

Teoria della Complessità Computazionale

Giacomo Calamai

Lezione del 27/11/2003

seminario *Tipi e Informazione*

Prof. A. Cantini, Prof. P. Minari

Università degli Studi di Firenze

Dipartimento di Filosofia

A.A. 2003-2004

3 dicembre 2003

TURING, CHURCH E GÖDEL: LA PRECISAZIONE DEL CONCETTO DI PROCEDURA ALGORITMICA

Macchine di Turing (TM)...

Turing-computabilità di una funzione...

TM deterministiche: il modello standard di calcolo nella Teoria della Computabilità...

Definizione. Una *Macchina di Turing deterministica ad un solo nastro (TM)* è una quadrupla $M = (Q, \Sigma, \delta, s_0)$. Q è l'insieme *finito* degli stati, dove $s_0 \in Q$ è lo stato iniziale, mentre Σ è l'insieme finito dei simboli, l'*alfabeto* della macchina M . Q e Σ sono insiemi disgiunti. Σ contiene inoltre i simboli speciali $\#$ e \triangleright : il simbolo vuoto e il 'primo simbolo'. δ è una funzione di transizione da $Q \times Q$ in

$(Q \cup \{H, sí, no\}) \times \Sigma \times \{\leftarrow, \rightarrow, \frown\}$. H (lo stato di arresto), $sí$ (lo stato di accettazione), no (lo stato di rifiuto) e i simboli di direzione del cursore \leftarrow (*left*), \rightarrow (*right*) e \frown (*stay*) non appartengono a $\Sigma \cup Q$. δ è il *programma* della macchina: per ogni combinazione di stati $q \in Q$ e simboli $s \in \Sigma$ correnti specifica una tripla $\delta(q, \sigma) = (p, \rho, D)$ dove p è lo stato successivo, ρ il simbolo da sovrascrivere a q e $D \in \{\leftarrow, \rightarrow, \frown\}$ specifica la direzione di movimento del cursore. Nel caso di \triangleright , se per q e p vale $\delta(q, \triangleright) = (p, \rho, D)$, allora $\rho = \triangleright$ e $D = \rightarrow$. Ovvero, \triangleright dirige sempre il cursore verso destra e non viene mai cancellato.

TURING-COMPUTABILITÀ PARZIALE E TOTALE

Definizione. *Una funzione numerica parziale n -aria $f(x_1, \dots, x_n)$ è **parzialmente Turing-computabile** se e soltanto se esiste una Macchina di Turing deterministica ad un solo nastro \mathbf{T} tale che $f(x_1, \dots, x_n) = \Psi_T^{(n)} x_1, \dots, x_n$. In questo caso diremo che la macchina \mathbf{T} **computa** la funzione f . Inoltre, se $f(x_1, \dots, x_n)$ è una funzione numerica n -aria totale, allora la corrispondente funzione $\Psi_T^{(n)} x_1, \dots, x_n$ calcolata dalla Macchina di Turing deterministica ad un solo nastro \mathbf{T} si dirà **Turing-computabile**.*

VARIANTI DI MACCHINE DI TURING

Le macchine di Turing deterministiche (nelle quali ad ogni istante del processo di calcolo è univocamente determinata l'operazione successiva che verrà compiuta) sono il modello di calcolo standard della Teoria della Computabilità (astratta).

Tuttavia nella Teoria della Complessità si incontrano frequentemente altri modelli di calcolo (spesso varianti di macchine di Turing):

1. **TM multinastro:** viene ammessa una molteplicità di nastri e di relative testine di lettura/scrittura.
2. **TM nondeterministiche:** viene tralasciato il vincolo sul determinismo; durante una computazione qualsiasi una macchina non

deterministica può prevedere un numero arbitrariamente grande di configurazioni deterministiche distinte. Otteniamo un albero di possibili configurazioni nel quale ogni ramo rappresenta una computazione in senso strettamente deterministico.

3. **TM con oracolo:** i processi computazionali vengono *relativizzati* ad universi alternativi; esiste un particolare stato $q? \in Q$ chiamato *stato interroga-oracolo* che può mutare il corso della computazione *relativizzandolo* al responso dell'oracolo.
4. **TM alternanti...**
5. **Macchine a Registri...**
6. **TM con Accesso Random...**
7. **Circuiti Booleani...**

STABILITÀ DEI MODELLI COMPUTAZIONALI

Fatto: tutti questi modelli sono equivalenti (convergono sulla stessa classe di funzioni computabili), anche se dal punto di vista della **efficienza** di calcolo ciascuno può simulare l'altro con **aumenti quadratici** o **cubici** del tempo di calcolo.

ULTERIORI CARATTERIZZAZIONI DEL CONCETTO DI PROCEDURA EFFETTIVA

- Funzioni generali ricorsive definite mediante il calcolo delle equazioni (Gödel-Herbrand-Kleene [1936])...
- Funzioni λ -definibili (Church [1936])...
- Funzioni μ -ricorsive e funzioni parziali ricorsive (Gödel-Kleene [1936])...
- Funzioni definite in sistemi di deduzione canonici (Post [1943])...
- Funzioni Markov-computabili (mediante algoritmi) su alfabeti finiti (Markov [1951])...

LA TESI DI CHURCH-TURING

Tesi di Church-Turing (Turing [1936], Church [1936])

Le funzioni intuitivamente calcolabili sono precisamente le funzioni Turing-computabili (o, in maniera equivalente, le funzioni parziali ricorsive, o le funzioni λ -rappresentabili, o le funzioni Markov-computabili, o le funzioni computabili mediante il calcolo delle equazioni, o le funzioni computabili mediante Macchine a Registri, o le funzioni flow-chart-computabili...)

È una *tesi*! Non è possibile darne una *dimostrazione*, in quanto dovremmo dimostrare l'equivalenza tra un concetto formale e un concetto intuitivo.

ARGOMENTI A FAVORE DELLA STABILITÀ DELLA CLASSE

- (a) Ogni funzione intuitivamente computabile escogitata fino ad ora si è dimostrata Turing-computabile
- (b) Tutte le formalizzazioni del concetto di *funzione intuitivamente computabile* introdotte fino ad ora sono dimostrabilmente equivalenti tra di loro (nonostante più o meno profonde distinzioni formali)
- (c) I programmi dei calcolatori moderni sono teoricamente (ma non praticamente!) simulabili mediante macchine di Turing

ARGOMENTO (DEBOLE?) CONTRO LA TESI

È possibile che vi siano funzioni computabili da un agente umano ma non da una macchina di Turing?

La mente umana potrebbe non essere simulabile mediante una macchina di Turing (almeno limitatamente all'attività scientifica)

LA COMPUTABILITÀ ASTRATTA

Il fondamentale concetto di computazione

Una **computazione** è un processo attraverso il quale, a partire da oggetti dati, detti **inputs**, in accordo con un insieme finito di **regole** (un **programma**, una **procedura**, un **algoritmo**), mediante una serie di **passi** (elementari), giungiamo al termine di questa serie ottenendo un risultato finale, l'**output**.

INPUT → **BLACK BOX** → OUTPUT

PREDICATI (TEORICAMENTE) DECIDIBILI E INDECIDIBILI

Supponiamo che $M(x_1, x_2, \dots, x_n)$ sia un predicato n -ario di numeri naturali. La *funzione caratteristica* $C_M(\vec{x})$ (dove $\vec{x} = x_1, \dots, x_n$) è data da

$$C_M(\vec{x}) = \begin{cases} 1 & \text{se vale } M(\vec{x}) \\ 0 & \text{se } M(\vec{x}) \text{ non vale} \end{cases}$$

Il predicato $M(\vec{x})$ è **decidibile** se la funzione C_M è computabile. $M(\vec{x})$ è **indecidibile** se $M(\vec{x})$ è non decidibile.

INDECIDIBILITÀ DEL PROBLEMA DELL'ARRESTO

Teorema. (Turing [1936])

Il problema 'Φ_x(y) è definita' (o, equivalentemente, 'P_x(y) ↓' o 'y ∈ W_x') è indecidibile.

Non esiste un metodo generale effettivo per scoprire se un dato programma che lavora su un certo input eventualmente non si arresta.

Non esiste un metodo generale perfetto per verificare in maniera esaustiva se i programmi sono liberi da possibili *loops* infiniti.

INDECIDIBILITÀ DELLA VALIDITÀ LOGICA NEL CALCOLO DEI PREDICATI AL PRIM'ORDINE (FOL)

Teorema. (Church [1936])

*La validità in **FOL** è indecidibile.*

Non esiste una procedura di decisione per
FOL.

La procedura di semidecisione riflessa dalla completezza deduttiva di **FOL** è tutto ciò che abbiamo.

DECIDIBILITÀ TEORICA vs DECIDIBILITÀ PRATICA

Si considerino i due seguenti problemi decisionali **decidibili**:

- **Il problema Massimo Comune Divisore (MCD)**

dati due numeri interi positivi a e b , qual è il massimo comune divisore tra a e b ?

- **Il problema Fattorizzazione (FAT)**

dato un intero $a > 1$, a è un numero composto (ovvero, a non è primo)? Se a è composto, trovarne la scomposizione in fattori primi, o *fattorizzazione*.

L'algoritmo **EUCLIDE** calcola il massimo comune divisore tra due numeri interi positivi:

EUCLIDE(a, b)

(a) **se** $b = 0$

(b) **allora il risultato è** a

(c) **in caso contrario il risultato è**
EUCLIDE($b, a \bmod b$)

(dove ' $a \bmod b$ ' è il resto della divisione intera di a per b)

Esempio. **EUCLIDE**(30, 21) =

EUCLIDE(21, 9)

EUCLIDE(9, 3)

EUCLIDE(3, 0) = 3

EUCLIDE è un algoritmo di *ricorrenza*.

Se vogliamo sapere se un numero intero a ($a > 1$) è un numero composto (ovvero non è primo), dobbiamo cercare *in maniera esaustiva* un divisore di a . L'algoritmo **ESAUSTIONE** compie un tipo di ricerca di questo genere:

ESAUSTIONE(a)

(a) $k \leftarrow 2$

(b) **while** $k^2 \leq a$

(c) **se** $a \bmod k = 0$

(d) **allora il risultato è sì**

(e) **in caso contrario** $k \leftarrow k + 1$

(f) **il risultato è no**

L'algoritmo **ESAUSTIONE** è *iterativo*: dato a , una iterazione può occorrere $\sqrt{a} - 1$ -volte.

Il numero dei cicli di **EUCLIDE** è molto minore: dati a e b , **EUCLIDE** compierà al più $2(1 + \log_2 a)$ -cicli.

Basta osservare, banalmente, che per ogni due cicli di **EUCLIDE** il resto diminuisce di almeno la metà.

Sia n il massimo della lunghezza (della rappresentazione binaria) di a e b ; allora il numero dei cicli computazionali compiuto in una esecuzione di **EUCLIDE** è limitato dal valore $2(n + 2)$ (in notazione standard, $\mathcal{O}(n)$).

Ad ogni ciclo, **EUCLIDE** effettua una divisione intera. Ora, l'algoritmo usuale di divisione intera è quadratico, ovvero lavora in $\mathcal{O}(n^2)$ -passi. Dunque **EUCLIDE** lavora in $\mathcal{O}(n^3)$ -passi: **EUCLIDE** è un algoritmo dal tempo cubico, e pertanto lavora in tempo polinomiale.

D'altro canto, per ogni entrata a , un algoritmo come **ESAUSTIONE** può compiere $\sqrt{a} - 1$ -iterazioni.

Se n è la lunghezza binaria di a , queste iterazioni corrispondono a circa $2^{\frac{n}{2}}$ -passi.

ESAUSTIONE non è un algoritmo che lavora in tempo polinomiale.

LOWER BOUNDS ASTRONOMICI

Un algoritmo come **ESAUSTIONE**, benchè garantisca la decidibilità del problema della primalità, non è efficiente da un punto di vista pratico.

Un computer che lavori ininterrottamente a 1 GHz durante un anno effettua circa $3,2 \times 10^{16}$ passi computazionali.

D'altro canto, per $n = 128$, $2^{\frac{n}{2}}$ corrisponde, all'incirca, a $1,8 \times 10^{19}$. Conseguentemente, un computer a 1 GHz impiegerebbe circa 563 anni a compiere 2^{64} passi computazionali.

Per $n = 256$ $2^{\frac{n}{2}}$ corrisponde, all'incirca, a $3,4 \times 10^{38}$. Si consideri il fatto che il nostro sistema solare ha circa 6×10^9 anni. Perciò nemmeno tutto questo tempo sarebbe sufficiente per compiere 2^{128} passi a 1 GHz!

Siamo di fronte ad un caso di decidibilità teorica, ma *indecibilità pratica* (*unfeasibility*).

COMPLESSITÀ DELLA ELIMINAZIONE DELLE CESURE

Un algoritmo effettivo, ma non trattabile, in **Teoria della Dimostrazione** è quello che scaturisce dalla procedura di *cut elimination* nel Calcolo dei Sequenti o di Normalizzazione in Deduzione Naturale.

Teorema.

(Paris-Wilkie [1989], Pudlak [1986])

Non esiste una costante c tale che 2_c^h rappresenti un upper bound per l'altezza dello Hauptsatz, indipendente dal grado del cut della prova che deve essere normalizzata.

Otteniamo dunque come corollario del Teorema precedente,

Corollario.

Esiste un algoritmo di cut elimination che è ricorsivo primitivo ma non elementarmente ricorsivo; l'assegnazione

prova \longrightarrow prova diretta

non è Kalmar-elementare.

MISURE DI COMPLESSITÀ COMPUTAZIONALE (DI UNA FUNZIONE RICORSIVA)

- **Misure di complessità statiche:**

Intuitivamente, il numero dell'indice di una funzione ricorsiva, oppure il numero dei simboli necessari per scrivere un programma (una tavola di istruzioni) per una **TM**...

- **Misure di complessità dinamiche:**

Esempio:

Il *numero dei passi compiuti* da una **TM**.
Il *numero delle celle lette* da una **TM**
durante un processo di calcolo.

Definizione. *Dato un sistema indicizzato accettabile $\{\varphi_e\}_{e \in \omega}$ per funzioni parziali ricorsive, una misura di complessità dinamica è una famiglia indicizzata $\{\Psi_e\}_{e \in \omega}$ di funzioni parziali ricorsive tale che*

- (a) *per ogni e e x , $\Psi_e(x) \downarrow$ sse $\varphi_e(x) \downarrow$
(solo una quantità finita di risorse è necessaria in una computazione che converge);*
- (b) *il predicato $\Psi_e(x) \simeq z$ è ricorsivo, uniformemente in e , x e z
(la restrizione ad una quantità finita di risorse z costringe la computazione di $\varphi_e(x)$ a convergere o ad avere un comportamento periodico).*

LE MISURE DI COMPLESSITÀ 'TEMPO' e 'SPAZIO' DETERMINISTICHE

Definizione.

(a) **Complessità temporale:**

$\mathbf{T}_e^{Det}(x) =_{def}$ numero di passi compiuti nella computazione di $\varphi_e(x)$, dove $\varphi_e(x) \downarrow$;

(b) **Complessità spaziale:**

$\mathbf{S}_e^{Det}(x) =_{def}$ numero di celle lette nella computazione di $\varphi_e(x)$, se $\varphi_e(x) \downarrow$, non definita altrimenti.

Osservazione. Esistono misure per il Tempo e lo Spazio non deterministici (in effetti, esistono misure associabili ad ogni modello di calcolo: **OTM**, **RTM**, **ATM**, circuiti booleani...)

COMPLESSITÀ NON DETERMINISTICA

Restringiamo l'attenzione al parametro tempo:

Definizione. Tempo non deterministico

Sia $\mathbf{T}_{e,k}^{Ndet}(x)$ il numero di istruzioni eseguite nel k -esimo cammino computazionale deterministico che termina in uno stato di accettazione;

- (a) se esiste almeno un cammino deterministico dell'albero che termina in uno stato di accettazione (oppure, se ogni cammino termina in uno stato di accettazione), allora $\mathbf{T}_e^{Ndet}(x)$ è il minimo tra $\mathbf{T}_{e,k}^{Ndet}(x)$;*
- (b) se esiste almeno un cammino deterministico dell'albero computazionale che non termina, allora $\mathbf{T}_e^{Ndet}(x)$ è non finito.*

CLASSI DI COMPLESSITÀ

(Hartmanis e Stearns [1965])

- **Classi temporali deterministiche**
 C_t^T o **TIMEF** $[t]$ è la classe delle funzioni computabili in tempo t , ovvero la classe delle funzioni ricorsive computabili mediante una qualche macchina di Turing ad un nastro infinito in entrambe le direzioni che non compie più di $t(x)$ -mosse durante una computazione sull'input x .
TIME $[t]$ è la restrizione di **TIMEF** $[t]$ a valori 0-1, ovvero a insiemi.

CLASSI NON DETERMINISTICHE

- **Accettazione non deterministica di insiemi**

*Un insieme A è accettabile in tempo non deterministico t se esiste una macchina di Turing non deterministica tale che, per ogni x , $x \in A$ se e solo se esiste un cammino dell'albero delle computazioni che termina in uno stato di accettazione sull'input x , che non richiede più di $t(x)$ -passi. **NTIME** $[t]$ è la classe degli insiemi accettabili in tempo non deterministico t .*

LA CLASSE \mathcal{PF} DELLE FUNZIONI COMPUTABILI IN TEMPO POLINOMIALE

(von Neumann [1953], Cobham [1964],
Edmonds [1965])

Definizione.

\mathcal{PF} è la classe delle funzioni computabili in tempo polinomiale nella lunghezza dell'input, ovvero

$$\mathcal{PF} = \bigcup_{n \in \omega} \mathbf{TIMEF}[|x|^n],$$

oppure

$$\mathcal{PF} = \{f : (\exists e)[f \simeq \varphi_e \wedge (\exists n)(\forall \infty x)(\mathbf{T}_e(x) \leq |x|^n)]\}$$

LA TESI DI COBHAM-EDMONDS (O TESI DI COOK-KARP)

(Edmonds [1965], Cobham [1964], Cook [1971], Karp [1971])

Ogni funzione computabile trattabile (feasible) è computabile in tempo polinomiale (nella lunghezza dell'input)

È ragionevole controporre la complessità polinomiale a quella non polinomiale nella caratterizzazione del concetto di *trattabilità*?

Oppure la distinzione è più o meno arbitraria?

ARGOMENTAZIONI A SOSTEGNO DELLA TESI

- (a) Gli algoritmi che lavorano in tempo esponenziale sembrano insensibili all'aumento della velocità di esecuzione (ovvero al progresso tecnologico)
- (b) Esiste una differenza intrinseca nel concetto matematico di '*crescita di una funzione*' quando si passa dai polinomi alla funzione esponenziale
- (c) Il Principio della Riduzione Polinomiale: l'esistenza di una 'invariante' che lascia inalterata una classe di complessità al variare dello strumento tecnico (il modello di calcolo, ad esempio) utilizzato per definirla.

ARGOMENTAZIONI CONTRO LA TESI DI COBHAM-EDMONDS

(a) Consideriamo un algoritmo generico che lavora su un computer capace di compiere un milione di operazioni al secondo. Tale algoritmo impiegherà:

i. **su un input di lunghezza 100:**

- 0,01 secondi se lavora in tempo $|x|^2$
o $|x|^{\log_{10} |x|}$
- 1 secondo se lavora in tempo $|x|^3$
- 1,5 minuti se lavora in tempo $|x|^4$
- 2,45 ore se lavora in tempo $|x|^5$ o $10^{\sqrt{|x|}}$
- 12 giorni se lavora in tempo $|x|^6$
- 3 anni se lavora in tempo $|x|^7$

ii. **su un input di lunghezza 1000:**

- 1 secondo, se lavora in tempo $|x|^2$
- 16,5 minuti se lavora in tempo $|x|^3$
 - $|x|^{\log_{10} |x|}$
- 12 giorni se lavora in tempo $|x|^4$
- 33 anni se lavora in tempo $|x|^5$

Dunque: le funzioni esponenziali sono sicuramente pessime, ma funzioni non polinomiali che crescono lentamente sono sicuramente migliori, anche nel caso di input grandi, rispetto a funzioni polinomiali (anche con esponenti piuttosto piccoli).

(b) **Qual è il significato di upper bounds polinomiali?**

Cosa significa per un algoritmo avere un limite superiore polinomiale nel tempo di esecuzione?

Definizione. *Un algoritmo lavora in tempo polinomiale se esiste un bound polinomiale $|x|^n$ che limita la sua complessità temporale per quasi ogni input.*

Questo significa allora che una quantità finita di argomenti viene tralasciata come insignificante.

La distinzione è ammissibile se l'attenzione è rivolta allo studio di situazioni infinitarie (come nella teoria matematica delle funzioni) ma diventa inaccettabile se cerchiamo di risolvere questioni pratiche.

Infatti, se ammettiamo una quantità finita di eccezioni, allora possiamo classificare un algoritmo come polinomiale (e quindi come trattabile) in base al fatto che il suo comportamento è buono su input che potrebbero non venir mai usati nella pratica.

(c) **Qual è il significato di lower bounds non polinomiali?**

Cosa significa per un algoritmo aver un limite inferiore non polinomiale nel tempo di esecuzione?

Definizione. *Un algoritmo non lavora in tempo polinomiale se, per ogni n , esiste una quantità infinita di input x tale che l'algoritmo impiega su x più tempo rispetto a $|x|^n$.*

Questo significa allora che una quantità infinita di input *worst-case* viene considerata come significativa.

Ancora una volta, la distinzione è ragionevole se l'attenzione è rivolta a comportamenti globali, ma diventa inaccettabile se l'enfasi è posta su questioni pratiche.

Infatti è possibile classificare un algoritmo come non polinomiale (e quindi come non trattabile) a partire dal fatto che il suo comportamento è pessimo su una quantità infinita di input che non verranno mai considerati nella pratica.

- i. A causa della **dimensione**: (gli input *worst-case* più piccoli sono più grandi di tutti gli input significativi dal punto di vista pratico);
- ii. A causa della **distribuzione**: (gli input *worst-case* si trovano in *regioni* diverse rispetto agli input significativi dal punto di vista pratico).

(d) IL PROBLEMA LINEAR PROGRAMMING (LP)

(LP): *data una matrice a di interi con n righe e m colonne, un vettore b di n interi, un vettore c di m interi e un intero k , decidere se esiste un vettore x di numeri razionali tale che $ax \leq b$ e $cx \geq k$.*

van Dantzig [1949] ha escogitato un algoritmo chiamato **Simplex Method** per la soluzione di LP.

Klee e Minty [1972] e Zadeh [1973] hanno dimostrato che Simplex Method è un algoritmo computabile in tempo esponenziale.

Khachian [1979] ha escogitato un algoritmo chiamato **Ellipsoid Method** per LP computabile in tempo polinomiale.

Fatto: benché Ellipsoid Method lavori in realtà in tempo quadratico, i coefficienti del limite quadratico sono ampi, e nella pratica l'algoritmo si comporta in maniera peggiore di Simplex Method!

Inadeguatezza della classificazione dei due algoritmi secondo i parametri standard della Teoria della Complessità.

La problematicità del concetto di **numero trattabile** affonda le sue radici in questioni propriamente logico-filosofiche (prima di toccare i problemi dell'Informatica Teorica o della scienza dei calcolatori nel suo complesso):

- (a) situazioni paradossali come quelle espresse dall'antico **Paradosso del Sorite** o dal **Paradosso di Wang** [1958], legate all'utilizzo di espressioni 'vaghe' come 'mucchio' o 'numero piccolo'.

- (b) concezioni **Ultra-Intuizioniste** e **Ultra-Finitiste** (o **Finitiste Strette**) in filosofia della matematica, dove il significato degli asserti matematici viene collegato a costruzioni eseguibili nella pratica, e non solo in linea di principio.

(c) **Critica all'impredicatività del Principio di Induzione**

Intuitivamente:

Problema: Perché ammettiamo che la funzione esponenziale rappresenti una operazione *significativa*, anche nel caso di espressioni senza possibili realizzazioni fisiche, come $5^{5^{5^5}}$?

Poiché si può dimostrare che la funzione esponenziale genera, solitamente, risultati numericamente significanti a partire da argomenti numericamente significanti: la dimostrazione di questo avviene per **induzione**.

Fatto: in questo caso l'utilizzo dell'induzione è dimostrabilmente **circolare** e **impredicativo** (in particolare, \mathbb{N} viene assunto come una totalità già data). (Nelson [1986], Isles [1992], Leivant [1993; 2001]).

MATEMATICA TRATTABILE

Fatto: è possibile trattare rigorosamente e formalmente il concetto di *numero trattabile*.

Esistono vari studi al riguardo:

Esenin-Volpin [1970]

Parikh [1971]

Simon [1977]

Nelson [1986]

Isles [1992]

Sazonov [1992]

Bellantoni-Hoffman [2000]

Leivant [2002]

CARATTERIZZAZIONI ALTERNATIVE DELLA CLASSE \mathcal{PF}

ALGEBRE DI FUNZIONI PER IL TEMPO POLINOMIALE

Ferreira [1988; 1990]

Sia \mathbb{W} l'algebra di parole generata a partire dai costruttori ϵ , 0 e 1 (di arietà 0,1 e 1, rispettivamente), essenzialmente isomorfa al linguaggio $\{0,1\}^*$ e adatta a descrivere l'albero binario di Cantor $2^{<\omega}$.

Un po' di terminologia...

Ammettiamo innanzitutto la **concatenazione** di due stringhe $\sigma, \tau \in 2^{<\omega}$, indicata mediante $\sigma \oplus \tau$ o, semplicemente, $\sigma\tau$.

Il **prodotto** $\sigma \otimes \tau$ di due stringhe è definito come

$$\sigma \otimes \tau = \underbrace{\sigma \oplus \sigma \oplus \sigma \oplus \dots \oplus \sigma}_{|\tau| \text{-volte}}$$

Ovviamente, per ogni $\sigma, \tau \in 2^{<\omega}$,

$$|\sigma \oplus \tau| = |\sigma| + |\tau|$$

$$|\sigma \otimes \tau| = |\sigma| \cdot |\tau|$$

Definizione. *La classe delle **Funzioni limitanti** è la più piccola classe di funzioni che include la funzione identità, le funzioni proiezione, le funzioni 'prodotto' e concatenazione, e che è chiusa rispetto alla composizione all'attribuzione di valori alle variabili.*

Fatto: ad ogni polinomio $P(X_1, \dots, X_r) \in \mathbb{N}[X_1, \dots, X_r]$ possiamo attribuire una funzione limite (non unica) r -aria t tale che per ogni $\sigma_1, \dots, \sigma_r \in 2^{<\omega}$,

$$|t(\sigma_1, \dots, \sigma_r)| = P(|\sigma_1|, \dots, |\sigma_r|)$$

Esempio: la funzione limitante

$$t(\sigma_1, \sigma_2, \sigma_3) = (\sigma_1 \otimes \sigma_2 \otimes \sigma_2)(\sigma_3 \otimes 11)$$

è associata al polinomio

$$P(X_1, X_2, X_3) = X_1 X_2^2 + 2X_3$$

La funzione **troncamento** $|$ è definita mediante

$$\sigma|_{\tau} = \begin{cases} \gamma & \text{se } \gamma \subseteq^{sw} \sigma \text{ e } |\gamma| = |\tau| \\ \sigma & \text{se } |\sigma| \leq |\tau| \end{cases}$$

dove ' $\gamma \subseteq^{sw} \sigma$ ' significa che γ è una **sottoparola iniziale** di σ , ovvero esiste un δ tale che $\gamma \oplus \delta = \sigma$.

Scriviamo $\gamma \subseteq^{sw^*} \sigma$ se γ è una **sottoparola** di σ , ovvero se esiste una ρ con $\rho\gamma \subseteq^{sw} \sigma$.

Introduciamo adesso uno schema che genera tutte le funzioni computabili in tempo polinomiale.

(a) **Funzioni Iniziali:**

- $E(x) = \emptyset$
- $P_i^n(x_1, \dots, x_n) = x_i$, con $1 \leq i \leq n$
- $C_0(x) = x \oplus 0$
- $C_1(x) = x \oplus 1$
-

$$Q(x, y) = \begin{cases} 1 & \text{se } x \subseteq^{sw} y \\ 0 & \text{altrimenti} \end{cases}$$

(b) Funzioni Derivate

i. $f(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_k(x_1, \dots, x_n)),$

ovvero f è definita a partire da g, h_1, \dots, h_k mediante **composizione**;

ii. $f(x_1, \dots, x_n, \emptyset) = g(x_1, \dots, x_n)$

$$f(x_1, \dots, x_n, y_0) = h_0(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))|_{t(x_1, \dots, x_n, y)}$$

$$f(x_1, \dots, x_n, y_1) = h_1(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))|_{t(x_1, \dots, x_n, y)},$$

dove t è un termine limitante.

Diciamo allora che f è definita a partire da g, h_0, h_1 mediante **iterazione limitata con bound t** .

TEOREMA DI CARATTERIZZAZIONE

Indichiamo con $\mathcal{P}\mathcal{F}^{\mathcal{F}\mathcal{E}\mathcal{R}}$ la classe delle funzioni generate mediante gli schemi i. e ii.

Teorema.

(Ferreira [1988; 1990])

La classe di funzioni $\mathcal{P}\mathcal{F}^{\mathcal{F}\mathcal{E}\mathcal{R}}$ è esattamente la classe $\mathcal{P}\mathcal{F}$ delle funzioni computabili in tempo polinomiale.

Dimostrazione. (intuitiva)

Chiaramente tutte le funzioni della classe $\mathcal{P}\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}$ sono computabili in tempo polinomiale: le funzioni iniziali lo sono, e la computabilità in tempo polinomiale è chiusa rispetto alla composizione e alla iterazione limitata.

Viceversa, supponiamo che $f : (2^{<\omega})^r \rightarrow 2^{<\omega}$ sia una funzione computabile in tempo polinomiale. Dobbiamo dimostrare allora che f è una $\mathcal{P}\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}$ -funzione.

Definiamo, mediante iterazione limitata, una $\mathcal{P}\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}$ -funzione $RUN(\cdot)$ tale che, per ogni $1 \leq j \leq k$ e ogni $\sigma \in 2^{<\omega}$, con $|\sigma| = j$, $RUN(\sigma)$ 'codifica' una configurazione del nastro C_j di una macchina di Turing deterministica *poly-bounded*... \boxtimes

CHIUSURA DELLA CLASSE $\mathcal{P}^{\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}}$

Fatto: la classe $\mathcal{P}^{\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}}$ dei predicati decibibili in tempo polinomiale è chiusa rispetto alle operazioni booleane e alla quantificazione su sottoparole, ovvero alla quantificazione del tipo

$$\begin{aligned} &\exists x(x \subseteq^{sw^*} f(\vec{y}) \wedge \dots) \\ &\forall x(x \subseteq^{sw^*} f(\vec{y}) \rightarrow \dots), \end{aligned}$$

dove f è una $\mathcal{P}^{\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}}$ -funzione.

Siano $\sigma, \tau \in 2^{<\omega}$; allora $\sigma \leq \tau$ sse $|\sigma| \leq |\tau|$.

Problema (fondamentale): la classe $\mathcal{P}^{\mathcal{F}\mathcal{F}\mathcal{E}\mathcal{R}}$ è chiusa rispetto alla quantificazione limitata, ovvero alla quantificazione del tipo

$$\begin{aligned} &\exists x(x \leq f(\vec{y}) \wedge \dots) \\ &\forall x(x \leq f(\vec{y}) \rightarrow \dots) \quad ??? \end{aligned}$$

Questo problema è meglio noto come *questione $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$* .

POLINOMIALITÀ NON DETERMINISTICA

Teorema. (Caratterizzazione)
(Ferreira [1988; 1990])

*La classe **NP** dei predicati accettati in tempo polinomiale (non deterministico) mediante una macchina di Turing non deterministica coincide con la classe dei predicati del tipo*

$$\exists x \leq f(\vec{y})Q(x, \vec{y}),$$

dove f è una funzione computabile in tempo polinomiale e Q è $\mathcal{P}\mathcal{F}\mathcal{E}\mathcal{R}$ -decidibile (predicati di questo tipo sono detti anche ' Σ_1^p -predicati').

*Dunque le classi **NP** e Σ_1^p coincidono.*

IL TEMPO POLINOMIALE DETERMINISTICO: LA CLASSE P

Definizione. \mathbf{P} è la classe degli insiemi computabili in tempo polinomiale deterministico nella lunghezza dell'input, ovvero

$$\mathbf{P}_n = \mathbf{TIME}[|x|^n].$$

In altri termini, \mathbf{P} coincide con l'unione infinita delle classi di complessità che hanno come funzione limite un qualunque polinomio di grado finito.

IL PROBLEMA 2-SAT (SODDISFACIBILITÀ DELLE FORMULE DI HORN)

Una formula proposizionale si dice **Horn** se è una congiunzione di disgiunzioni di letterali di cui al più uno è positivo. Ovvero, sia $U = \{u_1, u_2, \dots, u_n\}$ un insieme di n -variabili booleane. Una formula booleana f è detta in **forma normale congiuntiva (CNF)** se $f = c_1 \wedge c_2 \wedge \dots \wedge c_m$; le m -clausole c_i sono del tipo $c_i = (l_{i_1} \vee l_{i_2} \vee \dots \vee l_{i_k})$, dove ogni l_{i_j} indica un letterale, ovvero una variabile proposizionale o la negazione di una variabile proposizionale.

Una formula proposizionale è **soddisfacibile** se esiste una valutazione proposizionale che la rende vera.

Una assegnazione di valori per U è una funzione $t : U \rightarrow \{T, F\}$ che assegna ad ogni variabile il valore T (true) o F (false). Un letterale l è T se $l = (u_j)$ e $t(u_j) = T$ oppure se $l = \neg u_j$ e $t(u_j) = F$.

Una clausola è soddisfatta da un'assegnazione se almeno un letterale incluso in essa è in T . La formula f è **soddisfatta** se ogni m -clausola è soddisfatta.

2-SAT

2-SAT: il problema 2-SAT consiste nel trovare un algoritmo che decida, data una formula booleana Horn f in forma normale congiuntiva tale che ogni clausola contiene esattamente due letterali, se esiste un'assegnazione di valori che soddisfa f .

Teorema. (Cook [1971])

$$2 - SAT \in \mathbf{P}.$$

Se f è la formula Horn si può dimostrare che esiste un algoritmo che termina (dichiarando la formula soddisfacibile o dichiarandola insoddisfacibile) dopo un numero di passi pari al massimo alla lunghezza di f .

TEMPO POLINOMIALE NON DETERMINISTICO. LA CLASSE NP

La classe **NP** è la classe degli insiemi e dei linguaggi computabili in tempo polinomiale (non deterministico) mediante macchine di Turing non deterministiche.

Definizione (Bennett [1962], Edmonds [1965], Cook [1971])

*La classe **NP** è la classe degli insiemi accettati in tempo polinomiale non deterministico nella lunghezza dell'input, ovvero*

$$\mathbf{NP}_n = \mathbf{NTIME}[|x|^n], \text{ oppure}$$

$$\mathbf{NP} = \bigcup_{n \in \omega} \mathbf{NP}_n .$$

ESEMPI DI PROBLEMI IN NP

Il problema SAT: *data una formula proposizionale ϕ , si tratta di decidere se ϕ è soddisfacibile oppure no.*

Nel caso di SAT, ad esempio, non si conoscono algoritmi per scoprire se una espressione è soddisfacibile sostanzialmente migliori di quelli che effettuano tutte le prove, la cui complessità è dell'ordine di $\mathcal{O}(2^n)$.

Il problema Fattorizzazione...

IL PROBLEMA TSP

(*Travelling Salesman Problem*)

In una istanza di TSP abbiamo un intero $n > 0$ e la distanza tra ogni coppia di n -città nella forma di una matrice $n \times n [d_{i,j}]$, dove $d_{i,j} \in \mathbb{Z}^+$. Un *viaggio* è un cammino finito che visita ogni città esattamente una volta. Il problema TSP consiste nel trovare un viaggio di minima lunghezza totale. Possiamo scegliere: $F = \{\text{tutte le permutazioni cicliche } \pi \text{ su } n\text{-oggetti}\}$. Una permutazione ciclica π rappresenta un viaggio se interpretiamo $\pi(j)$ come la città visitata dopo la j -esima città, con $j = 1, \dots, n$. La funzione c rappresenterà una iniezione da π in

$$\sum_{j=1}^n d_{j,\pi(j)}$$

DETERMINISMO vs NONDETERMINISMO

Fatto: dato che le macchine di Turing deterministiche sono un caso particolare delle macchine di Turing non deterministiche, dalle definizioni introdotte segue immediatamente che un problema in **P** è anche in **NP**.

Il problema di decidere se valga o meno l'implicazione inversa costituisce uno dei più importanti problemi ancora irrisolti.

PROBLEMA:

$$P \stackrel{?}{\neq} NP$$

LA QUESTIONE $P \stackrel{?}{\neq} NP$

La classe degli insiemi computabili in tempo polinomiale mediante macchine di Turing deterministiche coincide con la classe degli insiemi accettabili in tempo polinomiale non deterministico nella lunghezza dell'input?

Ovvero, qual è il rapporto tra le classi **P** e **NP**?

È possibile risolvere problemi combinatori finiti senza utilizzare metodi che facciano leva sulla mera forza bruta? In generale, è possibile esibire un metodo algoritmico che permetta di evitare, nella risoluzione di un problema (apparentemente) intrattabile, il ricorso a ricerche esaustive mediante la sola forza bruta?

La questione $P \stackrel{?}{\neq} NP$ coincide con l'esistenza o meno di un tale metodo.

LE NOZIONI DI p -RIDUCIBILITÀ E NP-COMPLETEZZA

Definizione. Sia \mathcal{L}_i un linguaggio su Σ_i , con $i = 1, 2$. Allora $\mathcal{L}_1 \leq_p \mathcal{L}_2$ (\mathcal{L}_1 è **polinomialmente riducibile** a \mathcal{L}_2 , o **p -riducibile**) se e soltanto se esiste una funzione computabile in tempo polinomiale $f : \Sigma_1^* \rightarrow \Sigma_2^*$ tale che $x \in \mathcal{L}_1 \leftrightarrow f(x) \in \mathcal{L}_2$, per ogni $x \in \Sigma_1^*$.

Definizione. Un linguaggio \mathcal{L} è **NP-completo** se e soltanto se \mathcal{L} è in **NP** e vale $\mathcal{L}' \leq_p \mathcal{L}$, per ogni linguaggio \mathcal{L}' in **NP**.

Fatto: se Φ è un problema **NP**-completo, e se avessimo un algoritmo efficiente per Φ , allora avremmo un algoritmo efficiente per ogni problema in **NP**.

Un algoritmo *poly-time* per un qualsiasi problema **NP**-completo porterebbe immediatamente ad un algoritmo *poly-time* per tutti i problemi in **NP**.

Problema: esistono problemi dimostrabilmente **NP**-completi?

LA NP-COMPLETEZZA DI SAT

Teorema (Cook [1971])

SAT è **NP-completo**.

Dimostrazione. (intuitiva)

Occorre far vedere che

1. SAT è risolubile in tempo polinomiale mediante una **TM** non deterministica;
2. ogni problema combinatorio risolubile mediante una **TM** non deterministica in tempo polinomiale può essere ridotto in tempo polinomiale al problema di determinare se una data formula proposizionale appartiene a $\{TAUT\}$; ovvero, se A è un problema in **NP**, allora A è polinomialmente trasformabile in SAT.

Data una stringa x , dobbiamo costruire una formula $F(x)$ -utilizzando solo il fatto che $A \in \mathbf{NP}$ - tale che x è una stringa 'sì' di A sse $F(x)$ è soddisfacibile. Per fare ciò si deve considerare un algoritmo *verifica-certificati* \mathcal{A} per A il quale, per l'ipotesi secondo cui $A \in \mathbf{NP}$, opera entro un *bound* polinomiale $\rho(n)$

⊠

Fatto: se \mathcal{L} è in \mathbf{NP} , allora per dimostrare che \mathcal{L} è \mathbf{NP} -completo basta mostrare che SAT (o TSP o FAT , o...) $\leq_p \mathcal{L}$

Esistono centinaia di problemi \mathbf{NP} -completi (Karp [1972]...)

Osservazione: $\mathbf{P}=\mathbf{NP}$ sse SAT (o un qualsiasi altro problema \mathbf{NP} -completo) è in \mathbf{P} .

CRIPTOGRAFIA

Questioni di Teoria della Complessità hanno importanti applicazioni nella crittografia moderna:

- sistemi di crittatura pubblici e privati
- sicurezza delle transazioni finanziarie su Internet

Teorema. *Se $P=NP$, allora la crittografia è impossibile.*

Un algoritmo efficiente (polinomiale) per SAT potrebbe violare un qualsiasi sistema crittografico.

Esempio. Il sistema DES (*Data Encryption Standard*) utilizzato per la sicurezza delle password in UNIX:

DES: password $w \longrightarrow$ forma criptata E

E è pubblica, e l'inversione è computazionalmente intrattabile.

Fatto: ciascuna E può essere iniettata in una CNF formula F_E con circa 25.000 clausole con tre letterali: una qualsiasi attribuzione di verità (soddisfacibile) per F_E porta ad una corrispondente password w .

Un algoritmo dell'ordine di n^2 per SAT codificherebbe DES in 10 secondi (alla velocità di 10^9 operazioni al secondo).

Assunzione: SAT è in NP.

IL PROBLEMA FAT

Fattorizzazione è in **NP**, e non si è ancora riusciti a dimostrare se è o meno **NP**-completo.

Agrawal, Katalina e Saxena [2001?] hanno scoperto un algoritmo che riesce a stabilire, in tempo $\mathcal{O}(n^{12})$, se un numero di n -cifre binarie è primo oppure no (esistevano solo algoritmi probabilistici che vi riuscivano). La scoperta tuttavia non modifica il problema della fattorizzazione, che rimane in **NP**: infatti, dopo aver scoperto che un numero non è primo, occorre trovarne i fattori primi, e questo richiede ancora una ricerca per tentativi.

CURIOSITÀ: FATTORIZZAZIONI QUANTISTICHE

Teorema (Shor [1997])

Il problema della fattorizzazione in numeri primi può essere risolto in tempo polinomiale su un macchina di Turing quantistica.

Per fattorizzare un numero di n di l bit, il migliore algoritmo classico conosciuto è il *Number Field Sieve* (Lenstra [1993]), che impiega un tempo di circa

$$\mathcal{O}(\exp(c l^{1/3} \log^{2/3} l)).$$

L'algoritmo quantistico di fattorizzazione impiegherebbe invece, su un computer quantistico, $\mathcal{O}(l^2 \log l \log \log l)$ -passi.

P=NP??

Come dimostrarlo?

‘Semplicemente’ esibendo un algoritmo efficiente per SAT (o per Fattorizzazione, o per TSP...)

Nessuno ci è ancora riuscito.

Ad esempio, il migliore algoritmo deterministico per 3-SAT impiega un tempo dell'ordine di 1505^n , per n -variabili ($\approx 2^n$).

$P \neq NP??$

- **Metodi ‘classici’:**

1. *diagonalizzazione*

2. *riduzione (ex., ad HP (Halting Problem))*

Fatto: tali metodi sono però relativizzabili!

Baker, Gill e Solovay [1975]: non è possibile relativizzare la questione $P \stackrel{?}{\neq} NP$ (diversamente, ad esempio, da quanto accade nell’ambito della ricorsività relativa, dove per ogni oracolo A esiste un insieme ricorsivamente enumerabile in A che non è ricorsivo in A).

In particolare, BGS [1975] hanno dimostrato che

1. esiste un oracolo **PSPACE**-completo e ricorsivo A tale che $\mathbf{P}^A = \mathbf{NP}^A$;
2. esiste un oracolo **PSPACE**-completo e ricorsivo B tale che $\mathbf{P}^B \neq \mathbf{NP}^B$.

- **Complessità dei Circuiti Booleani:**

Qual è l'ampiezza di un circuito booleano per SAT??

(ogni problema in \mathbf{P} può essere risolto mediante circuiti booleani di dimensione polinomiale)

Risultato: nessun *bound* significativo. Il migliore limite superiore per SAT è esponenziale, e $\text{BC}(\text{SAT}) \leq 4^n$ (Razborov [1987]).

- **Natural Proofs:**

Il metodo standard per dimostrare limiti inferiori per circuiti booleani (Razborov e Rudich [1997]).

Risultato: nessun *upper* o *lower bound* significativo.

- **Algoritmi Random:**

A) Esistono $\mathcal{L} \in \mathbf{EXPTIME}$ e $\epsilon > 0$ tali che, per ogni famiglia di circuiti $\langle B_n \rangle$ che computa \mathcal{L} e per tutti gli n sufficientemente grandi, B_n ha almeno $2^{\epsilon n}$ -porte.

Impagliazzo e Wigderson [1997]:

1. se A, allora **BPP=P**

2. se $\neg A$, allora **P \neq NP**

- **Sistemi di Dimostrazione Proposizionale...**
- **Metodi di risoluzione...**

BIBLIOGRAFIA

Trattazioni Generali...

- Papadimitriou, C. H. *Computational Complexity*, Addison Wesley, 1994.
- Odifreddi, P. *Classical Recursion Theory, Volume II*, North Holland, Amsterdam, 1999.
- van Leeuwen ed. *Handbook of Theoretical Computer Science*, North Holland, 1990.
- Papadimitriou, C. H. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, 1982.

Origini...

- Cobham, A. *The Intrinsic Computational Difficulty of Functions*, Proc. of the 1964 International Congress for Logic, Methodology and the Philosophy of Science, North Holland, Amsterdam, 1964.
- Edmonds, J. *Paths, Trees and Flowers*, Canadian Journal of Mathematics, 17, 1965.
- Cook, S. A. *The Complexity of Theorem Proving Procedures*, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151-158, 1971.
- Karp, R. M. *Reducibility Among Combinatorial Problems*, Complexity of Computer Computations, Plenum Press, 85-103, 1972.

Trattazioni avanzate...

- Immerman, N. *Descriptive Complexity*, Springer Verlag, 1999.
- Leivant, D. ed, *Logic and Computational Complexity*, Lectures Notes in Computer Science, 960, 1994.
- Ferreira, F. *Polynomial Time Computable Arithmetic and Conservative Extension*, Ph. D. Dissertation, Pennsylvania State University, 1988.