

Contents

computational complexity	3
def: problema in computer science	3
tipologie di problema	3
complessità degli algoritmi e dei problemi	3
esempio: codice	4
def: tempo di esecuzione dell'algoritmo A	4
def: complessità temporale dell'algoritmo A	4
def: complessità di un problema	4
problemi di decisione e classi di complessità	5
def: un algoritmo A risolve π	5
def: classe dei problemi $TIME(g(n))$	5
algoritmi non-deterministici per i problemi di decisione	5
def: un algoritmo non-deterministico A risolve π	5
def: classe dei problemi $NTIME(g(n))$	6
esempio: algoritmo non-deterministico per il problema della clique . . .	6
osservazioni (algoritmi deterministici e non-deterministici)	6
corollario: $TIME(g(n)) \subseteq NTIME(g(n))$	6
efficienza e trattabilità	6
efficienza e trattabilità: ragione 1	6
efficienza e trattabilità: ragione 2	7
osservazione: macchina di turing non-deterministica	7
def: codici polinomialmente correlati	7
dimensione dell'input (def: codici correlati polinomialmente)	7
esempio: codici correlati polinomialmente	7
esempio: codifica non naturale	8
def: modelli computazionali simulabili in modo polinomiale	8
classi P e NP	8
problemi NP -completi	8
optimization problems	9
def: problema di ottimizzazione	9
osservazioni (problemi di ottimizzazione)	9
esempio: descrizione formale di un problema di ottimizzazione (max clique)	9
def: soluzione ottima	9
problema decisionale sottostante	10
esempio: descrizione formale di un problema decisionale sottostante (max clique)	10
osservazioni (problema decisionale sottostante)	10
classi di complessità dei problemi di ottimizzazione: PO	10
classi di complessità dei problemi di ottimizzazione: PO	10
PO e NPO : nella pratica	11
def: relazione $NPO \subseteq NP-HARD$	11
teorema: relazione tra $P \neq NP$ e risolvibilità polinomiale dei problemi $NP-HARD$	11
teorema: relazione tra $P = NP$ e $PO = NPO$	11
approximation	12
introduzione	12
def: algoritmo di r -approssimazione per problemi di minimizzazione	12
def: algoritmo di r -approssimazione per problemi di massimizzazione . . .	12
determinazione del fattore di approssimazione r	12

[illegible]

computational complexity

def: problema in computer science

un problema π é una relazione

$$\pi \subseteq I_\pi \times S_\pi$$

dove:

- I_π = insieme delle istanze di input del problema
- S_π = insieme delle soluzioni del problema

tipologie di problema

• decisione:

- si verifica se una data proprietà é valida per un determinato input
- $S_\pi = \{true, false\}$ o semplicemente $S_\pi = \{0,1\}$ e la relazione $\pi \subseteq I_\pi \times S_\pi$ corrisponde ad una funzione

$$f : I_\pi \rightarrow \{0,1\}$$

- esempi: soddisfacibilità, test di connettività di un grafo, etc....

• ricerca:

- data un'istanza $x \in I_\pi$, si chiede di determinare una soluzione $y \in S_\pi$ tale che la coppia $(x,y) \in \pi$ appartengono alla relazione che definisce il problema
- esempi: soddisfacibilità, clique, vertex cover, nei quali chiediamo in output un assegnamento di verità soddisfacente, rispettivamente una clique o un vertex cover, invece di semplicemente "si" o "no"

• ottimizzazione

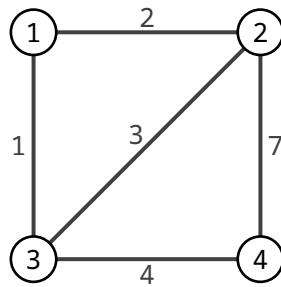
- data un'istanza $x \in I_\pi$, si chiede di determinare una soluzione $y \in S_\pi$ ottimizzando una data misura della funzione costo
- esempi: min spanning tree, max SAT, max clique, min vertex cover, min TSP, etc....

complessità degli algoritmi e dei problemi

- espressa in funzione della taglia dell'input (denotata come $|x|, \forall x \in I_\pi$)
- taglia dell'istanza x
 - quantità di memoria necessaria a memorizzare x in un computer
 - lunghezza $|x|_c$ della stringa che codifica x in un particolare codice naturale $c : I_\pi \rightarrow \Sigma$, dove Σ é l'alfabeto del codice c
- codice naturale
 - conciso: le stringhe che codificano le istanze non devono essere ridondanti o allungate inutilmente
 - numeri espressi in base ≥ 2

esempio: codice

- istanza: grafo G



- codice per G
 - $\Sigma = \{\{, \}, , 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (simboli)
 - $c(G) = \{1, 2, 3, 4, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, 2, 1, 3, 7, 4\}$
 - * $\{1, 2, 3, 4\}$ (nodi)
 - * $\{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$ (archi)
 - * $\{2, 1, 3, 7, 4\}$ (pesi)
 - $|G|_c = 49$

def: tempo di esecuzione dell'algoritmo A

sia $t_A(x)$ il tempo di esecuzione dell'algoritmo A per l'input x , allora il tempo di esecuzione nel caso peggiore di A é:

$$T_A(n) = \max\{t_A(x) \mid |x| \leq n\}, \quad \forall n > 0$$

def: complessità temporale dell'algoritmo A

l'algoritmo A ha complessità temporale

- $O(g(n))$ se $T_A(n) = O(g(n))$, ovvero

$$\lim_{n \rightarrow \infty} \frac{T_A(n)}{g(n)} \leq c, \text{ per una costante } c > 0$$

- $\Omega(g(n))$ se $T_A(n) = \Omega(g(n))$, ovvero

$$\lim_{n \rightarrow \infty} \frac{T_A(n)}{g(n)} \geq c, \text{ per una costante } c > 0$$

- $\Theta(g(n))$ se $T_A(n) = \Theta(g(n))$, ovvero

$$T_A(n) = \Omega(g(n)) \text{ e } T_A(n) = O(g(n))$$

def: complessità di un problema

un problema ha complessità

- $O(g(n))$ se esiste un algoritmo che lo risolve avente complessità $O(g(n))$
- $\Omega(g(n))$ se ogni algoritmo A che lo risolve ha complessità $\Omega(g(n))$
- $\Theta(g(n))$ se ha complessità $O(g(n))$ e $\Omega(g(n))$

problemi di decisione e classi di complessità

i problemi di decisione sono solitamente descritti da un'istanza di input (o semplicemente INPUT) e da una DOMANDA sull'input

esempi:

- soddisfacibilità
 - INPUT: CNF (Conjunctive Normal Form) formula definita su un insieme di variabili
 - DOMANDA: esiste un assegnamento di verità $\tau: V \rightarrow \{0,1\}$?
- clique
 - INPUT: un grafo non orientato $G=(V,E)$ di n nodi e un intero $k > 0$
 - DOMANDA: esiste in G una clique di almeno k nodi ($\geq k$), ovvero un sottoinsieme $U \subseteq V$ tale che $|U| \geq k$ e $\{u,v\} \in E, \forall u,v \in U$?
- vertex cover
 - INPUT: un grafo non orientato $G=(V,E)$ di n nodi e un intero $k > 0$
 - DOMANDA: esiste in G un vertex cover di al massimo k nodi ($\leq k$), ovvero un sottoinsieme $U \subseteq V$ tale che $|U| \leq k$ e $u \in U$ o $v \in U, \forall \{u,v\} \in E$?

nei problemi di decisione $I_\pi = Y_\pi \cup N_\pi$

- Y_π = insieme di istanze positive, ovvero con soluzione 1
- N_π = insieme di istanze negative, ovvero con soluzione 0

def: un algoritmo A risolve π

un algoritmo A risolve $\pi \iff \forall \text{ input } x \in I_\pi, A \text{ risponde } 1 \iff x \in Y_\pi$

def: classe dei problemi $TIME(g(n))$

$TIME(g(n))$ = classe dei problemi di decisione con complessità $O(g(n))$

algoritmi non-deterministici per i problemi di decisione

essi si compongono di 2 fasi

- fase 1
 - generano in modo non-deterministico un "certificato" y
- fase 2
 - partendo dall'input x e dal certificato y , verificano se x é un'istanza positiva

def: un algoritmo non-deterministico A risolve π

un algoritmo non-deterministico A risolve π se si ferma per ogni possibile certificato y ed esiste un certificato y per cui A risponde 1 (*true*) $\iff x \in Y_\pi$

- complessità
 - costo della fase 2
 - espressa in funzione di $|x|$

def: classe dei problemi $NTIME(g(n))$

$NTIME(g(n))$ = classe di problemi di decisione con complessità non-deterministica $O(g(n))$

esempio: algoritmo non-deterministico per il problema della clique

- fase 1
 - dato in input il grafo $G = (V, E)$, genera non-deterministicamente un sottoinsieme $U \subseteq V$ di k nodi
- fase 2
 - verifica se U è una clique, ovvero se $\{u, v\} \in E, \forall u, v \in U$, e in tal caso risponde 1, altrimenti risponde 0
- chiaramente l'algoritmo risolve il problema della clique, in quanto si ferma per ogni possibile sottoinsieme U ed esiste un sottoinsieme U per il quale risponde 1 se e solo se esiste una clique di k nodi in G , ovvero $\iff (G, K) \in Y_{clique}$
- complessità: $O(n^2)$, poiché $|U| \leq |V| = n$

osservazioni (algoritmi deterministici e non-deterministici)

- un algoritmo deterministico è meno potente di uno non-deterministico poiché non può eseguire la fase 1
- se esiste un algoritmo deterministico A che risolve π , allora esiste anche un algoritmo non-deterministico A' che risolve π con la stessa complessità come segue:
 - esso esegue al fase 1 e coincide con A nella fase 2, ignorando il certificato y

corollario: $TIME(g(n)) \subseteq NTIME(g(n))$

$$TIME(g(n)) \subseteq NTIME(g(n))$$

- dove:
 - $TIME(g(n))$ = classe dei problemi deterministicamente risolvibili in tempo $O(g(n))$
 - $NTIME(g(n))$ = classe dei problemi non-deterministicamente risolvibili in tempo $O(g(n))$

efficienza e trattabilità

- un problema è trattabile se può essere risolto efficientemente (deterministicamente)
- sono considerati trattabili o efficientemente risolvibili tutti i problemi aventi complessità limitata da un polinomio della dimensione dell'input

$$TRATTABILITÀ \equiv EFFICIENZA \equiv POLINOMIALITÀ$$

efficienza e trattabilità: ragione 1

la crescita delle funzioni polinomiali rispetto a quelle esponenziali (sia per ciò che riguarda il tempo di esecuzione sia per ciò che riguarda la dimensione delle istanze risolvibili entro un certo tempo di esecuzione)

efficienza e trattabilità: ragione 2

- la composizione di polinomi é un polinomio e dunque la risolvibilità in tempo polinomiale di un problema é indipendente da
 - il codice naturale utilizzato, poiché tutti i codici naturali sono correlati in maniera polinomiale
 - il modello computazionale adottato, se ragionevole (cioé costruibile nella pratica o meglio in grado di eseguire un lavoro limitato costante per step), in quanto tali modelli sono polinomialmente correlati, ovvero possono simularsi l'un l'altro in tempo polinomiale

osservazione: macchina di turing non-deterministica

la macchina di turing non-deterministica non é un modello di calcolo ragionevole, poiché la quantità di lavoro svolto in ogni fase (ciascun livello dell'albero delle computazioni) cresce in modo esponenziale

def: codici polinomialmente correlati

- 2 codici c_1 e c_2 per un problema π sono correlati polinomialmente se esistono 2 polinomi p_1 e p_2 tali che, $\forall x \in I_\pi$:
 - $|x|_{c_1} \leq p_1(|x|_{c_2})$
 - $|x|_{c_2} \leq p_2(|x|_{c_1})$
- se la complessità rispetto a c_1 é $O(q_1(|x|_{c_1}))$ per un dato polinomio q_1 , allora rispetto a c_2 é $O(q_1(p_1(|x|_{c_2}))) = O(q_2(|x|_{c_2}))$ dove q_2 é il polinomio tale che $\forall \lambda \ q_2(\lambda) = q_1(p_1(\lambda))$
- tutti i codici naturali sono correlati polinomialmente, ovvero la risolvibilità polinomiale non dipende dal particolare codice utilizzato

dimensione dell'input (def: codici correlati polinomialmente)

qualsiasi quantità polinomialmente correlata ad un codice naturale é dunque correlata ad un qualsiasi codice naturale possibile, dato che tutti i codici naturali sono correlati polinomialmente e che la composizione di polinomi é un polinomio

esempio: codici correlati polinomialmente

- assumiamo che per ogni grafo G di n nodi
 - $|G|_{c_1} = 10n^2$
 - $|G|_{c_2} = n^3$
- se $p_1(\lambda) = 10\lambda$ e $p_2(\lambda) = \lambda^2$ abbiamo che:
 - $|G|_{c_1} = 10n^2 \leq 10n^3 = p_1(|G|_{c_2})$
 - $|G|_{c_2} = n^3 \leq 100n^4 = p_2(|G|_{c_1})$
- dunque i 2 codici sono correlati polinomialmente
- regola pratica:
 - 2 quantità sono polinomialmente correlate se sono polinomi sulle stesse variabili

esempio: codifica non naturale

- test di primalità
 - INPUT: un numero intero $n > 0$
 - DOMANDA: n é un numero primo?
 - ALGORITMO (banale):
 - * scansiona tutti i numeri da 2 a $n-1$ e risponde 1 (*true*) se nessuno di essi lo divide
 - COMPLESSITÀ: $O(n)$, polinomiale?
 - CODICE c_1 (naturale): n espresso in base 2, ovvero $|n|_{c_1} = \log_2 n$
 - CODICE c_2 (non naturale): n espresso in base 1, ovvero $|n|_{c_2} = n$
- dunque la complessità dell'algoritmo é:
 - $O(2^{|n|_{c_1}})$ rispetto a c_1 , che é esponenziale
 - $O(|n|_{c_2})$ rispetto a c_2 , che é polinomiale!
- dimensione dell'input
 - correlata polinomialmente ai codici naturali $|n|_{c_1} = \log_2 n$

def: modelli computazionali simulabili in modo polinomiale

- 2 modelli computazionali M_1 e M_2 sono mutualmente simulabili in modo polinomiale se esistono 2 polinomi p_1 a p_2 tali che:
 1. ogni algoritmo A per M_1 con complessità $T_A(n)$ può essere simulato su M_2 in tempo $p_1(T_A(n))$
 2. ogni algoritmo A per M_2 con complessità $T_A(n)$ può essere simulato su M_1 in tempo $p_2(T_A(n))$
- dunque se A é polinomiale in M_1 allora é polinomiale anche in M_2 e viceversa
- tutti i modelli computazionali ragionevoli sono mutualmente simulabili in modo polinomiale, ovvero la risolvibilità polinomiale non dipende dal particolare modello utilizzato

classi P e NP

- P = classe di tutti i problemi risolvibili deterministicamente in tempo polinomiale, ovvero

$$P = \cup_{k=0}^{\infty} TIME(n^k)$$

- NP = classe di tutti i problemi risolvibili non-deterministicamente in tempo polinomiale, ovvero

$$NP = \cup_{k=0}^{\infty} NTIME(n^k)$$

- $P = NP$? nessuno lo a dimostrato

problemi NP -completi

- i problemi più difficili di NP e tali che se $P \neq NP$ non appartengono a P , viceversa, se 1 di essi appartiene a P , allora $P = NP$
- finora nessuno é riuscito a trovare un algoritmo polinomiale deterministico per nessun problema NP -completo
- congettura: $P \neq NP$

optimization problems

def: problema di ottimizzazione

un problema di ottimizzazione π é una quadrupla $(I_\pi, S_\pi, m_\pi, goal_\pi)$ con:

- I_π = insieme delle istanze di input di π
- $S_\pi(x)$ = insieme delle soluzioni ammissibili dell'istanza $x \in I_\pi$
- $m_\pi(x, y)$ = misura della soluzione ammissibile $y \in S_\pi(x)$ per l'input $x \in I_\pi$ (intera)
- $goal_\pi \in \{\min, \max\}$ = specifica se abbiamo un problema di minimizzazione o di massimizzazione

osservazioni (problemi di ottimizzazione)

- assumiamo che $m_\pi(x, y)$ é sempre un numero intero
 - i nostri modelli computazionali possono trattare solo l'approssimazione razionale dei reali
 - scalando tali reali possiamo ottenere numeri interi equivalenti
 - i valori interi rivelano già le difficoltà intrinseche dei problemi
- quando sono chiari dal contesto (in seguito):
 - π sarà omesso
 - $m(x, y)$ = sarà denotato semplicemente come m

esempio: descrizione formale di un problema di ottimizzazione (max clique)

- I = grafo $G = (V, E)$
- $S = \{U \subseteq V \mid \{u, v\} \in E, \forall u, v \in U\}$
- $m(G, U) = |U|$
- $goal = \max$

possiamo descrivere i problemi di ottimizzazione nella seguente forma, più semplice e informale

- MAX CLIQUE
 - INPUT: grafo $G = (V, E)$
 - SOLUZIONE: $U \subseteq V \mid \{u, v\} \in E, \forall u, v \in U$
 - MISURA: $|U|$

def: soluzione ottima

- data un'istanza $x \in I_\pi$, una soluzione $y^* \in S_\pi$ é ottima per x se $m(x, y^*) = goal\{m(x, y) \mid y \in S(x)\}$
- la misura di una soluzione ottima (o in modo analogo di tutte le soluzioni ottime) di x é denotata come $m^*(x)$ o semplicemente m^*

problema decisionale sottostante

ogni problema di ottimizzazione ha un problema decisionale sottostante che può essere ottenuto introducendo un intero k nell'istanza di input e chiedendo se esiste una soluzione ammissibile di misura $\leq k$ (per min) e $\geq k$ (per max)

- problema di ottimizzazione:
 - dato un input x , trova $y \in S(x) \mid m(x,y)$ sia min o max (secondo il *goal*)
- problema decisionale sottostante:
 - dato un input x e un intero $k \geq 0$, esiste $y \in S(x) \mid m(x,y) \leq k$ (min) o $\geq k$ (max)

esempio: descrizione formale di un problema decisionale sottostante (max clique)

- MAX CLIQUE
 - INPUT: grafo $G = (V, E)$
 - SOLUZIONE: $U \subseteq V \mid \{u, v\} \in E, \forall u, v \in U$
 - MISURA: $|U|$
- problema decisionale sottostante:
 - INPUT: grafo $G = (V, E)$ e un intero $k > 0$
 - DOMANDA: esiste una clique U in G tale che $|U| \geq k$

osservazioni (problema decisionale sottostante)

- se esiste un algoritmo polinomiale A per il problema di ottimizzazione, allora esiste un algoritmo polinomiale anche per il problema decisionale sottostante che funziona come segue:
 1. esegue A per determinare la soluzione ottimale y^* per l'input x
 2. risponde 1 (*true*) se $m(x, y^*) \leq k$ (min) o $\geq k$ (max)
- il problema di ottimizzazione è difficile almeno quanto il problema decisionale sottostante

classi di complessità dei problemi di ottimizzazione: PO

- un problema di ottimizzazione π appartiene alla classe PO se:
 - per ogni input x , $x \in I$ può essere verificato in tempo polinomiale
 - esiste un polinomio $p \mid \forall x \in I$ e $y \in S(x)$ vale $|y| \leq p(|x|)$
 - $\forall x \in I$ e $y \in S(x)$, $m(x, y)$ può essere calcolata in tempo polinomiale (rispetto a $|x|$)
 - $\forall x \in I$, una soluzione ottima y^* può essere calcolata in tempo polinomiale
- esempi: shortest path fra 2 nodi, min spanning tree, ecc...

classi di complessità dei problemi di ottimizzazione: PO

un problema di ottimizzazione π appartiene alla classe NPO se:

- per ogni input x , $x \in I$ può essere verificato in tempo polinomiale
- esiste un polinomio $p \mid \forall x \in I$ e $y \in S(x)$ vale $|y| \leq p(|x|)$
- $\forall x \in I$ e $y \in S(x)$, $m(x, y)$ può essere calcolata in tempo polinomiale (rispetto a $|x|$)

esempi: max clique, min vertex cover, min TSP, ecc...

***PO* e *NPO*: nella pratica**

- *PO*: classe dei problemi di ottimizzazione il cui problema decisionale sottostante appartiene a *P*
- *NPO*: classe dei problemi di ottimizzazione il cui problema decisionale sottostante appartiene a *NP*
- chiaramente $PO \subseteq NPO$

def: relazione *NPO* - *NP-HARD*

un problema di ottimizzazione in *NPO* é *NP-HARD* se il problema decisionale sottostante é *NP-Completo*

teorema: relazione tra $P \neq NP$ e risolvibilità polinomiale dei problemi *NP-HARD*

se $P \neq NP$, un problema di ottimizzazione *NP-HARD* non può essere risolto in tempo polinomiale (poiché é difficile almeno quanto il problema decisionale sottostante)

teorema: relazione tra $P = NP$ e $PO = NPO$

se $P = NP$ allora $PO = NPO$

- quasi tutti i problemi che verranno presentati in seguito sono *NP-HARD*, ovvero non efficientemente risolvibili
- verranno progettati algoritmi per tali problemi che restituiscono soluzioni "vicine" a quelle ottime

approximation

introduzione

- DOMANDA: supponiamo di dover risolvere un problema NP-HARD, cosa dovremmo fare?
- RISPOSTA: sacrificare 1 delle 3 caratteristiche desiderate
 1. risolvere istanze arbitrarie del problema
 2. risolvere il problema di ottimalità
 3. risolvere il problema in tempo polinomiale
- STRATEGIE:
 1. progettare algoritmi per casi speciali del problema
 2. progettare algoritmi di approssimazione o euristiche
 3. progettare algoritmi che possono richiedere tempo esponenziale
- d'ora in poi ci concentreremo sui problemi di ottimizzazione NP-HARD, ovvero problemi che non possono essere risolti in modo efficiente (a meno che $P = NP$)
- per tali problemi verranno progettati algoritmi in grado di determinare soluzioni prossime a quelle ottime, ovvero "buone approssimazioni"

def: algoritmo di r-approssimazione per problemi di minimizzazione

dato un problema di minimizzazione π e un numero $r \geq 1$, un algoritmo A é un algoritmo di r-approssimazione per π se per ogni input $x \in I$ restituisce sempre una soluzione r-approssimata, ovvero una soluzione ammissibile $y \in S(x)$ tale che

$$\frac{m(x, y)}{m^*(x)} \leq r$$

def: algoritmo di r-approssimazione per problemi di massimizzazione

dato un problema di massimizzazione π e un numero $r \leq 1$, un algoritmo A é un algoritmo di r-approssimazione per π se per ogni input $x \in I$ restituisce sempre una soluzione r-approssimata, ovvero una soluzione ammissibile $y \in S(x)$ tale che

$$\frac{m(x, y)}{m^*(x)} \geq r$$

determinazione del fattore di approssimazione r

- come possiamo determinare il fattore di approssimazione r se non conosciamo il valore m^* di una soluzione ottima?
- per problemi di minimizzazione (rispettivamente massimizzazione), confrontiamo il valore della soluzione restituita $m(x, y)$ con un lower bound (rispettivamente upper bound) appropriato $l(x)$ (rispettivamente $u(x)$) di $m^*(x)$
- se il loro rapporto é al massimo r (\leq) per min o almeno r (\geq) per max, allora l'algoritmo é r-approssimante

min (analogo per max) fattore di approssimazione r

- se

$$\frac{m(x, y)}{l(x)} \leq r$$

- allora

$$\frac{m(x, y)}{m^*(x)} \leq \frac{m(x, y)}{l(x)} \leq r$$

algoritmo: Approx-Cover per min vertex cover

Algorithm 1 Approx-Cover

```
 $M = \emptyset$   
 $U = \emptyset$   
repeat  
  seleziona un arco  $\{u, v\} \in E$   
   $U = U \cup \{u, v\}$   
   $E = E \setminus \{e \in E \mid e \text{ é incidente a } u \text{ o a } v\}$   
   $M = M \cup \{\{u, v\}\}$   
until ( $E = \emptyset$ )  
return  $U$ 
```

lemma: Approx-Cover forma un matching al termine dell'esecuzione

al termine dell'esecuzione dell'algoritmo di approssimazione Approx-Cover, M forma un matching, ovvero gli archi in M non condividono alcun nodo

dimostrazione:

- banalmente, ogni volta che un arco e é selezionato in M , tutti gli archi con un nodo in comune con e vengono eliminati da E
- pertanto nei passi successivi nessun arco con un nodo in comune con e può essere selezionato dall'algoritmo

□

teorema: Approx-Cover é 2-approssimante

Approx-Cover é 2-approssimante

dimostrazione:

- il valore della soluzione restituita dall'algoritmo é

$$m = |U| = 2|M|$$

- sia U^* il cover ottimo.
Poiché gli archi in M non condividono alcun nodo (M é un matching) e poiché ciascuno di essi deve avere un nodo in U^*

$$m^* = |U^*| \geq |M|$$

- dunque:

$$\frac{m}{m^*} \leq \frac{2|M|}{|M|} = 2$$

□

approximation

caratteristiche

- la soluzione viene determinata in step
- ad ogni step l'algoritmo esegue la scelta che sembra essere la migliore in quello step, senza considerare le possibili conseguenze nei futuri step

problema: max 0-1 knapsack

- INPUT:
 - un insieme finito di oggetti O
 - un profitto intero $p_i \forall o_i \in O$
 - un volume intero $a_i \forall o_i \in O$
 - un intero positivo b
- SOLUZIONE:
 - un sottoinsieme di oggetti $Q \subseteq O$ tale che $\sum_{o_i \in Q} a_i \leq b$
- MISURA:
 - profitto totale degli oggetti scelti, ovvero $\sum_{o_i \in Q} p_i$
- senza perdere di generalità, in seguito, assumeremo sempre che:
 - $a_i \leq b \forall o_i \in O$
 - $p_i > 0 \forall o_i \in O$

max 0-1 knapsack: descrizione della scelta greedy

- nella scelta greedy:
 - non possiamo considerare solo il profitto degli oggetti, in quanto il loro volume potrebbe essere troppo grande
 - non possiamo considerare solo il volume degli oggetti, in quanto il loro profitto potrebbe essere troppo basso
- idea: consideriamo gli oggetti in base al profitto per unità di volume, ovvero in base al rapporto
$$\frac{p_i}{a_i}$$
- l'algoritmo greedy seleziona gli oggetti in ordine decrescente di profitto per volume

algoritmo: Greedy-Knapsack

Algorithm 2 Greedy-Knapsack

```
 $Q = \emptyset$   
 $v = 0$   
//  $v$  = volume del sottoinsieme corrente degli oggetti scelti  
ordina gli oggetti in ordine decrescente di profitto per volume  $\frac{p_i}{a_i}$   
siano  $o_1, \dots, o_n$  gli oggetti elencati secondo tale ordine  
for  $i = 1$  to  $n$  do  
  if  $v + a_i \leq b$  then  
     $Q = Q \cup \{o_i\}$   
     $v = v + a_i$   
  end if  
end for  
return  $Q$ 
```

teorema: $\forall r < 1$ Greedy-Knapsack non é r -approssimante

$\forall r < 1$ dato, Greedy-Knapsack non é r -approssimante

dimostrazione:

- dato un intero $k = \lceil \frac{1}{r} \rceil$, consideriamo la seguente istanza di max 0-1 knapsack
- $\forall n \geq 2$
 - $b = kn$ é il volume del knapsack
 - $n - 1$ oggetti con profitto $p_i = 1$ e volume $a_i = 1$
 - 1 oggetto con profitto $b - 1$ e volume b
- soluzione restituita:
 - l'insieme dei primi $n - 1$ oggetti, ovvero $m = n - 1$
- soluzione ottima
 - l'insieme contenente solo l' n -esimo oggetto, ovvero

$$m^* = b - 1 = kn - 1$$

- quindi:

$$\frac{m}{m^*} = \frac{n - 1}{kn - 1}$$

- cosí che:

(<) poiché $\frac{1}{r} > 1$

$$\frac{m}{m^*} = \frac{n - 1}{kn - 1} \leq \frac{n - 1}{\frac{n}{r} - 1} < \frac{n - 1}{\frac{n}{r} - \frac{1}{r}} = \frac{n - 1}{\frac{1}{r}(n - 1)} = r$$

□

miglioramento algoritmo: Greedy-Knapsack

- osservazione:
 - intuitivamente, Greedy-Knapsack non restituisce una buona approssimazione, poiché ignora l'oggetto avente il profitto massimo

Greedy-Knapsack modificato

- calcola una soluzione greedy Q_{GR} e sia m_{GR} la misura di quest'ultima
- considera l'oggetto O_{\max} avente il massimo profitto p_{\max}
- se $m_{GR} \geq p_{\max}$ restituisci Q_{GR} altrimenti restituisci $Q = \{O_{\max}\}$

lemma 1: Greedy-Knapsack modificato

- sia o_j il primo oggetto che l'algoritmo Greedy-Knapsack non inserisce nel knapsack e sia:

$$m_j = \sum_{i=1}^{j-1} p_i$$

- allora:

$$m^* \leq m_j + p_j$$

dimostrazione:

- $m^* \leq m_j + p_j$ deriva direttamente osservando semplicemente che, denotando con v la somma dei volumi dei primi $j-1$ oggetti scelti, $m_j + p_j$ è il valore della soluzione ottima dell'istanza in cui il volume del knapsack è $v + a_j > b$

lemma 2: Greedy-Knapsack modificato

- $m^* \leq m_{GR} + p_{\max}$

dimostrazione:

- diretta conseguenza del precedente lemma osservando che $m_j \leq m_{GR}$ e $p_j \leq p_{\max}$, e quindi:

$$m^* \leq m_j + p_j \leq m_{GR} + p_{\max}$$

- intuizione: l'algoritmo restituisce una soluzione di valore $\max\{m_{GR}, p_{\max}\}$, che è almeno la metà di $m_{GR} + p_{\max}$, ovvero la metà di un upper bound di m^*

$$\max\{m_{GR}, p_{\max}\} \geq \frac{m_{GR} + p_{\max}}{2}$$

teorema: Greedy-Knapsack modificato è $\frac{1}{2}$ -approssimante

Greedy-Knapsack modificato è $\frac{1}{2}$ -approssimante

dimostrazione:

- $m_{Mod} \geq \max\{m_{GR}, p_{\max}\} \geq \frac{(m_{GR} + p_{\max})}{2} \geq \frac{m^*}{2}$

problema: min multiprocessor scheduling

- INPUT:

- insieme di n jobs P
- numero di processori h
- tempo di esecuzione $t_j \ \forall p_j \in P$

- SOLUZIONE:

- uno schedule per P , ovvero una funzione

$$f : P \rightarrow \{1, \dots, h\}$$

- MISURA:

- *makespan* o tempo di completamento di f , ovvero

$$\max_{i \in [1, \dots, h]} \sum_{p_j \in P \mid f(p_j)=i} t_j$$

algoritmo: Greedy-Graham

- scelta greedy: ad ogni step assegna un job al processore meno carico
- $T_i(j)$:
 - tempo di completamento (somma dei tempi di esecuzione dei jobs assegnati) del processore i al termine del tempo j , ovvero una volta schedulati i primi j jobs (in qualunque ordine)

Algorithm 3 Greedy-Graham

```
siano  $p_1, \dots, p_n$  i jobs elencati in un qualsiasi ordine
for  $j = 1$  to  $n$  do
  assegna  $p_j$  al processore  $i$  avente il minimo  $T_i(j-1)$  ovvero  $f(p_j)$ 
end for
return lo schedule  $i$ 
```

- osservazione:
 - se i jobs vengono schedulati in accordo con il tempo di arrivo, l'algoritmo assegna ciascun job senza conoscere quelli futuri, ovvero ONLINE

teorema: Greedy-Graham é $\frac{2-1}{h}$ -approssimante

l'algoritmo Greedy-Graham é $\frac{2-1}{h}$ -approssimante, dove h é il numero di processori

fatto:

- dato $s \geq 0$ e h numeri $a_1, \dots, a_h \mid a_1 + \dots + a_h = s$, allora esiste j , $1 \leq j \leq h$, tale che

$$a_j \geq \frac{s}{h} \qquad a_1 + \dots + a_h < h \frac{s}{h}$$

- altrimenti, contraddizione
- analogamente, esiste j' , $1 \leq j' \leq h$, tale che $a_{j'} \leq \frac{s}{h}$
- in altre parole, un numero é al massimo uguale alla media e uno maggiore o uguale alla media
- pertanto, $\min_j a_j \leq \frac{s}{h}$ e $\max_j a_j \geq \frac{s}{h}$

dimostrazione:

- sia T la somma di tutti i tempi di esecuzione dei job, ovvero

$$T = \sum_{j=1}^n t_j$$

- siano $T_1^*, T_2^*, \dots, T_h^*$ i tempi di completamento degli h processori nella soluzione ottima

- poiché $T_1^* + T_2^* + \dots + T_h^* = T$ dal precedente 'fatto', esiste j tale che $T_j^* \geq \frac{T}{h}$
- quindi:

$$m^* \geq T_j^* \geq \frac{T}{h}$$

- sia k il processore con il massimo tempo di completamento nello schedule f restituito dall'algoritmo, ovvero con $T_k(n)$ massimo
- in più sia p_l l'ultimo job assegnato al processore k
- dato che, per la scelta greedy, p_l è stato assegnato ad uno dei processori meno carichi all'inizio dello step l , sempre per il 'fatto' precedente, abbiamo:

$$T_k(l-1) \leq \frac{\sum_{j < l} t_j}{h} \leq \frac{T - t_l}{h}$$

- dato che la somma dei tempi di esecuzione di tutti i jobs assegnati prima di p_l è al massimo (\leq) $T - t_l$
- pertanto:

$$\begin{aligned} m = T_k(n) &= T_k(l-1) + t_l \leq \frac{T - t_l}{h} + t_l = \\ &= \frac{T - t_l + ht_l}{h} = \frac{T}{h} - \frac{1+h}{h}t_l = \frac{T}{h} + \frac{h-1}{h}t_l \leq \dots \end{aligned}$$

- poiché $\frac{T}{h} \leq m^*$ e $t_l \leq m^*$

$$\begin{aligned} \dots &\leq m^* + \frac{h-1}{h}m^* = \frac{hm^* + (h-1)m^*}{h} = \frac{hm^* + hm^* - m^*}{h} = \\ &= \frac{2hm^* - m^*}{h} = \frac{2h-1}{h}m^* = (2 - \frac{1}{h})m^* \end{aligned}$$

- e quindi:

$$\frac{m}{m^*} \leq 2 - \frac{1}{h}$$

□

- osservazioni:

- quando h cresce, il rapporto di approssimazione $\frac{2-1}{h}$ tende a 2
- l'analisi è stretta, ovvero vale il seguente teorema

teorema: Greedy-Graham non è r -approssimante per $r < \frac{2-1}{h}$

Greedy-Graham non è r -approssimante per $r < \frac{2-1}{h}$

dimostrazione:

- considera la seguente istanza:
 - $h(h-1)$ jobs con tempo di esecuzione 1
 - 1 job con tempo di esecuzione h
- Greedy-Graham assegna i jobs nella seguente maniera:
- e quindi:

$$m = 2(h-1)$$

- la soluzione ottima può essere ottenuta assegnando il job più lungo ad un processore e distribuendo ugualmente i jobs più corti tra i processori restanti:
- e quindi:

$$m^* = h$$

- in conclusione:

$$\frac{m}{m^*} = \frac{2(h-1)}{h} = 2 - \frac{1}{h} \quad (\text{diverso da } \leq 2 - \frac{1}{h})$$

migliorare il rapporto di approssimazione r per Greedy-Graham

- DOMANDA: come possiamo migliorare il rapporto di approssimazione r
- richiamiamo rapidamente gli step base della dimostrazione del rapporto di approssimazione di Greedy-Graham
- abbiamo utilizzato i seguenti *lower bounds* per il valore della soluzione ottima:
 - $m^* \geq \frac{T}{h}$, come in qualsiasi soluzione almeno 1 processore deve avere tempo di completamento $\frac{T}{h}$ (richiamiamo che $T = \sum_j t_j$)
 - $m^* \geq t_j$, per ogni job p_j , come in qualsiasi soluzione uno dei processori deve eseguire p_j
- abbiamo utilizzato il seguente *upper bounds* per il valore della soluzione restituita:
 - per limitare superiormente il valore della soluzione restituita, se k è uno dei processori più carichi e p_l è l'ultimo job assegnato a k , per la scelta greedy:

$$T_k(l-1) \leq \frac{\sum_{j < l} t_j}{h} \leq \frac{T - t_l}{h}$$

- quindi possiamo derivare la seguente disuguaglianza:

$$\begin{aligned} m = T_k(n) &= T_k(l-1) + t_l \leq \frac{T - t_l}{h} + t_l = \\ &= \frac{T - t_l + ht_l}{h} = \frac{T}{h} - \frac{1-h}{h}t_l = \frac{T}{h} + \frac{h-1}{h}t_l \leq \dots \end{aligned}$$

- poiché $\frac{T}{h} \leq m^*$ e $t_l \leq m^*$

$$\begin{aligned} \dots &\leq m^* + \frac{h-1}{h}m^* = \frac{hm^* + (h-1)m^*}{h} = \frac{hm^* + hm^* - m^*}{h} = \\ &= \frac{2hm^* - m^*}{h} = \frac{2h-1}{h}m^* = (2 - \frac{1}{h})m^* \end{aligned}$$

- idea per il miglioramento: decrementa t_l il più possibile e trova un rapporto di approssimazione migliore sfruttando le disuguaglianze

$$m \leq \frac{T}{h} + \frac{h-1}{h}t_l \leq m^* + \frac{h-1}{h}t_l$$

- modificando l'algoritmo e/o migliorando l'analisi vedremo come limitare superiormente t_l progressivamente con:
 - $\frac{m^*}{2}$ ($\frac{3}{2}$ -approssimante),
 - $\frac{m^*}{3}$ ($\frac{4}{3}$ -approssimante),
 - e arbitrariamente piccolo, ovvero ϵm^* $((1+\epsilon)$ -approssimante), cioè un PTAS

Greedy-Graham, primo miglioramento

- assegnare i jobs dal piú lungo al piú corto
- ciò ci consente di evitare il caso peggiore dell'algoritmo di Graham, ovvero il fatto che un job lungo arrivi alla fine, sbilanciando significativamente il carico dei processori

algoritmo: Ordered-Greedy

Algorithm 4 Ordered-Greedy

siano p_1, p_2, \dots, p_n i job elencati in ordine decrescente di tempo di esecuzione, ovvero tale che $t_1 \geq t_2 \geq \dots \geq t_n$
for $j = 1$ to n **do**
 assegna p_j al processore i con il minimo $T_i(j-1)$, ovvero $f(p_j) = i$
end for
return schedule f

- vediamo un'analisi piú semplice che porta ad un rapporto di approssimazione di circa $\frac{3}{2}$

lemma: Ordered-Greedy

se $n > h$, allora $t_{h+1} \leq \frac{m^*}{2}$

dimostrazione:

- dall'ordinamento dei job, i primi $h+1$ hanno tutti un tempo di esecuzione $\geq t_{h+1}$
- ma allora $m^* \geq 2t_{h+1}$, poiché in ogni schedule almeno 1 degli h processori deve ricevere almeno 2 dei primi $h+1$ job

□

teorema: Ordered-Greedy é $(\frac{3}{2} - \frac{1}{2h})$ -approssimante

Ordered-Greedy é $(\frac{3}{2} - \frac{1}{2h})$ -approssimante

dimostrazione:

- di nuovo sia k uno dei processori piú carichi (alla fine)
- se k ha 1 solo job, allora chiaramente la soluzione ritornata é ottima
- altrimenti considera l'ultimo job p_l assegnato a k
- dato che p_l non é il primo job assegnato a k , $l \geq h+1$ e quindi $t_l \leq t_{h+1} \leq \frac{m^*}{2}$, e cosí:

$$\begin{aligned} m &\leq \frac{T}{h} + \frac{h-1}{h} t_l \leq m^* + \frac{h-1}{h} \frac{m^*}{2} = \\ &= m^* + \frac{m^*(h-1)}{2h} = \frac{2hm^* + m^*(h-1)}{2h} = \frac{2hm^* + hm^* - m^*}{2h} = \\ &= \frac{3hm^* - m^*}{2h} = \left(\frac{3h-1}{2h}\right)m^* = \left(\frac{3h}{2h} - \frac{1}{2h}\right)m^* = \left(\frac{3}{2} - \frac{1}{2h}\right)m^* \end{aligned}$$

□

problema: max cut

- INPUT: grafo $G = (V, E)$
- SOLUZIONE: una partizione di V in 2 sottoinsiemi V_1 e V_2 , ovvero tale che:

$$V_1 \cup V_2 = V \text{ e } V_1 \cap V_2 = \emptyset$$

- MISURA: la cardinalità del taglio, ovvero il numero di archi con un estremo (nodo) in V_1 e un estremo in V_2 , cioè:

$$|\{\{u, v\} \mid u \in V_1 \text{ e } v \in V_2\}|$$

algoritmo: Greedy-Max-Cut

Algorithm 5 Greedy-Max-Cut

```
 $V_1 = V_2 = \emptyset$ 
for  $i = 1$  to  $n$  do
  //  $\Delta_i$  = set di archi tra  $i$  e i nodi  $j < i$  (adiacenti)
   $\Delta_i = \{\{i, j\} \in E \mid j < i\}$ 
end for
return  $V_1, V_2$ 
```

local search

rounding

primal dual

dynamic programming

approximation schemes

alternative approaches

social networks and bibliography

centrality measures

spectral analysis and prestige index

link analysis

web structure

search and advertising

matching markets

auctions

vsg mechanism

gsp mechanism