Algoritmi di approssimazione

Corso di Laurea Magistrale in Matematica

A.A. 2014/2015

Algoritmi di approssimazione: elementi di complessità

Sommario

Algoritmi di approssimazione: elementi di complessità

Sommario

Informazioni generali sul corso

- Docenti: Mauro Leoncini, Manuela Montangero
- Sito web del gruppo di algoritmi: http://algogroup.unimore.it
- Materiale didattico, avvisi, esami, ...: dal sito del gruppo seguire il link alla didattica
- Ricevimento: dal sito del gruppo seguire i link alle pagine dei docenti
- Modalità d'esame: solo orale
- CFU assegnati: 6

Argomenti trattati

- Problemi *computazionalmente difficili*
- Elementi di Teoria della **NP**-completezza
- *Algoritmi di approssimazione* per problemi **NP**-hard
- Algoritmi basati su programmazione lineare intera
- Algoritmi euristici

Algoritmi di approssimazione: elementi di complessità

Sommario

Nozioni base di algoritmica

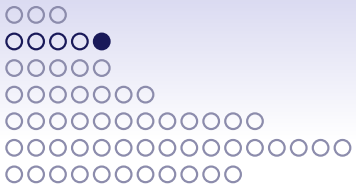
- Problemi e istanze
  - il *problema* dell'ordinamento
  - un ben preciso insieme di chiavi da ordinare (*istanza*)
- Tipi di problemi
  - *decisionali*: sono quelli che ammettono risposta SI/NO, o VERO/FALSO (ad esempio: il numero p è primo?)
  - *di ricerca*: sono quelli che chiedono di trovare una, o eventualmente l'unica, soluzione (ad esempio, ordinare un insieme di chiavi, trovare un cammino fra due vertici assegnati di un grafo, ecc.)
  - *di ottimizzazione*: chiedono di trovare la migliore soluzione possibile in base ad dato criterio di costo/guadagno (ad esempio, trovare il cammino di lunghezza minima che congiunge due vertici di un grafo orientato pesato)

Nozioni base di algoritmica (2)

- *Dimensione di un'istanza*: numero di bit (in alcuni casi possiamo usare anche le word) sufficiente a rappresentare l'istanza *in modo compatto*
- Esempio: un numero intero n è rappresentato in modo compatto mediante notazione binaria; la dimensione è dunque $\log n$
- Esempio: un grafo $G = (V, E)$ tale che $V = \{1, 2, ..., n\}$ ed $|E| = m$ è rappresentabile in modo compatto mediante $n + 2m$ numeri in notazione binaria
- Se π è un'istanza del problema Π, la sua dimensione verrà indicata con $|\pi|$

Nozioni base di algoritmica (3)

- Misuriamo il costo computazionale di un dato algoritmo
  - contando il numero di operazioni elementari (a livello di *bit* o di *word*)
  - in funzione della *dimensione* delle istanze di input
  - considerando, per ogni data dimensione, il costo massimo (analisi *worst-case*)
- Considerare le operazioni a livello di bit può essere necessario nel caso di problemi che coinvolgono (anche) numeri
- Importanza di codificare i problemi in modo *compatto*



Nozioni base di algoritmica (4)

- La *complessità computazionale* di un problema à la quantità minima di risorse (spazio o tempo) sufficienti per risolverlo, ovvero il costo dell'algoritmo di minimo costo che risolve il problema
- Anche la complessità può quindi essere analizzata nel worst-case, sia a livello di bit che di word
- Molti autori usano anche la locuzione "complessità di un algoritmo", che noi invece eviteremo

Algoritmi di approssimazione: elementi di complessità

Sommario

Stabilire se un numero è primo

- Usiamo il crivello di Eratostene
- L'input è il numero n
- Tentativamente, marca i numeri $2, 3, 4, \ldots, n$ come primi
- Poni $i \leftarrow 2$
- Finché $i \leq \lfloor \sqrt{n} \rfloor$
  - trova il primo numero $j \geq i$ marcato come primo
  - smarca i numeri $2j, 3j, \ldots, \lfloor n/j \rfloor j$
  - Poni $i \leftarrow j + 1$
- Stampa i numeri ancora marcati

- Esempio: $n = 18$ ($\lfloor \sqrt{18} \rfloor = 4$)
- **2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18**

Stabilire se un numero è primo

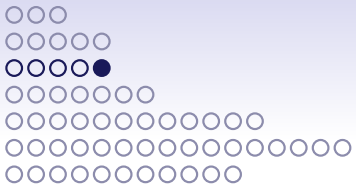
- Usiamo il crivello di Eratostene
- L'input è il numero n
- Tentativamente, marca i numeri $2, 3, 4, \ldots, n$ come primi
- Poni $i \leftarrow 2$
- Finché $i \leq \lfloor \sqrt{n} \rfloor$
  - trova il primo numero $j \geq i$ marcato come primo
  - smarca i numeri $2j, 3j, \ldots, \lfloor n/j \rfloor j$
  - Poni $i \leftarrow j + 1$
- Stampa i numeri ancora marcati

- Esempio: $n = 18$ ($\lfloor \sqrt{18} \rfloor = 4$)
- **2 3** 4 **5** 6 **7** 8 **9** 10 **11** 12 **13** 14 **15** 16 **17** 18

Stabilire se un numero è primo

- Usiamo il crivello di Eratostene
- L'input è il numero n
- Tentativamente, marca i numeri $2, 3, 4, \ldots, n$ come primi
- Poni $i \leftarrow 2$
- Finché $i \leq \lfloor \sqrt{n} \rfloor$
  - trova il primo numero $j \geq i$ marcato come primo
  - smarca i numeri $2j, 3j, \ldots, \lfloor n/j \rfloor j$
  - Poni $i \leftarrow j + 1$
- Stampa i numeri ancora marcati

- Esempio: $n = 18$ ($\lfloor \sqrt{18} \rfloor = 4$)
- **2 3** 4 **5** 6 **7** 8 9 10 **11** 12 **13** 14 15 16 **17** 18



Il crivello di Eratostene

- Il costo dell'algoritmo è $O(n)$
- Tuttavia, la dimensione dell'input non è n, bensì $\log n$
- Ne consegue (correttamente) che il crivello di Eratostene è un algoritmo (di costo) esponenziale
- Considerare n come dimensione dell'input equivale a scrivere l'input in unario, quindi in modo non compatto
- Qual è la complessità del problema della primalità?

Algoritmi di approssimazione: elementi di complessità

Sommario

Proposizioni logiche in forma normale congiuntiva

- Dato un insieme di proposizioni logiche elementari (p.e.) $I = \{x_1, x_2, \ldots, x_n\}$, un letterale L su I è una p.e. oppure la negazione di una p.e. In altri termini

$$L \in \{x_i, \bar{x}_i\}$$

per un qualche $i \in \{1, 2, \ldots, n\}$.

- Una formula booleana in *Forma Normale Congiuntiva* (*FNC*) su I è del tipo

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

dove ogni elemento C_i della congiunzione, detto *clausola*, ha la seguente forma:

$$C_i = L_{i1} \vee L_{i2} \vee \ldots \vee L_{ik_i}, \quad k_i \geq 1$$

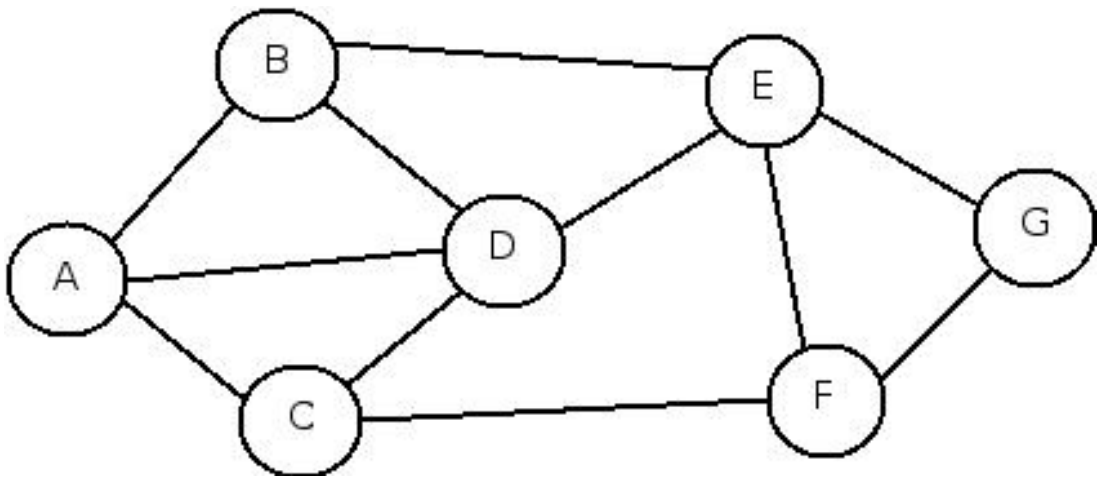
$i = 1, \ldots, m$, e a sua volta L_{ij} è un letterale su I.

Definiamo alcuni problemi modello

- *Satisfiability* (o, più semplicemente, *SAT*): data una formula proposizionale F in FNC, dire se esiste un assegnamento di verità alle proposizioni elementari che rende vera F
- Esempio: $F = (a \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{a} \vee \bar{c})$ è soddisfattibile assegnando ad a il valore **T** (vero) e a b e c il valore **F** (falso).
- *k-SAT*: come *SAT*, ma con il vincolo che ogni clausola contenga esattamente k letterali.

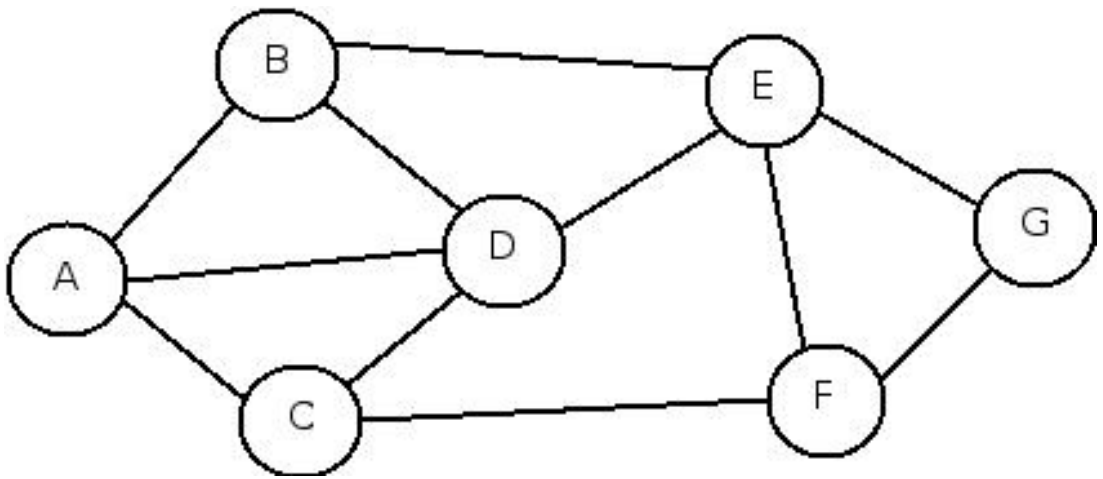
Definiamo alcuni problemi modello (2)

- *INDEPENDENT SET* (*IS*): dato un grafo $G = (V, E)$ e un intero k non maggiore di $|V|$, dire se esiste un sottoinsieme V' di V, di dimensione almeno k, tale che in G non ci sono archi che uniscono vertici di V'.
- Esempio: nel grafo seguente i vertici B, C e G formano un insieme indipendente di dimensione 3



Definiamo alcuni problemi modello (3)

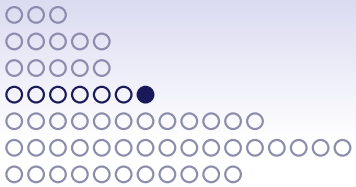
- *Vertex Cover* (*VC*): dato un grafo $G = (V, E)$ e un intero k non maggiore di $|V|$, dire se esiste un sottoinsieme V' di V, di dimensione almeno k, tale che ogni arco di G ha almeno un estremo in V'.
- Esempio: nel grafo seguente i vertici *A*, *D*, *E* ed *F* formano una copertura di dimensione 4



Definiamo alcuni problemi modello (4)

- *Set Cover* (*SC*): data una collezione $\{B_1, B_2, ..., B_m\}$ di sottoinsiemi di uno stesso insieme S ed un intero $k \leq m$, dire se ci sono k sottoinsiemi la cui unione coincide con S.
- Esempio: sia $S = \{1, 2, 3, 4, 5, 6\}$, $B1 = \{1, 4\}, B2 = \{2, 4, 6\}, B3 = \{1, 5\}, B4 = \{3, 5, 6\}$; è facile vedere che 3 sottoinsiemi sono necessari e che

$$S = B1 \cup B_2 \cup B4 = B2 \cup B3 \cup B4$$



Definiamo alcuni problemi modello (5)

- *Travelling Salesman Problem* (*TSP*): dato l'insieme di tutte le $N = n(n-1)$ distanze fra n città e un valore t, dire se esiste un cammino chiuso di lunghezza non maggiore di t che tocca tutte le città una e una sola volta
- *Hamiltonian Graph* (*HG*): dato un grafo con n vertici, dire se è Hamiltoniano, cioè se contiene un ciclo semplice di lunghezza n (cioè che passa una e una sola volta da ogni nodo).

Algoritmi di approssimazione: elementi di complessità

Sommario

Stabilire la difficoltà dei problemi

- Sulla base delle conoscenze attuali, per ciascuno dei problemi introdotti vale quanto segue
  - non si dispone di alcun algoritmo efficiente di risoluzione, mediante il quale si possa stabilire che il problema è computazionalmente *facile*
  - non si dispone di nessuna dimostrazione teorica che consenta di classificare il problema come *difficile*
- Ma che cosa vuol dire, per un problema, essere (computazionalmente) facile/difficile?

Una scala di difficoltà

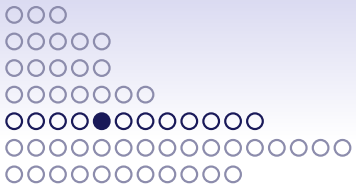
- Cominciamo col fissare una scala di difficoltà
- Definiamo alcune importanti famiglie di funzioni
  - *polinomi*: sono del tipo n^k, con $k > 0$ costante
  - *superpolinomi*: sono del tipo $n^{log^k n}$, con $k > 0$ costante
  - *esponenziali*: sono del tipo b^{n^k}, con $b > 1$ e $k > 0$ costanti
- Due problemi le cui complessità sono funzioni nella stessa famiglia hanno (essenzialmente) la *stessa difficoltà*
- Diversamente, *esponenziale è più difficile di superpolinomiale che è più difficile di polinomiale*

Una scala di difficoltà (2)

- Ricordiamo che l'analisi di complessità prende in considerazione il comportamento *asintotico* di un algoritmo
- Ad esempio, consideriamo due algoritmi (per uno stesso problema) che impiegano tempo $T_1(n) = n^{100}$ e $T_2(n) = \frac{1}{10}2^{n/2}$
- Sebbene per un ampio insieme di valori risulti $T_1(n) > T_2(n)$, esiste tuttavia un valore n_0 tale che, per ogni $n > n_0$, si ha $T_1(n) < T_2(n)$
- n_0 si trova risolvendo l'equazione

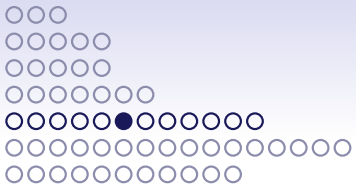
$$n^{100} = \frac{1}{10}2^{n/2}$$

la cui soluzione è un valore compreso fra 2231 e 2232 (e dunque $n_0 = 2231$)



Classi di problemi

- I problemi vengono classificati utilizzando le ampie classi di funzioni introdotte.
- Alcune classi importanti
  - $\mathbf{P} = \{\Pi : \exists k$ e un algoritmo A che risolve ogni istanza $\pi \in \Pi$ in tempo $O(|\pi|^k)\}$
  - $\mathbf{EXPTIME} = \{\Pi :$ $\exists k$ e un algoritmo A che risolve ogni istanza $\pi \in \Pi$ in tempo $O(2^{|\pi|^k})\}$
  - $\mathbf{PSPACE} = \{\Pi :$ $\exists k$ e un algoritmo A che risolve ogni istanza $\pi \in \Pi$ usando $O(|\pi|^k)$ bit di memoria$\}$

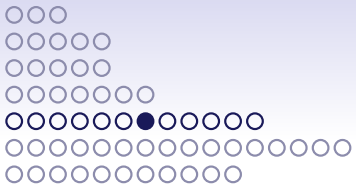


Classi di problemi (2)

- Il nucleo della teoria che svilupperemo considererà solo problemi decisionali
- Le classi **P**, **PSPACE** e **EXPTIME** sono definite in termini di problemi decisionali
- Si può dimostrare che vale

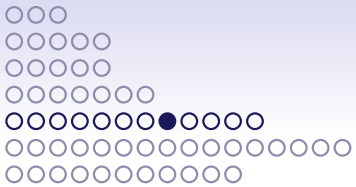
$$\mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$$

ed inoltre $\mathbf{P} \neq \mathbf{EXPTIME}$



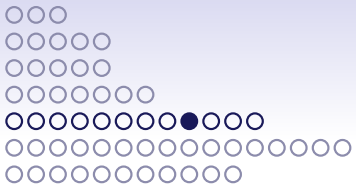
Tesi di Church-Turing

- È lecita una domanda: le definizione che abbiamo dato di classi di complessità (e prima ancora di costo di un algoritmo) *è una buona definizione*?
- Non è possibile che il costo dipenda *in maniera sostanziale* dall'esecutore, cioè dal modello di calcolo utilizzato?
- Non c'è forse un'enorme differenza fra un supercomputer moderno e un PC di 30 anni orsono?
- La tesi di Church-Turing afferma che, se M_1 ed M_2 sono due *ragionevoli* modelli di calcolo universali, allora ogni computazione di M_1 può essere simulata da una computazione di M_2 con uno slowdown polinomiale, e viceversa.



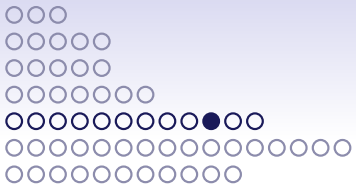
Di nuovo la domanda fondamentale

- Quand'è che un problema Π è facile?
- In algoritmica si utilizza il seguente criterio
  - Se il bit-cost di ogni possibile algoritmo per Π è esponenziale, allora Π è certamente difficile
  - Se invece esiste un algoritmo di bit-cost polinomiale, allora il problema Π è facile
- Non sembra un criterio tanto sensato. Se il polinomio in questione fosse del tipo, poniamo, n^{100}?



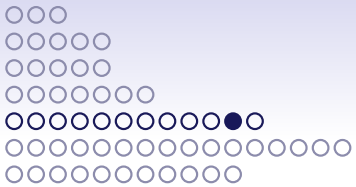
Una (parte della) spiegazione

- I problemi combinatoriali che stiamo esaminando ammettono un numero esponenziale di possibili soluzioni.
- Ad esempio, nel caso di *SAT*, se la formula contiene n proposizioni elementari, i possibili assegnamenti di verità sono 2^n
- Un altro esempio. Nel caso del *TSP* ci sono $(n-1)!$ possibili permutazioni che corrispondono ad altrettanti ordini di attraversamento delle città (avendo fissato partenza e arrivo dalla città 1).
- Il costo esponenziale sembra allora emergere quando non esiste una strategia "sostanzialmente" migliore dell'esame esaustivo di tutte le possibili soluzioni $\Rightarrow$ approccio *brute-force*



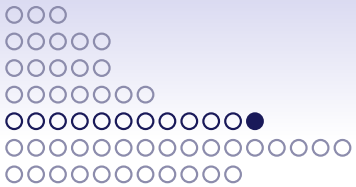
Una (parte della) spiegazione (2)

- Per i problemi in questione, un algoritmo polinomiale non potrà comunque esaminare tutte le possibili soluzioni, perché queste sono in numero esponenziale, e dunque "sono troppe"
- In altri termini, un algoritmo polinomiale non può essere di tipo a forza bruta; esso sfrutta in un qualche modo *intelligente* il fatto che il problema ammette una soluzione più efficiente della semplice enumerazione di tutte le possibilità
- Possiamo concludere che un algoritmo di costo n^{100}, per quanto inefficiente, indica comunque che il problema che esso risolve ammette una strategia diversa dalla forza bruta



Una (parte della) spiegazione (3)

- La precedente osservazione teorica è supportata da considerazioni pratiche, cioè il criterio polinomiale/esponenziale funziona davvero
- Infatti, se un problema ammette un algoritmo polinomiale, cioè può essere risolto in modo “sostanzialmente” diverso dalla forza bruta, allora esso ammette quasi sempre un algoritmo con polinomio di *grado basso*
- Quasi sempre di detto algoritmo esistono implementazioni concrete efficienti



Ricapitolando

- Se prendiamo per buona la spiegazione data, possiamo concludere che
  - i problemi nella classe **P** sono facili
  - i problemi nell'insieme differenza **EXPTIME**$\setminus$**P** sono difficili
- Sfortunatamente, tutti i problemi che abbiamo elencato, come pure molti altri di grande interesse applicativo, non sono a tutt'oggi classificabili ne' come facili ne' come difficili
- Ad esempio, per *SAT* non disponiamo di un algoritmo polinomiale $\Rightarrow$ *è un problema difficile*?
- Non abbiamo però neppure una dimostrazione che ogni possibile algoritmo richieda tempo esponenziale $\Rightarrow$ *è un problema facile*?
- Per gli altri problemi definiti la situazione è la stessa

Algoritmi di approssimazione: elementi di complessità

Sommario

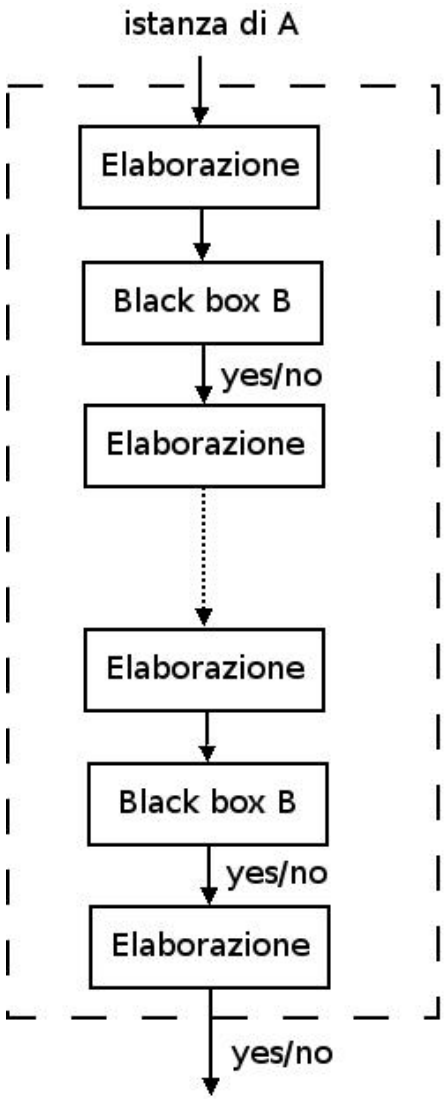
Un obiettivo più “modesto”

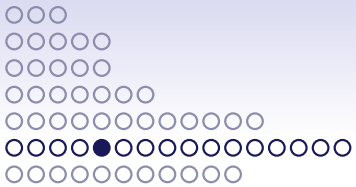
- Esplorare lo spazio dei problemi computazionalmente difficili
- ... dando evidenza della difficoltà di alcuni problemi
- ... caratterizzando almeno una ampia classe di essi da un punto di vista matematico
  - il problema A è almeno tanto difficile quanto B
  - il problema A è il più difficile nell’ambito di una determinata classe C

Uno strumento fondamentale: le riduzioni

- Diremo che un problema A è *riducible* a B *in tempo polinomiale* (o che A è *polinomialmente riducibile* a B), e scriveremo $A \leq_P B$, se esiste un algoritmo per A con le seguenti proprietà
  - A può invocare l'esecuzione di un sottoprogramma (*oracolo*) in grado di risolvere B
  - il costo attribuito a B è costante
  - il costo complessivo di A (incluse le chiamate a B) è polinomiale
- Esempio: calcolo del mediano di un insieme di numeri.
  - Il problema è polinomialmente riducibile all'ordinamento: si chiama il sottoprogramma per l'ordinamento e si prende l'elemento centrale della sequenza ordinata.

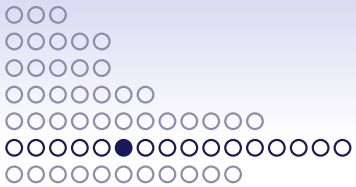
Schema generale di una riduzione





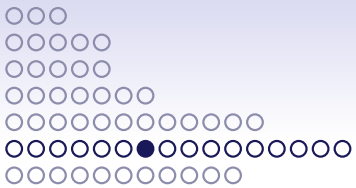
Significato delle riduzioni

- Se $A \leq_P B$ allora B è almeno tanto difficile quanto A
- Infatti se esiste un algoritmo polinomiale per B (da sostituire concretamente all'oracolo), cioè se B è facile, allora il costo di A *incluse le chiamate di* B è anch'esso polinomiale, e dunque anche A è facile
- La simbologia $A \leq_P B$ vuole proprio esprimere questo fatto
- Tutto ciò dipende semplicemente dalle proprietà di chiusura dei polinomi



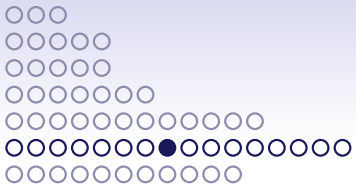
Significato delle riduzioni (2)

- Supponiamo ancora che $A \leq_P B$
- Se possiamo dimostrare che la complessità di A è esponenziale (o superpolinomiale) allora non può esistere un algoritmo polinomiale per B, perché abbiamo appena visto che, se esistesse, allora anche A sarebbe risolvibile in tempo poliniamo $\Rightarrow$ contraddizione
- e $A \leq_P B$ e $B \leq_P A$ allora A e B hanno la stessa difficoltà (scriveremo $A \equiv_P B$)



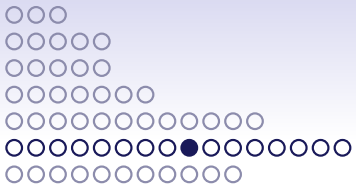
Generalità dei problemi decisionali

- Consideriamo il problema *Independent Set*
- La versione di ottimizzazione di questo problema chiede di trovare il massimo insieme indipendente nel grafo di input
- La verisone decisionale chiede di sapere se, nel grafo di input, esiste uninsieme indipendente di cardinalità assegnata k.
- Chiaramente la versione di ottimizzazione non è più facile di quella decisionale
- Infatti, supponiamo di voler sapere se un certo grafo G ha un insieme indipendente di cardinalità almeno k
- Se risolviamo il problema di ottimizzazione, troviamo l'insieme (o un insieme) indipendente di cardinalità massima, che suppomiamo essere k'
- A questo punto, la risposta al problema decisionale è affermativa se e solo se $k' > k$



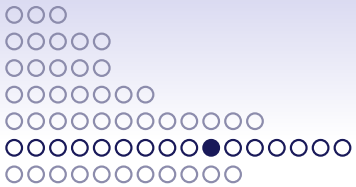
Una (auto)riduzione per *Independent Set*

- Supponiamo di disporre di un sottoprogramma D per risolvere la versione decisionale di *IS*
- Dato G con n vertici, determiniamo dapprima la dimensione massima k di un insieme indipendente di G usando un processo di ricerca binaria
  - Invochiamo $D(G, n/2)$
  - Se la risposta è affermativa proviamo con $D(G, (3/4)n)$, altrimenti con $D(G, n/4)$
  - ...
- Si procede quindi alla individuazione di k vertici che formano un IS, nel modo seguente



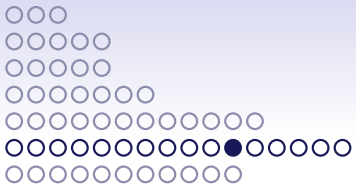
Una (auto)riduzione per *Independent Set* (2)

- Si considera un vertice v e lo si elimina dal grafo
- Si verifica (con una chiamata a D) se la dimensione del massimo IS si riduce
- In caso affermativo, v è necessariamente parte dell'IS di dimensione massima, e dunque si procede cercando, nel sottografo di G ottenuto eliminando v e i vertici ad esso adiacenti, un independent set di dimensione $k - 1$
- Altrimenti si procede eliminando un altro vertice nel grafo originale
- Il numero *totale* di chiamate a D è al più $n + \log n$



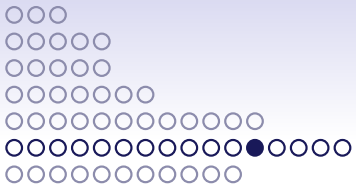
Prime riduzioni per problemi decisionali

- Stabiliremo i seguenti risultati, che illustrano l'uso delle riduzioni
- *IS* e *VC* hanno la stessa difficoltà
- *SC* è almeno tanto difficile quanto *VC*
- *IS* è almeno tanto difficile quanto *3-SAT*



Difficoltà relativa di *IS* e *VC*

- Un semplice risultato: dato un grafo $G = (V, E)$, un sottoinsieme S di V è un independent set se e solo se $V \setminus S$ è un vertex cover
  - se (x, y) è un arco di G, allora al più solo uno fra i vertici x e y può stare in S, e quindi almeno uno fra x e y sta in $V \setminus S$, ne consegue che l'arco (x, y) è coperto e dunque $V \setminus S$ è un vertex cover
  - anche il viceversa è immediato
- Questo implica che un grafo G ha un independent set di dimensione almeno k se e solo se G ha un vertex cover di dimensione non maggiore di $n - k$ ($n = |V|$)
- Vale quindi banalmente $IS \equiv_P VC$



SC è almeno tanto difficile quanto *VC* ($VC \leq_P SC$)

- Dato un problema di vertex covering su un grafo $G = (V, E)$, poniamo

$$S = E \quad \text{e} \quad B_i = \{\text{archi incidenti nel nodo } i\}$$

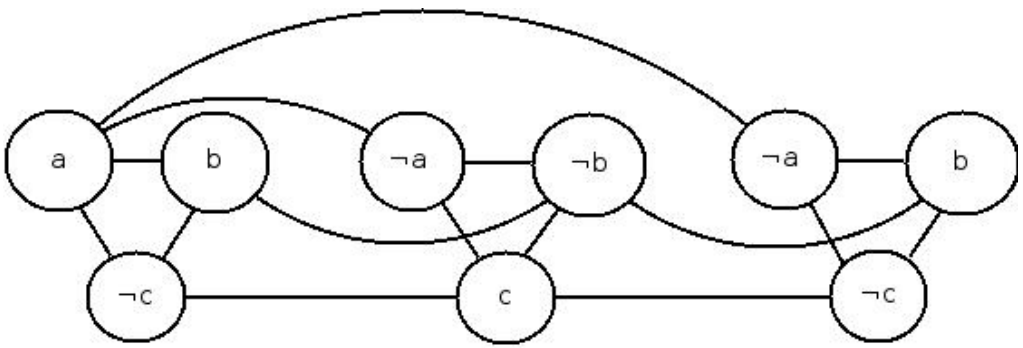
  cioè ogni sottoinsieme è in corrispondenza biunivoca con un vertice
- Se esiste una collezione di k sottoinsiemi che copre S, allora i vertici corrispondenti formano un vertex cover

IS è almeno tanto difficile quanto *3-SAT* ($3\text{-}SAT \leq_P IS$)

- Data una formula booleana in FNC con k clausole, ognuna formata esattamente da 3 letterali, costruiamo un grafo composto da k triangoli opportunamente collegati
- Esempio: alla formula

$$F = (a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c)$$

corrisponde il seguente grafo

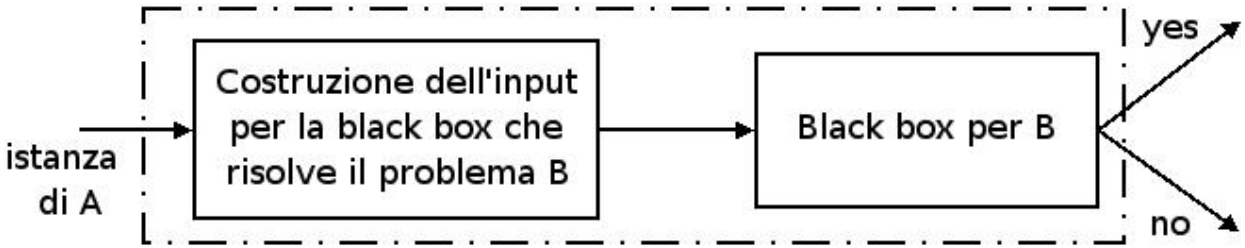


3-$SAT \leq_P IS$ (2)

- Algoritmicamente, la riduzione è un po' più complessa di quelle viste in precedenza
- Ad esempio, per risolvere *IS* usando la black box per *VC* è sufficiente cambiare l'input: da k ad $n - k$, dove n è il numero di vertici del grafo di input e k è la dimensione minima richiesta per il più grande insieme indipendente
- Solo poco più complessa è la riduzione da *VC* a *SC*
- Nel caso in questione, data la formula, il programma deve "costruire" il grafo corrispondente prima di invocare la black-box per *IS*

Struttura comune delle riduzioni

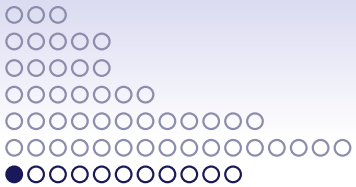
- Tutte le riduzioni viste finora (con l'unica eccezione dell'autoriduzione per *IS*) sono comunque costituite da una sola chiamata alla black-box
- $A \leq_P B$



- Si noti che la definizione di riduzione consentirebbe un numero polinomiale di chiamate alla black box

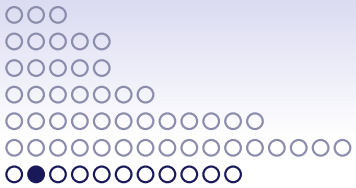
Il punto sulle riduzioni

- Riduzioni viste finora:
  3-$SAT \leq_P IS \leq_P VC \leq_P SC$ e $VC \equiv_P IS$
- Inoltre, poiché la riducibilità è una relazione *transitiva* (discende dalla proprietà di chiusura dei polinomi rispetto alla composizione), vale ad esempio: 3-$SAT \leq_P SC$
- Abbiamo ora uno strumento per (tentare di) stabilire la difficoltà relativa di alcuni problemi
- Non siamo però ancora in grado di caratterizzare *classi di problemi (apparentemente) difficili*



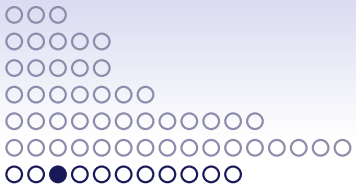
Algoritmi di approssimazione: elementi di complessità

Sommario



Ricerca e verifica di soluzioni

- Sembra esserci una chiara differenza, sperimentabile in molti contesti, fra lo sforzo necessario a *trovare* una soluzione di un problema e *verificare* che una certa soluzione proposta è effettivamente una soluzione
- Qualche esempio:
  - in matematica, *calcolare un integrale* e verificare che una funzione è *primitiva* di un'altra funzione
  - nell'algoritmica, scrivere un programma di ordinamento e un programma che verifica se una sequenza è ordinata
  - negli scacchi, trovare le mosse che portano ad un matto e verificare che una sequenza porta effettivamente allo scacco matto

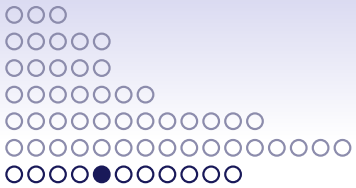


Ricerca e verifica di soluzioni (2)

- Per i problemi computazionali introdotti
  - *SAT*: è facile verificare (in tempo proporzionale alla lunghezza della formula F) se un assegnamento di verità agli n letterali di F produce il valore **true**
  - *TSP*: è facile verificare (in tempo proporzionale al numero di città) se la lunghezza di un ciclo proposto è minore della soglia richiesta
  - *IS*: è facile verificare (in tempo proporzionale al numero di archi) se un insieme di vertici è indipendente
  - ...

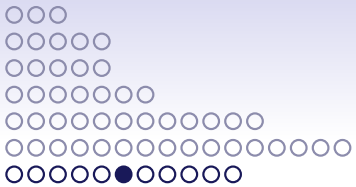
Alcune definizioni

- Dato un problema decisionale A, diremo che gli *input legali* per A in corrispondenza dei quali la risposta è *yes* costituisce l'insieme delle *istanze positive* per A (*yes-instances*)
- Un programma C è detto *certificatore efficiente* per A se soddisfa le seguenti condizioni
  (1) C accetta due parametri in input, I ed S, dove si intende che I rappresenti un'*istanza* di A
  (2) C impiega tempo polinomiale in $|I| + |S|$
  (3) Per ogni istanza positiva I di A, esiste una stringa S (detta *certificato*), la cui lunghezza è al più $|I|^k$ (dove k è una costante che dipende solo da A e da C), tale che $C(I, S) = \textbf{yes}$



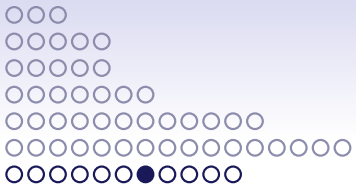
La classe **NP**

- **NP** è la classe dei problemi che posseggono un certificatore efficiente, cioè un algoritmo polinomiale di verifica delle soluzioni
- *SAT* è in **NP**: per ogni istanza positiva il certificato è costituito dagli n valori logici dati alle variabili
- *VC* è in **NP**: per ogni istanza positiva il certificato è costituito da una lista di k vertici
- *TSP* è in **NP**: per ogni istanza positiva il certificato è costituito dall'ordine di visita delle città
- ...



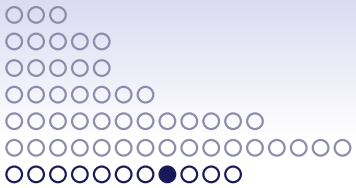
La classe **NP** (2)

- Un risultato immediato: $\mathbf{P} \subseteq \mathbf{NP}$: infatti, per ogni problema è risolvibile in tempo polinomiale il certificato è la stringa vuota
- Stabilire se $\mathbf{P} = \mathbf{NP}$ o se $\mathbf{P} \neq \mathbf{NP}$ è uno dei principali problemi aperti in Informatica e uno dei sette “Millennium Prize Problems” del *Clay Mathematics Institute* (http://www.claymath.org/millennium/)
- **NP** è in realtà l’acronimo di **N**ondeterministic **P**olinomial (Time)



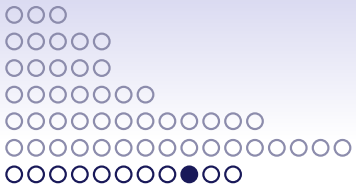
L'interrogativo fondamentale

- Circa quattro decenni di tentativi di risolvere l'enigma non hanno prodotto la soluzione
- Per dimostrare che **P** $\neq$ **NP** sarebbe ovviamente sufficiente dimostrare che un problema in **NP** (poniamo, 3-*SAT*) richiede tempo esponenziale (non polinomiale)
- Naturalmente, se dimostrassimo che 3-*SAT* richiede tempo esponenziale avremmo anche dimostrato che ci sono anche altri problemi difficili, ad esempio *IS*, *VC* e *SC*
- Il viceversa, cioè che se (poniamo) 3-*SAT* fosse risolvibile in tempo polinomiale allora **P** $=$ **NP**, non sembra implicitamente valere da ciò che abbiamo detto finora



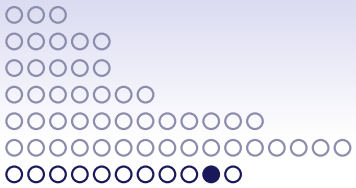
L'interrogativo fondamentale (2)

- Dalle riduzioni che abbiamo visto sappiamo che *SC* è almeno tanto difficile quanto *IS*, *VC* e 3-*SAT*
- Il fatto che *SC* sia il più difficile dei quattro problemi significa che una sua eventuale *facilità* implicherebbe la facilità degli altri
- Ci domandiamo allora: esiste un problema nella classe **NP** che possiamo dimostrare essere *il più difficile della classe*
- Se un tale problema esistesse, potremmo concentrare su quel problema gli eventuali sforzi per dimostrare che $\mathbf{P} = \mathbf{NP}$
- In realtà di tali problemi ne esistono tanti, e vengono detti *problemi* **NP***-completi*



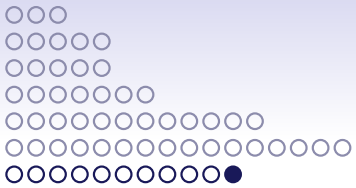
Problemi **NP**-completi

- I problemi “più difficili” per una data classe vengono detti problemi *completi per quella classe*
- Un problema A è completo per la classe **NP** (o, più semplicemente, **NP**-completo) se e solo se
  - $A \in \mathbf{NP}$
  - se $B \in \mathbf{NP}$ allora $B \leq_P A$
- Un problema completo “racchiude” tutta la difficoltà della classe
- Trovare un problema completo (a patto che esista) sembra sensibilmente più difficile che stabilire la difficoltà relativa di due problemi specifici



Il risultato fondamentale

- Esistono problemi **NP**-completi (risultato dovuto indipendentemente a due studiosi, uno americano e uno russo)
- In particolare *SAT* è **NP**-completo (teorema di Cook)
- La dimostrazione (a posteriori ...) non è neppure troppo complessa, ma necessita di molte nozioni preliminari
- $\Rightarrow$ la omettiamo



Altri problemi **NP**-completi

- Per dimostrare che un problema $A \in$ **NP** è **NP**-completo è ora (cioè "dopo" il Teorema di Cook) sufficiente trovare un problema B, già noto essere **NP**-completo, e provare che $B \leq_P A$
- Ad esempio, alla luce delle riduzioni già viste, se provassimo che $SAT \leq_P 3\text{-}SAT$ dimostreremmo in un sol colpo che 3-*SAT*, *IS*, *VC* e *SC* sono problemi **NP**-completi
- È questo il modo mediante il quale la collezione di problemi **NP**-completi cresce
- Oggi sono noti migliaia di problemi **NP**-completi, in aree quali logica, informatica, ricerca operativa, economia, ecc.