albero T di altezza 1, con foglie etichettate x_1, x_2, ... , x_k associo il numero

$$G_1(T) = 2^{G_0(x_1)} \cdot 3^{G_0(x_2)} \cdot 5^{G_0(x_3)} \cdot ... \cdot P_n^{G_0(x_k)}$$

  dove 2, 3, ..., p_k è la sequenza dei primi k numeri primi

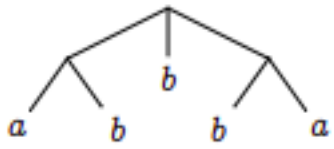
- In generale, se la radice dell'albero T ha, da sinistra a destra, i figli v_1, . . . , v_t, radici di sottoalberi alberi con numeri di Gödel n_1, . . . , n_t, allora il numero di Gödel di T è $2^{n1}\ 3^{n2}...p_t^{ni}$

Esempio

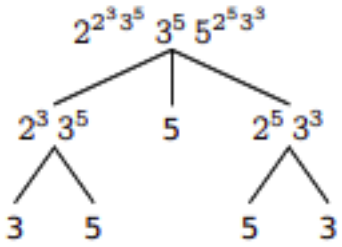
Posso anche fare il percorso inverso:

- Un numero dispari rappresenta una foglia.
- Un numero pari va decomposto come prodotto di potenze di primi.
- Gli esponenti sono i numeri di Gödel dei sottoalberi

Posso quindi distinguere gli interi che non sono numeri di Gödel.



Se $G_0(a) = 3$, $G_0(b) = 5$, allora



Il Teorema Fondamentale dell'Aritmetica assicura che la decodifica è unica.

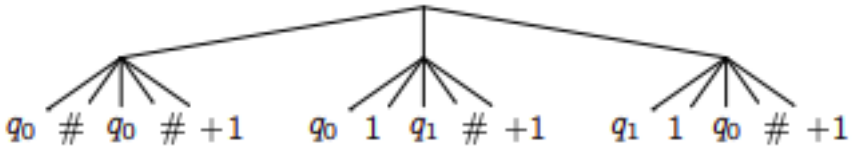
3.0.13 Numerazione della macchina di turing

Una macchina di Turing è rappresentata da una lista di 5-ple e, in definitiva, da un albero.

Esempio
La macchina che accetta i numeri pari ha le 5-ple (in ordine lessicografico).

$$(q_0, \#, q_0, \#, +1),(q_0, 1, q_1, \#, +1),(q_1, 1, q_0, \#, +1)$$

Possiamo identificarlo come albero:



Per numerare le macchine di Turing su un alfabeto A,
associamo: un intero dispari > 1 a ognuno dei simboli necessari.

-1	$+1$	$\#$	a_1,	a_2	$\cdots$	a_k	q_0	q_1	$\cdots$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	$\downarrow$	
3	5	7	9	11	$\cdots$	$2k+7$	$2k+9$	$2k+11$	$\cdots$

Per ogni macchina di Turing M:

- Ordiniamo le 5-ple in ordine lessicografico.
- Calcoliamo il numero di Gödel dell'albero corrispondente.

Otteniamo una funzione totale e iniettiva, effettivamente calcolabile $f : \mathcal{M}_A \to \mathbb{N}$, ove $\mathcal{M}_A$ denota l'insieme delle macchine di Turing sull'alfabeto A.

Inoltre una procedura effettiva permette, per ogni $n \in \mathbb{N}$, di sapere se $n \in f(\mathcal{M}_A)$ e, in tal caso, di calcolare il programma della macchina $M = f^{-1}(n)$

Una biezione $g : \mathcal{M}_A \to \mathbb{N}$ si ottiene nel modo seguente: per ogni $M \in \mathcal{M}_A$, se $f(M)$ è l'n-esimo elemento di $f(M_A)$, $n \geq 1$, allora poniamo

$$g(M) = n - 1.$$

Osservazione

Una procedura effettiva per calcolare x = g(M) è la seguente:

$x \leftarrow 0$
for $n \leftarrow 0, \ldots, f(M) - 1$
    if $n \in f(\mathcal{M}_A)$
        $x \leftarrow x + 1$

Teorema

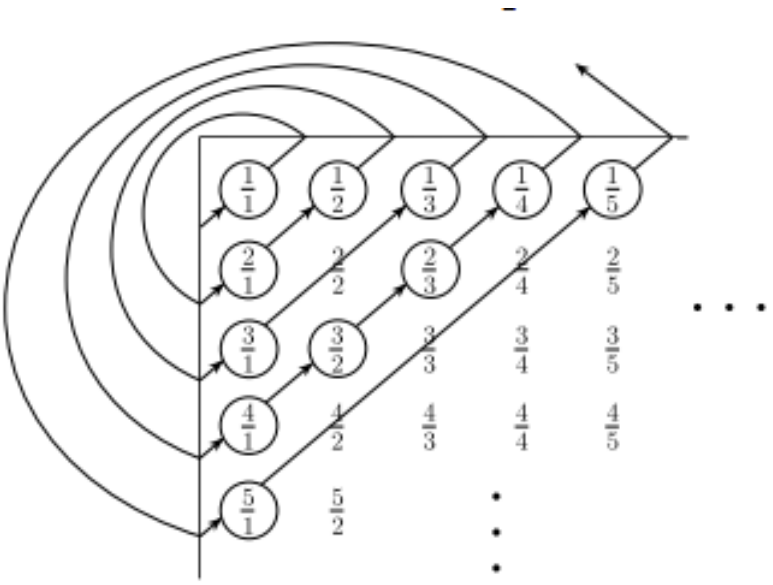
C'è una corrispondenza biunivoca effettiva tra le macchine di Turing su un alfabeto A e i numeri naturali.

3.0.14 Coppie di interi

Teorema di Cantor: Esiste una biiezione effettiva tra N^2 ed N data da:

$$(n, m) = \frac{(n + m)(n + m + 1)}{2} + n$$

Basta osservare che ogni naturale compare infinite volte come ascissa di una coppia $(n, m) \in N^2$ al variare di m, intuitivamente quindi ci sono tante coppie ordinate di naturali quanti naturali.



Corollario

C'è una corrispondenza biunivoca effettiva tra le macchine di Turing su qualsiasi alfabeto e i numeri naturali. Per il teorema precedente possiamo enumerare le macchine di Turing

$$\mathrm{M}_0,\mathrm{M}_1,\ldots,\ \mathrm{M}_n$$

Pertanto, è possibile enumerare tutte le funzioni calcolabili

$$\phi_0,\phi_1,\ldots,\phi_n$$

Nel secondo caso, però, la corrispondenza non è biunivoca, perchè la medesima funzione può essere calcolata da più macchine di Turing.

3.0.15 Descrizione delle macchine di turing

Descrizione formale
Lista delle quintuple

Descrizione implementativa
Descrive in linguaggio corrente, il modo in cui la macchina di Turing muove la testina e registra i dati sul nastro

Descrizione di alto livello
Descrive un algoritmo, ignorando i dettagli implementativi.

Esempio: Palindrome

Descrizione implementativa su input w

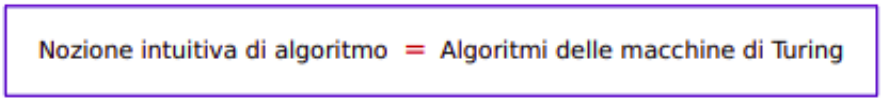
1. cancella la prima lettera
2. se non ci sono altre lettere accetta
3. sposta la testina alla fine della parola
4. se l'ultima lettera è diversa dalla prima rifiuta
5. cancella l'ultima lettera
6. se non ci sono altre lettere accetta
7. sposta la testina all'inizio della parola
8. ricomincia dal passo 1

Descrizione di alto livello su input w

1. se l(w) <= 1, accetta
2. se la prima e l'ultima lettera di w sono distinte, rifiuta
3. cancella la prima e l'ultima lettera di w
4. ricomincia al passo 1

3.0.16 Tesi di Church-Turing

Una funzione è calcolabile se e solo se esiste una macchina di Turing che la calcola.
Argomenti



- L'inclusione $\supseteq$ è evidente
- Assenza di controesempi
- Tecniche molto generali
- Numerosi modelli equivalenti
- Numerose generalizzazioni rivelatesi equivalenti
- La metafora dell'impiegato diligente

Convenzione
D'ora in poi considereremo solo insiemi di numeri naturali e funzioni di N in N (ipotesi non restrittiva). Le macchine di Turing si enumerano

$$M_0, M_1, \ldots, M_n$$

e le funzioni da esse calcolate saranno denotate, rispettivamente

$$\phi_0, \phi_1, \ldots, \phi_n$$

Osservazioni

- Ogni macchina di Turing calcola una funzione (parziale) di N in N
- Ogni macchina di Turing calcola una funzione (parziale) di N^2 in N
- Esiste una procedura effettiva che, data una macchina di Turing M, calcola l'indice x tale che $M = M_x$
- Esiste una procedura effettiva che, dato un numero naturale x , calcola il programma della macchina di Turing M_x

3.0.17 Funzioni e problemi

La funzione caratteristica di un insieme L $\subseteq N$ è la funzione f_L definita da: **Lemma**

$$f_L(x) = \begin{cases} 1 & \text{se } x \in L, \\ 0 & \text{altrimenti.} \end{cases}$$

Un insieme L $\subseteq N$ è decidibile se e solo se la sua funzione caratteristica è computabile.

Corollario: Esistono insiemi indecidibili

Dimostrazione
Per il Teorema di Cantor, non ci sono 'abbastanza' macchine di Turing.

Esempio

La seguente funzione è calcolabile:

$$g(x) = \begin{cases} 1 & \text{se nell'espansione decimale di } \pi \\ & \text{esistono almeno } x \text{ 5 consecutivi,} \\ 0 & \text{altrimenti.} \end{cases}$$

Invero, se il massimo numero di 5 consecutivi nell'espansione decimale di π è k, allora

$$g(x) = \begin{cases} 1 & \text{se } x \leq k, \\ 0 & \text{altrimenti.} \end{cases}$$

Se invece tale massimo non esiste, allora

$$g(x) = 1, \quad \text{per ogni } x \geq 0.$$

Non è noto se la seguente funzione sia calcolabile:

$$g(x) = \begin{cases} 1 & \text{se nell'espansione decimale di } \pi \text{ c'è una} \\ & \text{sequenza di esattamente } x \text{ 5 consecutivi,} \\ 0 & \text{altrimenti.} \end{cases}$$

Capitolo 4

Problemi insolubili

4.0.1 La macchina di turing Universale

Teorema
La funzione g : $N^2 \mapsto N$ definita ponendo, per ogni x , y $\in N$

$$g(x,y) = \phi_x(y)$$

è calcolabile secondo turing.

Dimostrazione Su un input (x,y) $\in N^2$

1. Decodifica le istruzioni di M_x
2. Simula M_x sull'input y
3. Se M_x si arresta, restituisci l'output della computazione

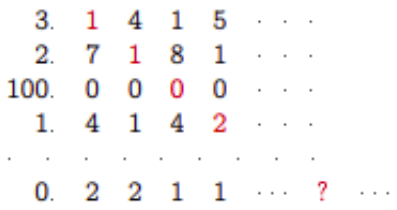
Definizione
La macchina di Turing μ che calcola la funzione g si dice Macchina di Turing universale.
La macchina di Turing universale ha la capacità di eseguire qualunque algoritmo.
Calcolatore all purpose (modello di Von Neumann).

4.0.2 La diagonalizzazione

Teorema L'insieme R non è numerabile.

Dimostrazione

Per assurdo. Se R fosse numerabile, avremmo una tabella infinita con tutti i numeri reali:



Costruiamo un numero reale x = 0,c_1,c_2,c_3 ... in cui la i-esima cifra decimale è diversa dalla i-esima cifra rossa (e anche da 0 e 9).

Per esempio: 0.2211 ...

Dove può comparire nella nostra tabella?

4.0.3 Il problema dell'arresto

Data una macchina di Turing M e un input y, decidere se M si arresta sull'input y.

$$\mathrm{H} = \{(\mathrm{x},\mathrm{y}) \in N^2 \mid \mathrm{M}_x \downarrow \mathrm{y}\}$$

	0	1	2	3	· · ·		
M_0	↓	↓	↑	↑	· · ·		
M_1	↓	↑	↑	↓	· · ·		
M_2	↑	↑	↓	↑	· · ·		
M_3	↓	↓	↑	↓	· · ·		
· ·	·	·	·	·	· · ·	·	·
M	↑	↓	↑	↑	· · ·	?	· · ·

Se sapessi risolvere il problema dell'arresto, potrei costruire una macchina di Turing M che su ciascun input x si arresta se e solo se l'elemento x-esimo della diagonale è ↑.
Come prima, tale macchina non può comparire nella nostra tabella!

Teorema

Il problema dell'arresto non è decidibile.

Dimostrazione

Per assurdo supponiamo che H sia decidibile.
Allora anche il problema:

$$\mathrm{K} = \{\ \mathrm{x} \in \mathrm{N} \mid (\mathrm{x}\ ,\ \mathrm{x}\ ) \in \mathrm{H}\}$$

è decidibile.
Invero, sia M una macchina di Turing che decide H. Il problema H è deciso dalla macchina M′ che esegue il seguente programma:
Su input x $\in N$

1. simula M su (x , x)
2. restituisci l'output della computazione

4.0.4 Il problema dell'arresto2

Anche il complemento di K è decidibile e, di conseguenza semidecidibile. Più precisamente, K è accettato dalla macchina M" che esegue il seguente programma.
Su input x $\in N$

1. Simula M su (x , x)
2. Se l'output è NO, accetta; se l'output è SI, eseguire un loop infinito

Sia z l'indice della macchina M". Allora

$$M_z \downarrow z \iff (z, z) \in H \iff z \in K \iff z \notin \overline{K} \iff M'' \uparrow z$$

Ma $M'' = M_z$, contraddizione!

4.0.5 Riduzioni

Siano S e T due problemi. Una funzione totale e calcolabile $f{:}N{\to}N$ si dice riduzione del problema S al problema T se, per ogni x ∈ N si ha:

$$\text{x} \in \text{S se e soltanto se } f(x) \in \text{T}$$

Proposizione
Siano S e T due problemi. Se esite una riduzione di S a T e T è decidibile, allora S è decidibile.

Dimostrazione
Siano M e M′ rispettivamente la macchina di Turing che calcola f e quella che decide T. Allora S è deciso dalla macchina che esegue il seguente algoritmo.
Su input x

1. simula M su x
2. simula M′ sull'output di M
3. restituisci l'output di M′

4.0.6 Riduzioni-2

Corollario
Siano S e T due problemi. Se esite una riduzione di S a T e S è indecidibile, allora anche T è indecidibile.

Proposizione
Siano S e T due problemi. Se esite una riduzione di S a T e T è semidecidibile, allora S è semidecidibile.

Esempio
La funzione x $\in N \mapsto (x, x) \in N^2$ è una riduzione del problema K al problema dell'arresto H.

Osservazione
Il problema dell'arresto è semi-decidibile.
Invero, è accettato dalla macchina di Turing universale.

Il problema$\{$(x,y, z) $\in N^3$ | $M_x \downarrow$ y in al più z passi$\}$ **è decidibile**

Teoria della programmazione
Non è possibile costruire un sistema di debugging in grado di stabilire se un programma termini o meno

4.0.7 Equivalenza di macchine di Turing

Problema dell'equivalenza di macchine di Turing
Date due macchine di Turing, decidere se calcolano la medesima funzione:

$$\mathrm{E} = \{(\mathrm{x}\ ,\ \mathrm{y}) \mid \phi_x = \phi_y\}.$$

Osservazione
Due funzioni sono uguali se

- hanno lo stesso dominio,
- hanno lo stesso valore su ogni elemento del dominio.

La dimostrazione richiede vari passi che utilizzano diagonalizzazione e riduzione

4.0.8 Funzioni Totali

Proposizione

Il problema T = {x ∈ N | ϕ_x è totale} è indecidibile.

Dimostrazione

Per assurdo, sia T decidibile.

Allora c'è una funzione calcolabile totale f tale che T = f(N).

Invero f è calcolata dal seguente algoritmo

su input x

1 $y \leftarrow -1$
2 for $z = 0, \ldots, x$
3 repeat
4 $y \leftarrow y + 1$
5 until $y \in T$
6 return y

La funzione g : $N \mapsto N$ definita da

$$g(x) = \phi_f(x)\ (x)+1$$

è calcolabile totale. Pertanto, g = $\phi_f(z)$ per un opportuno z ∈ N.

Ma allora

$$\phi_{f(z)}(z) = g(z) = \phi_{f(z)}(z) + 1,$$

assurdo!

4.0.9 Funzione Nulla

La funzione nulla è la funzione calcolabile totale Z : x ∈ N ↦ 0.

Proposizione
Per ogni x ∈ N, consideriamo la funzione g_x definita da:

$$g_x(y) = \begin{cases} 0 & \text{se } \phi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Si verifica facilmente che g_x è calcolabile e anche il suo indice è calcolabile.
Invero esso è calcolato dal seguente algoritmo:
Su input x

1. calcola il programma di M_x
2. aggiungi al programma le istruzioni seguenti:
    - se la macchina originale si arresta, output zero
3. restituisci il codice della macchina modificata

Quindi $g_x = \phi_h(x)$, con h calcolabile totale. Inoltre

$$x \in T \iff g_x = Z \iff h(x) \in S.$$

Pertanto h è una riduzione di T a S.
Poiché T è indecidibile, lo sarà anche S.

4.0.10 Equivalenza di macchine di Turing

Proposizione

Il problema E = $\{(x , y) \in N^2 \mid \phi_x = \phi_y\}$ è **indecidibile**

Dimostrazione

Sia z l'indice della funzione nulla.

La funzione f : x $\in$ N $\mapsto$ (x,z) $\in N^2$ è una riduzione di S a E.

Poiché S è indecidibile, lo sarà anche E.

Osservazione

Non è possibile decidere se due generici programmi calcolano la stessa funzione: la correttezza semantica di un programma è indecidibile.

Capitolo 5

Problemi semi-decidibili

5.0.1 Insiemi semi-decidibili

Definizione
Un problema L $\subseteq$ N è decidibile se esiste una macchina di Turing che computa la sua funzione caratteristica
Il problema L è **semidecidibile** se esiste una macchina di Turing M che accetta L,

$$\chi_s(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{se } x \notin L \end{cases}$$

ovvero, per ogni x $\in$ N,

M $\downarrow$ x se e solo se x $\in$ L

Teorema
Sia L $\subseteq$ N. Allora le proposizioni seguenti sono equivalenti:

- L è semidecidibile
- L è il dominio di una funzione calcolabile
- L è vuoto o c'è una funzione totale calcolabile $f : N \mapsto N$ tale che $L = f(N)$

5.0.2 Enumerabilità effettiva

Osservazione

Se $L \neq \emptyset$, la condizione 3 esprime il fatto che esiste una enumerazione effettiva degli elementi di L:

$$\text{f(0),f(1),f(2), . . . , f (n), . . .}$$

Dimostrazione

1. L è semidecidibile
2. L è il dominio di una funzione calcolabile
3. L è vuoto o c'è una funzione totale calcolabile $f : N \mapsto N$ tale che $L = f(N)$

L'equivalenza tra 1 e 2 è immediata.
Mostriamo che 1 implica 3. Se $L \neq \emptyset$ è accettato dalla macchina di Turing M , allora la seguente procedura infinita produce tutti gli elementi di L (con ripetizioni).
(next genera la coppia successiva in una enumerazione di NxN)

```
1  (x, y) ← (0, 0)
2  repeat
3        if M ↓ x in y passi
4              output x
5        next (x, y)
6  forever
```

Poniamo $f(x)$ uguale all'$(x + 1)$-esimo output della procedura precedente.
Mostriamo che 3 implica 1. Se $L = f(N)$, allora L è accettato dalla macchina di Turing che esegue il seguente algoritmo.

```
Su input x
1  y ← 0
2  while x ≠ f(y)
3        y ← y + 1
4  accetta
```

5.0.3 Insiemi semidecidibili e decidibili

Teorema

Un insieme L è decidibile se e solo se sia L che il suo complemento sono semidecidibili.

Dimostrazione

Se L è decidibile, lo è anche il complemento. Pertanto, sono semidecidibili entrambi. Viceversa, se L e il suo complemento sono accettati dalle macchine di Turing M e M', rispettivamente, la seguente procedura decide L:

```
Su input x
1  repeat
2       simula un passo di M su x
3       if M si arresta
4            accetta
5       simula un passo di M' su x
6       if M' si arresta
7            rifiuta
8  forever
```

Esempio

Il problema:

$$\mathrm{K} = \{x \in N | M_x \downarrow x\}$$

è semidecidibile ma non decidibile (invero è riducibile al problema dell'arresto, che è semidecidibile in quanto accettato dalla macchina di Turing universale).
Pertanto:

$$\overline{k} = \{x \in N | M_x \uparrow x\}$$

non è semidecidibile.

Osservazione

Esistono macchine di Turing M per cui è indecidibile il problema.

$$\{x \in N | M \downarrow x\}.$$

In effetti, qualunque macchina di Turing che accetti un linguaggio semidecidibile ma non decidibile (come, ad es., K) ha questa proprietà.

5.0.4 Il 10 problema di Hilbert

Data un'equazione Diofantea, determinare se ammette soluzioni intere.
Il problema seguente si riduce al decimo problema di Hilbert.

Problema
Data un'equazione Diofantea, determinare se ammette soluzioni in numeri naturali

Dimostrazione
L'equazione

$$D(x, y, z) = 0$$

ha una soluzione naturale se e soltanto se l'equazione.

$$D(x_1^2 + x_2^2 + x_3^2 + x_4^2, y_1^2 + y_2^2 + y_3^2 + y_4^2, z_1^2 + z_2^2 + z_3^2 + z_4^2) = 0$$

ha una soluzione intera (per il Teorema dei quattro quadrati di Lagrange).

5.0.5 Insiemi diofanteei

Definizione
Un insieme $L \subseteq N$ si dice Diofanteo se esiste un'equazione Diofantea $D(x_0, x_1, ..., x_m) = 0$ tale che

$$L = \{a \in N | D(a, x_1, ..., x_m) = 0 \text{ ha soluzione}\}$$

Osservazione
Una soluzione del 10. problema di Hilbert implicherebbe la decidibilità di tutti gli insiemi Diofantei.

Esempi
L'insieme dei quadrati perfetti è rappresentato da

$$a - x^2 = 0$$

L'insieme dei numeri composti è rappresentato da

$$a - (x_1 + 2)(x_2 + 2) = 0$$

L'insieme dei numeri che non sono potenze di 2 è rappresentato da

$$a - (2x_1 + 3)x_2 = 0$$

Proposizione
Gli insiemi Diofantei sono semidecidibili. (perchè è semidecidibile il 10. problema di Hilbert)

5.0.6 Insiemi Diofantei e insiemi semidecidibili

Osservazione
L'equazione $D(a, x_1, ..., x_m) = 0$ ha soluzioni se e solo se ha soluzioni l'equazione:

$$a = (x_0 + 1)(1 - (D(x_0, x_1, ..., x_m))2) - 1$$

Quindi, un insieme Diofanteo è l'insieme dei valori positivi assunti da un polinomio con variabili in N.

Congettura di M. Davis

Diofanteo = semidecidibile

Teorema (M. Davis, H. Putnam, J. Robinson)
Per ogni insieme semidecidibile L esiste un'equazione Diofantea esponenziale E tale che

$$L = \{a \in N | Ehaunasoluzione(a, x_1, ..., x_n)\}$$

Un'equazione Diofantea esponenziale è, ad esempio

$$(x + 1)^{y+2} + x^3 = y^{(x+1)^x} + y^4$$

5.0.7 Prova della congettura di Davis

Come corollario del teorema di Davis, Putnam e Robinson, per provare la congettura di Davis ci si può ridurre al solo insieme:

$$(a, b, a^b)|a, b \in N$$

Il fatto che quest'ultimo insieme è Diofanteo è stato definitivamente dimostrato da Y. Matiyasevich (1970).

Teorema (Davis, Putnam, Robinson, Matiyasevich)
Ogni insieme semidecidibile è Diofanteo. Pertanto il 10. problema di Hilbert è indecidibile.

Osservazione
L'intera teoria della computabilità si può riscrivere in termini di equazioni Diofantee.

Problema
Data un'equazione Diofantea, determinare se ammette soluzioni razionali.

Capitolo 6

Auto-riferimento

6.1 Autoriferimento

6.1.1 Teorema di ricursione di Turing

Teorema
Sia $f : N$ x $N \mapsto N$ una funzione calcolabile (parziale). Esiste effettivamente $z \in N$ tale che:

$$\phi_z(y) = f(z, y).$$

Osservazione

- Il programma di una macchina di Turing può accedere al suo codice
- Si può calcolare una funzione che dipende anche dalla macchina che la calcola

Esempio
Cosa calcola la seguente macchina di Turing?.
Su input x

```
1  z ← mio indice
2  se x = 0, output 1
3  altrimenti output x(φ_z(x − 1))
```

6.1.2 Teorema s-m-n

Teorema

Sia $f : N$ x $N \mapsto N$ una funzione (parziale) calcolabile. Allora esiste una funzione calcolabile totale $t : N \mapsto N$ tale che

$$\phi_{t(x)}(y) = f(x, y) \text{ con } x, y \in N.$$

Osservazione

Questo teorema ci dice che ogni funzione di due variabili f(x,y) si può calcolare nel modo seguente:

- costruisco una macchina di Turing $M = M_t(x)$ che dipende solo da x
- eseguo M sull'input y

6.1.3 Dimostrazione del Teorema s-m-n

Dimostrazione

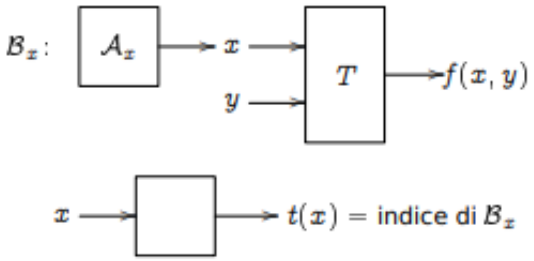
Per ogni $x \in N$, sia $g_x : N \mapsto N$ la funzione definita da $g_x(y) = f(x, y)$.

Una macchina di Turing per calcolare gx si ottiene concatenando

- le istruzioni per stampare x sul nastro prima di y
- le istruzioni della macchina di Turing che calcola f .

Dato x, si può effettivamente costruire tale macchina e calcolarne l'indice.
Se t è la funzione che esegue questo calcolo, si avrà

$$f(x, y) = g_x(y) = \phi_{t(x)}(y).$$



6.1.4 Dimostrazione del Teorema di Ricursione

Per ogni $x \in N$, sia $g_x : N \mapsto N$ la funzione definita da $g_x(y) = f(x, y)$.

$$g(x, y) = \begin{cases} f(\phi_x(x), y), & \text{se } \phi_x(x)\downarrow, \\ \uparrow & \text{altrimenti.} \end{cases}$$

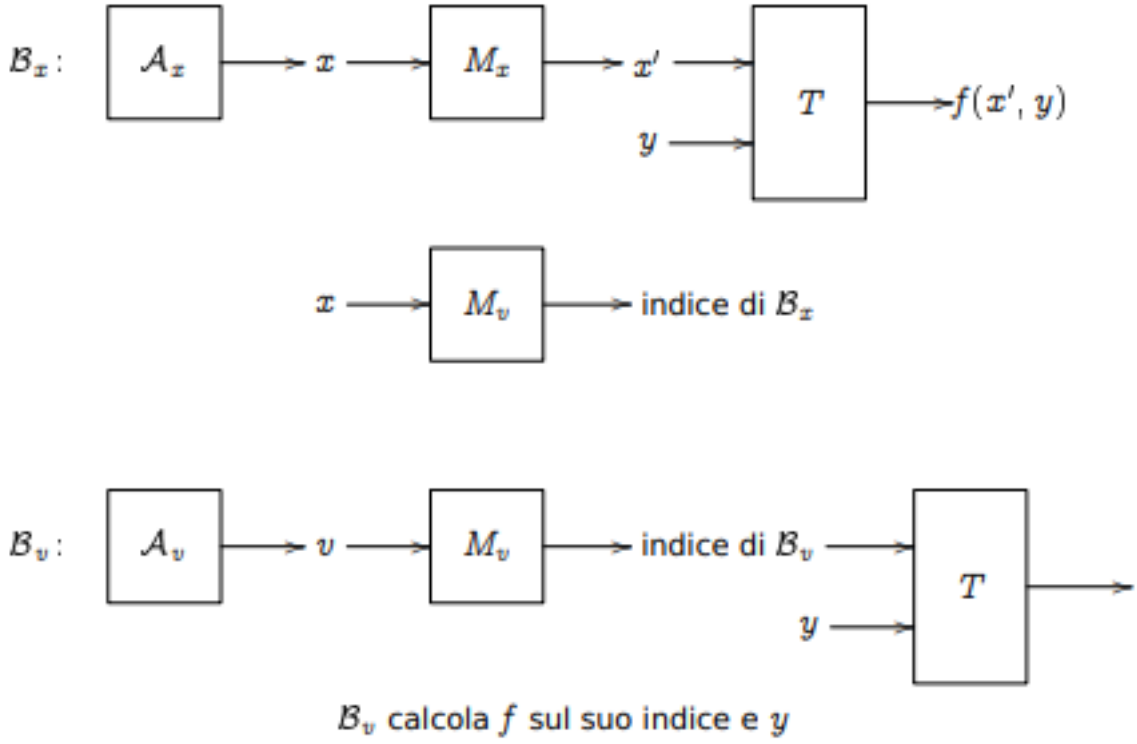
Una macchina di Turing per calcolare g_x si ottiene concatenando.
Per il Teorema s-m-n, c'è una funzione totale t $= \phi_v$ tale che

$$\phi_{t(z)}(y) = g(x, y) \text{ con } x, y \in N$$

Posto $z = t(v) = \phi_v(v)$, si ha

$$\phi_z(y) = \phi_t(v)(y) = g(v, y) = f(\phi_v(v), y) = f(z, y)$$

6.1.5 Costruzione



6.1.6 Esempio

È indecidibile se una macchina di Turing accetta 1.
Se così non fosse, potremmo costruire una macchina di Turing con il seguente programma
Su input x.

```
1  z ← mio indice
2  se M_z accetta 1, rifiuta
3  altrimenti accetta
```

Contraddizione!

Nuova dimostrazione dell'indecidibilità del problema dell'arresto:
Se fosse decidibile, potremmo costruire una macchina di Turing con il seguente programma.
Su input x.

```
1  z ← mio indice
2  se M_z ↑ x, arresto
3  altrimenti loop infinito
```

Contraddizione!

6.1.7 Teorema del punto fisso

Teorema

Sia $t : N \mapsto N$ una funzione calcolabile totale.
Esiste effettivamente $z \in N$ tale che

$$\phi_z = \phi_t(z)$$

Dimostrazione

La funzione $f : N$ x $N \mapsto N$ definita da $f(x, y) = \phi_{t(x)}(y)$ con $x, y \in N$, è calcolabile.
Per il Teorema di Recursione ci sarà $z \in N$ tale che

$$\phi_z(y) = f(z, y) = \phi_t(z)(y), y \in N.$$

Oppure

Basta considerare la funzione calcolata dalla procedura.
Su input x

1 $z \leftarrow$ mio indice
2 output $\phi_{t(z)}(y)$

6.1.8 Teorema di Radice

Teorema
Sia P una famiglia di funzioni calcolabili. L'insieme.

$$R = \{x \in N | \phi_x \in P\}$$

è indecidibile, purchè $R \neq \varnothing$, N.

Dimostrazione
Per assurdo, supponiamo che R sia decidibile e $R \neq \varnothing$, N.

- Possiamo trovare $i \in R$ e $j \notin R$.
- Definiamo la funzione $f : N$ x $N \mapsto N$ con

$$f(x, y) = \begin{cases} \phi_i(y) & \text{se } x \notin R, \\ \phi_j(y) & \text{se } x \in R. \end{cases}$$

Per il Teorema di Recursione ci sarà $z \in N$ tale che $\phi_z(y) = f(z, y)$.
Ma allora se $z \notin R$, si ha $\phi_z = \phi_i \in R$, mentre se $z \in R$, si ha $\phi_z = \phi_j \notin R$.

Contraddizione!.

6.1.9 Macchina di Turing minimali

Definizione

Una macchina di Turing è minimale se non esistono macchine di Turing con un minor numero di stati che calcolano la stessa funzione.

Osservazione

Legata alle nozioni di complessità di Kolmogorov, entropia, compressibilità.

Proposizione

È indecidibile se una macchina di Turing è minimale.

Dimostrazione

Se così non fosse, avremmo una macchina di Turing che:
Su input x

```
1  z ← mio indice
2  j ← 0
3  repeat
4        j ← j + 1
5  until M_j è minimale e ha più stati di M_z
6  output φ_j(x)
```

Ma allora Mz è equivalente a una macchina minimale con più stati: **assurdo**

Capitolo 7

Il problema di corrispondenza di post

7.0.1 PCP

Un'istanza del problema di corrispondenza di Post (PCP)è una (2n)-pla di parole:

$$(u_1, v_1; u_2, v_2; ...; u_n, v_n)$$

Su un alfabeto A, $n >= 1$.
Un match di tale istanza è una sequenza $i_1, i_2, ..., i_k$ con$k > 0$, tale che

$$u_{i1}, u_{i2}...u_{ik} = v_{i1}, v_{i2}...v_{ik}$$

Denotiamo con PCP l'insieme delle istanze che hanno un match.

Osservazione
In termini algebrici, PCP è l'insieme delle coppie di omomorfismi di semigruppi liberi $g, h : B^+ \mapsto A^+$ che coincidono almeno su una parola di A^+.

7.0.2 Problema corrispondenza di post

Esempio

Consideriamo l'istanza (b,ca;a,ab;ca,a;abc,c) che possiamo rappresentare nella forma

$$\left[\begin{smallmatrix} b \\ ca \end{smallmatrix}\right], \left[\begin{smallmatrix} a \\ ab \end{smallmatrix}\right], \left[\begin{smallmatrix} ca \\ a \end{smallmatrix}\right], \left[\begin{smallmatrix} abc \\ c \end{smallmatrix}\right]$$

Un match si ottiene concatenando

$$\left[\begin{smallmatrix} a \\ ab \end{smallmatrix}\right]\left[\begin{smallmatrix} b \\ ca \end{smallmatrix}\right]\left[\begin{smallmatrix} ca \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ ab \end{smallmatrix}\right]\left[\begin{smallmatrix} abc \\ c \end{smallmatrix}\right]$$

Esempio

L'istanza

$$\left[\begin{smallmatrix} abc \\ ab \end{smallmatrix}\right], \left[\begin{smallmatrix} ca \\ a \end{smallmatrix}\right], \left[\begin{smallmatrix} acc \\ ba \end{smallmatrix}\right]$$

non ha match.

Teorema

PCP è indecidibile.

Dimostrazione

Mostreremo che il problema dell'arresto si riduce a PCP.
Consideriamo il seguente problema di corrispondenza di Post modificato:

Definizione

MPCP è l'insieme delle istanze di PCP che hanno un match tale che $i_1 = 1$.

Proposizione

Il problema dell'arresto si riduce a MPCP.

7.0.3 Ridurre l'arresto MPCP

Dobbiamo costruire una funzione che a ogni macchina di Turing M e ogni input w associ un'istanza $P_{M,w}$ di MPCP, in modo tale che M $\downarrow$ w se e solo se $P_{M,w}$ ha un match con $i_1 = 1$.
Possiamo ridurci al caso di macchine di Turing che

- si arrestano solo in un fissato stato q_h
- ogni volta che entrano nello stato q_h si arrestano

Costruiremo un'istanza di MPCP in cui un match riflette, in qualche modo, la computazione di M su w.

7.0.4 Costruzione dell'istanza

Poniamo

$$\left[\begin{matrix} u_1 \\ v_1 \end{matrix}\right] = \left[\begin{matrix} \$\$ \\ \$\$q_0w\$ \end{matrix}\right]$$

Per ogni quintupla $(q, a, r, b, +1)$ aggiungiamo la coppia

$$\left[\begin{matrix} qa \\ br \end{matrix}\right] \quad \text{e se } a = \#, \text{ anche } \quad \left[\begin{matrix} q\$ \\ br\$ \end{matrix}\right]$$

Per ogni quintupla $(q, a, r, b, -1)$ aggiungiamo le coppie

$$\left[\begin{matrix} cqa \\ rcb \end{matrix}\right], \quad c \in A, \quad \left[\begin{matrix} \$qa \\ \$r\#b \end{matrix}\right]$$

Per ogni $a \in A$ U $\{\#, \$\}$, aggiungiamo la coppia

$$\begin{bmatrix} a \\ a \end{bmatrix}$$

Infine aggiungeremo le coppie

$$\begin{bmatrix} q_h \\ cq_h \end{bmatrix}, \begin{bmatrix} q_h \\ q_h c \end{bmatrix}, \quad c \in A \cup \{\#\}, \qquad \begin{bmatrix} q_h \$\$ \\ \$ \end{bmatrix}$$

Esempio

Istruzioni:

$$(q_0, 1, q_1, \#, +1), (q_0, \#, q_0, \#, +1), (q_1, 1, q_0, \#, +1), (q_1, \#, q_h, \#, +1)$$

Input: w = 111

Istanza di MPCP:

$$\begin{bmatrix} \$\$ \\ \$\$ q_0 111 \$ \end{bmatrix}, \quad \begin{bmatrix} q_0 1 \\ \# q_1 \end{bmatrix}, \begin{bmatrix} q_0 \# \\ \# q_0 \end{bmatrix}, \begin{bmatrix} q_1 1 \\ \# q_0 \end{bmatrix}, \begin{bmatrix} q_1 \# \\ \# q_h \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} \# \\ \# \end{bmatrix}, \begin{bmatrix} \$ \\ \$ \end{bmatrix}, \begin{bmatrix} \# q_h \\ q_h \end{bmatrix}, \begin{bmatrix} 1 q_h \\ q_h \end{bmatrix}, \begin{bmatrix} q_h \# \\ q_h \end{bmatrix}, \begin{bmatrix} q_h 1 \\ q_h \end{bmatrix}, \begin{bmatrix} q_h \$\$ \\ \$ \end{bmatrix}$$

$\$\$ q_0 111\$$
$\$\$ q_0 111\$ \# q_1 11\$$

Simula $(\Lambda, q_0, 1, 11) \vdash_M (\Lambda, q_1, 1, 1)$

Se M ↑ w non ci sono match !

7.0.5 Ridurre MPCP a PCP

$$\left[\begin{matrix} \$\$ \\ \$\$q_0111\$ \end{matrix}\right], \left[\begin{matrix} q_01 \\ \#q_1 \end{matrix}\right], \left[\begin{matrix} q_0\# \\ \#q_0 \end{matrix}\right], \left[\begin{matrix} q_11 \\ \#q_0 \end{matrix}\right], \left[\begin{matrix} q_1\# \\ \#q_h \end{matrix}\right], \left[\begin{matrix} 1 \\ 1 \end{matrix}\right], \left[\begin{matrix} \# \\ \# \end{matrix}\right], \left[\begin{matrix} \$ \\ \$ \end{matrix}\right],$$
$$\left[\begin{matrix} \#q_h \\ q_h \end{matrix}\right], \left[\begin{matrix} 1q_h \\ q_h \end{matrix}\right], \left[\begin{matrix} q_h\# \\ q_h \end{matrix}\right], \left[\begin{matrix} q_h1 \\ q_h \end{matrix}\right], \left[\begin{matrix} q_h\$\$ \\ \$ \end{matrix}\right]$$

Se invece M ↓ w, continuando, otterremo

$$\begin{matrix} \cdots\$ \\ \cdots\$\,\#\,\#\,\#\,\#\,q_h\,\$ \end{matrix} \quad \text{e poi,} \quad \begin{matrix} \cdots\$\,q_h\,\$\,\$ \\ \cdots\$\,q_h\,\$\,\$ \end{matrix}$$

C'è un match !

Asserto
Si ha M ↓ w se e solo se $P_{M,w} \in$ MPCP.

Corollario
Il problema dell'arresto si riduce a MPCP.
Ora costruiamo una riduzione di MPCP a PCP. Per ogni parola $w = a_1a_2...a_l$, denotiamo:

$$*w = *a_1 * a_2... * a_l,\ w* = a_1 * a_2 * ...a_l*$$

Dobbiamo costruire una funzione che a ogni istanza P di MPCP associ un'istanza P' di PCP, in modo tale che P ha un match con $i_1 = 1$ se e solo se P' ha un match.
Sia $P = (u_1, v_1; u_2, v_2; ...; u_n, v_n)$. Definiamo

$$\text{P'} = (*u_1, v_1*; *u_1, v_1*; *u_2, v_2*; ..., *u_n, v_n*; *o, o)$$

Un match di P' inizia necessariamente con $\begin{bmatrix} \star u_1 \\ * v_1 \star \end{bmatrix}$ e corrisponde a un match di P con $i_1 = 1$.

Viceversa, se P ha un match con $i_1 = 1$, allora ci sarà anche un match di P'. Se ne conclude che P' $\in$ PCP se e solo se P $\in$ MPCP.

Conclusione
Abbiamo costruito una riduzione del problema dell'arresto a MPCP e una riduzione di MPCP a PCP.
Poichè il problema dell'arresto è indecidibile, lo sono anche MPCP e PCP.

7.0.6 Matrici con angolo zero in alto a destra

Problema (Zero nell'angolo)
Dato un insieme finito M di matrici 3 x 3 a coefficienti interi, esiste un prodotto di matrici di M che ha uno 0 nell'angolo in alto a destra?.

Teorema
Il problema Zero nell'angolo è indecidibile.

Dimostrazione
PCP si riduce al problema Zero nell'angolo.

7.0.7 Riduzione di PCP a Zero nell'angolo

Sia $P = (u_1, v_1; ...; u_n, v_n)$ un'istanza di PCP.
Costruiremo un insieme di matrici M = $\{M_1, .., M_n\}$ tali che P $\in$ PCP.
se e solo se M contiene una matrice con zero nell'angolo in alto a destra.
Possiamo supporre, senza perdita di generalità, che l'alfabeto sia $\{1, 2, ..., d-1\}$.
Per $i = 1, ..., n$ definiamo:

$$M_i = \begin{pmatrix} 1 & (v_i)_d & (u_i)_d - (v_i)_d \\ 0 & 2^{\ell(v_i)} & 2^{\ell(u_i)} - 2^{\ell(v_i)} \\ 0 & 0 & 2^{\ell(u_i)} \end{pmatrix}$$

dove $(w)_d$ è il numero denotato da w in base d. Si verifica facilmente che

$$M_{i_1} \cdots M_{i_m} = \begin{pmatrix} 1 & (v_{i_1} \cdots v_{i_m})_d & (u_{i_1} \cdots u_{i_m})_d - (v_{i_1} \cdots v_{i_m})_d \\ 0 & 2^{\ell(v_{i_1} \cdots v_{i_m})} & 2^{\ell(u_{i_1} \cdots u_{i_m})} - 2^{\ell(v_{i_1} \cdots v_{i_m})} \\ 0 & 0 & 2^{\ell(u_{i_1} \cdots u_{i_m})} \end{pmatrix}$$

$i_1, ..., i_m \in \{1, ..., n\}$.
Ma allora $M_{i1}...M_{im}$ ha zero nell'angolo in alto a destra se e solo se $(i_1, ..., i_m)$ è un match di P.
Questo prova che la funzione $P \mapsto M$ è una riduzione di PCP a Zero nell'angolo.
Con tecniche simili, si dimostra l'indecidibilità del **problema di mortalità** per le matrici 3 x 3 a coefficienti interi:

Problema (mortalità)
Dato un insieme finito M di matrici 3 x 3 a coefficienti interi, esiste un prodotto di matrici di M uguale alla matrice nulla?

7.0.8 CFG ambigue

Definizione

Una grammatica non contestuale è ambigua se esiste una parola con due derivazioni sinistre.

Teorema

È indecidibile se una grammatica non contestuale è ambigua.

Dimostrazione

PCP si riduce all'ambiguità delle grammatiche non contestuali.
Sia $P = (u_1, v_1; ...; u_n, v_n)$ un'istanza di PCP.
Costruiremo una grammatica non contestuale G che è ambigua se e solo se P $\in$ PCP.
Siano c1,..., cn n nuove lettere e G la grammatica con le produzioni

$$S \mapsto X \mid Y, X \mapsto u_i X_{ci} \mid uici, Y \mapsto v_i Y_{ci} \mid vici, \text{ i} = 1, \ldots, \text{n}.$$

Si verifica facilmente che G genera il linguaggio: Non è poi difficile convincersi che G

$$L = \{u_{i_1} \cdots u_{i_\ell} c_{i_\ell} \cdots c_{i_1} \mid 1 \leq i_1, \ldots, i_\ell \leq n\} \cup \{v_{i_1} \cdots v_{i_\ell} c_{i_\ell} \cdots c_{i_1} \mid 1 \leq i_1, \ldots, i_\ell \leq n\}$$

è ambigua se e soltanto se

$$u_{i1}...u_{il} = v_{i1}...v_{il}$$

per opportuni $i_1, ..., i_l$ cioè se e soltanto se P ha un match.

Capitolo 8

Funzioni ricorsive

8.0.1 Approcci alternativi alla nozione di calcolabilità

- Calcolo equazionale (Gödel,Herbrand,Kleene)
- lambda-calcolo (Church)
- funzioni ricorsive parziali (Gödel,Kleene)
- macchina di Turing
- sistemi di deduzioni canonici (Post)
- sistemu di Markov
- URM (Sheperdson, Sturgis)

Terorema
Ognuno dei metodi precedenti genera la medesima classe di funzioni

8.0.2 Funzioni ricorsive di base

Nel seguito, considereremo esclusivamente funzioni (parziali o totali)

$$f : N^k \mapsto N$$

con $k >= 0$ (l'intero k si dice arietà di f ; nel caso k = 0, f è una costante).

Definizione
Chiameremo funzioni primitive di base le funzioni seguenti.

1. La funzione **successore**

$$s : N \mapsto N \text{ definita da } s(x) = x + 1$$

2. La funzione **zero**

$$z : N \mapsto N \text{ definita da } z(x) = 0$$

3. Le **proiezioni**

$$P_i^k : N^k \mapsto N \text{ definita da } P_i^k(x_1, ..., x_k) = x_i \text{ con } 1 <= i <= k$$

8.0.3 Composizione e recursione

Definizione
Siano $k, n > 0$ con , $h : N^k \mapsto N$ e $g_1, ..., g_n : N^k \mapsto N$ funzioni totali.
La funzione $f : N^k \mapsto N$ definita da:

$$f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), ..., g_n(x_1, ..., x_k))$$

si dice la funzione ottenuta per composizione dalle funzioni $h, g_1, ..., g_n$.

Definizione
Sia $k >= 0, h : n^k + 2 \mapsto N$ e $g : N^k \mapsto N$ funzioni totali.
La funzione $f : N^k + 1 \mapsto N$ definita da

$$f(x_1, .., x_k, 0) = g(x_1, ..., x_k)$$
$$f(x_1, .., x_k, y + 1) = h(x_1, ..., x_k, y, f(x_1, ..., x_k, y)$$

si dice la funzione ottenuta per **recursione** dalle funzioni g e h.

8.0.4 Funzioni ricorsive primitive

Definizione

Una funzione si dice ricorsiva primitiva se si può ottenere dalle funzioni ricorsive di base con un numero finito di operazioni di composizione e recursione.

Osservazione

Le funzioni ricorsive primitive sono Turing-calcolabili.

Esempi

La funzione costante $c_i : \mathbb{N} \to \mathbb{N}$ definita da $c_i(x) = i$ è primitiva.
Infatti $c_i(x) = \underbrace{s(s(\cdots s}_{i \text{ volte}}(z(x))\cdots))$,

Addizione:
$add(x, 0) = P_1^1(x), \quad add(x, y+1) = s(P_3^3(x, y, add(x, y)))$,

Moltiplicazione:
$mult(x, 0) = 0, mult(x, y+1) = add(x, mult(x, y))$,

Predecessore: $pred(0) = 0, \quad pred(y+1) = P_1^1(y)$,

Sottrazione tronca:
$sub(x, 0) = P_1^1(x), \quad sub(x, y+1) = pred(sub(x, y))$,

Elevamento a potenza:
$exp(x, 0) = s(z(x)), \quad exp(x, y+1) = mult(x, exp(x, y))$,

$exponent(i, y)$ = l'esponente dell'i-esimo primo nella decomposizione di y in fattori primi,

è una funzione ricorsiva primitiva.

8.0.5 La funzione di Ackermann

Problema

La nozione di funzione ricorsiva primitiva coincide con la nozione intuitiva di funzione calcolabile? La funzione $A(n, x) = x \uparrow^n x$ è la funzione di Ackermann.

$$x \uparrow^0 y = x + 1,$$

$$x \uparrow^1 y = x + y = x + \underbrace{1 + \cdots + 1}_{y \text{ volte}},$$

$$x \uparrow^2 y = x \cdot y = \underbrace{x + \cdots + x}_{y \text{ volte}},$$

$$x \uparrow^3 y = x^y = \underbrace{x \cdot \cdots \cdot x}_{y \text{ volte}},$$

$$x \uparrow^4 y = x^y = \underbrace{x^{x^{\cdot^{\cdot^{x}}}}}_{y \text{ volte}},$$

$$\vdots$$

$$x \uparrow^{n+1} y = \underbrace{x \uparrow^n \cdots \uparrow^n x}_{y \text{ volte}},$$

$$\vdots$$

Osservazione

- La funzione di Ackermann è effettivamente calcolabile
- cresce molto velocemente

Teorema

Per ogni funzione ricorsiva primitiva f esiste $|x_0 >= 0$ tale che

$$f(x) < A(x_1, x_1), \text{ per ogni } x >= x0$$

Corollario

La funzione di Ackermann non è ricorsiva primitiva.

8.0.6 Minimalizzaione

Definizione
Sia $k > 0$ e $g : N^k + 1 \mapsto N$ una funzione totale. La funzione parziale $f : N^k \mapsto N$ definita da

$$f(x_1, ..., x_k) = min\{y \mid g(x_1, .., x_k, y) = 0\}$$

si dice ottenuta da g per minimalizzazione.

Osservazione
Se g è calcolabile, allora lo è anche f.

Osservazione
Le nozioni di composizione e recursione si estendono naturalmente alle funzioni parziali.

8.0.7 Composizione e recursione

Definizioni
Sia $k > 0$ e $h : N^n \mapsto N$ una funzione ottenuta per composizione dalle funzioni $h_1, g_1, ..., g_n$. avrà dominio

$$\{\vec{x} \in \mathbb{N}^k \mid \vec{x} \in \mathrm{dom}(g_i),\ 1 \leq i \leq n,\ (g_1(\vec{x}), \ldots, g_n(\vec{x})) \in \mathrm{dom}(h)\}$$

è definito da:

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

Sia $k >= 0$, $h : N^{k+2} \mapsto N$ e $g : N^k \mapsto N$ funzioni parziali.
La funzione f ottenuta per recursione dalle funzioni g e h avrà dominio definito da

$$(\vec{x}, 0) \in \mathrm{dom}(f) \text{ se } \vec{x} \in \mathrm{dom}(g)$$
$$(\vec{x}, y + 1) \in \mathrm{dom}(f) \text{ se } (\vec{x}, y) \in \mathrm{dom}(f) \text{ e } (\vec{x}, y, f(\vec{x}, y)) \in \mathrm{dom}(h)$$

e sarà definito da:

$$f(\vec{x}, 0) = g(\vec{x}), \quad f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y)),$$

8.0.8 Funzioni ricorsive parziali

Definizioni

Una funzione si dice ricorsiva parziale se si può ottenere dalle funzioni ricorsive di base con un numero finito di operazioni di composizione, recursione e minimalizzazione. Si dice funzione ricorsiva una funzione parziale ricorsiva che sia anche totale. Infine, un sottoinsieme di N si dice ricorsivo se la sua funzione caratteristica è ricorsiva.

Proposizione

Le funzioni ricorsive parziali sono calcolabili.

8.0.9 Funzioni ricorsive parziali e calcolabilità

Teorema

Una funzione è ricorsiva parziale se e solo se è Turing-calcolabile.

Corollario

Un insieme è ricorsivo se e solo se è decidibile.
Diremo che un insieme è ricorsivamente enumerabile se è vuoto o è l'immagine di una funzione ricorsiva.

Corollario

Un insieme è ricorsivamente enumerabile se e solo se è semidecidibile.

8.0.10 Dimostrazione

Si è già visto che le funzioni ricorsive parziali sono Turing-calcolabili. Dobbiamo provare che le funzioni Turing-calcolabili sono ricorsive parziali. Sia quindi f una funzione Turing-calcolabile e sia M la macchina di Turing che la calcola. Per ogni descrizione istantanea C di M, denoteremo con $gn(C)$ il suo numero di Gödel.
Consideriamo le funzioni

$$\mathtt{start}(x) = \mathtt{gn}(\text{config. iniziale con input } x)$$

$$\mathtt{next}(x) = \begin{cases} \mathtt{gn}(C') & \text{se } x = \mathtt{gn}(C) \text{ e } C \vdash_M C', \\ 0 & \text{se } x = \mathtt{gn}(C) \text{ e } C \not\vdash_M, \\ 1 & \text{altrimenti.} \end{cases}$$

$$\mathtt{tape}(x) = \begin{cases} y & \text{se } x = \mathtt{gn}(C) \text{ e } C \text{ è una configurazione con } y \text{ sul nastro,} \\ 0 & \text{altrimenti.} \end{cases}$$

8.0.11 Dimostrazione-2

Le funzioni start, next, tape sono ricorsive primitive. Definiamo $g : N^2 \mapsto N$ con

$$g(x_1, 0) = start(x), g(x, t + 1) = next(g(x_1, t)).$$

La funzione g calcola il numero di Gödel della configurazione istantanea di M su input x dopo t passi.
E' ricorsiva primitiva.
La funzione

$$h(x) = min\{t|g(x, t) = 0\}$$

calcola il numero di passi in cui M si arresta su input x , diminuito di 1; è ricorsiva parziale.
La funzione

$$k(x) = tape(pred(h(x)$$

calcola l'output di M su input x, cioè $f(x)$; è ricorsiva parziale.

Capitolo 9

Teoria della complessità

9.0.1 Costo della computazione

Problema
Cosa si può calcolare con un costo ragionevole?

Osservazione
Il costo dipende dall'algoritmo

Esempio
Vogliamo determinare il valore del polinomio

$$x^7 + 5x^6 + 2x^4 + 12x^3 + 5x^2 + 2x + 21$$

per certi input x . Quante moltiplicazioni servono?

- per ciascun monomio, nell'ordine, 6,6,4,3,2,1, totale: 22
- se calcolo prima tute le potenze di x (6 moltiplicazioni), per ciascun monomio, nell'ordine, 0,1,1,1,1,1, totale: 11
- se scrivo il polinomio nella forma

$$(((((x + 5)x^2 + 2)x + 12)x + 5)x + 2)x + 21$$

Esempio

Per calcolare il massimo comun divisore di due interi positivi a e b si può:

- decomporre a e b in fattori primi
- selezionare i fattori primi comuni, con il minimo esponente

oppure usare l'algoritmo di Euclide:

```
repeat
    c ← a mod b
    a ← b
    b ← c
until c = 0
return a
```

$$\begin{cases} 2x + 3y = 5 \\ x - y = 0 \end{cases}$$

Per risolvere un sistema lineare si può utilizzare il metodo di Cramer:

$$x = \frac{\det \begin{pmatrix} 5 & 3 \\ 0 & -1 \end{pmatrix}}{\det \begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix}}, \quad y = \frac{\det \begin{pmatrix} 2 & 5 \\ 1 & 0 \end{pmatrix}}{\det \begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix}},$$

oppure ridurre il sistema in forma triangolare:

$$\begin{cases} 2x + 3y = 5 \\ 5y = 5 \end{cases}$$

9.0.2 Teoria della complessità

Quale risorsa misurare?
Tempo, memoria, energia,...

Quale modello di calcolo?
Macchina di Turing, altri?.

Quali problemi considerare?

- Problemi di **decisione**

  dato $S \subseteq A*$, decidere per ogni input $w \in A*$ se $w \in S$.

- Problemi di **computazione**

  Data $f : A* \mapsto A*$ calcolare, per ogni input $w \in A*$, f(w)

- problemi di ottimizzazione

- problemi di ricerca

9.0.3 Teoria della complessità2

Che risultati cerchiamo?
Vogliamo conoscere il costo minimo per risolvere un problema (non un particolare algoritmo). Ci aspettiamo

- Risultati negativi

  il problema non si può risolvere senza...

- Risultati di confronto

  per risolvere il problema S servono almeno le stesse risorse necessarie a risolvere il problema T

9.0.4 Come misurare la complessità

Vogliamo valutare il costo della computazione in funzione della complessità dell'input. Sia S un problema (di decisione) sull'alfabeto A. Consideriamo una funzione $f : N \mapsto N$ calcolata come segue:

- per ogni $n \in N$ consideriamo tutti gli input di lunghezza minore o uguale a n e il costo delle relative computazioni (escludendo quelle che non terminano)
- f(n) restituirà il massimo costo di tali computazioni

Proprietà delle funzioni costo

- una funzione costo è definita per n >= n_0, con $n_0 \in N$ opportuno
- non decrescente
- non limitata

Esempi
$n, n^2, n^k, 2^n, 2^{2n}, |\log_2 n|$, polinomi a coefficienti non negativi.

Esempio

Quante divisioni servono per l'algoritmo Euclideo del MCD?
Supponiamo a >= b e calcoliamo MCD(a, b).

$$\begin{aligned} r_0 &= a \mod b \\ r_1 &= b \mod r_0 \\ r_2 &= r_0 \mod r_1 \\ &\cdots \\ r_k &= r_{k-1} \mod r_{k-2} = 0 \end{aligned}$$

Pertanto.

$$\begin{aligned} a &\geq b + r_0 > 2r_0 \\ &\geq 2(r_1 + r_2) > 4r_2 \\ &\geq \cdots > 2^{k/2}. \end{aligned}$$

Ne segue

$$k < 2 \log_2 a$$

Pertanto:
Il numero di divisioni necessario per calcolare MCD(a, b) è linearmente limitato dalla lunghezza della rappresentazione binaria di a.

Osservazione

Per le funzioni numeriche, la complessità dell'input è misurata dal numero di bit (o cifre decimali) necessaria per rappresentarlo:

$$1 + |\log_2(n)|$$

9.0.5 Complessità temporale

Definizione
Sia M una macchina di Turing. Si dice complessità temporale di M la funzione $c_M : N \mapsto N$ definita come segue:
per ogni $n \in N$, $c_M(n)$ è il massimo numero di passi di una computazione convergente di M su un input di lunghezza minore o uguale a n.

Osservazione
La funzione c_M è definita per $n >= n_0$ con $n_0 \in N$, non decrescente e non limitata (purchè c_M abbia un dominio infinito e M esamini l'intero input su ogni computazione).

9.0.6 La classe P

Definizione
Denotiamo con P la classe dei problemi S accettati da una macchina di Turing M tale che $c_M(n) = O(n^k)$ per qualche intero positivo k.

Osservazione
Se $S \in P$, allora c'è una macchina di Turing che, per ogni input w di lunghezza n, si comporta nel modo seguente:

- se $w \in S$, allora M accetta w in al più $cl(w)^k$ passi (con c, k costanti positive fissate)
- se $w \notin S$, allora $M \uparrow w$

In realtà, c'è anche una macchina di Turing M' che decide S con $c_{M'}(n) = O(n^k)$.

9.0.7 Tesi di Edmonds-Cook-Karp

Un problema S ha un algoritmo rapido di decisione se e solo se $S \in$ P.

Osservazione

Riferita ai problemi, non agli algoritmi.

Osservazione

La definizione di P è robusta:
se esiste un algoritmo che decide S e richiede l'esecuzione di $O(n^k)$ istruzioni (comunque specificate, purchè simulabili da una macchina di Turing in tempo polinomiale), allora $s \in P$.

Capitolo 10

La classe P

10.0.1 La classe P

Definizione
Denotiamo con P la classe dei problemi S accettati da una macchina di Turing M tale che $c_M(n) = O(n^k)$ per qualche intero positivo k.

Osservazione
Se $S \in P$, allora c'è una macchina di Turing che, per ogni input w di lunghezza n, si comporta nel modo seguente:

- se $w \in S$, allora M accetta w in al più $cl(w)^k$ passi (con c, k costanti positive fissate)
- se $w \notin S$, allora M $\uparrow$ w

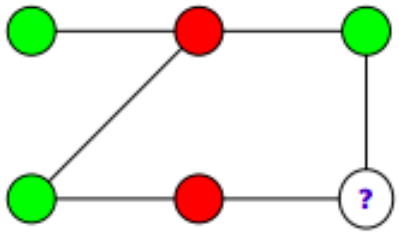
In realtà, c'è anche una macchina di Turing M' che decide S con $c'_M(n) = O(n^k)$.

10.0.2 2Col

Definizione
Un grafo $G = (V, E)$ è 2-colorabile se esiste una funzione $c : V \mapsto \{1, 2\}$ tale che per ogni $(v, w) \in E$, $f(v) \neq f(w)$.
La classe dei grafi 2-colorabili è denotata 2COL.

Proposizione
2COL $\in$ P.

Dimostrazione
È sufficiente trovare un algoritmo che risolve 2COL in tempo polinomiale.



10.0.3 Algoritmo

Su input G.

```
1  per ogni componente connessa C
2        seleziona un vertice e assegnagli il colore 1
3  while c'è un vertice v non colorato adiacente a uno colorato
4        if v adiacente a due vertici di colore diverso
5              rifiuta
6        else
7              assegna a v colore opposto a quello del vertice adiacente
8  accetta
```

10.0.4 SAT

Siano $x_0, x_1, x_2, ...$una famiglia, potenzialmente infinita di variabili. Chiameremo letterali le variabili x_i e le loro negazioni $\bar{x}_i$. Una clausola è una sequenza di letterali che non contiene letterali col medesimo indice.
Un sistema di clausole si dice soddisfacibile se esiste una clausola che le interseca tutte (ossia contiene un letterale di ogni clausola del sistema) La famiglia dei sistemi di clausole soddisfacibili è denotata con SAT.

Esempio
Il sistema.

$$\{(x_0 + \bar{x}_1)(x_0 + \bar{x}_1 + x_2)(x_3 + x_4 + \bar{x}_0)(\bar{x}_4 + \bar{x}_5)\}$$

Una clausola che soddisfa il sistema non è altro che un implicante dell'espressione medesima. Quindi un sistema è soddisfacibile se la corrispondente espressione Booleana non è nulla.

10.0.5 2SAT

Definizione
Una clausola di lunghezza 2 si dice 2-clausola.
L'insieme dei sistemi di 2-clausole soddisfacibili si denota con 2SAT.
Proposizione
2SAT $\in$ P.

Dimostrazione
Dobbiamo trovare un algoritmo che risolve 2SAT in tempo polinomiale.

Esempio

$\{x_0\overline{x}_1, x_0x_2, \overline{x}_1x_3, x_4\overline{x}_0, \overline{x}_4\overline{x}_5\}$

soluzione: $x_0x_4\overline{x}_5x_3$

$\{x_0x_1, x_0\overline{x}_1, \overline{x}_0x_2, \overline{x}_0\overline{x}_2\}$

soluzione: $x_0\overline{x}_0$

10.0.6 Ridurre 2COL a 2SAT

Costruiamo una riduzione di 2COL a 2SAT.
Devo costruire una funzione calcolabile totale f che a ogni grafo G associa un sistema di 2-clausole f (G) tale che.
foto.
Sia G = (V , E) un grafo.
Per ogni vertice $v_i, i = 1, ..., n$, introduco una variabile x_i (x_i = "v_i ha colore verde") e per ogni lato $(v_i, v_j) \in E$, le due clausole

$$x_i, x_j, \bar{x}_i, \bar{x}_j$$

(uno dei vertici ha colore verde, uno non ha colore verde).

Algoritmo alternativo per 2COL
Su input G

1 costruisci $f(G)$
2 se $f(G) \in$ 2SAT accetta, altrimenti rifiuta

10.0.7 Equazioni lineari

Probelma
Input: Un'equazione $ax + by = c$ con $a, b, c \in Z$
Output: si se l'equazione ha una soluzione intera, no altrimenti.

Osservazione

- Se MCD(a, b) = 1, l'equazione ha soluzione intera (teorema di Bezout)
- Se MCD(a, b) divide c, ci si può ricondurre al caso precedente
- Se l'equazione ha una soluzione intera, allora MCD(a, b) divide c.

Soluzione
Per risolvere il problema basta calcolare MCD(a, b) e verificare se divide c.
Sono necessarie

- $O(n^2)$ divisioni, n = $\log_2$ (max$\{a, b, c\}$)
- Per ogni divisione, $O(n)$ sottrazioni
- Per ogni sottrazione $O(n)$ operazioni elementari sui bit.
- totale: $O(n^4)$ operazioni elementari: il problema è nella classe P

10.0.8 Numeri primi

Problema
Input: Un numero a ∈ N.
Output: Si se a è primo, no altrimenti.

Soluzione
Dividi a per tutti gli interi $q = 2, 3, .., |\sqrt{a}|$.

- ma sono $|\sqrt{a} - 1| = O(2^{n/2})$ divisioni, $n = \log_2$a

L'algoritmo di Agrawal, Kayal, Saxena (2002), ha complessità $O(n^1 1)$. Quindi il problema è nella classe P.

Osservazione
Nella pratica, la complessità $n^1 1$ è troppo alta, per cui per la ricerca di grandi primi si preferiscono algoritmi probabilistici.

Problema Input: Un numero $a \in N$.
Output: la decomposizione di a in fattori primi.

10.0.9 coP

Definizione
Denotiamo coP la classe dei problemi il cui complemento è in P.

Proposizione
coP = P.

Dimostrazione
Se $S \in coP$, c'è una macchina di Turing M che decide S con $c_M(n) = O(n^k)$, per k opportuno.
Ma allora c'è una macchina M' che accetta S, con $c'_M(n) = c_M(n) = O(n^k)$

(basta aggiungere le istruzioni affinchè M con output SI inizi un loop infinito).

Capitolo 11

La classe NP

11.0.1 Macchine di Turing non deterministiche

Una Macchina di Turing M è una quadrupla (Q, A, σ, q_0), dove:

- Q è un insieme finito di stati
- A è un alfabeto, cui si aggiunge il simbolo bianco #
- σ è una funzione da Q $\times$ (A U {#}) a p(Q $\times$ (A U {#}) $\times$ $\{-1, +1\}$), chiamata funzione di transizione
- $q_0 \in Q$ è lo stato iniziale

Le quintuple (q, a, q', a', x) tali che (q', a', x) $\in \sigma(q, a)$ sono dette le istruzioni di M.

11.0.2 Configurazioni consecutive

Definizione
Una configurazione istantanea di una Macchina di Turing M = (Q, A, σ, q_0) è un elemento dell'insieme (A U {#})* x Q x (A U {#}) x (A U {#}*. Nell'insieme delle configurazioni istantanee di M introduciamo la relazione binaria $\vdash_M$ che associa alla configurazione C quelle che possono seguirla in una computazione di M. Qualora non esistesse nessuna configurazione di questo tipo, scriveremo $C \vdash /_M$

Esempio
Se M ha l'istruzione (q, a, q', b, −1) e (q, a, q", a, +1), allora.

$$(abbaa, q, a, ababb) \underset{M}{\vdash} (abba, q', a, babab)$$

e

$$(abbaa, q, a, ababb) \underset{M}{\vdash} (abbaaa, q'', a, babb)$$

11.0.3 Computazioni

Definizioni

- La configurazione iniziale C0 di una macchina di Turing non deterministica M sull'input w è definita come nel caso deterministico,
- una sequenza

  $C_0 \vdash_M, C_1 \vdash_M \ldots C_n \vdash /_M$

  si dirà una computazione convergente di M sull'input w.
- Diremo che M accetta w se esiste una computazione convergente di M sull'input w.

Osservazione
Quindi M accetta w se almeno una computazione di M su w converge, M rifiuta w se tutte le computazioni di M su w divergono.

11.0.4 Complessità temporale per macchine di Turing non deterministiche

Definizione
Sia M una macchina di Turing non deterministica. Per ogni parola w accettata da M, denotiamo con t_w il minimo numero di passi di una computazione convergente di M su input w. Si dice complessità temporale di M la funzione $c_M : N \mapsto N$ definita come segue: per ogni $n \in N$

$$c_M(n) = \max\{t_w | waccettatada M_1, l(w) <= n\}.$$

Osservazione
Ovviamente, una macchina di Turing deterministica si può considerare come una particolare macchina di Turing non deterministica in cui, per ogni input, c'è un'unica computazione.
In tal caso, la definizione di $c_M(n)$ coincide con quella già data in precedenza.

11.0.5 Macchine e linguaggi

Definizione
L'insieme delle parole accettate da una macchina di Turing non deterministica M si dirà il linguaggio accettato da M.

Proposizione
I problemi accettati da macchine di Turing non deterministiche sono semi-decidibili.

Dimostrazione
Una macchina di Turing deterministica può simulare, parallelamente, le computazioni di una macchina di Turing non deterministica su un dato input.
Tale macchina accetterà se almeno una delle computazioni della macchina non deterministica termina.

11.0.6 La classe NP

Definizione
Denotiamo con NP la classe dei problemi S accettati da una macchina di Turing non deterministica M tale che $c_M(n) = O(n^k)$ per qualche intero positivo k.

Osservazione
Se $S \in NP$, allora ci sono una macchina di Turing non deterministica M, e due interi positivi c e k tali che, per ogni input w di lunghezza n, M si comporta nel modo seguente:

- se $w \in S$, allora esiste almeno una computazione convergente M su w con al più c l(w) k passi

- se $w \notin S$, allora tutte le computazioni di M su w divergono.

Osservazione
Riferita ai problemi, non agli algoritmi.

Osservazione
Come quella di P, anche la definizione di NP è robusta:
se esiste una procedura non deterministica che accetta le parole di S e richiede l'esecuzione di $O(n^k)$ istruzioni (comunque specificate, purchè simulabili da una macchina di Turing in tempo polinomiale), allora $S \in NP$.

11.0.7 SAT

Il problema SAT è accettato dalla seguente procedura non-deterministica:
Dato un sitema di clausole S

1. assegna (non deterministicamente) dei valori di verità alle variabili
2. se tutte le clausole sono soddisfatte accetta, altrimenti divergi

Ciascuna esecuzione del primo passo richiede tempo lineare (in termini di operazioni elementari)
Il secondo passo si può eseguire in tempo quadratico.

$$\textbf{SAT} \in \textbf{NP}$$

Osservazione
Il passo 1 può essere eseguito in 2^k modi diversi, dove k è il numero delle variabili.

11.0.8 3COL

Il problema 3COL è accettato dalla seguente procedura non-deterministica:
Dato un grafo G

1. assegna (non deterministicamente) dei colori ai vertici
2. verifica se, per ogni lato, i vertici agli estremi hanno colore diverso
3. in caso affermativo accetta, altrimenti divergi.

Ciascuna esecuzione del primo passo richiede tempo lineare (in termini di operazioni elementari, rispetto al numero dei vertici)
Il secondo passo si può eseguire in tempo quadratico.

3COL $\in$ NP

Osservazione
Il passo 1 può essere eseguito in 3^k modi diversi, dove k è il numero dei vertici.

11.0.9 Grafi

Sia G = (V,E) un grafo e $V_0 \subseteq V$
L'insieme V_0 è indipendente se per ogni v, $v' \in V_0$, $(v, v') \notin E$.
L'insieme V_0 è una clique se per ogni v, v' $\in V_0$, (v, v') $\in$ E.
L'insieme V_0 è un ricoprimento di vertici se per ogni (v, v') $\in$ E almeno uno dei vertici v,v' sta in V_0.

Problema IS
Input: Un grafo G = (V ,E) e un intero $k > 0$
Output: si se G ha un insieme
indipendente di ordine k, no altrimenti.

Problema CLIQUE
Input: Un grafo G=(V,E) e un intero $k > 0$
Output: si se G ha una clique di ordine k, no altrimenti.

Problema VC
Input: Un grafo G = (V,E) e un intero $k > 0$
Output: si se G ha un ricoprimento di vertici di ordine k, no altrimenti.

11.0.10 Grafi-2

Proposizione
Sia G = (V , E) un grafo e $V_0 \subseteq V$
Le seguenti condizioni sono equivalenti:

1. V_0 è indipendente,
2. V_0 è una clique del grafo complementare $\bar{G} = (V, \bar{E})$
3. V / V_0 è un ricoprimento di vertici.

Corollario
Ognuno dei problemi IS, CLIQUE e VC si riduce agli altri due.
Inoltre la funzione di riduzione si calcola rapidamente.

Dimostrazione
Sia G un grafo di ordine n. Allora.

$$(G, k) \in IS \iff (\overline{G}, k) \in CLIQUE \iff (G, n - k) \in VC.$$

Osservazione
I sottoinsiemi di V di ordine k sono $\begin{pmatrix} n \\ k \end{pmatrix}$ e quindi non polinomialmente limitati da n (se k non è fissato).

11.0.11 Grafi-3

Il problema IS è accettato dalla seguente procedura non-deterministica:
Dato un grafo G e un intero $k > 0$.

1. seleziona (non deterministicamente) k vertici
2. verifica se, per ogni coppia di vertici selezionati v, v', (v, v') $\notin$ E
3. in caso affermativo accetta, altrimenti divergi.

Ciascuna esecuzione del primo passo richiede tempo polinomiale (in termini di operazioni elementari, rispetto al numero dei vertici) come pure il secondo passo.
In maniera simile, si trovano procedure non-deterministiche che accettano CLIQUE e VC, con la medesima complessità:

IS, CLIQUE, VC $\in$ NP

11.0.12 coNP

Definizione
coNP è la classe dei problemi il cui complemento sta nella classe NP.

Osservazione
Mentre P = coP, non è noto se NP = coNP.

Proposizione
$P \subseteq NP \cap coNP$

11.0.13 Un'altra definizione di Np

Proposizione
Sia $S \subseteq A^*$ un problema. Le seguenti condizioni sono equivalenti

1. Esistono una macchina di Turing non deterministica M e un intero $k > 0$ tali che M accetta S e $c_M(n) = O(n^k)$
2. Esistono un problema $T \subseteq A^* x B^*$ e un polinomio p(n) tali che $T \in P$ e per ogni $w \in A^*$

$$w \in S \iff \exists y \in B^* \ (w, y) \in T, \ \ell(y) \leq p(\ell(w)).$$

Osservazione
In altri termini, ogni problema di classe NP sull'alfabeto A si ottiene da un sottoinsieme di classe P di $A^* x B^*$ con le due operazioni seguenti:

1. Si scartano le coppie (w, y) con y' molto più lungo di w
2. si proietta su A^*

Sia S un problema che verifica la condizione 2
Per ogni $w \in S$, ci sarà una parola y tale che $(w, y) \in T$ e $l(y) <= p(l(w))$.
Chiameremo y il testimone (o soluzione) di w.

11.0.14 Il testimone

Sia S un problema che verifica la condizione 2.
Per ogni $w \in S$, ci sarà una parola y tale che (w, y) ∈ T e l(y)<=p(l(w)). Chiameremo y il testimone (o soluzione) di w.

Osservazione
La limitazione l(y)<= p(l(w)) serve ad assicurare che la condizione (w, y) ∈ T possa essere accertata in tempo polinomiale rispetto a l(w). Invero, T è accettato da una macchina di Turing deterministica M tale che $c_M(n) <= q_1(n)$, per un opportuno polinomio a coefficienti non negativi q_1. Sia $q_2(n) = q_1(n + (p(n)))$. Se

$$(\text{w, y}) \in \text{T, l(y)} <= \text{p(l(w))},$$

Allora M converge su (w, y) in al più $q_1(l(w) + l(y)) <= q_2(l(w))$ passi.

Esempi

$$T = \{(S, v) \mid S \text{ sistema di clausole, } v \text{ soluzione di } S\} \in P$$
$$S \in \text{SAT} \iff \exists v \ (S, v) \in T, \ \ell(v) \leq \ell(S).$$

$$T = \{(G, c) \mid G \text{ grafo, } c \text{ 3-colorazione dei vertici di } G\} \in P$$
$$G \in \text{3COL} \iff \exists c \ (G, c) \in T, \ \ell(c) \leq 2\ell(G).$$

$$T = \{(G, k, V_0) \mid G \text{ grafo, } k \geq 0, \ V_0 \text{ clique di } G \text{ di ordine } k\} \in P$$
$$(G, k) \in \text{CLIQUE} \iff \exists V_0 \ (G, k, V_0) \in T, \ \ell(V_0) \leq \ell((G, k)).$$

11.0.15 Dimostrazione

1 ⇒ 2

Esistono una macchina di Turing non deterministica M e un polinomio q(n) tali che M accetta S e $c_M(n) <= q(n)$.
Poniamo p(n) = q(n)(n + q(n) + 2).
Ogni $w \in S$ ha una computazione convergente con non più di q(l(w)) passi.
Tale computazione è descritta da una parola di lunghezza al più p(l(w)).
Poniamo

$$T = \{(w, y) \mid y \text{ computazione convergente di M su input w}\}$$

Si ha $T \in P$ e

$$w \in S \iff \exists y \ (w, y) \in T, \ \ell(y) \leq p(\ell(w)).$$

2 ⇒ 1

Esistono un problema $T \in P$ e un polinomio p(n) tali che Il problema S è accettato dalla seguente procedura non-deterministica:
Dato w

1. genera (non deterministicamente) y tale che $l(y) <= p(l(w))$
2. verifica se (w, y) ∈ T
3. in caso affermativo accetta, altrimenti divergi

Ciascuna esecuzione del primo passo richiede tempo polinomiale rispetto alla lunghezza di w.

$$w \in S \iff \exists y \ (w, y) \in T, \ \ell(y) \leq p(\ell(w)).$$

L'esecuzione del secondo passo richiede tempo polinomiale rispetto alla lunghezza di (w, y), che è maggiorata da $l(w) + p(l(w))$.
Quindi l'esecuzione del secondo passo richiede tempo polinomiale rispetto alla lunghezza di w.

Capitolo 12

NP-completezza

12.0.1 P = NP?

- **P**

  è la classe dei problemi che ammettono un algoritmo di decisione rapido (= polinomiale)

- **NP**

  è la classe dei problemi che ammettono un algoritmo di verifica rapido (= polinomiale)

Quindi P $\subseteq$ NP significa la verifica è più rapida della decisione.
Ma P = NP significherebbe che se c'è una procedura rapida di verifica per S, ce n'è una di decisione, veloce anch'essa.

- P è la classe dei problemi accettati da macchine di Turing deterministiche in tempo polinomiale

- NP è la classe dei problemi accettati da macchine di Turing non deterministiche in tempo polinomiale

Ogni macchina di Turing non deterministica è simulata da una macchina di Turing deterministica, ma la complessità cresce esponenzialmente. Si può fare meglio?

Osservazione
Se trovo un problema S di classe NP che non sta in P, allora P $\neq$ NP, Per dimostrare P = NP, dovrei dimostrare che tutti i problemi S $\in$ NP sono di classe P. Vedremo invece che esiste una famiglia di problemi di classe NP (tra i quali, p.es., SAT) tale che se anche uno solo di essi appartiene a P, allora P = NP.

12.0.2 Riduzioni polinomiali

Definizione
Siano S, T due problemi su alfabeti A, B, rispettivamente.
Una riduzione polinomiale di S a T è una funzione totale $f : A^* \mapsto B^*$ tale che calcola-

$$w \in S \iff f(w) \in T,$$

ta da una macchina di Turing deterministica con complessità temporale polinomiale.
Se esiste una riduzione polinomiale di S a T, scriveremo $S <=_p T$.

Osservazione
La relazione $<=_p$ è riflessiva e transitiva (pre-ordine).

Esempio
2COL $<=_p$ 2SAT, IS $<=_p$ CLIQUE $<=_p$ VC $<=_p$ IS.

Proposizione
Siano S, T due problemi. Se S $<=_p$ T e T $\in$ P (risp., T $\in$ NP), allora S $\in$ P (risp., S $\in$ NP).

12.0.3 Problemi NP-ardui

Definizione
Un problema T si dice NP-arduo se, per ogni S $\in$ NP si ha S $<=_p$ T. Un problema T si dice NP-completo se è NP-arduo e appartiene alla classe NP.

Proposizione
Sia S un problema NP-completo. Se S $\in$ P, allora P = NP.

12.0.4 Un problema NP-completo

Si consideri il seguente problema LHalt:

Input: una macchina di Turing non deterministica M, un input w di M, un'altra parola u;
Output: SI se una computazione di M converge su w in al più l(u) passi, NO altrimenti.

Proposizione
Si ha LHalt $\in$ NP.

Dimostrazione
Il problema è accettato dalla seguente procedura non deterministica in tempo polinomiale:

1. calcolare k = l(u)
2. eseguire (non deterministicamente) M su w per un massimo di k passi; se M non si arresta, proseguire con un ciclo infinito.

Proposizione
Il problema LHalt è NP-arduo

Dimostrazione
Sia S $\in$ NP. Allora esiste una macchina di Turing non deterministica M che accetta S e tale che

$$c_M(n) <= P(n)$$

per un opportuno polinomio P. La funzione f definita da è una riduzione polinomiale

$$w \mapsto \left(M, w, 1^{P(\ell(w))}\right)$$

di S a LHalt.
Data l'arbitrarietà di S, possiamo concludere che LHalt è NP-arduo. Dalle proprietà precedenti segue immediatamente che

Corollario
LHalt è NP-completo.

12.0.5 Il teorema di Cook-Levin

Teorema (Cook-Levin, 1971)
SAT è NP-completo.

Proposizione
Si ha SAT $\in$ NP.

Dimostrazione
L'insieme è di classe P e si ha S $\in$ SAT $\Longleftrightarrow$(S,V) $\in$ SATsol per un opportuno V.

$$\mathsf{SATsol} = \{(S, V) \mid S \text{ sistema di clausole, } V \text{ clausola che soddisfa } S\}$$

Proposizione

SAT è NP-arduo.

Dimostrazione

Sia S $\in$ NP. Allora esiste una macchina di Turing non deterministica M che

$$c_M(n) <= p(n)$$

per un opportuno polinomio p. Costruiremo una riduzione polinomiale di S a SAT.

12.1 Il Teorema di Cook-Levin - Approfondito

Si consideri il seguente problema LHalt:
INPUT: una macchina di Turing non deterministica M, un input w di M, un'altra parola u;
OUTPUT: SI se M converge su w in al più l(u) passi, NO altrimenti.
Vedremo che LHalt è un problema NP-completo.

Proposizione 1
Si ha LHalt $\in$ NP.

DIMOSTRAZIONE
Una procedura non deterministica che accetta LHalt in tempo polinomiale è la seguente:

1. calcolare k = l(u)
2. eseguire (non deterministicamente) M su w per un massimo di k passi; se M non si arresta, proseguire con un ciclo infinito.

Chiaramente, si ha (M, w, u) $\in$ LHalt se e solo se una delle possibili esecuzioni della suddetta procedura termina.
Poiché la simulazione è limitata a k = l(u) passi, la classe di complessità è NP.

Proposizione 2
Il problema LHalt è **NP-arduo.**
DIMOSTRAZIONE
Sia S $\in$ NP. Allora esiste una macchina di Turing non deterministica M che accetta S e tale che

$$c_M(n) <= P(n)$$

per un opportuno polinomio P. La funzione f definita da:

$$w \mapsto, w, 1^{P(l(w)}$$

è una riduzione di S a LHalt. Il calcolo di tale funzione si riduce al calcolo di $k = P(l(w))$ e alla concatenazione di M, w e 1^k. Si calcola pertanto in tempo costante. Data l'arbitrarietà di S, possiamo concludere che ogni problema di classe NP si riduce a LHalt in tempo polinomiale (anzi, costante). Questo dimostra che LHalt è NP-arduo.
Dalle proprietà precedenti segue immediatamente che

Corollario 1
LHalt è NP-completo.
Vedremo ora un esempio 'naturale' di problema NP-completo.

Teorema 1 (Cook-Levin, 1971)
SAT è NP-completo. Per dimostrare il Teorema di Cook-Levin dobbiamo verificare che SAT $\in$ NP (Proposizione 3) e che è un problema NP-arduo (Proposizione 4).

Proposizione 4
SAT è NP-arduo.

Dimostrazione
Sia S $\in$ NP. Allora esiste una macchina di Turing non deterministica M che accetta S e tale che

$$c_M(n) <= p(n)$$

per un opportuno polinomio p.
Detti Q, A e I rispettivamente l'alfabeto, l'insieme degli stati e l'insieme delle istruzioni di M, poniamo

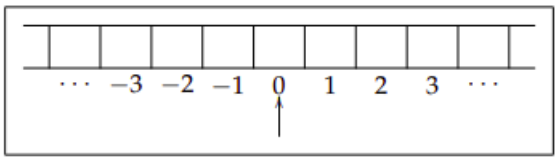
$$A = \{a_0, a_1, ..., a_m\}, Q = \{q_0, q_1, ..., q_h\}, I = \{\alpha_0, \alpha_1, ..., \alpha_l\}$$

con $a_0 = \#, m, h, l >= 0$. Senza perdita di generalità, possiamo ridurci al caso in cui M si arresta se e solo se entra nello stato q_h. Possiamo inoltre supporre che risulti

$$p(n) >= n$$

per ogni $n >= 0$ Passiamo ora alla costruzione di una riduzione di S a SAT. Dobbiamo costruire una funzione totale f che associa a ogni parola w $\in A^*$ un sistema di clausole

f(w) in modo tale che f(w) sia soddisfacibile se e solo se w ∈ S. Evidentemente, tale sistema dovrà in qualche modo 'rappresentare' le computazioni di M.
Poniamo T = p(l(w)). È possibile associare un indice a ogni cella del nastro di M nel modo seguente:
la cella in cui si trova inizialmente la testina ha indice 0, quella alla sua destra 1, la successiva 2, e così via, mentre le celle alla sinistra della cella di indice 0 avranno gli indici -1, -2, ecc Si osservi che in una computazione che duri non più di T passi,



la testina potrà visitare solo le celle di indice compreso tra -T e T. Potremo quindi disinteressarci delle celle al di fuori di tale intervallo.
Definiamo i letterali del nostro sistema e contemporaneamente una valutazione, detta valutazione standard associata a una generica computazione di M. Ora cerchiamo di

variabili		valutazione standard
c_{ijt}	$-T \le i \le T$ $0 \le j \le m$ $0 \le t \le T$	$V(c_{ijt}) = 1$ se al passo t nella cella i c'è a_j
s_{rt}	$0 \le r \le h$ $0 \le t \le T$	$V(s_{rt}) = 1$ se al passo t lo stato è q_r
d_{it}	$-T \le i \le T$ $0 \le t \le T$	$V(d_{it}) = 1$ se al passo t la testina è sulla cella di indice i
b_{kt}	$0 \le k \le l$ $0 \le t \le T$	$V(b_{kt}) = 1$ se al passo t la computazione usa l'istruzione α_k

costruire il nostro sistema f(w) in modo tale che sia soddisfatto esclusivamente dalle valutazioni standard delle computazioni di M. Iniziamo introducendo le clausole.

$$\bar{c}_{ij_1t}\,\bar{c}_{ij_2t}\,, \qquad 0 \le t \le T\,,\ -T \le i \le T,\ 0 \le j_1 < j_2 \le m,$$

Una valutazione V soddisfa tali clausole se e solo se comunque fissati i e t c'è al più un indice j per cui $V_{(cijt)} = 1$.

Ora aggiungiamo al nostro sistema le clausole

$$c_{i0t}\, c_{i1t} \ldots c_{imt}\,, \qquad 0 \leq t \leq T\,,\ -T \leq i \leq T\,.$$

Una valutazione V soddisfa tali clausole se e solo se comunque fissati i e t c'è almeno un indice j per cui $V_{(cijt)} = 1$. In conclusione una valutazione V soddisfa le clausole precedenti se e solo se per ogni i e t, esiste uno e un solo j tale che $V_{(cijt)} = 1$. In altri termini, tale condizione esprime il fatto che a ogni passo di una computazione di una macchina di Turing ogni cella del nastro contiene una e una sola lettera.
Allo stesso modo, aggiungiamo al nostro sistema le clausole.
Aggiungiamo ancora le clausole

$$\begin{array}{ll}
\overline{d}_{i_1t}\,\overline{d}_{i_2t}\,, & 0 \leq t \leq T\,,\ -T \leq i_1 < i_2 \leq T, \\
d_{-Tt}\, d_{-T+1\,t} \cdots d_{Tt} & 0 \leq t \leq T\,, \\
\overline{s}_{r_1t}\,\overline{s}_{r_2t}\,, & 0 \leq t \leq T\,,\ 0 \leq r_1 < r_2 \leq h\,, \\
s_{0t}s_{1t} \cdots s_{ht}\,, & 0 \leq t \leq T\,.
\end{array}$$

Tali clausole assicurano che se V è una valutazione che soddisfa il sistema, allora, per ogni t, ci sono esattamente un indice i e un indice r tale che $V_{(dit)} = 1$ e $V_{(srt)} = 1$.
In altri termini queste clausole esprimono l'unicità della posizione della testina e dello stato a ogni istante di una computazione di una macchina di Turing.
Aggiungiamo ancora le clausole

$$\begin{array}{ll}
\overline{b}_{k_1t}\,\overline{b}_{k_2t} & 0 \leq t \leq T\,,\ 0 \leq k_1 < k_2 \leq l, \\
b_{0t}b_{1t} \cdots b_{lt}s_{h,t} & 0 \leq t \leq T\,.
\end{array}$$

Tali clausole assicurano che se V è una valutazione che soddisfa il sistema, allora, per qualsiasi t tale che $V_{(sht)} = 0$, c'è esattamente un indice k tale che $V_{(bkt)} = 1$. Ciò esprime il fatto che una MdT applica un'unica istruzione a ogni passo di computazione fino a un eventuale arresto. Introduciamo ora le clausole che esprimono il fatto che il contenuto di una cella non viene modificato quando la testina non si trova su quella cella.
Ciò si ottiene con le clausole

$$\bar{c}_{ijt}c_{ij\,t+1}d_{it}, \qquad -T \leq i \leq T,\ 0 \leq j \leq m, 0 \leq t \leq T$$

Inoltre per ogni istruzione$\alpha_k = (q_{r1}, a_{j1}, q_{r2}, a_{j2}, \varepsilon), 0 <= k <= l$, introduciamo le clausole: Tali clausole esprimono il fatto che lo stato, il contenuto della cella sotto la

$$\begin{array}{ll} \bar{b}_{kt}s_{r_1 t}, \quad \bar{b}_{kt}s_{r_2\,t+1}, & 0 \leq t \leq T, \\ \bar{b}_{kt}\bar{d}_{it}c_{ij_1 t}, \quad \bar{b}_{kt}\bar{d}_{it}c_{i\,j_2\,t+1}, \quad \bar{b}_{kt}\bar{d}_{it}d_{i+\epsilon\,t+1}, & -T \leq i \leq T,\ 0 \leq t \leq T. \end{array}$$

testina e la posizione della testina cambiano in accordo con l'istruzione utilizzata all'istante t. Ancora posto $w = a_{j0}, a_{j1}, ..., a_{jn-1}$, introduciamo le clausole che esprimono

$$\begin{array}{ll} s_{00}, \ d_{00}, \ c_{0j_0 0}, \ c_{1j_1 0}, \ \ldots, \ c_{n-1j_{n-1}0}, & \\ c_{i00}, \qquad -T \leq i < 0 \quad \text{o} \quad n \leq i \leq T, & \end{array}$$

le condizioni iniziali della computazione e la clausola

$$s_{h0}, s_{h1}, ..., s_{hT}$$

che esprime il fatto che la computazione si arresta in tempo $<= T$.
Dalla costruzione segue che una valutazione V soddisfa f(w) se e solo se V è la valutazione standard associata a una computazione di M su w che converge in non più di T passi. Ne segue che w ∈ S se e solo se f(w) ∈ SAT. Quindi f è una riduzione di S a SAT. Osserviamo poi che f(w) è costituito da $O(T^2)$ clausole ciascuna di lunghezza O(T). Ne segue che il calcolo di f richiede tempo polinomiale. Possiamo così concludere che S $<=_p$ SAT.
Data l'arbitrarietà di S, abbiamo dimostrato che ogni problema di classe NP si riduce a SAT in tempo polinomiale. Questo dimostra che SAT è NP-arduo

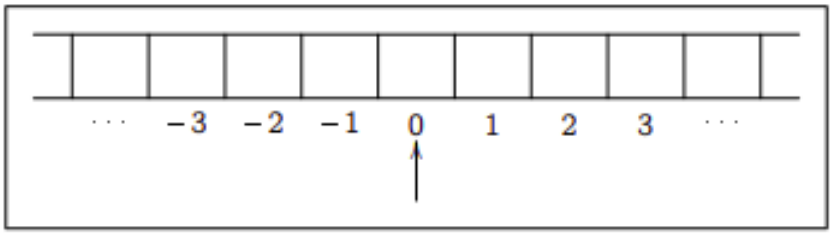
12.1.1 Riduzione di S e SAT

Siano

$$\text{A} = \{a_0, a_1, ..., a_m\} \text{ , Q} = \{q_0, q_1, ..., q_h\} \text{ , I} = \alpha_0, \alpha_1, ..., \alpha_l.$$

rispettivamente l'alfabeto, l'insieme degli stati e l'insieme delle istruzioni di M, con $a_0 = \#$.
Possiamo supporre che M si arresta se e solo se entra nello stato q_h e che $p(n) >= n$, per ogni $n >= 0$.
Dobbiamo costruire una funzione totale f che associa a ogni parola w $\in$ A* un sistema di clausole f(w) in modo tale che f(w) sia soddisfacibile se e solo se w $\in$ S.
Faremo in modo di 'rappresentare' le computazioni di M nelle clausole.

Poniamo T = p(l(w))



12.1.2 Variabili

Definiamo le variabili del nostro sistema e contemporaneamente una valutazione, detta valutazione standard associata a una generica computazione di M.

12.1.3 Clausole

Costruiamo il sistema f (w) in modo tale che sia soddisfatto esclusivamente dalle valutazioni standard delle computazioni convergenti di M. sono soddisfatte se e solo se per ogni i e t c'è uno e un solo j tale che $V(c_{ijt}) = 1$: ogni cella, a ogni istante contiene una e una sola lettera. a ogni istante c'è un'unica la posizione della testina e un unico stato.

	variabili	valutazione standard
c_{ijt}	$-T \leq i \leq T$ $0 \leq j \leq m$ $0 \leq t \leq T$	$V(c_{ijt}) = 1$ se al passo t nella cella i c'è a_j
s_{rt}	$0 \leq r \leq h$ $0 \leq t \leq T$	$V(s_{rt}) = 1$ se al passo t lo stato è q_r
d_{it}	$-T \leq i \leq T$ $0 \leq t \leq T$	$V(d_{it}) = 1$ se al passo t la testina è sulla cella di indice i
b_{kt}	$0 \leq k \leq l$ $0 \leq t \leq T$	$V(b_{kt}) = 1$ se al passo t la computazione usa l'istruzione α_k

$$\overline{c}_{ij_1t}\,\overline{c}_{ij_2t}, \qquad 0 \leq t \leq T,\ -T \leq i \leq T,\ 0 \leq j_1 < j_2 \leq m,$$
$$c_{i0t}\,c_{i1t}\,\cdots\,c_{imt}, \qquad 0 \leq t \leq T,\ -T \leq i \leq T.$$

$$\overline{d}_{i_1t}\,\overline{d}_{i_2t}, \qquad 0 \leq t \leq T,\ -T \leq i_1 < i_2 \leq T,$$
$$d_{-Tt}\,d_{-T+1\,t}\,\cdots\,d_{Tt} \qquad 0 \leq t \leq T,$$
$$\overline{s}_{r_1t}\,\overline{s}_{r_2t}, \qquad 0 \leq t \leq T,\ 0 \leq r_1 < r_2 \leq h,$$
$$s_{0t}\,s_{1t}\cdots s_{ht}, \qquad 0 \leq t \leq T.$$

12.2 Altre clausole

$$\begin{array}{ll} \overline{b}_{k_1 t}\,\overline{b}_{k_2 t} & 0 \le t \le T,\ 0 \le k_1 < k_2 \le l, \\ b_{0t}\, b_{1t} \cdots b_{lt}\, s_{h,t} & 0 \le t \le T. \end{array}$$

(un'unica istruzione a ogni passo di computazione, fino a un eventuale arresto). Per ogni istruzione $a_k = (q_{r1}, a_{j1}, q_{r2}, a_{j2}, \varepsilon)$, $0 <= k <= l$, introduciamo le clausole: (all'istante t lo stato, il contenuto della cella sotto la testina e la posizione della testina

$$\begin{array}{ll} \overline{b}_{kt}\, s_{r_1 t}, \quad \overline{b}_{kt}\, s_{r_2\, t+1}, & 0 \le t \le T, \\ \overline{b}_{kt}\,\overline{d}_{it}\, c_{ij_1 t}, \quad \overline{b}_{kt}\,\overline{d}_{it}\, c_{i\, j_2\, t+1}, \quad \overline{b}_{kt}\,\overline{d}_{it}\, d_{i+\varepsilon\ t+1}, & -T \le i \le T,\ 0 \le t \le T. \end{array}$$

cambiano in accordo con l'istruzione α_t) (nessun'altra cella è modificata).

$$\overline{c}_{ijt}\, c_{ij\ t+1}\, d_{it}, \qquad -T \le i \le T,\ 0 \le j \le m, 0 \le t \le T$$

12.2.1 Ancora clausole

Posto $w = a_{j_0} a_{j_1} \cdots a_{j_{n-1}}$, introduciamo le clausole

$$s_{00}\,,\ d_{00}\,,\ c_{0j_00}\,,\ c_{1j_10}\,,\ \cdots\,,\ c_{n-1j_{n-1}0}\,,$$

$$c_{i00}\,, \qquad -T \leq i < 0 \quad \text{o} \quad n \leq i \leq T\,,$$

$$s_{h0} s_{h1} \cdots s_{hT}.$$

(configurazione iniziale e condizione di arresto)

12.2.2 Conclusione

- Una valutazione V soddisfa f (w) se e solo se V è la valutazione standard associata a una computazione di M su w che converge in al più T passi.
- Quindi w $\in$ S $\iff$ f(w) $\in$ SAT.
- Ci sono $O(T^2)$ clausole ciascuna di lunghezza O(T). Quindi f si calcola in tempo polinomiale

$$\text{S} <=_p \text{SAT}.$$

- SAT è NP-arduo.

Capitolo 13

Problemi NP-completi

13.0.1 3SAT

Proposizione
Il problema 3SAT è NP-completo.

Dimostrazione

- Chiaramente 3SAT $\in$ NP.
- Per provare che 3SAT è NP-arduo, basta SAT $<=_p$ 3SAT.
- Dobbiamo associare a ogni sistema di clausole un sistema di 3-clausole, calcolabile in tempo polinomiale, preservando soddisfacibilità e insoddisfacibilità.

Esempio
Se sostituisco la clausola $x_1, \bar{x}_2, x_3, x_4$ con le due clausole

$$x_1, \bar{x}_2 z_1, \bar{z} x_3 x_4$$

ove z è una nuova variabile, la (in)soddisfacibilità è preservata.

- Si può usare la medesima tecnica per tutte le clausole di lunghezza >= 4.

Esempio
Se sostituisco la clausola $x_1, \bar{x}_2$ con le due clausole

$$x_1, \bar{x}_2 y, x_1, \bar{x}_2 \bar{y}$$

ove y è una nuova variabile, la (in)soddisfacibilità è preservata.

- Si può usare la medesima tecnica per tutte le clausole di lunghezza 2 e anche di lunghezza 1.

Conclusione
Con le regole precedenti si ottiene una riduzione di SAT a 3SAT. È facile convincersi che e la riduzione si calcola in tempo polinomiale. Quindi SAT $<=_p$ 3SAT.

13.0.2 Insieme indipendente

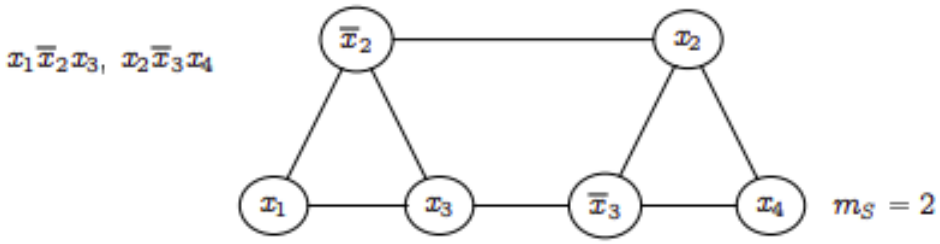
Proposizione
Il problema IS è NP-completo

Dimostrazione

- Si è già visto che IS $\in$ NP
- Per provare che IS è NP-arduo, basta 3SAT $<=_p$ IS.
- Dobbiamo associare a ogni sistema di 3-clausole S un grafo G_S e un intero m_S , calcolabili in tempo polinomiale, tale che

$$S \in \mathsf{3SAT} \iff (G_S, m_S) \in IS.$$

Esempio



13.0.3 Ridurre 3SAT a IS

- 3 vertici per ogni clausola (corrispondenti ai 3 letterali)
- I lati connettono i 3 letterali di ogni clausola e ogni vertice con letterale x a tutti i vertici con letterale $\bar{x}$
- m_S = numero delle clausole

Osservazione
Un insieme di m_S vertici indipendenti contiene esattamente un vertice di ogni 'triangolo' e non contiene vertici etichettati con un letterale e il suo opposto.

Conclusione
Con la costruzione precedente si ottiene una riduzione di 3SAT a IS. È facile convincersi che la riduzione si calcola in tempo polinomiale. Quindi SAT $<=_p$ 3SAT.

13.0.4 CLIQUE e VC

Proposizione
CLIQUE e VC sono NP-completi.

Dimostrazione

- Si è già visto che questi due problemi sono in NP.
- Inoltre IS $<=_p$ VC e IS $<=_p$ CLIQUE

13.0.5 3COL

Proposizione
Il problema 3COL è NP-completo.

Dimostrazione

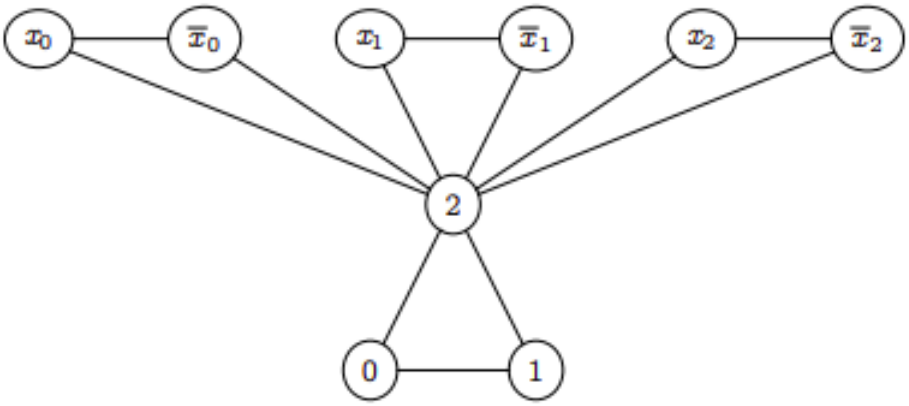
- Si è già visto che 3COL $\in$ NP.

- Per provare che 3COL è NP-arduo, basta 3SAT $<=_p$ 3COL.
- Dobbiamo associare a ogni sistema di 3-clausole S un grafo G_S calcolabile in tempo polinomiale, tale che

$$S \in \text{3SAT} \iff G_S \in \text{3COL}.$$

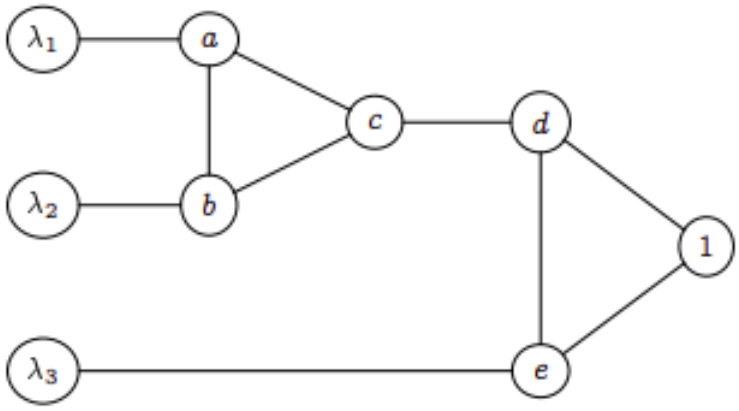
13.0.6 Ridurre 3SAT a 3COL

- 3 vertici tutti collegati fra loro (chiamiamoli 0,1,2)
- Un vertice per ogni letterale, tutti collegati al vertice 2, inoltre ogni letterale collegato alla sua negazione



13.0.7 Ridurre 3SAT a 3COL

per ogni clausola $\lambda_1, \lambda_2, \lambda_3$, 5 nuovi vertici e le connessioni



Osservazioni

- Assegnamo i colori 0,1,2 ai vertici 0,1,2
- I letterali avranno colore 0 o 1 e ognuno colore diverso dalla sua negazione
- Se e ha colore 0, allora $\lambda_3 = 1$
- Se invece e ha colore 2, allora, d ha colore 0, una fra a e b ha colore 0, uno fra $\lambda_1, \lambda_2, \lambda_3$ ha colore 1.
- Quindi almeno uno tra $\lambda_1, \lambda_2, \lambda_3$ ha colore 1, si possono assegnare colori coerenti ai vertici, $a,\ b,\ c,\ d,\ e$

Conclusione
Se G_S ha una 3-colorazione, un letterale di ogni clausola ha colore 1: i letterali di colore 1 soddisfano S Viceversa se S $\in$ 3SAT, G_S ha una 3-colorazione coi letterali veri di colore 1 e i letterali falsi di colore 0.

13.0.8 Problemi NP-intermedi

Definizione
Un problema è NP-intermedio se appartiene alla classe NP-P, ma non è NP-completo. Non è noto se esistano problemi NP-intermedi (potrebbero non esistere anche se P $\neq$ NP)

Candidati

- Isomorfismi di grafi:

  Input: Due grafi G e G'
  Output: Si se esiste un isomorfismo tra G e G', NO altrimenti

- Decomposizione di un intero in fattori primi
- Problemi radi
- Problemi unari

13.0.9 Problemi NP-intermedi

Definizione
Un problema S è rado se il numero delle parole di S di lunghezza n è polinomialmente limitato. Un problema S $\subseteq$ a* si dice unario.

Proposizione
Se esiste un problema rado in NP $-$ P, allora è necessariamente un problema NP-intermedio.

Proposizione
Se esiste un problema rado in NP $-$ P, allora ne esite uno unario.

13.0.10 Congettura di Berman-Hartmanis

Definizione

Due problemi S, T su alfabeti A, B sono P-isomorfi se c'è una biiezione $f : A* \mapsto B*$ tale che

- $fef^{-}1$ sono computabili in tempo polinomiale
- per ogni $w \in A*$, si ha w $\in$ S $\iff$ f(w) $\in$ T.

Congettura

Due problemi NP-completi sono sempre P-isomorfi.

Capitolo 14

Il criptosistema Merkle-Hellman

14.1 Il problema dello zaino (KNAPSACK

Input: Gli interi positivi $v_1, v_2, ...v_n$
Output: SI/NO secondo che esista una sequenza $e1, e_2...., e_n$ di bit 0,1 per cui:

$$\sum_{i=1}^{n} e_i v_i = V$$

Teorema
l problema KNAPSACK è NP-completo

Dimostrazione

- KNAPSACK $\in$ NP. Invero

$$\{(V, v_1, v_2, \ldots, v_n, e_1, \ldots, e_n) \mid \sum_{i=1}^{n} e_i v_i = V\} \in \mathsf{P}$$

- Per provare KNAPSACK è NP-arduo, basta mostrare che 3SAT $<=_p$ KNAPSACK.

- Dobbiamo associare a ogni sistema di 3-clausole un'istanza di KNAPSACK, calcolabile in tempo polinomiale, in modo che sistemi soddisfacibili siano associati a elementi di KNAPSACK, e sistemi insoddisfacibili a elementi del suo complemento.

Esempio

$$c_0 = x_0 x_1 \overline{x}_2, \quad c_1 = \overline{x}_1 x_2 \overline{x}_3$$

	x_0	x_1	x_2	x_3	c_0	c_1
x_0	1	0	0	0	1	0
$\overline{x}_0$	1	0	0	0	0	0
x_1	0	1	0	0	1	0
$\overline{x}_1$	0	1	0	0	0	1
x_2	0	0	1	0	0	1
$\overline{x}_2$	0	0	1	0	1	0
x_3	0	0	0	1	0	0
$\overline{x}_3$	0	0	0	1	0	1
					1	0
					1	0
					0	1
					0	1
	1	1	1	1	3	3

(numeri decimali)

Selezioniamo le righe corrispondenti ai letterali di una valutazione che soddisfa il sistema:
Ci sarà esattamente un 1 in ogni colonna x_i e da uno a tre 1 nelle colonne c_j. Aggiungendo qualcuna delle righe rimanenti otteniamo la soluzione di KNAPSACK
Viceversa, una soluzione di KNAPSACK seleziona le righe corrispondenti ai letterali di una valutazione che soddisfa il sistema, più qualcuna delle righe ulteriori

E' una riduzione polinomiale

14.1.1 Crittografia

Due processi

1. Il mittente cifra il suo messaggio **(codifica)**
2. Il destinatario decifra il testo originale **(decodifica)**

Esempio
Codice cesareo

- **Codifica:** $A \mapsto D, B \mapsto E, C \mapsto F, D \mapsto G, ...$
- **Decofica:** $A \mapsto U, B \mapsto V, C \mapsto Z, D \mapsto A, ...$

  Lo scambio di chiavi è un aspetto critico

Crittografia chiave pubblica

- Ogni utente dispone di due chiavi
  - Una pubblica, con cui gli interlocutori cifrano i messaggi a lui destinati
  - Una privata (segreta e personale) che serve per decifrarli
- Chiunque può codificare, ma uno solo può decodificare
- Dalla funzione di codifica f non è facile calcolare la funzione di decodifica f^{-1}

14.1.2 Successioni supercrescenti

Definizione

Una sequenza di interi $v_1, v_2, ..., v_n$ si dice supercrescente se Nel caso di sequenze

$$v_i > \sum_{j=1}^{i-1} v_j, \quad i = 2, \ldots, n.$$

supercrescenti, la soluzione di KNAPSACK è facile:

```
1  for j = n, ..., 1
2      if V ≥ v_j
3          e_j = 1
4          V ← V − v_j
5      else
6          e_j = 0
7  if V = 0 then accetta else rifiuta
```

Esempio

La successione 2, 3, 7, 15, 31, 62 è supercrescente.
Vediamo se $(24, 2, 3, 7, 15, 31, 62) \in$ KNAPSACK Poichè $24 < 62 e 24 < 31$, scartiamo 62 e 31.
Poi si ha 24 >=15 e $24 - 15 = 9$.
Proseguendo, 9 >= 7 e $9 - 7 = 2$ Ancora $2 < 3$, poi 2 >= 2 e $2 - 2 = 0$.
In conclusione, $60 = 15 + 7 + 2$

14.1.3 Creazione delle chavi

- Parto da una sequenza supercrescente $v_1, v_2, ..., v_n$, m e un intero a tale che gcd(m, a) = 1.

$$(w_1, w_2, ..., w_n)$$

con $w_i = av_i$ mod m, i = 1,..., n

- La chiave privata è

$$(a', v_1, v_2, ..., v_n)$$

con a' inverso di a modulo m.

- Il messaggio è una sequenza di n bit $e_1, ..., e_n$ che si codificherà con

$$V = \sum_{i=1}^{n} e_i w_i$$

Esempio

Se la sequenza supercrescente è 2, 3, 7, 15, 31, 62 e a = 9 allora m = 62, a' = 7 e la chiave pubblica sarà

18, 27, 1, 11, 31

Il messaggio 10011 sarà codificato con

$$18 + 11 + 31 = 60$$

14.1.4 Decodifica

- Ricordiamo che

$$V = \sum_{i=1}^{n} e_i w_i$$

- Pertanto

$$a' V = \sum_{i=1}^{n} e_i a' w_i \equiv \sum_{i=1}^{n} e_i v_i \pmod{m}.$$

  Tenendo conto del fatto che:

$$\sum_{i=1}^{n} e_i v_i < m,$$

- Conoscendo a' e V posso calcolare **la sommatoria precedente**
- Infine, per la supercrescenza delle v_i, posso ottenere rapidamente $e_1, ..., e_n$

Esempio
Con la sequenza supercrescente è 2, 3, 7, 15, 31, a = 9, m = 62, a' = 7 riceviamo il messaggio V = 60.
Per prima cosa calcoliamo a'V mod m = 7 · 60 mod 62 = 48.
Con l'algoritmo per le successioni supercrescenti otteniamo

$$48 = 31 + 15 + 2$$

e quindi $e_1, e_2, e_3, e_4, e_5 = 10011$.

14.1.5 La classe UP

Definizione

Una macchina di Turing non deterministica si dice non ambigua se ogni input al più una computazione convergente.

La classe dei problemi accettati da macchine di Turing non ambigue con complessità temporale polinomiale si denota UP.

Il problema KNAPSACK appartiene alla classe UP

- Non è noto se P = UP, né se NP = UP.

Capitolo 15

La gerarchia polinomiale

15.0.1 NP e coNP

Definizione
Ricordiamo che coP è la classe dei problemi il cui complemento appartiene alla classe P.
Poniamo:

$$\Sigma_0^P = P, \quad \Pi_0^P = \mathbf{coP}$$

Come già osservato,

$$\Sigma_0^P = \Pi_0^P$$

Definzione
Poniamo:

$$\Sigma_1^P = \mathbf{NP}, \quad \Pi_1^P = \mathbf{coNP}$$

NP = coNP?

Esempio
Come ben noto, SAT $\in$ NP:

- Per dimostrare che un sistema di clausole è soddisfacibile, basta esibire una clausola che lo soddisfa (e la lunghezza di tale clausola è polinomialmente limitata dalla dimensione del sistema)
- Per dimostrare che un sistema di clausole è insoddisfacibile, è necessario verificare che tutte le clausole non soddisfano il sistema (e il numero di tali clausole può essere esponenziale rispetto alla dimensione del sistema)

$$\text{UNSAT} \in \text{NP ?}$$

Definizione
Sia S un problema sull'alfabeto A. Si ha S $\in$ NP se esistono un problema T $\in$ P e un polinomio p_s tali che per ogni input w

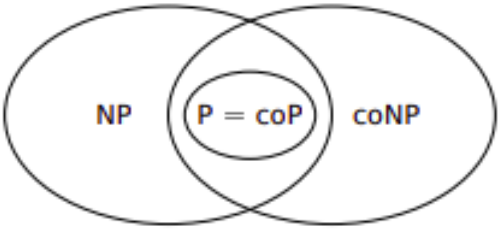
$$w \in S \Longleftrightarrow \exists y \in A* \text{ t.c. } l(y) <= p_s(l(w)), (w, y) \in T$$

Si ha S $\in$ coNP se esistono un problema T $\in$ P e un polinomio p_s tali che per ogni input w

$$w \in S \Longleftrightarrow \exists y \in A* \text{ t.c. } l(y) <= p_s(l(w)), (w, y) \in T$$

Osservazione

- P = coP $\subseteq$ coNP
- se P = NP, allora coNP = coP = P = NP.



Definizione

Sia S un problema sull'alfabeto A.

Si ha S $\in \sum_1^p$ se esistono un problema T $\in \prod_0^P$ e un polinomio p_s tali che per ogni input w

$$w \in S \Longleftrightarrow \exists y \in A* \text{ t.c. } l(y) <= p_s(l(w)), (w, y) \in T$$

Si ha S $\in \prod_1^P$ se esistono un problema T $\in \sum_0^p$ e un polinomio p_s tali che per ogni input w

$$w \in S \Longleftrightarrow \exists y \in A* \text{ t.c. } l(y) <= p_s(l(w)), (w, y) \in T$$

15.0.2 $\sum_2^p \prod_2^P$

Definizione

Sia S un problema sull'alfabeto A.

Si ha S $\in \sum_2^p$ se esistono un problema T $\in \prod_2^P$ e un polinomio p_s tali che per ogni input w

$$w \in S \iff \exists y \in A* \text{ t.c. } l(y) <= p_s(l(w)), (w, y) \in T$$

Si ha S $\in \prod_2^P$ se esistono un problema T $\in \sum_1^p$ e un polinomio p_s tali che per ogni input w

$$w \in S \iff \exists y_1 \in A^* \; \forall y_2 \in A^* \; \exists y_3 \in A^* \; \cdots \; Q_k y_k \in A^* \text{ con}$$
$$\ell(y_1), \ldots, \ell(y_k) \leq p_s(\ell(w)), \; (w, y_1, \ldots, y_k) \in T$$

15.0.3 $\sum_k^p \prod_k^P$

Sia S un problema sull'alfabeto A e $k >= 1$. Si ha S $\in \sum_k^p$ se esistono un problema T $\in \prod_p^{k-1}$e un polinomio p_s tali che per ogni input w

$$w \in S \iff \exists y_1 \in A^* \; \forall y_2 \in A^* \; \exists y_3 \in A^* \cdots Q_k y_k \in A^* \text{ con } \ell(y_1), \ldots, \ell(y_k) \leq p_s(\ell(w)), \; (w, y_1, \ldots, y_k) \in T$$

15.0.4 Osservazioni

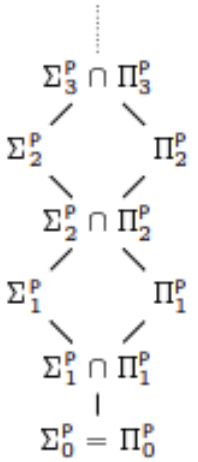
- $\prod_k^p$ = co$\sum_k^p$, k >= 0
- $\sum_k^p \subseteq \sum_{k+1}^p$, $\prod_{k+1}^P$ k >= 0
- $\sum_k^p \subseteq \sum_{k+1}^p$, $\prod_{k+1}^P$ k >= 0

Definiamo

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^p = \bigcup_{k \geq 0} \Pi_k^p.$$

La sequenza delle classi $\sum_k^p$, $\prod_k^P$, k >= 0 prende il nome di gerarchia polinomiale.

15.0.5 Gerarchia Polinomiale



Proposizione

Si ha PH = P se e solo se P = NP.

Dimostrazione

- Supponiamo PH = P. Poichè P $\subseteq$ NP $\subseteq$ PH, si ha P = NP.
- Viceversa supponiamo P = NP, i.e., $\sum_0^p = \sum_1^p$. Ne segue

$$\Sigma_0^P = \Sigma_1^P = \cdots = \Sigma_k^P = \cdots = \mathsf{PH}$$

e, di conseguenza, $\sum_1^p = \sum_2^p$. Iterando,

$$\Sigma_0^P = \Sigma_1^P = \cdots = \Sigma_k^P = \cdots = \mathsf{PH}$$

Proposizione

1. Se $\sum_k^p = \prod_k^p$ per qualche k >= 0 , allora $\sum_p^j = \prod_p^j = \sum_k^p$ per ogni j >= k. Pertanto, PH = $\sum_k^p$.
2. Se $\sum_k^p = \prod_k^p$ oppure $\prod_k^p = \prod_{k+1}^p$ per qualche k >= 0, allora $\sum_p^j = \prod_p^j = \sum_k^p$ per ogni j >= k. Pertanto, PH = $\sum_k^p$.

Definizione
Un problema S si dice $\sum_k^p$ arduo se per ogni problema T $\in \sum k$ si ha T $<_p$ S. Un problema si dice $\sum_k^p$-completo se è $\sum_k^p$-arduo e appartiene alla classe $\sum_k^p$

Proposizione
Per ogni k >= 0 esistono problemi $\sum_k^p$-completi

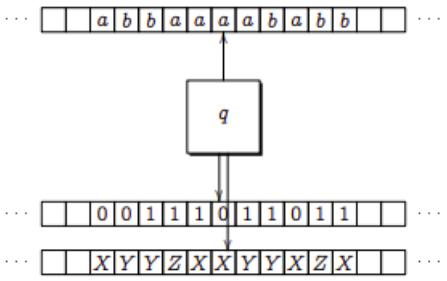
15.0.6 Tempi Esponenziali

PEXP è la classe dei problemi accettati da una macchina di Turing deterministica con complessità temporale $O(2^n)^k$, per un opportuno intero positivo k.
LEXP è la classe dei problemi accettati da una con complessità temporale $O(c^n)$, per un opportuno intero positivo c.
Le classi NPEXP e NLEXP sono definite analogamente, ma utilizzando macchine di Turing non deterministiche.

Capitolo 16

La complessità spaziale

16.0.1 Macchina di turing a piu' strati



Una macchina di Turing M a più nastri è composta da:

- un'unità di controllo a stati finiti
- un nastro N_0 di input-output di lunghezza infinita con relativa testina
  - I nastri sono suddivisi in celle, e ogni cella contiene un simbolo di un certo alfabeto A oppure il simbolo bianco #.
- Inizialmente, il nastro di input-output N_0 contiene l'input, la testina è sul simbolo più a sinistra, lo stato è q_0, gli altri nastri sono vuoti.
- Ogni successiva mossa di M è determinata da
  - stato

  - simboli letti dalle testine sugli m nastri

- consiste in:
  - aggiornare lo stato
  - modificare i simboli esaminati su ogni nastro
  - spostare in ogni nastro la testina di un passo verso destra, verso sinistra o lasciarla ferma

Definizione
Una Macchina di Turing a m nastri è una quadrupla $M = (Q, A, \sigma, q_0)$, dove:

- Q è un insieme finito di stati
- A è un alfabeto, cui si aggiunge il simbolo bianco #
- σ è una funzione parziale da Q $\times$ (A U $\{\#\}$)^m a $Qx(A \text{ U } \{\#\})mx\{-1, 0, 1\}^m$, chiamata funzione di transizione
- $q_0 \in Q$ è lo stato iniziale

Le (2m + 1)-tuple

$$(q, a_1..., a_m, q', a_1^{'}, ..., a_m^{'}, x_1, ..., x_m)$$

tali che

$\sigma(q, a_1, ..., a_m) = (q', a_1^{'}, ..., a_m^{'}, x_1, ..., x_m)$

sono dette le istruzioni di M.

Esempio

Una macchina di Turing a due nastri che accetta le palindrome

Startegia

- Copiamo l'input, capovolto, sul nastro ausiliario
- Riportiamo la testina del nastro di input/output all'inizio dell'input
- Esaminiamo i due nastri verificando se contengono la stessa parola

$$
\begin{array}{ll}
(q_0, a, \#, q_0, a, a, +1, -1) & (q_2, a, a, q_2, \#, \#, +1, +1) \\
(q_0, b, \#, q_0, b, b, +1, -1) & (q_2, b, b, q_2, \#, \#, +1, +1) \\
(q_0, \#, \#, q_1, \#, \#, -1, 0) & (q_2, \#, \#, q_2, Y, \#, 0, 0) \\
(q_1, a, \#, q_1, a, \#, -1, 0) & (q_2, a, b, q_3, N, \#, +1, 0) \\
(q_1, b, \#, q_1, b, \#, -1, 0) & (q_2, b, a, q_3, N, \#, +1, 0) \\
(q_1, \#, \#, q_2 \#, \#, +1, +1) & (q_3, a, \#, q_3, \#, \#, +1, 0) \\
 & (q_3, b, \#, q_3, \#, \#, +1, 0)
\end{array}
$$

Osservazione

Con due nastri, il tempo necessario alla verifica è 3n, con un nastro,$n(n-1)/2$

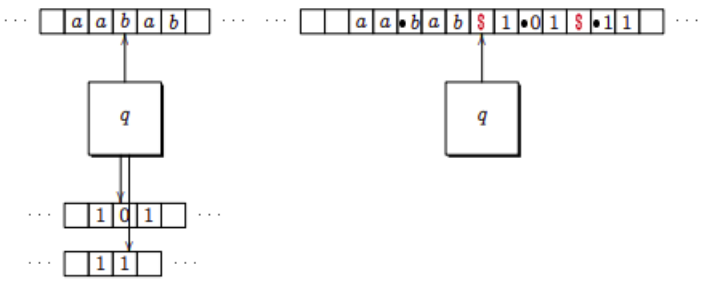
16.0.2 Equivalenza tra modelli

Teorema

Per ogni macchina di Turing M= (Q, A, σ, q_0) con m nastri ausiliari, esiste una macchina di Turing $M' = (Q', A', \sigma', q_0')$ tale che

- per ogni linguaggio L $\subseteq$ A, L è accettato o deciso da M se e solo se lo è da M'
- per ogni funzione $f : A* \mapsto A*$, f è calcolata da M se e solo se lo è da M'

16.0.3 Dimostrazione



16.0.4 Complessità spaziale

Valutare la quantità di memoria richiesta per la risoluzione di un problema

Il massimo numero di celle visitate dalla macchina di Turing nelle computazioni convergenti su input di lunghezza <= n ?

Sarebbe sempre >= n

Esempio
Per sommare due interi (espressi in binario) è necessario tenere traccia del riporto: un solo bit, indipendentemente dalla dimensione dell'input.
È opportuno distinguere

- lo spazio necessario per l'input
- lo spazio usato dallo sviluppo della computazione

16.0.5 Il modello

Macchina di Turing a piu' nastri:

- un nastro di sola lettura su cui collocare l'input
- un nastro ausiliario di lavoro su cui svolgere la computazione
- un eventuale terzo nastro di sola scrittura, da usare una sola volta per l'output.

Definizione
La complessità spaziale di una macchina di Turing deterministica M (come sopra) è la funzione $s_M(n)$ che associa ad ogni naturale n il numero massimo di quadri impiegato da M sul nastro di lavoro nelle computazioni convergenti su input di lunghezza <= n.

16.0.6 Il modello non deterministico

Definizione
Sia M una macchina di Turing non deterministica (come sopra).
Per ogni parola w accettata da M, denotiamo con s_w il minimo numero di quadri impiegato sul nastro di lavoro da una computazione convergente di M su input w.
Si dice complessità spaziale di M la funzione $s_M : N \mapsto N$ definita come segue: per ogni $n \in N$,

$$s_M(n) = max\{sw|waccettatadaM, l(w) <= n\}.$$

Qualche osservazione

$$s_M(n) <= c_M(n)$$

Invero, in una computazione di t passi, si possono visitare al più t celle sul nastro di lavoro.
Viceversa

$$c_M(n) = n * 2^{O(s_M(n))}$$

16.0.7 Dimostrazione

Una configurazione istantanea di una macchina di Turing in una computazione convergente su un input di lunghezza n è determinata da

Lo stato (k = Card Q possibilità)

La posizione della testina sul nastro di input (n possibili posizioni)

La posizione della testina sul nastro di lavoro ($s_M(n)$ possibili posizioni)

Il contenuto del nastro di lavoro ($d^{sM(n)}$ possibilità, con d = 1 + Card A)

Pertanto il numero di tali configurazioni è maggiorato da

$$kns_M(n)d^{s_M(n)} = n \cdot 2^{\log_2 k + \log_2 s_M(n) + s_M(n)\log_2 d} = n \cdot 2^{O(s_M(n))}$$

In una computazione convergente, una configurazione istantanea non può ripetersi. Ne segue l'asserto

16.0.8 Classi di complessità spaziale

PSPACE è la classe dei problemi di decisione S che sono accettati da macchine di Turing (deterministiche) M tali che $s_M(n) = O(n^k)$ per qualche intero positivo k.
LOGSPACE (o, brevemente, L) è la classe dei problemi di decisione S che sono accettati da macchine di Turing (deterministiche) M tali che $s_M(n) = O(\log n)$.

Osservazione

$$L \subseteq P \subseteq PSPACE \subseteq PEXP.$$

Si ha L $\neq$ PSPACE e quindi vale almeno una delle disuguaglianze

$$L \neq PoppureP \neq PSPACE$$

Congettura

$$L \neq P$$

16.0.9 Classi non deterministiche

Definizione
PSPACE è la classe dei problemi di decisione S che sono accettati da macchine di Turing (deterministiche) M tali che $s_M(n) = O(n^k)$ per qualche intero positivo k.
LOGSPACE (o, brevemente, NL) è la classe dei problemi di decisione S che sono accettati da macchine di Turing (deterministiche) M tali che $s_M(n) = O(\log n)$.

Osservazione
Si ha

- PSPACE $\subseteq$ NPSPACE
- L $\subseteq$ NL
- NP $\subseteq$ NPSPACE

16.0.10 Il grafo delle computazioni

Le computazioni di una macchina di Turing M su un particolare input w sono descritte dal grafo diretto G(M, w) definito da

- I vertici di G(M, w) sono le configurazioni istantanee di M nelle computazioni convergenti su input w e un ulteriore vertice t(M, w)
- Le frecce connettono le configurazioni consecutive (legate dalla relazione $\vdash$M). Inoltre, ci sono frecce da tutte le configurazioni di arresto a t(M, w).

Osservazione
Come si è visto, il numero degli stati di G(M(w)) è $2^{O(s_M(n))}$, ove $n = l(w)$. Il medesimo limite vale per il numero delle frecce.
Inoltre ogni vertice può essere descritto con $O(s_M(n) + \log n)$ bit di informazione

Capitolo 17

Problemi NL-completi

17.0.1 Il grafo delle computazioni

Data una macchina di Turing M, un input w e un intero $K >= s_M(n)$, costruiamo il grafo G(M, w) con

- I vertici di G(M, w) sono le configurazioni istantanee di M con input w e spazio <= K sul nastro di lavoro, oltre a un ulteriore vertice t(M, w)
- Le frecce sono le coppie (C, C') tali che C $\vdash$M C' e, inoltre, le coppie (C,t(M, w)) con C $\vdash$ /M

Osservazione
G(M, w) ha n $2^{O(K)}$ vertici e n^2 $2^{O(K)}$ frecce.

Applicazione

$$c_M(n) = n \cdot 2^{O(s_M(n))}$$

Invero, una computazione che accetta w corrisponde a un cammino in G(M, w) dalla configurazione iniziale fino a t(M, w). Se priva di cicli, la sua lunghezza è minore del numero dei vertici.

17.0.2 Un'applicazione

Proposizione

$$\text{NL} \subseteq \text{P}.$$

Dimostrazione
Sia S $\in$ NL e sia M una macchina di Turing non deterministica che accetta S con

$$s_M(n) <= c \log 2(n)$$

Per decidere S si può usare la seguente procedura deterministica su input w

1. costruire G(M, w), con $K = |c \log 2(n)|$
2. cercare un cammino dalla configurazione iniziale a t(M, w)
3. se tale cammino esiste, accettare, altrimenti rifiutare.

Il numero dei vertici di G(M, w) è $n \cdot 2^{O(\log 2(n))} = n^{O(1)}$, e ognuno di essi è descritto da O(log n) bit.
Pertanto la procedura descritta è in P.

17.0.3 L=NL?

Definizione
Siano S, S' due problemi su alfabeti finiti A, A' rispettivamente.
Scriviamo

$$S <=_{\log} S'$$

se esiste una funzione totale $f : A^* \mapsto A'^*$, computabile deterministicamente in spazio logaritmico, tale che per ogni w $\exists$ A*

$$\text{w} \in \text{S} \iff \text{f(w)} \in \text{S'}$$

Osservazione
Non è evidente che la relazione <= log sia un pre-ordine.
In effetti, non è banale trovare un algoritmo che calcoli in spazio polinomiale la composizione di due funzioni calcolabili in spazio polinomiale.

17.0.4 Riduzioni in spazio logaritmico

Data una funzione $f : A^* \mapsto A'^*$ denotiamo con $\hat{f}$ la funzione di A^* x N in A' definita da

- $\hat{f}$(w, n)=n-esima lettera di f(w)

Lemma
La funzione f si può calcolare in spazio logaritmico se e solo se $\hat{f}$ si può calcolare in spazio logaritmico

17.0.5 Dimostrazione

Supponiamo che f sia calcolata da una macchina di Turing deterministica M in spazio logaritmico. Ovviamente, M avrà un nastro di output write-only. La funzione $\hat{f}$ è calcolata da una macchina di Turing M' ottenuta modificando M nel modo seguente:

- aggiungiamo sul nastro di lavoro un contatore, con valore iniziale 1
- le istruzioni di spostamento della testina sul nastro di output sono sostituite da incrementi/decrementi del contatore
- si esegue l'istruzione di scrittura sul nastro di output solo se il valore del contatore è n

Lo spazio per il contatore è $\log 2$ l(f(w)).
Poichè NL $\subseteq$ P, si ha $\log 2$ l(f (w)) = O($\log 2$ l(w))

17.0.6 La relazione <= log

Proposizione
Siano S, S', S" problemi definiti su alfabeti finiti A, A', A" rispettivamente.

- Se S $<=_{\log}$ S' e S' $<=_{\log}$ S", allora S $<=_{\log}$ S"
- Se S $<=_{\log}$ S' e S $\in$L, allora S $\in$ L.
- Se S $<=_{\log}$ S' e S' $\in$ NL, allora S $\in$ NL.

Dimostrazione
Siano f e g, rispettivamente, le riduzioni di S a S' e di S' a S" calcolabili in spazio logaritmico.
Denotiamo con M la macchina di Turing che calcola $\hat{f}$ se con M' la macchina di Turing che calcola g. Costruiamo una macchina di Turing M" che calcola f ° g, simulando simultaneamente M e M'

17.0.7 Dimostrazione

- Sul nastro di input scriviamo la parola w su cui calcolare f ° g
- Sul nastro di lavoro registriamo
  - il contenuto del nastro di lavoro della macchina M'
  - la posizione della testina di lettura sul nastro di input di M'
  - una parte del nastro è riservato alla simulazione di M
- La computazione funziona nel modo seguente
  - si legge la posizione n della testina del nastro di input di M'
  - si calcola f (w, n) usando il nastro di input e la parte riservata sul nastro di lavoro
  - si simula un passo della computazione di M' utilizzando l'input f (w, n) e il nastro di lavoro e si aggiorna il valore di n
- lo spazio necessario è logaritmico

Le altre due dimostrazioni sono analoghe.

17.0.8 NL-completezza

Definizione

Un problema T si dice NL-arduo se per ogni S $\in$ NL si ha S $<=_{\log}$ T.
Un problema T si dice NL-completo se sta in NL ed è NL-arduo.

Osservazione

Sia T un problema NL-completo.
Allora si ha T $\in$ L se e soltanto se L = NL

17.0.9 Gap

Problema di accessibilità nei grafi

Input: Un grafo diretto G = (V, E) e due vertici $t_0, t_1 \in V$
Output: SI se in G c'è un cammino da t_0 a t_1, NO altrimenti.

Teorema

GAP è un problema NL-completo.

Dimostrazione

$$\text{GAP} \in \text{NL}$$

Una procedura non deterministica che accetta GAP in spazio logaritmico è la seguente.

```
1  t ← t0
2  while t ≠ t0
3        t ← un vertice s tale che (t, s) ∈ E (o s = t)
```

17.0.10 GAP è NL-arduo

Sia S $\in$ NL.
Dobbiamo trovare una riduzione di S a GAP, calcolabile in spazio logaritmico.
Sia M una macchina di Turing non deterministica che accetta S in spazio logaritmico.
A ogni input w del problema S associamo la tripla

$$f(w) = (G(M, w), t_0, t(M, w)) \text{ ove}$$

- G(M, w) è il grafo delle computazioni di M su w
- t_0 è la configurazione iniziale

È facile verificare che f è una riduzione di S a GAP.
Un'analisi attenta mostra che f si può calcolare in spazio logaritmico.

Capitolo 18

Teorema di Savitch

18.0.1 Teorema di Savitch

Collega le minime risorse di memoria necessarie per risolvere deterministicamente o non deterministicamente un problema.

Teorema
Sia S un problema che si accetta non deterministicamente in spazio s dove $s(n) >=$ $\log_2$ n è una funzione a sua volta computabile in spazio $0(s(n))$. Allora S è accettato deterministicamente in spazio $0(s2(n))$.

Corollario

$$\text{PSPACE} = \text{NPSPACE}$$

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE$$

18.0.2 Gap

Proposizione
Il problema GAP può essere accettato deterministicamente in spazio $0(log^2 n)$.

Dimostrazione
Sia G = (V , E) un grafo diretto finito.

Consideriamo la funzione Acc: $VxVxN \mapsto B$ definita da

$$\mathrm{Acc}(t_0, t_1, k) = \begin{cases} 1 & \text{se esiste un cammino da } t_0 \text{ a } t_1 \text{ di lunghezza } \leq 2^k, \\ 0 & \text{altrimenti.} \end{cases}$$

Si verifica facilmente che:

$$\mathrm{Acc}(t_0, t_1, k) = \sum_{v \in V} \mathrm{Acc}(t_0, v, k-1)\mathrm{Acc}(v, t_1, k-1)$$

$$\mathrm{Acc}(t_0, t_1, k) = \sum_{v \in V} \mathrm{Acc}(t_0, v, k-1)\mathrm{Acc}(v, t_1, k-1)$$

```
Function Acc(t0, t1, k)
 1  if k = 0
 2       if t0 = t1 o (t0, t1) ∈ E
 3            return 1
 4       else
 5            return 0
 6  for v ∈ V
 7       if Acc(t0, v, k − 1)
 8            if Acc(v, t1, k − 1)
 9                 return 1
10  return 0
```

18.0.3 Analisi

```
6  for v ∈ V
7      if Acc(t0, v, k − 1)
8          if Acc(v, t1, k − 1)
9              return 1
```

Sia s_k lo spazio per l'esecuzione dell'algoritmo.
La variabile v richiede spazio $\log_2 n$.
L'istruzione 2 richiede spazio $s_k - 1$.
L'istruzione 3 può essere eseguita nello spazio già usato per l'istruzione 2.
Se ne deduce $sk = s_k - 1 + \log_2 n$ e, risolvendo la ricorrenza.

$$s_k = klog_2n$$

18.0.4 Conclusione

Detto n il numero di vertici di G e k = $|log_2n|$, si ha $(G, t_0, t_1) \in$ GAP se e solo se $Acc(t_0, t_1, k) = 1$. Quindi il problema si risolve in spazio

- $klog_2n <= (log_2n)^2$

18.0.5 Il caso NL

Proposizione
Sia $S \in NL$. Allora S è accettato deterministicamente in spazio $0((log_2n)^2)$.

Dimostrazione
Adattando le tecniche sviluppate nelle lezioni precedenti, si ha che se $S <=_{\log} T$ e T si calcola in spazio $O(log^2n)$, allora anche S si calcola in spazio $O(log^2n)$. Poichè GAP è NL-completo, si ha $S <=_{log}$ GAP e l'asserto segue dal lemma precedente.

18.0.6 Il caso generale

Sia M una macchina di Turing che accetta S in spazio s(n).
Osserviamo che si ha w ∈ S se e solo se c'è un cammino nel grafo G(M, w) dalla

configurazione iniziale alla configurazione di arresto.
Adattando l'algoritmo usato per GAP, si può verificare questa condizione in spazio $O(s^2(n))$.

18.0.7 Il complemento di GAP

Lemma

$$\text{GAP} \in \text{coNL}$$

Dimostrazione
Dobbiamo trovare una procedura non deterministica che accetta in spazio logaritmico il complemento di GAP.
Sia G = (V , E) un grafo diretto e $t_0 \in V$ un vertice. Per ogni intero j >= 0, diremo che un vertice v ∈ V è j -accessibile se esiste un cammino di lunghezza j da t_0 a v.

18.0.8 Vertici j-accessibili

Supponiamo di conoscere il numero r dei vertici j -accessibili.
Allora possiamo realizzare una procedura non deterministica, che ci dice se un dato vertice v è $(j+1)-$accessibile o meno.
Invero, possiamo procedere nel modo seguente:

- Per ogni vertice u ∈ V , generiamo non deterministicamente un cammino di lunghezza j che parte da t_0
- Se il numero dei vertici u per cui il cammino generato termina proprio in u è uguale a r , allora possiamo concludere che gli u per cui questo avviene sono esattamente i vertici j -accessibili; in tal caso, il vertice v è $(j+1)-$accessibile se e solo se v è adiacente a uno di tali vertici
- Se invece il numero delle volte in cui il cammino generato termina in u è minore di r , allora non possiamo concludere nulla.

18.0.9 Procedura Accesso

Input: Un grafo diretto $G = (V, E), t_0$, $v \in V$, $j >= 0$, r = numero dei vertici j-accessibili.
Output: **VERO** se v è $(j+1)-$accessibile, **FALSO** altrimenti (nelle esecuzioni che terminano)

```
1  Accesso ← FALSE
2  for u ∈ V
3       s ← 0
4       t ← t_0
5       for i ← 1, . . . , j
6            scegli (t, t') ∈ E
7            t ← t' // genera cammino di j passi da t_0
8       if u = t
9            s ← s + 1
10           If (u, v) ∈ E Then Accesso ← TRUE
11 if s < r
12      error
```

18.0.10 Calcolo di r

Ora siamo in grado di calcolare il numero dei vertici j -accessibili per $j = 1, 2, ..., n-1$. Invero, una volta calcolato il numero r dei vertici j-accessibili, applicando la procedura Accesso a tutti i vertici v ∈ V , potremo conoscere il numero dei vertici $(j+1)$-accessibili.

Procedura ContaAccessibili
Input: Un grafo diretto G = (V , E), $t_0 \in$ V.
Output: Genera il numero r dei vertici (j+1)-accessibili per ogni j.

```
1  n ← card(V)
2  r ← 1
3  for j ← 0, . . . , n − 1
4       r_new ← 0
5       for v ∈ V
6            if Accesso                          // può fallire
7                 r_new ← r_new + 1
8       r ← r_new                  // numero vertici (j+1)-accessibili
```

18.0.11 Accettare il complemento di GAP

Una semplice modifica del procedimento permette anche di determinare se un vertice è inaccessibile.
Otteniamo così la seguente procedura che può terminare solo se t_1 è inaccessibile da t_0.

```
1  if t_1 = t_0 then error
2  n ← Card(V)
3  r ← 1
4  for j ← 1, ..., n − 1
5       r_new ← 0
6       for v ∈ V
7            if Accesso
8                 if v = t_1 then error
9                 r_new ← r_new + 1
10      r ← r_new
```

18.0.12 NL = coNL

Teorema

$$\text{NL} = \text{coNL}$$

Dimostrazione

Poichè GAP è NL-completo, per ogni problema $S \in$ NL si ha $S \leq_{\log}$ GAP e, di conseguenza, $\overline{S} \leq_{\log} \overline{\text{GAP}}$.
Avendo dimostrato che $\overline{\text{GAP}} \in$ NL, concludiamo che $\overline{S} \in$ NL e quindi $S \in$ coNL

18.1 NL = coNL Approfondito

Vogliamo mostrare che le classi di complessità spaziale NL e coNL coincidono. Il primo passo consiste nel mostrare che il problema NL-completo GAP sta nella classe coNL.

Lemma 1

$$GAP \in coNL$$

Dimostrazione
Per dimostrare l'asserto dobbiamo trovare una procedura non deterministica che accetta in spazio logaritmico il complemento di GAP, ossia l'insieme delle triple (G, t_0, t_1), ove G è un grafo diretto e t_0 e t_1 sono due vertici di G con t_1 inaccessibile da t_0.
Sia dunque G = (V, E) un grafo diretto e $t_0 \in V$ un vertice. Per ogni intero $j >= 0$, diremo che un vertice $v \in V$ è j-accessibile se esiste un cammino di lunghezza j da t_0 a v. Supponiamo di conoscere il numero r dei vertici j-accessibili.
Allora possiamo realizzare una procedura non deterministica, che ci dice se un dato vertice v è (j + 1)-accessibile o meno. Invero, possiamo procedere nel modo seguente:

- Per ogni vertice u ∈ V, generiamo non deterministicamente un cammino di lunghezza j che parte da t_0
- Se il numero dei vertici u per cui il cammino generato termina proprio in u è uguale a r, allora possiamo concludere che gli u per cui questo avviene sono esattamente i vertici j-accessibili; in tal caso, il vertice v è (j + 1)-accessibile se e solo se v è adiacente a uno di tali vertici
- Se invece il numero delle volte in cui il cammino generato termina in u è minore di r, allora non possiamo concludere nulla.

Abbiamo così a disposizione la seguente procedura non deterministica che per ogni vertice v termina almeno una volta determinando correttamente se v è (j + 1)-accessibile o non lo è.
Ora siamo in grado di calcolare il numero dei vertici j-accessibili per $j = 1, 2, ..., n-1$.

Procedura Accesso

Ingresso: Un grafo diretto $G = (V, E)$, $t_0, v \in V$, $j \geq 0$, r = numero dei vertici j-accessibili;

Uscita: VERO se v è $(j+1)$-accessibile, FALSO altrimenti (nelle esecuzioni che terminano);

```
Accesso ← FALSO;
s ← 0;
per ciascun u ∈ V fai
    t ← t_0;
    per i ← 1, ..., j fai
        scegli (t, t') ∈ E;
        t ← t';                          /* genera cammino di j passi da t_0 */
    se u = t allora
        s ← s + 1;
        se (u, v) ∈ E allora Accesso ← VERO;
se s < r allora ERRORE;
```

Invero, una volta calcolato il numero r dei vertici j-accessibili, applicando la procedura Accesso a tutti i vertici $v \in V$, potremo conoscere il numero dei vertici (j + 1)-accessibili.
La procedura è la seguente:

Procedura ContaAccessibili

Ingresso: Un grafo diretto $G = (V, E)$, $t_0 \in V$

```
n ← Card(V);
r ← 1;                                   /* vertici 0-accessibili */
per j ← 0, ..., n − 2 fai
    r_nuovo ← 0;
    per ciascun v ∈ V fai
        se Accesso allora                          /* può fallire */
            r_nuovo ← r_nuovo + 1
    r ← r_nuovo;                  /* numero vertici (j+1)-accessibili */
```

Naturalmente, in molte esecuzioni del procedimento testé descritto, qualche chiamata a Accesso determinerà un errore. Esiste però almeno una esecuzione in cui tutte le chiamate terminano correttamente. In tal caso, il procedimento determina esattamente tutti i vertici j-accessibili per $j = 1, 2, ..., n-1$. Tenuto conto che in un grafo con n vertici, un vertice è accessibile se e solo se è j-accessibile per qualche $j < n$, una semplice modifica del procedimento permette anche di determinare se un vertice è inaccessibile. Otteniamo così la seguente procedura che può terminare solo se t_1 è inaccessibile da t_0.

```
Procedura Inaccessibile
  Ingresso: Un grafo diretto G = (V, E), t0, t1 ∈ V
  Accetta: (G, t0, t1) ∉ GAP

  se t1 = t0 allora ERRORE;                          /* t0 è accessibile */
  n ← Card(V);
  r ← 1;
  per j ← 1, ..., n − 2 fai
     r_nuovo ← 0;
     per ciascun v ∈ V fai
        se Accesso allora
           se v = t1 allora ERRORE;                   /* t1 accessibile */
           r_nuovo ← r_nuovo + 1;
     r ← r_nuovo;                        /* numero vertici (j+1)-accessibili */
```

Adesso occupiamoci di studiare la complessità spaziale della procedura Inaccessibile. Tale procedura fa uso delle variabili n, r, r_{nuovo}, v, ciascuna delle quali richiede spazio logaritmico rispetto al numero dei vertici di G, e inoltre necessita dello spazio per l'esecuzione della procedura accesso. Anche le variabili della procedura Accesso richiedono spazio logaritmico rispetto al numero dei vertici di G. Possiamo concludere quindi che $GAP \in coNL$.
Ora utilizzeremo il fatto che $GAP \in coNL$ per mostrare che $NL = coNL$.
Iniziamo col seguente:

Lemma 2
Siano S e T due problemi. Se $S <=_{\log} T$ e $T \in coNL$, allora $S \in coNL$.

Dimostrazione
Si osservi che una riduzione f di S a T è anche una riduzione del complemento $\bar{S}$ di S al complemento $\bar{T}$ di T. Quindi se $S <=_{\log} T$, si ha anche $\bar{S} <=_{\log} \bar{T}$.
Tenuto conto del fatto che $T \in NL$, si ha $S \in NL$ e quindi $S \in coNL$.
Ora siamo pronti per dimostrare il nostro risultato principale.

Teorema 1

$$\text{NL} = \text{coNL}$$

Dimostrazione
Sia S ∈ NL. Poiché GAP è NL-completo, si ha $S <=_{\log} GAP$ e quindi, per i lemmi

precedenti, S $\in$ coNL.
Viceversa, sia S $\in$ coNL. Allora il suo complemento S sta in NL. Per l'argomento precedente, si ha S $\in$ coNL e quindi S $\in$ NL. Se ne conclude che le classi NL e coNL coincidono.

Capitolo 19

La somma di sottoinsiemi è NP-completa

19.1 La somma di sottoinsiemi è NP-completa

Il problema della somma di sottoinsiemi è il seguente: dati n numeri interi non negativi

$$w1, ..., wn$$

e una somma obiettivo W, si tratta di decidere se esiste un sottoinsieme $I \subset \{1, ..., n\}$ tale che $\sum_{i \in I} w_i = W$. Si tratta di un caso molto particolare del problema di **Knapsack**.
Nel problema di Knapsack, anche gli oggetti hanno valori v_i, e il problema consiste nel massimizzare $\sum_{i \in I} v_i i \in Ivi$ soggetto a $\sum_{i \in I} w_i = W$.
Se impostiamo $v_i = w_i$ per tutti gli i, la Somma di sottoinsiemi è un caso speciale del problema di Knapsack che abbiamo discusso quando abbiamo considerato la programmazione dinamica. In quella sezione, abbiamo fornito un algoritmo per il problema che viene eseguito in tempo $O(n * W)$.
Questo algoritmo funziona bene quando W non è troppo grande, ma notiamo che questo algoritmo non è un algoritmo a tempo polinomiale. Per scrivere un intero W, abbiamo bisogno solo di log W cifre. È naturale assumere che tutti i $w_i <= W$, e quindi la lunghezza dell'ingresso è $(n+1) \log W$, e il tempo di esecuzione di $O(n * W)$ non è polinomiale per questa lunghezza di input.
In questa dispensa mostriamo che, in effetti, la somma di sottoinsiemi è NP-completa.

Innanzitutto dimostriamo che La somma di sottoinsiemi è in NP.

Proposizione 1
La somma di sottoinsiemi è in NP.

Dimostrazione
Dato un insieme proposto I, basta verificare se effettivamente $\sum_{i \in I} w_i = W$. Sommare al massimo n numeri, ciascuno di dimensione W, richiede un tempo $O(n \log W)$, lineare rispetto alla dimensione dell'input. Per stabilire che la somma di insiemi è NP-completa, dimostreremo che è almeno altrettanto difficile di SAT.

Teorema

SAT <= Somma di sottoinsiemi.

Dimostrazione
Per dimostrare l'affermazione è necessario considerare una formula ϕ, un input di SAT, e trasformarla in un input equivalente alla somma dei sotto-insieme.
Supponiamo che ϕ abbia n variabili $x_1, ..., x_n$, e m clausole $c_1, ..., c_m$, dove la clausola c_j ha k_j letterali.
Definiremo il nostro problema della somma di sottoinsiemi utilizzando una base B molto grande, quindi scriveremo i numeri come $\sum_{j=0}^{n} + ma_j B^j$ e impostiamo la base B come $B = 2max_j k_j$ che assicura che le addizioni tra i numeri tra i nostri numeri non causeranno mai un riporto.
Scritte in base B, le cifre $i = 1, ..., n$ corrisponderanno alle n variabili $x_1, ...x : n$, e l'obiettivo di queste cifre sarà quello di assicurarsi di impostare ogni variabile su vero o falso (e non su entrambi). Avremo due numeri w_i e w_{i+n} che corrispondono alla variabile x_i che viene impostata vera o falsa, e la cifra i farà in modo di utilizzare uno dei due numeri w_{i+n} in qualsiasi soluzione. Per fare ciò, impostiamo la decima cifra di W, w_i e w_{i+n} come 1 e impostiamo questa cifra in tutti gli altri numeri come 0. Le successive m cifre corrisponderanno alla variabile x_i. Le successive m cifre corrispondono alle m clausole e la cifra obiettivo n + j serve a garantire che la j-esima clausola sia soddisfatta dalla nostra impostazione delle variabili.
Il valore obiettivo sarà $W = \sum_{j=1}^{n} B^i + \sum_{j=1}^{m} k_j B^{n+j}$.
Si inizia definendo 2n numeri, per ciascuno dei letterali x_i e $x\bar{i}$. Le cifre 1,...,n faranno

in modo che ogni sottoinsieme che somma a W utilizzi esattamente solo uno dei due numeri x_i e $x\bar{i}$, mentre le successive m cifre avranno lo scopo di garantire che ogni clausola sia soddisfatta. Avremo bisogno di alcuni numeri aggiuntivi che definiremo in seguito.
Il numero corrispondente al letterale x_i è il seguente $wi = B^i + \sum_{j:x_i \in c_j} B^{n+j}$, mentre il numero corrispondente al letterale $x\bar{i}$ è il seguente corrispondente al letterale $w_i = B^i + \sum_{j:x_i \in c_j} B^{n+j}$. Se aggiungiamo un insieme di n numeri corrispondente a un'assegnazione di verità soddisfacente per ϕ, si ottiene una qualche forma di $\sum_{i=1}^{n} b_j B^i + \sum_{j=1}^{m} b_j B^{n+j}$ dove b_j è il numero di letterali veri nella clausola c_j. Poiché si trattava di un'assegnazione soddisfacente, dobbiamo avere $b_j >= 1$.
Come ultimo dettaglio, aggiungeremo $k_j - 1$ copie del numero B^{n+j} per tutte le clausole c_j . In questo modo definito il nostro problema della somma dei sottoinsiemi, con l'obiettivo W e i $2n + \sum_j (k + j - 1)$ numeri definiti, l'aggiunta di questi numeri aggiuntivi ci permetterà di raggiungere esattamente W. Per dimostrare che si tratta di una riduzione valida, dobbiamo stabilire due affermazioni di seguito riportate che stabiliscono rispettivamente la direzione del se e del solo se della prova.

Proposizione 2
Se il problema SAT definito dalla formula ϕ è risolvibile, allora anche il problema della Somma di sottoinsiemi appena definito con $2n - m + \sum_j k_j$ numeri è anch'esso risolvibile.

Dimostrazione
Supponiamo di avere un'assegnazione soddisfacente per la formula ϕ, prima consideriamo di aggiungere i numeri che corrispondono a_i letterali veri. Abbiamo usato esattamente uno dei numeri w_i e w_{n+i}, quindi avremo 1 nella cifra i-esima e otterremo una somma della forma $\sum_{i=1}^{n} B_i + \sum_{j=1}^{m} a_j B^{n+j}$.
Inoltre, si avrà $1 <= a_i <= k_i$, dove a_i è almeno 1, in quanto l'assegnazione ha soddisfatto la formula, quindi almeno uno dei numeri aggiunti ha un 1 nella (n+j) terza cifra, e al massimo k_j in quanto anche sommando tutti i numeri al massimo k_j di essi ha un 1 nella (n+j)terza cifra. In particolare, con $B > k_j$, non ci saranno riporti.
Per fare in modo che la somma sia esattamente W, aggiungiamo $k_j - a_j$ copie del numero B^{n+j} che abbiamo aggiunto alla fine della costruzione.

Dobbiamo poi dimostrare l'altra direzione:

Proposizione 3
Se il problema della somma di sottoinsiemi che abbiamo appena definito con $2n - m + \sum_j k_j$ numeri è risolvibile, allora la Somma di sottoinsiemi definita dalla formula ϕ è risolvibile.

Dimostrazione
Si noti innanzitutto che per qualsiasi sottoinsieme si possa aggiungere, non ci sarà mai un riporto in nessuna cifra.
Per capire perché, si noti che tutti i numeri da sommare hanno tutte le cifre 0 o 1; per la cifra $i = 1, ..., n$ abbiamo due numeri con un 1 in quella cifra $w_i e w_{i+n}$; la cifra 0 è sempre 0; e per la cifra n + j abbiamo esattamente $2max_j\ k_j - 1$ numeri che hanno un 1 in quella cifra: k_j corrispondenti ai k_j letterali nella clausola e $k_j - 1$ numeri extra B^{n+j} che abbiamo aggiunto alla fine.
Quindi, anche se anche se aggiungiamo tutti i numeri, non possiamo causare un riporto in nessuna delle cifre!.
In base all'osservazione di cui sopra sull'assenza di riporti, per ottenere il numero W dobbiamo trovare un sottoinsieme I che abbia esattamente il numero giusto di 1 in ogni cifra. Per prima cosa concentriamoci sulle cifre 1,..., n. Questa cifra in W è un 1 e i due numeri che hanno un 1 in questa cifra sono w_i e w_{i+n}, per sommare a W, dobbiamo usare esattamente uno di questi, sia I' $\subset$ I corrispondente ai letterali. Questo Questo dimostra che i numeri selezionati tra i primi 2n corrispondono a un'assegnazione di verità delle variabili $x_1, ..., x_n$.
Infine, dobbiamo dimostrare che questa assegnazione di verità soddisfa la formula ϕ. Consideriamo la somma $W' = \sum_{j \in I'} wj$ aggiungendo solo il sottoinsieme I che corrisponde alle variabili. Si noti che $W' = \sum_{j=1}^{n} B_i + \sum_{j=1}^{m} a'_j B^{n+j>}$ con $a'_j j <= k_j$.
Dobbiamo dimostrare che $a'_j >= 1$, il che dimostrerà che abbiamo un'assegnazione soddisfacente. Ricordiamo che il sottoinsieme I somma esattamente a W. Per essere in grado di estendere I' con un sottoinsieme di numeri aggiuntivi che sommano a W, dobbiamo avere a'_j, poiché ci sono ci sono solo $k_j - 1$ copie di B^{n+j} .