NP COMPLETEZZA

[Deme, seconda edizione] cap. 16

fino a Sezione 16.3.1 inclusa

Teoria della Complessità

Abbiamo spesso parlato di complessità (algoritmica) polinomiale, dicendo che eravamo «contenti» se un algoritmo aveva complessità polinomiale (cioè, se risolveva un'istanza del problema in tempo polinomiale). D'altro canto, eravamo scontenti se aveva complessità esponenziale.

La **Teoria della Complessità** è la branca dell'informatica che studia la complessità dei **problemi**.

La **complessità** di un **problema** (non di un algoritmo!) è la complessità del «**migliore**» **algoritmo che lo risolve**, sia che questo algoritmo sia noto, sia che esso sia ipotizzato.

In altre parole, la complessità di un problema è rappresentata dalle risorse di calcolo minime necessarie a risolvere una sua istanza.

Turing Completezza

Siamo sicuri di fare uno studio «accurato» e «generale».

La complessità di un problema, infatti, **non dipende** nemmeno dal **linguaggio di programmazione** che usiamo, né dal **tipo di calcolatore**, a patto che esso sia un modello di calcolo **Turing-completo** o Turing-equivalente
(tutti i modelli di calcolo che conoscete sono al massimo Turing-completi)

Esistono modelli di calcolo «migliori», ma questo non è argomento di un corso di algoritmi…

Problemi (matematici)

Per prima cosa, definiamo più formalmente un **problema P** come una relazione

$$\mathbf{P} \subseteq \mathbf{I} \times \mathbf{S}$$

Dove **I** è l'insieme delle possibili **istanze** in ingresso ed **S** è l'insieme delle possibili **soluzioni** del problema.

A questo punto possiamo formalizzare meglio i tipi di problemi che già conosciamo

Tipi di Problemi

Problemi di Decisione: problemi che richiedono di **verificare** una certa **proprietà** sull'input. Sono problemi per cui $\mathbf{S = \{0, 1\}}$.

Esempio: dato un grafo G, dire se è connesso.

Problemi di Ricerca: data un'istanza di un problema, restituire una **soluzione ammissibile**. Data un'istanza $\mathrm{x} \in \mathrm{I}$ di un problema P, **trovare un s tale che $(\mathbf{x}, \boldsymbol{s}) \in \mathbf{P}$**

Esempio: dato G connesso, trovare un albero ricoprente di G.

Problemi di Ottimizzazione: data un'istanza di un problema, restituire una **soluzione ottima** secondo un determinato criterio. Possono essere di massimizzazione o minimizzazione. Equivale a **trovare un s* tale che $(\mathbf{x}, \boldsymbol{s}^*) \in \mathbf{P}$** ed **s* è il migliore degli s tali che $(\mathbf{x}, \boldsymbol{s}) \in \mathbf{P}$ (secondo un dato criterio)**

Esempio: dato G connesso, trovare un minimo albero ricoprente di G.

Classi di Complessità

Per semplicità, in queste slide ci focalizziamo su problemi di decisione. Tuttavia, il discorso può essere generalizzato agli altri.

Vogliamo caratterizzare i problemi in base alle **risorse di calcolo richieste per risolverli**.

Una **classe di complessità** è un **insieme di problemi** che possono essere risolti usando le **stesse risorse** di calcolo.

Dato n, che indica la dimensione dell'input (ad es. dimensione del vettore di input per problemi di ordinamento), possiamo classificare i problemi in base al tempo o allo spazio richiesti.

TIME(f(n)) è l'insieme dei problemi che possono essere risolti in tempo O(f(n))

SPACE(f(n)) è l'insieme dei problemi che possono essere risolti con spazio O(f(n))

ESEMPIO: verificare se un elemento appartiene ad un albero binario di ricerca ha complessità TIME(log n) e SPACE(n).

Classe P

Per semplicità, le classi di complessità sono "raggruppate" nelle cosiddette **classi unione**.

La classe **P** è la classe dei **problemi risolvibili** in **tempo polinomiale** rispetto a n

$$P = \bigcup_{c=0}^{\infty} TIME(n^c)$$

(notate che $O(\mathrm{n} \log n) = O(n^{1+\epsilon})$ per ogni $\epsilon > 0$, quindi i problemi risolvibili in tempo n log n appartengono a P)

Nota. La classe P è storicamente la più importante. Infatti rappresenta la classe dei **problemi** "**trattabili**" (anche se questa definizione è un po' approssimativa).

Classe PSPACE

La classe **PSPACE** è la classe dei **problemi risolvibili** in **spazio polinomiale** rispetto a n

$$PSPACE = \bigcup_{c=0}^{\infty} SPACE(n^c)$$

Notate che potendo fare al più un numero polinomiale di operazioni, e quindi anche di accessi in memoria, $\boldsymbol{P \subseteq PSPACE}$

Classe EXPTIME

La classe **EXPTIME** è la classe dei **problemi risolvibili** in **tempo esponenziale** rispetto a n

$$EXPTIME = \bigcup_{c=0}^{\infty} TIME(2^{n^c})$$

Immaginiamo di avere n^c locazioni di memoria (per semplicità binarie). Il sistema si può trovare in al massimo 2^{n^c} stati diversi. Quindi $\boldsymbol{PSPACE \subseteq EXPTIME}$.

Inclusione Propria

L'unica inclusione propria dimostrata è

$$P \subset EXPTIME$$

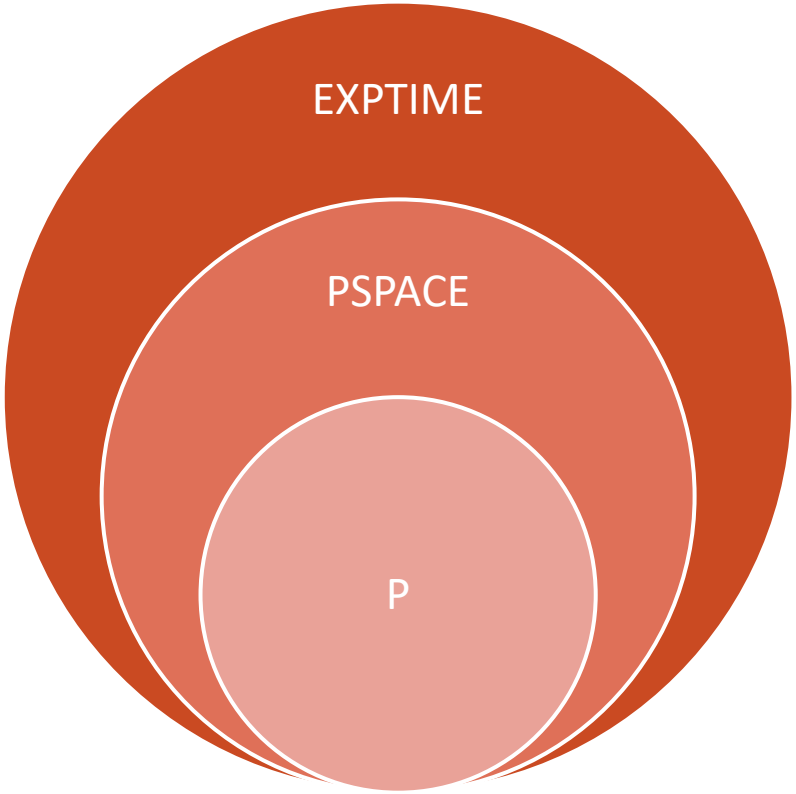
(cioè esiste almeno un problema che non può essere risolto in tempo polinomiale)

Capire se

$P \subset PSPACE$ e

$PSPACE \subset EXPTIME$

sono problemi aperti (ma si pensa che sia così).



$P \subset PSPACE$?

Introduciamo un problema appartenente a **PSPACE** per cui **non esistono algoritmi** risolventi **polinomiali**.

Espressione in Forma Normale Congiuntiva: congiunzione di disgiunzioni (**clausole**) di variabili booleane (**letterali**). Ad esempio

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Il problema della soddisfacibilità (dire se **esiste almeno un'assegnazione $x_1 = z_1, x_2 = z_2, \dots x_n = z_n$ tale che l'espressione sia vera**) di espressioni di questo tipo appartiene a **PSPACE**.

Non si conoscono algoritmi che lo risolvono in tempo polinomiale.

$P \subset PSPACE$?
Un algoritmo (esponenziale – quindi non P – ma che appartiene a PSPACE)

Identifichiamo con e$(x_1, x_2, \dots x_n)$ la formula.

soddisfacibilità (e$(\mathbf{x_1}, \mathbf{x_2}, \dots \mathbf{x_n})$, assegnazioni $\mathbf{z_1}, \mathbf{z_2}, \dots \mathbf{z_{i-1}}$, i)
  if i = n+1 **return** e$(z_1, z_2, \dots z_n)$
  else
    v_0 <- soddisfacibilità (e$(x_1, x_2, \dots x_n)$, $z_1, z_2, \dots z_{i-1}$, **0**, i+1)
    v_1 <- soddisfacibilità (e$(x_1, x_2, \dots x_n)$, $z_1, z_2, \dots z_{i-1}$, **1**, i+1)
    return $v_0 \vee v_1$

L'algoritmo ha complessità $O(2^n)$ (prova tutte le alternative).

Al termine dell'esecuzione, se la formula è soddisfacibile, $z_1, z_2, \dots z_n$ **è una assegnazione che rende vera la formula** (quindi una soluzione dell'equivalente problema di ricerca)

Certificati

Un **certificato** è un oggetto y, dipendente dall'istanza x e dal problema P, che possa giustificare $(x, 1) \in P$ (per problemi **decisionali**).

Ad esempio, nel problema della soddisfacibilità, l'assegnazione $z_1, z_2, \dots z_n$ è un certificato.

Per il problema di dire se un grafo è connesso, un **albero ricoprente** è un certificato.

Possiamo ora riscrivere i nostri algoritmi in 2 fasi:

una di **costruzione** ed

una di **verifica**

for ogni clausola c appartenente ad e **then**
    if c non è vera dati $z_1, z_2, \dots z_n$ **return** 0
return 1

} **fase di verifica**

Non Determinismo

Definiamo una funzione **indovina z** $\in \{0, 1\}$ che può indovinare (in tempo costante) un **giusto** valore booleano.

(a volte indovina è chiamata **oracolo**).

L'algoritmo precedente diventerebbe

soddisfacibilità (e$(x_1, x_2, \ldots x_n)$**)**
  for i = 1..n
    z_i **<- indovina** x_i **} fase di costruzione**
  for ogni clausola c appartenente ad e **then**
    if c non è vera dati $z_1, z_2, \ldots z_n$ **return** 0 **} fase di verifica**
  return 1

L'algoritmo ha complessità $O(n + m)$ dove m è il numero di clausole.

Non Determinismo - II

La funzione **indovina** ovviamente **non esiste**. Tuttavia ci permette di introdurre una nuova classe di algoritmi (e di problemi)

Chiamiamo gli algoritmi sviluppati con il suo uso **non deterministici**, perché la loro computazione non è deterministica (cioè **il passo successivo della computazione non è determinato solo dallo stato della computazione stessa**).

Notiamo però nell'algoritmo precedente che solo la fase di costruzione fa uso di indovina, mentre **la fase di verifica è deterministica**…

La Classe NP

Chiamiamo **NTIME(f(n))** la classe di problemi risolvibili da un algoritmo **non deterministico** in **tempo O(f(n))**.

La classe **NP** è la classe di **problemi risolvibili** in **tempo polinomiale** da un algoritmo **non deterministico**

$$NP = \bigcup_{c=0}^{\infty} NTIME(n^c)$$

O, equivalentemente, NP è la **classe** di **problemi** per cui la fase di **verifica** è fatta (**deterministicamente**) in **tempo polinomiale**.

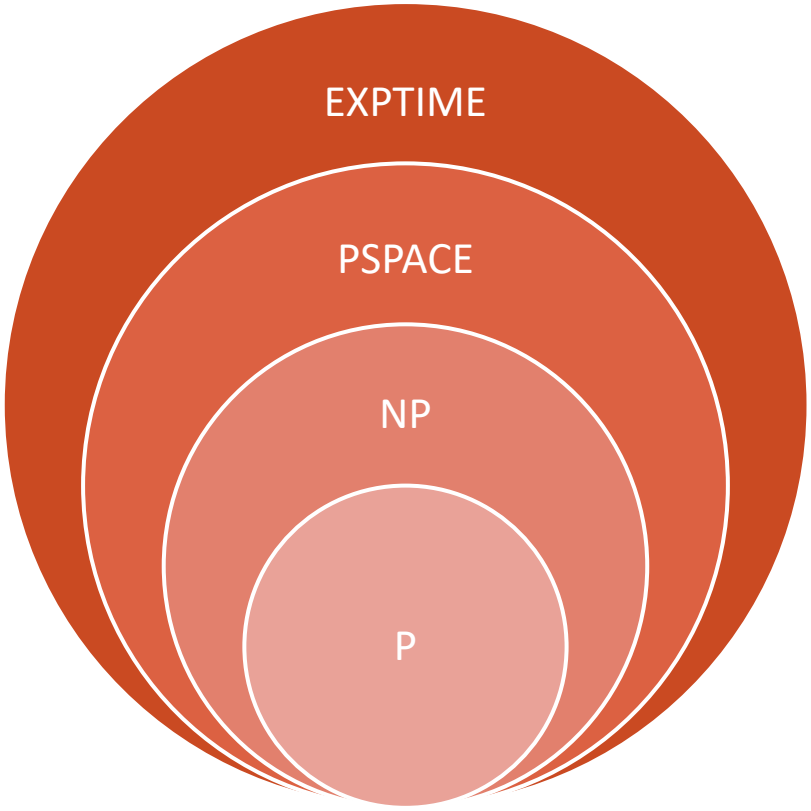
Ovviamente, un algoritmo deterministico è un caso particolare di uno non deterministico, quindi $P \subseteq NP$. Inoltre, affinché la verifica sia fatta (deterministicamente) in tempo polinomiale, il certificato deve occupare spazio polinomiale, quindi $NP \subseteq PSPACE$.

La nuova gerarchia

Come per la gerarchia vista precedentemente, non è noto, ma si sospetta, che

$\boldsymbol{P \subset NP}$ e

$NP \subset PSPACE$



$NP \subset PSPACE$?

Introduciamo un problema appartenente a **PSPACE** ma che si sospetta non appartenere a NP.

Formula Booleana Quantificata: è un'espressione in forma normale congiuntiva preceduta da una serie di quantificatori universali ∀ ed esistenziali ∃ che legano tutte le variabili. Ad esempio

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

Data tale formula, il problema delle formule booleane quantificate richiede di verificare se essa è **vera**.

$NP \subset PSPACE$?
Un algoritmo (non NP, ma che appartiene a PSPACE)

Identifichiamo con e$(x_1, x_2, \dots x_n)$ la formula e $\mathcal{E}_i x_i$ i quantificatori.

quantificata ($\boldsymbol{\mathcal{E}_i x_i, \dots \mathcal{E}_n x_n}$: **e**$\boldsymbol{(x_1, x_2, \dots x_n)}$, $\boldsymbol{z_1, z_2, \dots z_{i-1}}$, **i**)
  if i = n+1 **return** e$(z_1, z_2, \dots z_n)$
  else
    v_0 <- quantificata ($\mathcal{E}_{i+1} x_{i+1}, \dots \mathcal{E}_n x_n$: e$(x_1, x_2, \dots x_n)$, $z_1, z_2, \dots z_{i-1}$, 0, i+1)
    v_1 <- quantificata ($\mathcal{E}_{i+1} x_{i+1}, \dots \mathcal{E}_n x_n$: e$(x_1, x_2, \dots x_n)$, $z_1, z_2, \dots z_{i-1}$, 1, i+1)
    if $\mathcal{E}_i = \exists$ **return** $v_0 \vee v_1$
    else return $\boldsymbol{v_0 \wedge v_1}$ //la grande differenza è nel caso di $\mathcal{E}_i = \forall$

L'algoritmo ha complessità $O(2^n)$.

Notiamo che è molto simile a quello proposto per le Espressioni in Forma Normale Congiuntiva.

$NP \subset PSPACE$? - II

Notiamo che l'algoritmo per le Formule Booleane Quantificate è molto simile a quello proposto per le Espressioni in Forma Normale Congiuntiva.

Diversamente dal primo, la sua fase di **costruzione** **non può essere eseguito in tempo polinomiale nemmeno usando la funzione indovina**.

In questo caso il **certificato** da **verificare** ha esso stesso **dimensione esponenziale** (per ogni $\forall x_i$, dobbiamo verificare la formula sia per $x_i = 0$ che per $x_i = 1$). Di conseguenza la **verifica** ha essa stessa una complessità **esponenziale** $O(2^n)$.

Problemi come questo (appartenenti a PSPACE ma probabilmente non ad NP) suggeriscono (ma non ne siamo sicuri!) che

$NP \subset PSPACE$.

$P \subset NP$ o $P = NP$?

Il confine tra P ed NP è quello più importante in teoria della complessità. Infatti, è quello che separa i problemi «trattabili» da quelli «non trattabili» (almeno con facilità).

Cerchiamo di capire come arrivare ad una conclusione per questa domanda.

Prima di tutto cerchiamo un modo per **confrontare i problemi** tra di loro

Riducibilità Polinomiale

Diciamo che un problema (decisionale) $P1 \subseteq I1 \times \{0,1\}$ **è riconducibile polinomialmente ad un problema** (decisionale) $P2 \subseteq I2 \times \{0,1\}$ se e solo se

- Esiste una funzione che, data un'istanza $x1 \in I1$ di P1, la **trasforma** in un'istanza $x2 \in I2$ di P2 usando **tempo ("abbastanza") polinomiale**
- Per ogni coppia di istanze x1 e x2 così costruite, **la soluzione di x1 è la stessa soluzione di x2**

PROPRIETA': **Se P2 appartiene alla classe P, anche P1 appartiene a P**.

DIMOSTRAZIONE: la definizione di riducibilità polinomiale suggerisce un algoritmo polinomiale per P1. Basta risolvere ogni istanza x1 di P1 trasformandola (polinomialmente) in x2 e risolvendo x2 con il rispettivo algoritmo per P2 di complessità polinomiale.

Riducibilità Polinomiale - esempio

Problema del Cammino (lungo) semplice (P2): dato un grafo G, due vertici s e t ed una lunghezza k, verificare se esiste un cammino semplice in G da s a t di lunghezza almeno k.

Problema del Ciclo Hamiltoniano (P1): dato un grafo G, verificare se esiste un **ciclo che visita ciascun vertice una volta sola**.

È facile notare che per risolvere il Problema del Ciclo Hamiltoniano basta trovare un arco (u,v) tale che esista tra i due vertici un cammino semplice di lunghezza k = n-1.

Banalmente, dobbiamo fare questa ricerca per ogni arco (u,v), quindi la complessità della trasformazione è O(m), che è polinomiale.

Allora il Problema del Ciclo Hamiltoniano (P1) è riconducibile polinomialmente al Problema del Cammino semplice (P2)

$P \subset NP$ o $P = NP$? – II

Possiamo definire le seguenti classi.

Problemi NP-hard: un problema decisionale D si definisce NP-hard (o NP-arduo) se **ogni problema $Q \in NP$** è **riconducibile** a D in **tempo polinomiale**.

Esistono problemi NP-hard sia appartenenti ad NP che a classi superiori.

Problemi NP-completi: un problema decisionale D si definisce NP-completo se è **NP-hard ed appartiene alla classe NP**.

I problemi NP-completi sono quelli di maggiore importanza: **basterebbe dimostrare che uno di essi appartiene a P per poter dire che $P = NP$**. Tuttavia, ciò non è mai stato dimostrato e si sospetta che ciò non sia nemmeno possibile.

Esistono problemi NP-completi?

Sì, ma in genere la dimostrazione è complessa ed esula dallo scopo di questo corso. Ne vediamo un semplice esempio:

Halt limitato: dato un **programma X** ed un **intero k**, verificare se **esiste <u>un input</u> per cui X termina in al massimo k passi**.

$$L - HALT(X,k) = \begin{cases} 1 \; se \; esiste \; input \; i \; tale \; che \; X(i) \; termina \; in \; al \; più \; k \; passi \\ 0 \; altrimenti \end{cases}$$

Appartiene ad **NP** perché la fase di verifica (usando come certificato i) può essere effettuata in tempo **polinomiale** simulando X(i) per k passi.

Esistono problemi NP-completi?

Ora dimostriamo che **ogni problema $\boldsymbol{Prob} \in \boldsymbol{NP}$ è riducibile polinomialmente ad halt limitato (cioè L-HALT è NP-hard)**.

Sia $Prob \in NP$, allora **esiste un programma C** che verifica un certificato in tempo polinomiale **p(n)**.

Allora è possibile costruire un **programma C'** che va in **loop** ogniqualvolta la risposta di **C** sia **negativa**, **termina altrimenti**.

C'(i)

if C(i) **return** 1
else loop

Per semplicità, consideriamo solo problemi Prob decisionali ($s \in \{0,1\}$)

Esistono problemi NP-completi?

Ma allora possiamo riscrivere Prob come il problema dell'halt limitato, con C' come programma X e p(n) come k (il numero di passi). L'halt limitato restituirà **1 se esistono dei dati (un certificato) per cui C' termina in p(n) passi**, **0 altrimenti** (quindi, quando non esiste una soluzione).

$$L - HALT(C', p(n))$$
$$= \begin{cases} 1 \; se \; esiste \; i \; t.c. \; C'(i) \; termina \; in \; p(n) \; passi \\ 0 \; altrimenti \end{cases}$$

Ma se $L - HALT(C', p(n)) = 1$ vuol dire che Prob ha soluzione (s = 1), e se $L - HALT(C', p(n)) = 0$ Prob non ha soluzione (s = 0)!

quindi **Prob è riconducibile polinomialmente al problema dell'halt limitato**

Quindi il problema dell'halt limitato è **NP-hard**

Essendo sia NP che NP-hard, è **NP-completo**

Problemi Indecidibili

Concludiamo questo discorso con un'osservazione. EXPTIME non è la classe più «difficile».

Esistono infatti una serie di problemi definiti **Indecidibili**: non solo **non si conoscono algoritmi per risolverli**, ma è stato dimostrato che tali algoritmi **non possono esistere**, nemmeno utilizzando tempo e spazio infiniti.

Problemi Indecidibili – Halt

Il **problema dell'halt è un problema indecidibile**.

Definire un programma **halt(P, i)** che restituisca 1 se il programma Prog (con un certo input i) termina in un numero finito di passi, 0 altrimenti.

DIMOSTRAZIONE: se potessimo scrivere halt, potremmo scrivere il programma g come

g(Prog)

if halt(Prog, Prog) **loop**

else return 0

(è possibile dare in input ad un algoritmo se stesso)

A questo punto, quale sarebbe l'output di g(g)? **g(g) termina se e solo se g(g) non termina**, e questa è una **contraddizione**.

Cosa devo aver capito fino ad ora

- Problemi decisionali, di ricerca e di ottimizzazione
- Complessità di un problema
- Classi di complessità
- Algoritmi non deterministici
- Problemi NP-hard ed NP-completi e P = NP?
- Problemi Indecidibili (problema dell'halt)

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente