

# Contents

<b>computational complexity</b>	<b>5</b>
def: problema in computer science . . . . .	5
tipologie di problema . . . . .	5
complessità degli algoritmi e dei problemi . . . . .	5
esempio: codice . . . . .	6
def: tempo di esecuzione dell'algoritmo $A$ . . . . .	6
def: complessità temporale dell'algoritmo $A$ . . . . .	6
def: complessità di un problema . . . . .	6
problemi di decisione e classi di complessità . . . . .	7
def: un algoritmo $A$ risolve $\pi$ . . . . .	7
def: classe dei problemi $TIME(g(n))$ . . . . .	7
algoritmi non-deterministici per i problemi di decisione . . . . .	7
def: un algoritmo non-deterministico $A$ risolve $\pi$ . . . . .	7
def: classe dei problemi $NTIME(g(n))$ . . . . .	8
esempio: algoritmo non-deterministico per il problema della clique . . .	8
osservazioni (algoritmi deterministici e non-deterministici) . . . . .	8
corollario: $TIME(g(n)) \subseteq NTIME(g(n))$ . . . . .	8
efficienza e trattabilità . . . . .	8
efficienza e trattabilità: ragione 1 . . . . .	8
efficienza e trattabilità: ragione 2 . . . . .	9
osservazione: macchina di turing non-deterministica . . . . .	9
def: codici polinomialmente correlati . . . . .	9
dimensione dell'input (def: codici correlati polinomialmente) . . . . .	9
esempio: codici correlati polinomialmente . . . . .	9
esempio: codifica non naturale . . . . .	10
def: modelli computazionali simulabili in modo polinomiale . . . . .	10
classi $P$ e $NP$ . . . . .	10
problemi $NP$ -completi . . . . .	10
<b>optimization problems</b>	<b>11</b>
def: problema di ottimizzazione . . . . .	11
osservazioni (problemi di ottimizzazione) . . . . .	11
esempio: descrizione formale di un problema di ottimizzazione (max clique)	11
def: soluzione ottima . . . . .	12
problema decisionale sottostante . . . . .	12
esempio: descrizione formale di un problema decisionale sottostante (max clique) . . . . .	12
osservazioni (problema decisionale sottostante) . . . . .	12
classi di complessità dei problemi di ottimizzazione: $PO$ . . . . .	13
classi di complessità dei problemi di ottimizzazione: $NPO$ . . . . .	13
$PO$ e $NPO$ : nella pratica . . . . .	13
def: relazione $NPO \neq NP-HARD$ . . . . .	13
teorema: relazione tra $P \neq NP$ e risolvibilità polinomiale dei problemi $NP-HARD$ . . . . .	13
teorema: relazione tra $P = NP$ e $PO = NPO$ . . . . .	13
<b>approximation</b>	<b>14</b>
introduzione . . . . .	14
def: algoritmo di r-approssimazione per problemi di minimizzazione . . .	14
def: algoritmo di r-approssimazione per problemi di massimizzazione . .	14
determinazione del fattore di approssimazione $r$ . . . . .	14

min (analogo per max) fattore di approssimazione $r$ . . . . .	15
algoritmo: Approx-Cover per min vertex cover . . . . .	15
lemma: Approx-Cover forma un matching al termine dell'esecuzione . . . . .	15
teorema: Approx-Cover é 2-approssimante . . . . .	15
<b>algorithmic techniques: greedy</b> . . . . .	<b>16</b>
caratteristiche . . . . .	16
problema: max 0-1 knapsack . . . . .	16
max 0-1 knapsack: descrizione della scelta greedy . . . . .	16
algoritmo: Greedy-Knapsack . . . . .	17
teorema: $\forall r < 1$ Greedy-Knapsack non é $r$ -approssimante . . . . .	17
miglioramento algoritmo: Greedy-Knapsack . . . . .	17
Greedy-Knapsack modificato . . . . .	18
lemma 1: Greedy-Knapsack modificato . . . . .	18
lemma 2: Greedy-Knapsack modificato . . . . .	18
teorema: Greedy-Knapsack modificato é $\frac{1}{2}$ -approssimante . . . . .	18
problema: min multiprocessor scheduling . . . . .	19
algoritmo: Greedy-Graham . . . . .	19
teorema: Greedy-Graham é $\frac{2-1}{h}$ -approssimante . . . . .	19
teorema: Greedy-Graham non é $r$ -approssimante per $r < \frac{2-1}{h}$ . . . . .	21
migliorare il rapporto di approssimazione $r$ per Greedy-Graham . . . . .	21
Greedy-Graham, primo miglioramento . . . . .	22
algoritmo: Ordered-Greedy . . . . .	22
lemma: Ordered-Greedy . . . . .	22
teorema: Ordered-Greedy é $(\frac{3}{2} - \frac{1}{2h})$ -approssimante . . . . .	23
problema: max cut . . . . .	23
algoritmo: Greedy-Max-Cut . . . . .	23
teorema: Greedy-Max-Cut é $\frac{1}{2}$ -approssimante . . . . .	24
conclusioni sulla tecnica greedy . . . . .	25
<b>algorithmic techniques: local search</b> . . . . .	<b>26</b>
caratteristiche . . . . .	26
schema di un algoritmo di ricerca locale . . . . .	26
complessità . . . . .	26
approssimazione . . . . .	26
definizione dell'intorno . . . . .	26
definizione dell'intorno: casi estremi . . . . .	27
problema: max cut (già definito precedentemente) . . . . .	27
algoritmo di ricerca locale per max cut . . . . .	27
complessità (algoritmo di ricerca locale per max cut) . . . . .	27
approssimazione (algoritmo di ricerca locale per max cut) . . . . .	28
fatto (approssimazione (algoritmo di ricerca locale per max cut)) . . . . .	28
teorema: l'algoritmo di ricerca locale é $\frac{1}{2}$ -approssimante . . . . .	28
TODO: esempio esecuzione algoritmo di ricerca locale su grafo . . . . .	29
conclusioni sulla tecnica della ricerca locale . . . . .	29
<b>algorithmic techniques: linear programming (rounding)</b> . . . . .	<b>30</b>
caratteristiche . . . . .	30
rounding: caratteristiche . . . . .	30
problema: min weighted vertex cover . . . . .	30
ILP: min weighted vertex cover . . . . .	31
LP: min weighted vertex cover (rilassamento lineare) . . . . .	31
algoritmo: Round-Vertex-Cover . . . . .	31

teorema: l'algoritmo Round-Vertex-Cover é 2-approssimante . . . . .	31
problema: min weighted set cover . . . . .	32
ILP: min weighted set cover . . . . .	32
LP: min weighted set cover (rilassamento lineare) . . . . .	32
algoritmo: Round-Set-Cover . . . . .	33
teorema: l'algoritmo Round-Set-Cover é $f$ -approssimante ( $f \geq 1$ ) . . . . .	33
<b>algorithmic techniques: dynamic programming (part 1)</b>	<b>34</b>
caratteristiche . . . . .	34
uno sguardo piú ravvicinato... . . . .	34
algoritmo: Fibonacci . . . . .	34
algoritmo: Fibonacci 2 . . . . .	35
algoritmo: Fibonacci 3 . . . . .	35
riassumendo . . . . .	35
top-down vs. bottom-up . . . . .	36
divide-and-conquer vs. dynamic programming . . . . .	36
<b>algorithmic techniques: dynamic programming (part 2)</b>	<b>36</b>
progettazione di algoritmi di programmazione dinamica . . . . .	36
complessità degli algoritmi di programmazione dinamica . . . . .	37
problema: max 0-1 knapsack (già definito precedentemente) . . . . .	37
algoritmo brute force . . . . .	37
progettazione dell'algoritmo di programmazione dinamica . . . . .	37
definizione ricorsiva per $OPT$ . . . . .	38
definizione ricorsiva per la misura $m$ della soluzione ottima $OPT(i, w)$ . . . . .	38
riepilogo definizioni ricorsive per $m$ e $OPT$ . . . . .	38
algoritmo: Progr-Dyn-Knapsack . . . . .	39
algoritmo: Progr-Dyn-Knapsack (trovare gli oggetti inseriti) . . . . .	39
teorema: l'algoritmo Progr-Dyn-Knapsack ha complessità temporale $O(nb)$ . . . . .	39
domanda: l'algoritmo Progr-Dyn-Knapsack é polinomiale? . . . . .	40
algoritmo Progr-Dyn-Knapsack-Dual: approccio duale . . . . .	40
definizione ricorsiva per $OPT$ (duale) . . . . .	40
definizione ricorsiva per la misura $m$ della soluzione ottima $OPT(i, w)$ . . . . .	40
algoritmo: Progr-Dyn-Knapsack-Dual . . . . .	41
teorema: l'algoritmo Progr-Dyn-Knapsack-Dual ha complessità temporale $O(n^2 p_{\max})$ . . . . .	41
<b>approximation schemes: polynomial time approximation scheme (PTAS)</b>	<b>42</b>
definizione: PTAS . . . . .	42
problema: min multiprocessor scheduling (già definito precedentemente) . . . . .	42
richiamiamo l'algoritmo Greedy-Graham . . . . .	42
ottenere un PTAS... . . . .	43
lemma: $t_i \leq \frac{T}{i}$ . . . . .	43
PTAS: idea sottostante . . . . .	44
algoritmo: PTAS-Scheduling . . . . .	44
teorema: l'algoritmo PTAS-Scheduling ritorna sempre una soluzione $(1 + \epsilon)$ -approssimata . . . . .	44
problema: min $h$ -processor scheduling . . . . .	45
teorema: l'algoritmo PTAS-Scheduling é un PTAS per il problema min $h$ -processor scheduling . . . . .	45
problema: min partition . . . . .	46
ottenere un PTAS per il problema min partition . . . . .	46

lemma: algoritmo di programmazione dinamica polinomiale (schedule approssimato per i primi $q$ jobs) . . . . .	46
teorema: esiste un PTAS per il problema min multiprocessor scheduling . . . . .	46
<b>approximation schemes: fully polynomial time approximation scheme (FPTAS)</b>	<b>47</b>
definizione: FPTAS . . . . .	47
FPTAS-Knapsack . . . . .	47
algoritmo: FPTAS-Knapsack . . . . .	48
lemma: $m \geq m^* - n \cdot k$ (FPTAS-Knapsack) . . . . .	48
teorema: FPTAS-Knapsack é un FPTAS per il problema max 0-1 knapsack . . . . .	48
é possibile ridurre la complessit� temporale? . . . . .	49
come migliorare i <i>bound</i> per $m^*$ . . . . .	49
algoritmo: New-FPTAS-Knapsack . . . . .	50
<b>alternative approaches</b>	<b>51</b>
performance garantite . . . . .	51
restrizione dell'insieme delle istanze . . . . .	51
media o analisi probabilistica . . . . .	51
euristiche . . . . .	51
algoritmi randomizzati . . . . .	52
algoritmi randomizzati . . . . .	52
definizione: algoritmi randomizzati e $r$ -approssimazione . . . . .	52
problema: max weighted cut . . . . .	52
algoritmo: Random-Cut . . . . .	53
teorema: Random-Cut é $\frac{1}{2}$ -approssimante . . . . .	53
problema: min weighted set cover (gi� definito precedentemente) . . . . .	53
algoritmo greedy per il problema min weighted set cover . . . . .	54
scelta greedy . . . . .	54
algoritmo: Greedy-Min-Weighted-Set-Cover . . . . .	54
lemma: $m = \sum_{S_j \in \hat{C}} c_j = \sum_{i=1}^n price(o_i)$ . . . . .	55
lemma: $price'(o_i) \geq eff(S'_j)$ . . . . .	55
lemma: $price(o_i) \leq \frac{m^*}{n-i+1}$ . . . . .	55
teorema: Greedy-Min-Weighted-Set-Cover é $H_n$ -approssimante . . . . .	56
esempio: $H_n \cdot m^*$ ! . . . . .	56

# computational complexity

## def: problema in computer science

un problema  $\pi$  é una relazione

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

dove:

- $I_\pi$  = insieme delle istanze di input del problema
- $S_\pi$  = 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

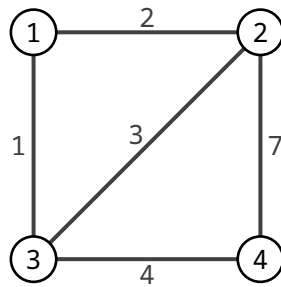
- data un'istanza  $x \in I_\pi$ , si chiede di determinare una soluzione  $y \in S_\pi$  ottimizzando una data misura della funzione costo
- esempi: min spanning tree, max SAT, max clique, min vertex cover, min TSP, etc....

## 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  $\Sigma$  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  $\geq 2$

## 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  $I_\pi = Y_\pi \cup N_\pi$

- $Y_\pi$  = insieme di istanze positive, ovvero con soluzione 1
- $N_\pi$  = insieme di istanze negative, ovvero con soluzione 0

**def: un algoritmo  $A$  risolve  $\pi$**

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  $\pi$**

un algoritmo non-deterministico  $A$  risolve  $\pi$  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  $\pi$ , allora esiste anche un algoritmo non-deterministico  $A'$  che risolve  $\pi$  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  $\pi$  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  $\pi$  é una quadrupla  $(I_\pi, S_\pi, m_\pi, goal_\pi)$  con:

- $I_\pi$  = insieme delle istanze di input di  $\pi$
- $S_\pi(x)$  = 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):
  - $\pi$  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:  $|U|$
- MIN VERTEX COVER
  - INPUT: grafo  $G = (V, E)$
  - SOLUZIONE:  $U \subseteq V \mid \forall \{u, v\} \in E, u \in U \vee v \in U$
  - MISURA:  $|U|$
- MIN TSP (Traveling Salesman Problem, problema del commesso viaggiatore)
  - INPUT:
    - \* insieme di città  $C = \{c_1, c_2, \dots, c_n\}$
    - \* distanza  $d(c_i, c_j) \in \mathbb{N}$ , per ogni coppia di città  $(c_i, c_j) \in C$

- SOLUZIONE: un tour di tutte le città, ovvero una permutazione  $\langle c_{p(1)}, c_{p(2)}, \dots, c_{p(n)} \rangle$  che descriva l'ordine di visita delle città
- MISURA: lunghezza del tour, ovvero

$$\left( \sum_{i=1}^{n-1} d(c_{p(i)}, c_{p(i+1)}) \right) + d(c_{p(n)}, c_{p(1)})$$

### def: soluzione ottima

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

un problema di ottimizzazione  $\pi$  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  $NPO$  è  $NP$ -HARD se il problema decisionale sottostante è  $NP$ -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 è 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  $\pi$  e un numero  $r \geq 1$ , un algoritmo  $A$  é un algoritmo di r-approssimazione per  $\pi$  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  $\pi$  e un numero  $r \leq 1$ , un algoritmo  $A$  é un algoritmo di r-approssimazione per  $\pi$  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 = archi scelti dall'algoritmo
M = ∅
// U = nodi scelti nel cover
U = ∅
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:

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

□

# algorithmic techniques: greedy

## 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  $b$  ( $b > 0$ )
- 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
$$\frac{p_i}{a_i}, \forall o_i \in O$$
- l'algoritmo greedy seleziona gli oggetti in ordine decrescente di profitto per volume



## algoritmo: Greedy-Knapsack

---

**Algorithm 2** Greedy-Knapsack

---

```
// Q = insieme degli oggetti scelti
Q = ∅
// v = volume del sottoinsieme corrente degli oggetti scelti
v = 0
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  $\frac{1}{r} > 1$

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

- $\forall r < 1 \rightarrow \frac{m}{m^*} < r$ , invece di  $\forall r \leq 1 \rightarrow \frac{m}{m^*} \geq r$

□

## 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,  $m_j + p_j$  é il valore della soluzione ottima dell'istanza in cui il volume del knapsack é  $v + a_j > b$

□

## 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 é 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 é $\frac{1}{2}$ -approssimante

Greedy-Knapsack modificato é  $\frac{1}{2}$ -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  $f(p_j) = i$ 
end for
return 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 $2 - \frac{1}{h}$ -approssimante

l'algoritmo Greedy-Graham é  $2 - \frac{1}{h}$ -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

$$a_j \geq \frac{s}{h}$$

- altrimenti, contraddizione ( $a_1 + \dots + a_h < h \frac{s}{h} = s$ )
- 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 piú sia  $p_l$  l'ultimo job assegnato al processore  $k$
- dato che, per la scelta greedy,  $p_l$  é 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  $p_l$  é 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  $2 - \frac{1}{h}$  tende a 2
- l'analisi é stretta, ovvero vale il seguente teorema

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

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

**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:

$$m = h + h - 1 = 2h - 1$$

- 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:

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

□

**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 bound* per il valore della soluzione restituita:
  - per limitare superiormente il valore della soluzione restituita, se  $k$  é uno dei processori più carichi e  $p_l$  é 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  $\epsilon m^*$  ( $(1+\epsilon)$ -approssimante), cioè un PTAS

## Greedy-Graham, primo miglioramento

- assegnare i jobs dal più lungo al più 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 più 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 jobs, 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:

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

- quindi:

$$\frac{m}{m^*} \leq \frac{3}{2} - \frac{1}{2h}$$

□

**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 cardinalitá del taglio, ovvero il numero di archi con un estremo (nodo) in  $V_1$  e un estremo in  $V_2$ , cioé:

$$|\{\{u, v\} \mid (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)\}|$$

**algoritmo: Greedy-Max-Cut**

- per semplicitá sia  $V = \{1, \dots, n\}$
- l'algoritmo ad ogni step inserisce un nuovo nodo in  $V_1$  o in  $V_2$
- scelta greedy:
  - allo step  $i$ , il nodo  $i$  viene inserito in modo da massimizzare il numero di archi nuovi nel taglio, ovvero in  $V_1$  se il numero di archi che ha verso i nodi giá inseriti in  $V_2$  é maggiore ( $\geq$ ) del numero di archi che ha verso quelli in  $V_1$ , altrimenti in  $V_2$  ( $<$ )

---

**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)
   $\Delta_i = \{\{i, j\} \in E \mid j < i\}$ 
  //  $U_i$  = set di nodi già inseriti (adiacenti ad  $i$ , all'inizio dello step  $i$ )
   $U_i = \{j \mid \{i, j\} \in \Delta_i\}$ 
   $\delta_i = |\Delta_i| = |U_i|$ 
   $\delta_{1i} = |V_1 \cap U_i|$ 
   $\delta_{2i} = |V_2 \cap U_i|$ 
  // chiaramente  $\delta_{1i} + \delta_{2i} = \delta_i$ 
  if  $\delta_{1i} > \delta_{2i}$  then
     $V_2 = V_2 \cup \{i\}$ 
  else
     $V_1 = V_1 \cup \{i\}$ 
  end if
end for
return  $V_1, V_2$ 
```

---

**teorema:** Greedy-Max-Cut é  $\frac{1}{2}$ -approssimante

Greedy-Max-Cut é  $\frac{1}{2}$ -approssimante

**dimostrazione:**

- chiaramente poiché quel taglio può solo contenere un sottoinsieme di tutti gli archi in  $E$

$$m^* \leq |E|$$

- mostriamo ora che la misura  $m$  del taglio restituita dall'algoritmo é almeno la metà del numero totale di archi, ovvero:

$$m \geq \frac{|E|}{2}$$

- ciò implica chiaramente l'affermazione, poiché

$$\frac{m}{m^*} \geq \frac{\frac{|E|}{2}}{|E|} = \frac{1}{2}$$

- poiché gli insiemi  $\Delta_i$  determinati dall'algoritmo formano una partizione di  $E$  e per definizione  $\delta_i = |\Delta_i|$ :

$$\sum_{i=1}^n \delta_i = \sum_{i=1}^n |\Delta_i| = |E|$$

- inoltre, il numero di archi aggiunti al taglio durante lo step  $i$ , ovvero con un estremo in  $V_1$  e l'altro in  $V_2$  (dopo l'esecuzione dell'  $i$ -esima iterazione dell'istruzione *for*), é:

$$\max(\delta_{1i}, \delta_{2i}) \geq \frac{(\delta_{1i} + \delta_{2i})}{2} = \frac{\delta_i}{2}$$

- quindi:

$$m = \sum_{i=1}^n \max(\delta_{1i}, \delta_{2i}) \geq \sum_{i=1}^n \frac{\delta_i}{2} = \frac{|E|}{2}$$

□



## **conclusioni sulla tecnica greedy**

- tutti gli algoritmi visti fin ora hanno complessità temporale polinomiale
- gli algoritmi greedy hanno buone performance in pratica poiché possono essere implementati in modo semplice
- ma come abbiamo visto, compiere la scelta che sembra migliore a ciascun singolo step, senza badare alle conseguenze future, in generale non permette di trovare la soluzione ottima

# algorithmic techniques: local search

## caratteristiche

- definiamo, per ogni soluzione ammissibile  $y$ , un sottoinsieme di soluzioni ammissibili "vicine" chiamato intorno di  $y$  o semplicemente  $neighborhood(y)$
- partendo da una soluzione iniziale, si passa ripetutamente ad una soluzione migliore nell'intorno corrente, finché possibile

## schema di un algoritmo di ricerca locale

- risolve una soluzione iniziale  $y$  ammissibile per l'input  $x$  (di solito una banale)
- fintanto che esiste una  $y' \in neighborhood(y)$  migliore di  $y$ 
  - sia  $y = y'$
- ritorna  $y$
- per definire un algoritmo di ricerca locale per un determinato problema é quindi sufficiente definire:
  - la soluzione iniziale
  - l'intorno delle soluzioni ammissibili

## complessità

- per ottenere una complessità temporale polinomiale:
  - la soluzione iniziale deve essere determinata in tempo polinomiale
  - il test della condizione di guardia del *while* e l'eventuale conseguente determinazione di una soluzione migliore nell'intorno deve essere eseguito in tempo polinomiale
  - NOTA: l'intorno può avere una cardinalità esponenziale rispetto alla dimensione dell'input!
  - il numero di iterazioni del *while* deve essere polinomiale

## approssimazione

- OTTIMO LOCALE: la soluzione  $y$  restituita é la migliore nell'intorno considerato
- per limitare il rapporto di approssimazione é sufficiente limitare il rapporto tra il valore di un qualsiasi ottimo locale con quello della misura di una soluzione ottima globale

## definizione dell'intorno

- $neighborhood(y)$ :
  - sufficientemente "ricco", per ottenere buone soluzioni (ottimi locali)
  - sufficientemente "povero", per garantire una complessità temporale polinomiale

## definizione dell'intorno: casi estremi

- $neighborhood(y) = \emptyset$ 
  - tempo di esecuzione polinomiale (se la soluzione iniziale viene determinata in tempo polinomiale)
  - cattiva approssimazione (ogni soluzione é un ottimo locale)
- $neighborhood(y) = S(x)$ , ovvero l'insieme di tutte le soluzioni ammissibili per  $x$ 
  - tempo di esecuzione non polinomiale (se il problema é NP-HARD)
  - buona approssimazione (poiché ogni ottimo locale é anche un ottimo globale)

## problema: max cut (giá definito precedentemente)

- 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:

$$|\{ \{u, v\} \mid (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1) \}|$$

## algoritmo di ricerca locale per max cut

- per definire l'algoritmo di ricerca locale, é sufficiente determinare:
  - la soluzione iniziale:

$$V_1 = V, V_2 = \emptyset$$

- l'intorno:

\* dati  $V = \{v_1, \dots, v_n\}$  e  $V_1, V_2$ , le soluzioni dell'intorno di  $(V_1, V_2)$  sono tutte le coppie  $(V_{1i}, V_{2i})$  con  $1 \leq i \leq n$  che possono essere ottenute muovendo un nodo  $v_i$  da  $V_1$  a  $V_2$  o viceversa, ovvero:

$$\text{if } (v_i \in V_1) \ V_{1i} = V_1 \setminus \{v_i\} \text{ e } V_{2i} = V_2 \cup \{v_i\}$$

$$\text{else } (v_i \in V_2) \ V_{1i} = V_1 \cup \{v_i\} \text{ e } V_{2i} = V_2 \setminus \{v_i\}$$

## complessità (algoritmo di ricerca locale per max cut)

- la soluzione iniziale viene banalmente ottenuta in tempo polinomiale
- il test della guardia *while* e l'eventuale determinazione di una migliore soluzione nell'intorno viene effettuata in tempo polinomiale come segue:
  - per ciascuna delle  $n$  soluzioni dell'intorno ( $n$  iterazioni), controlla se la soluzione corrente é migliore ( $n^2$  iterazioni)  $\rightarrow O(n^3)$
- le iterazioni nel *while* sono al massimo  $(\leq) |E| = O(n^2)$ , poiché ogni iterazione migliora la soluzione corrente, ovvero aumenta almeno di 1 il numero di archi del taglio, e vi sono  $|E|$  archi nel taglio (al massimo)
- quindi l'algoritmo ha complessità temporale:

$$O(n^3 n^2) = O(n^5)$$

## approssimazione (algoritmo di ricerca locale per max cut)

- vediamo una proprietà utile a mostrare il rapporto di approssimazione dell'algoritmo:

## fatto (approssimazione (algoritmo di ricerca locale per max cut))

dato un grafo  $G = (V, E)$ , sia  $\delta_i$  il grado di un generico nodo  $v_i \in V$ , allora:

$$\sum_{i=1}^n \delta_i = 2|E|$$

### dimostrazione:

- banalmente vero, poiché ogni arco viene contato 2 volte nella somma, ovvero incrementa la somma di 2

□

## teorema: l'algoritmo di ricerca locale è $\frac{1}{2}$ -approssimante

l'algoritmo di ricerca locale è  $\frac{1}{2}$ -approssimante

### dimostrazione:

- mostriamo che ogni ottimo locale  $(V_1, V_2)$  ha misura:

$$m \geq \frac{|E|}{2}$$

- ciò implica:

$$\frac{m}{m^*} \geq \frac{\frac{|E|}{2}}{|E|} = \frac{1}{2}$$

- poiché  $m^* \leq |E|$
- dato un ottimo locale  $(V_1, V_2)$  denotiamo con  $h$  il numero di archi interni, ovvero con entrambi gli estremi in  $V_1$  o in  $V_2$
- chiaramente,  $m + h = |E|$
- per ogni nodo  $v_i \in V$  definiamo i gradi interni ed esterni del nodo come segue:

- $\delta_i^{int}$  = numero di archi che  $v_i$  ha verso i nodi nella sua stessa partizione, ovvero:

$$\delta_i^{int} = |\{v_k | \{v_i, v_k\} \in E \text{ e } (v_i, v_k \in V_1) \text{ o } (v_i, v_k \in V_2)\}|$$

- $\delta_i^{ext}$  = numero di archi che  $v_i$  ha verso i nodi nell'altra partizione, ovvero:

$$\delta_i^{ext} = |\{v_k | \{v_i, v_k\} \in E \text{ e } (v_i \in V_1, v_k \in V_2) \text{ o } (v_i \in V_2, v_k \in V_1)\}|$$

- poiché la soluzione nell'intorno  $(V_1, V_2)$  ha misura non maggiore ( $\leq$ ) di quella di  $(V_1, V_2)$  (ottimo locale), abbiamo:

$$m - \delta_i^{ext} + \delta_i^{int} \leq m$$

- e quindi:

$$\delta_i^{int} - \delta_i^{ext} \leq 0$$

- riassunto, su tutti i nodi, abbiamo:

$$\sum_{v_i \in V} \delta_i^{int} - \sum_{v_i \in V} \delta_i^{ext} = \sum_{v_i \in V} (\delta_i^{int} - \delta_i^{ext}) \leq 0$$

- dal fatto precedente:

$$\sum_{v_i \in V} \delta_i^{int} = 2h$$

(perché é come sommare i gradi dei nodi del grafo contenente solo gli archi interni)

- e (sempre dal fatto precedente):

$$\sum_{v_i \in V} \delta_i^{ext} = 2m$$

(perché é come sommare i gradi dei nodi del grafo contenente solo gli archi esterni)

- quindi:

$$0 \geq \sum_{v_i \in V} \delta_i^{int} - \sum_{v_i \in V} \delta_i^{ext} = 2h - 2m$$

- ovvero  $m \geq h$

- quindi (aggiungendo  $m$  su entrambi i lati e dividendo per 2), otteniamo:

$$\frac{2m}{2} \geq \frac{(m+h)}{2} = m \geq \frac{(m+h)}{2} = \frac{|E|}{2}$$

□

## **TODO: esempio esecuzione algoritmo di ricerca locale su grafo**

### **conclusioni sulla tecnica della ricerca locale**

- come gli algoritmi greedy, gli algoritmi di ricerca locale hanno buone performance nella pratica e portano alla determinazione di buone euristiche (algoritmi che eseguono bene nella pratica ma che di solito non hanno prestazioni garantite in termini di tempo o approssimazione)

# algorithmic techniques: linear programming (rounding)

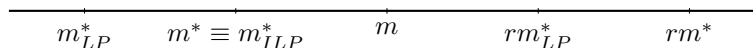
## caratteristiche

- il problema é formulato come un programma lineare intero (ILP: integer linear program)
- programma lineare intero: programmi lineare + vincoli di interezza
- esiste un algoritmo con complessità temporale polinomale (algoritmo ellissoide) per risolvere problemi lineari, ma...
- risolvere un programma lineare intero é un problema NP-HARD
- la formulazione come ILP consente di utilizzare potenti mezzi generali che, in base alle proprietà dell'ILP, sono in grado di fornire algoritmi con buona approssimazione:
  - rounding (arrotondamento)
  - primal-dual (primale-duale)

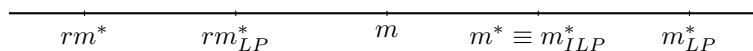
## rounding: caratteristiche

- il problema é formulato come un programma lineare
- il rilassamento lineare (LP) viene ottenuto dall'ILP rilassando i vincoli d'interezza, ovvero sostituendoli con adeguati vincoli lineari (sugli interi)
- la soluzione ottenuta (ottima per LP) é arrotondata ad una vicina soluzione intera ammissibile per ILP
- la misura  $m$  della soluzione ottenuta é in seguito confrontata con quella della soluzione ottima LP, ovvero  $m_{LP}^*$ , cioè un limite inferiore (min) o superiore (max) per  $m^*$

### min problems



### max problems



## problema: min weighted vertex cover

- INPUT:
  - un grafo  $G = (V, E)$
  - un costo intero  $c_j$  associato ad ogni  $v_j \in V$
- SOLUZIONE:
$$U \subseteq V \mid v_j \in U \vee v_k \in U, \forall \{v_j, v_k\} \in E$$
- MISURA: costo totale di  $U$ , ovvero

$$\sum_{v_j \in U} c_j$$

## ILP: min weighted vertex cover

- funzione obiettivo:  $\min \sum_{j=1}^n c_j x_j$
- vincoli:  $x_j + x_k \geq 1, \forall \{v_j, v_k\} \in E$
- vincoli interi:  $x_j \in \{0,1\}, \forall v_j \in V, \forall j \text{ con } 1 \leq j \leq n$

## LP: min weighted vertex cover (rilassamento lineare)

- $\min \sum_{j=1}^n c_j x_j$
- $x_j + x_k \geq 1, \forall \{v_j, v_k\} \in E$
- ~~$x_j \leq 1, \forall v_j \in V$~~  (superfluo)
- $x_j \geq 0, \forall v_j \in V$

## algoritmo: Round-Vertex-Cover

---

### Algorithm 6 Round-Vertex-Cover

---

determina l'ILP associato all'istanza in input

risolvi il rilassamento lineare LP dell'ILP

sia  $\langle x_1^*, x_2^*, \dots, x_n^* \rangle$  la risultante soluzione ottima dell'LP

$\forall v_j$  sia  $x_j = 1$  se  $x_j^* \geq \frac{1}{2}$  e  $x_j = 0$  se  $x_j^* < \frac{1}{2}$

**return** il cover  $U$  associato a  $\langle x_1^*, x_2^*, \dots, x_n^* \rangle$ , ovvero tale che  $U = \{v_j \in V \mid x_j = 1\}$

---

## teorema: l'algoritmo Round-Vertex-Cover é 2-approssimante

l'algoritmo Round-Vertex-Cover é 2-approssimante

### dimostrazione:

- é sufficiente mostrare che:
  1.  $x_1, x_2, \dots, x_n$  é ammissibile per l'ILP (esso soddisfa tutti i vincoli), ovvero  $U$  é un cover
  2.  $\frac{m}{m_{LP}^*} \leq 2$  e quindi anche  $\frac{m}{m^*} \leq \frac{m}{m_{LP}^*} \leq 2$
- DIMOSTRIAMO 1.
  - dall'ammissibilitá di  $\langle x_1^*, x_2^*, \dots, x_n^* \rangle$  per LP, per ogni arco  $\{v_j, v_k\} \in E$  vale  $x_j^* + x_k^* \geq 1$
  - ovvero  $x_j^* \geq 0.5$  o  $x_k^* \geq 0.5$ , cosí che  $x_j = 1$  o  $x_k = 1$ , e quindi  $x_j + x_k \geq 1$  é soddisfatto in ILP
- DIMOSTRIAMO 2.

$$m = \sum_{j=1}^n c_j x_j \dots$$

- (dall'arrotondamento:  $x_j \leq 2x_j^*$ )

$$m = \sum_{j=1}^n c_j x_j \leq \sum_{j=1}^n c_j 2x_j^* = 2 \sum_{j=1}^n c_j x_j^* \dots$$

$$- (\sum_{j=1}^n c_j x_j^* = m_{LP}^*)$$

$$m = \sum_{j=1}^n c_j x_j \leq \sum_{j=1}^n c_j 2x_j^* = 2 \sum_{j=1}^n c_j x_j^* = 2m_{LP}^*$$

- ovvero:

$$\frac{m}{m^*} \leq \frac{m}{m_{LP}^*} \leq 2$$

■

□

## problema: min weighted set cover

• INPUT:

- un universo  $U = \{o_1, o_2, \dots, o_n\}$  di  $n$  oggetti
- una famiglia  $\hat{S} = \{S_1, S_2, \dots, S_h\}$  di  $h$  sottoinsiemi di  $U$
- un costo intero  $c_j$  associato ad ogni  $S_j \in \hat{S}$

• SOLUZIONE: un cover di  $U$ , ovvero una sottofamiglia  $\hat{C} \subseteq \hat{S}$  tale che:

$$\cup_{S_j \in \hat{C}} S_j = U$$

• MISURA: costo totale del cover, ovvero

$$\sum_{S_j \in \hat{C}} c_j$$

- $f$  = frequenza massima di un oggetto nel sottoinsieme  $\hat{S}$ , ovvero ciascun oggetto occorre in al massimo  $f$  sottoinsiemi
- dato un insieme di  $n$  elementi  $\{1, 2, \dots, n\}$  (chiamato universo) e una collezione  $S$  di  $m$  insiemi, la cui unione eguaglia l'universo, il problema del set cover consiste nell'identificare il più piccolo sottoinsieme di  $S$  la cui unione eguaglia l'universo

## ILP: min weighted set cover

- funzione obiettivo:  $\min \sum_{j=1}^h c_j x_j$
- vincoli:  $\sum_{S_j | o_i \in S_j} x_j \geq 1, \forall o_i \in U$
- vincoli interi:  $x_j \in \{0, 1\}, \forall S_j \in \hat{S}$

## LP: min weighted set cover (rilassamento lineare)

- $\min \sum_{j=1}^h c_j x_j$
- $\sum_{S_j | o_i \in S_j} x_j \geq 1, \forall o_i \in U$
- $x_j \leq 1, \forall S_j \in \hat{S}$  (superfluo)
- $x_j \geq 0, \forall S_j \in \hat{S}$



## algoritmo: Round-Set-Cover

---

**Algorithm 7** Round-Set-Cover

---

determina l'ILP associato all'istanza in input  
risolvi il rilassamento lineare LP dell'ILP  
sia  $\langle x_1^*, x_2^*, \dots, x_n^* \rangle$  la risultante soluzione ottima dell'LP  
 $\forall S_j$  sia  $x_j = 1$  se  $x_j^* \geq \frac{1}{f}$  e  $x_j = 0$  se  $x_j^* < \frac{1}{f}$   
**return** il cover risultante, ovvero  $\hat{C} = \{S_j \in \hat{S} \mid x_j = 1\}$

---

**teorema: l'algoritmo Round-Set-Cover é  $f$ -approssimante ( $f \geq 1$ )**

l'algoritmo Round-Set-Cover é  $f$ -approssimante ( $f \geq 1$ )

**dimostrazione:**

• é sufficiente mostrare che:

1.  $x_1, x_2, \dots, x_n$  é ammissibile per l'ILP
2.  $\frac{m}{m_{LP}^*} \leq f$  e quindi anche  $\frac{m}{m^*} \leq \frac{m}{m_{LP}^*} \leq f$

• DIMOSTRIAMO 1.

- dall'ammissibilit  di  $\langle x_1^*, x_2^*, \dots, x_n^* \rangle$  per LP,  $\forall o_i \in U$

$$\sum_{S_j | o_i \in S_j} x_j^* \geq 1$$

- e poich  la somma ha al massimo ( $\leq$ )  $f$  termini, deve esistere  $S_j$  contenente  $o_i$  tale che  $x_j^* \geq \frac{1}{f}$ , ovvero tale che  $x_j = 1$ , e quindi:

$$\sum_{S_j | o_i \in S_j} x_j \geq 1$$

■

• DIMOSTRIAMO 2.

$$m = \sum_{j=1}^h c_j x_j \dots$$

- (dall'arrotondamento:  $x_j \leq f x_j^*$ )

$$m = \sum_{j=1}^h c_j x_j \leq \sum_{j=1}^h c_j f x_j^* = f \sum_{j=1}^h c_j x_j^* \dots$$

- ( $\sum_{j=1}^h c_j x_j^* = m_{LP}^*$ )

$$m = \sum_{j=1}^h c_j x_j \leq \sum_{j=1}^h c_j f x_j^* = f \sum_{j=1}^h c_j x_j^* = f m_{LP}^*$$

- ovvero:

$$\frac{m}{m^*} \leq \frac{m}{m_{LP}^*} \leq f$$

■

□

# algorithmic techniques: dynamic programming (part 1)

## caratteristiche

- come nel paradigma divide-and-conquer, suddividi il problema in sottoproblemi piú piccoli, risolvi ricorsivamente ciasun sottoproblema e combina le soluzioni dei sottoproblemi per formare la soluzione al problema originale
- ricorrenza facile da calcolare che consente di determinare la soluzione ad un sottoproblema dalla soluzione di sottoproblemi piú piccoli
- differentemente da divide-and-conquer, i sottoproblemi non sono indipendenti, ma si sovrappongono, ovvero durante le decomposizioni occorrono frequentemente gli stessi sottoproblemi
- idea: ciascun sottoproblema viene risolto solo una volta, ciò riduce la complessitá temporale
- differentemente da divide-and-conquer, di solito é con approccio bottom-up invece che top-down, ovvero partendo da sottoproblemi piú piccoli e risolvendo progressivamente quelli piú grandi, fino al problema iniziale

## uno sguardo piú ravvicinato...

- il paradigma divide-and-conquer é basato sulla decomposizione dei problemi in sottoproblemi piú piccoli:
  - risolvi ricorsivamente i sottoproblemi
  - combina le soluzioni dei sottoproblemi per determinare la soluzione del problema iniziale
- se un problema di taglia  $n$  é decomposto in  $k$  sottoproblemi di taglie  $n_1, n_2, \dots, n_k < n$ , rispettivamente, allora la complessitá temporale può essere espressa dall ricorrenza

$$T(n) = T(n_1) + T(n_2) + \dots + T(n_k) + C(n)$$

con  $C(n)$  = tempo per combinare le  $k$  sottosoluzioni

- la ricorrenza può essere risolta con metodi differenti, come ad esempio il ricorso al celebre Master Theorem
- un classico esempio di applicazione di divide-and-conquer é il calcolo dei numeri di Fibonacci
- l'algoritmo deriva direttamente dalla definizione ricorsiva di tali numeri

## algoritmo: Fibonacci

- caso base:  $(n \leq 2) \ F(1) = F(2) = 1$
- caso induttivo:  $(n > 2) \ F(n) = F(n-1) + F(n-2), n$

---

### Algorithm 8 Fibonacci

---

```
if  $n = 1$  or  $n = 2$  then
    return 1
else
    return  $Fibonacci(n-1) + Fibonacci(n-2)$ 
end if
```

---

- complessità temporale:

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

- che restituisce:

$$T(n) = O(2^n)$$

- albero delle chiamate ricorsive:

- nota:

- \* inefficiente: gli stessi sottoproblemi vengono risolti ripetutamente per molte volte

## algoritmo: Fibonacci 2

- programmazione dinamica:

- memorizza la soluzione di ciascun sottoproblema in una tabella o in un array, così da evitare di risolverli ripetutamente
- nell'algoritmo risultante,  $F$  é un array esterno globale visibile a tutte le chiamate ricorsive

- nuovo albero delle chiamate ricorsive

---

### Algorithm 9 Fibonacci 2

---

```

if  $n = 1$  or  $n = 2$  then
     $F[n] = 1$ 
    return  $F[n]$ 
else
    if  $F[n]$  é stato già assegnato then
        return  $F[n]$ 
    else
         $F[n] = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$ 
        return  $F[n]$ 
    end if
end if

```

---

## algoritmo: Fibonacci 3

---

### Algorithm 10 Fibonacci 3

---

```

 $F[1] = 1$ 
 $F[2] = 1$ 
for  $i = 3$  to  $n$  do
     $F[i] = F[i-1] + F[i-2]$ 
end for
return  $F[n]$ 

```

---

## riassumendo

- in programmazione dinamica:

- il problema iniziale può essere ricorsivamente decomposto in sottoproblemi
- gli stessi sottoproblemi occorrono molte volte e sono risolti una volta soltanto

- la soluzione di un sottoproblema può essere ottenuta combinando quelle dei sottoproblemi più piccoli
- 2 possibili implementazioni:
  - top-down (con annotazione in tabella)
  - bottom-up

### **top-down vs. bottom-up**

- top-down
  - sfrutta l'annotazione in tabella
  - PRO: risolve solo i sottoproblemi strettamente necessari
  - CON: overhead derivante dalla catena di chiamate ricorsive
- bottom-up
  - é la scelta tipica nella programmazione dinamica
  - PRO: é in ogni caso generalmente più efficiente perché elimina il peso della ricorsione, il quale incide maggiormente sulle prestazioni
  - CON: risolve anche i problemi non necessari

### **divide-and-conquer vs. dynamic programming**

- divide-and-conquer
  - tecnica ricorsiva
  - approccio top-down (problemi divisi in sottoproblemi)
  - utile quando i sottoproblemi sono indipendenti (ovvero differenti)
  - altrimenti, gli stessi sottoproblemi vengono risolti più volte
- dynamic programming
  - tecnica iterativa
  - tipicamente approccio bottom-up
  - utile quando i sottoproblemi si sovrappongono (ovvero coincidono)
  - ciascun sottoproblema viene risolto una volta soltanto

## **algorithmic techniques: dynamic programming (part 2)**

### **progettazione di algoritmi di programmazione dinamica**

- fornire una decomposizione ricorsiva dei sottoproblemi
- calcolare le sottosoluzioni in maniera bottom-up, ovvero partendo dai sottoproblemi di taglia più piccola
  - utilizzare una tabella per memorizzare i risultati dei sottoproblemi
  - evitare il calcolo delle stesse soluzioni sfruttando la tabella
- combinare le soluzioni dei sottoproblemi già risolti per costruire quelle dei sottoproblemi di taglia maggiore, fino alla risoluzione del problema originale

## complessità degli algoritmi di programmazione dinamica

- consideriamo la seguente tabella:
  - $n$  = taglia dei sottoproblemi  $(1, 2, \dots, n)$
  - $k$  = parametri dei sottoproblemi  $(p_1, p_2, \dots, p_k)$
- taglia della tabella = numero di sottoproblemi =  $nk$
- complessità:
  - [ taglia della tabella ]  $\times$  [ tempo per combinare le soluzioni ]
  - il tempo per combinare le soluzioni é sempre banalmente polinomiale
  - la complessità é polinomiale se la tabella ha taglia polinomiale, ovvero se é presente un numero polinomiale di differenti sottoproblemi

## problema: max 0-1 knapsack (giá definito precedentemente)

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

## algoritmo brute force

- semplice algoritmo che enumera tutti i possibili  $2^n$  sottoinsiemi degli  $n$  elementi
- sceglie la migliore combinazione (miglior profitto)
- l'algoritmo di programmazione solitamente ha performance migliori

## progettazione dell'algoritmo di programmazione dinamica

- definizione:
  - $OPT(i, w)$  = sottoinsieme con profitto massimo di oggetti  $1, 2, \dots, i$  con limite di peso  $w$
- fatto:
  - $OPT(n, b)$  = soluzione ottima del problema iniziale
- le seguenti alternative possono occorrere per  $OPT(i, w)$ :
  1.  $OPT$  non seleziona l'oggetto  $i$

- $OPT$  seleziona il migliore tra  $\{1, 2, \dots, i-1\}$  utilizzando il limite di peso  $w$
- 2.  $OPT$  seleziona l'oggetto  $i$ 
  - $OPT$  seleziona il migliore tra  $\{1, 2, \dots, i-1\}$  utilizzando il limite di peso  $w - w_i$
- assumiamo che  $OPT(k, w)$  sia la soluzione ottima per gli elementi  $\{o_1, o_2, \dots, o_k\}$
- nota: la soluzione ottima  $OPT(k+1, w)$  potrebbe non corrispondere a  $OPT(k, w)$
- anche perché  $OPT(k+1, w)$  potrebbe non essere un superset di  $OPT(k, w)$

### **definizione ricorsiva per $OPT$**

- possiamo dunque fornire la seguente definizione ricorsiva per  $OPT$ :
  - $OPT(i, w) = \emptyset$  se  $i = 0$
  - $OPT(i, w) = OPT(i-1, w)$  se  $w_i > w$
  - $OPT(i, w) =$  scelta migliore tra  $OPT(i-1, w)$  e  $OPT(i-1, w - w_i) \cup \{o_i\}$  (altrimenti)

### **definizione ricorsiva per la misura $m$ della soluzione ottima $OPT(i, w)$**

- in termini di misura  $m(i, w)$  della soluzione ottima  $OPT(i, w)$ 
  - $m(i, w) = 0$  se  $i = 0$
  - $m(i, w) = m(i-1, w)$  se  $w_i > w$
  - $m(i, w) = \max\{m(i-1, w), m(i-1, w - w_i) + p_i\}$  (altrimenti)
- chiaramente,  $m^* = m(n, b)$

### **riepilogo definizioni ricorsive per $m$ e $OPT$**

- come risultato, questo significa che il miglior sottoinsieme di  $k$  oggetti con vincolo di peso  $w$  è (mutua esclusione):
  - il miglior sottoinsieme di  $(k-1)$  oggetti con peso totale  $w$
  - il miglior sottoinsieme di  $(k-1)$  oggetti con peso totale  $w - w_k$ , più il contributo (il suo profitto) del  $k$ -esimo oggetto
  - quindi per quando riguarda la seguente formula ricorsiva:
    - \*  $OPT(i, w) = \emptyset$  se  $i = 0$
    - \*  $OPT(i, w) = OPT(i-1, w)$  se  $w_i > w$
    - \*  $OPT(i, w) =$  scelta migliore tra  $OPT(i-1, w)$  e  $OPT(i-1, w - w_i) \cup \{o_i\}$  (altrimenti)
  - o il  $k$ -esimo oggetto non può essere parte della soluzione (poiché il suo solo peso è così grande che l'oggetto stesso non entra nel knapsack)
  - altrimenti, scegliamo la soluzione migliore tra:
    - \* la soluzione che include il nuovo oggetto
    - \* la soluzione migliore che non include il nuovo oggetto

---

**Algorithm 11** Progr-Dyn-Knapsack

---

```
// inizializzazione a 0 della prima riga dell'array bidimensionale  $M$  (da 1 a  $b$ )
for  $w = 1$  to  $b$  do
     $M[0, w] = 0$ 
end for
// inizializzazione a 0 della prima colonna dell'array bidimensionale  $M$  (da 0 a  $n$ )
for  $i = 0$  to  $n$  do
     $M[i, 0] = 0$ 
end for
for  $i = 1$  to  $n$  do
    for  $w = 1$  to  $b$  do
        if  $w_i > w$  then
             $M[i, w] = M[i - 1, w]$ 
        else
             $M[i, w] = \max\{M[i - 1, w], M[i - 1, w - w_i] + p_i\}$ 
        end if
    end for
end for
return  $M[n, b]$ 
```

---

**algoritmo: Progr-Dyn-Knapsack**

- l'algoritmo Progr-Dyn-Knapsack per il problema max 0-1 knapsack, trova il massimo valore che può essere inserito dentro il knapsack
- il valore viene memorizzato in  $M[n, b]$  al termine della procedura
- per scoprire quali sono gli oggetti che sono stati inseriti nella soluzione ottima, bisogna tornare indietro nella tabella:
  - dobbiamo memorizzare in qualche modo ciascun oggetto aggiunto

**algoritmo: Progr-Dyn-Knapsack (trovare gli oggetti inseriti)**

---

**Algorithm 12** Progr-Dyn-Knapsack (trovare gli oggetti inseriti)

---

```
 $i = n$ 
 $k = w$ 
if  $M[i, k] \neq M[i - 1, k]$  then
    marca l'oggetto  $i$  come 'inserito nel knapsack'
     $i = i - 1$ 
     $k = k - w_i$ 
else
    // assumi che l'oggetto  $i$ -esimo non sia stato inserito nel knapsack
     $i = i - 1$ 
end if
```

---

**teorema: l'algoritmo Progr-Dyn-Knapsack ha complessità temporale  $O(nb)$**

la complessità temporale dell'algoritmo Progr-Dyn-Knapsack é  $O(nb)$

**dimostrazione:**

- l'algoritmo impiega  $O(1)$  per ciascuna entry della tabella
- vi sono  $O(nb)$  entry nella tabella

- dopo aver calcolato i valori possiamo risalire per trovare la soluzione ottima:
  - prendi l'elemento  $o_i$  in  $OPT(i, w) \iff M[i-1, w] < M[i, w]$

□

### domanda: l'algoritmo Progr-Dyn-Knapsack é polinomiale?

l'algoritmo Progr-Dyn-Knapsack é polinomiale?  
(suggerimento: considera il caso in cui  $b = 2^n$ )

### risposta:

- per essere polinomiale, la complessità dovrebbe essere polinomiale nel logaritmo dei valori codificati nell'istanza in input, ovvero rispetto a  $\log b$
- questa complessità é chiamata **pseudo-polinomiale**, ovvero polinomiale nella dimensione e nei valori in input, non solo nella dimensione dell'input

### algoritmo Progr-Dyn-Knapsack-Dual: approccio duale

- definizione:
  - $OPT(i, p)$  = sottoinsieme di peso minimo di oggetti  $1, 2, \dots, i$  con profitto almeno pari a  $p$  ( $\geq$ )
- domanda: quale sottoproblema corrisponde alla soluzione ottima?
- le seguenti alternative possono occorrere per  $OPT(i, p)$ :
  1.  $OPT$  non seleziona l'oggetto  $i$ 
    - $OPT$  seleziona il migliore tra  $\{1, 2, \dots, i-1\}$  utilizzando il limite di profitto  $p$
  2.  $OPT$  seleziona l'oggetto  $i$ 
    - $OPT$  seleziona il migliore tra  $\{1, 2, \dots, i-1\}$  utilizzando il limite di profitto  $p - p_i$

### definizione ricorsiva per $OPT$ (duale)

- possiamo dunque fornire la seguente definizione ricorsiva per  $OPT$ :
  - $OPT(i, p)$  = non definito se  $i = 0$
  - $OPT(i, p)$  = scelta migliore tra  $OPT(i-1, p)$  e  $\{o_i\}$  se  $p_i \geq p$
  - $OPT(i, p)$  = scelta migliore tra  $OPT(i-1, p)$  e  $OPT(i-1, p - p_i) \cup \{o_i\}$  (altrimenti, non definito se entrambi non sono definiti)

### definizione ricorsiva per la misura $v$ della soluzione ottima $OPT(i, p)$

- in termini di peso  $v(i, p)$  della soluzione ottima  $OPT(i, p)$ 
  - $v(i, p) = \infty$  se  $i = 0$
  - $v(i, p) = \min\{v(i-1, p), w_i\}$  se  $p_i \geq p$
  - $v(i, p) = \min\{v(i-1, p), v(i-1, p - p_i) + w_i\}$  (altrimenti)



## algoritmo: Progr-Dyn-Knapsack-Dual

---

**Algorithm 13** Progr-Dyn-Knapsack-Dual

---

```
// inizializzazione a  $\infty$  della prima riga dell'array bidimensionale  $V$  (da 1 a  $P$ )
for  $p = 1$  to  $P$  do
     $V[0, p] = \infty$ 
end for
// inizializzazione a  $\infty$  della prima colonna dell'array bidimensionale  $V$  (da 0 a  $n$ )
for  $i = 0$  to  $n$  do
     $V[i, 0] = \infty$ 
end for
for  $i = 1$  to  $n$  do
    for  $p = 1$  to  $P$  do
        if  $p_i \geq p$  then
             $V[i, p] = \min\{V[i-1, p], w_i\}$ 
        else
             $V[i, p] = \min\{V[i-1, p], V[i-1, p-p_i] + w_i\}$ 
        end if
    end for
end for
return  $\max p \mid V[n, p] \leq b$ 
```

---

- problema: come dovremmo scegliere  $P$ ?
- risposta: grande abbastanza da includere l'ottimo, ovvero qualunque *upper bound* di  $m^*$ , cioè  $P \geq m^*$
- scelta:

$$P = np_{\max} \geq \sum_{i=1}^n p_i \geq m^*, \quad (p_{\max} = \max p_j)$$

**teorema: l'algoritmo Progr-Dyn-Knapsack-Dual ha complessità temporale  $O(n^2 p_{\max})$**

l'algoritmo Progr-Dyn-Knapsack-Dual ha complessità temporale  $O(n^2 p_{\max})$

**dimostrazione:**

- l'algoritmo impiega  $O(1)$  per ogni entry della tabella
- vi sono  $O(nP) = O(n^2 p_{\max})$  entry nella tabella
- dopo il calcolo dei valori, possiamo risalire per trovare la soluzione ottima:
  - prendi l'elemento  $o_i$  in  $OPT(i, p) \iff V[i-1, p] > V[i, p]$

□

## approximation schemes: polynomial time approximation scheme (PTAS)

### definizione: PTAS

- un algoritmo  $A$  per un problema di ottimizzazione  $\pi \in NPO$ , è un polynomial time approximation scheme per  $\pi$  se, data un'istanza in input  $x \in I_\pi$  e un numero razionale  $\epsilon > 0$ , esso ritorna una soluzione  $(1+\epsilon)$ -approssimata (per min) o  $(1-\epsilon)$ -approssimata (per max) in tempo polinomiale rispetto alla dimensione dell'istanza  $x$
- **nota:** la complessità temporale può essere esponenziale in  $\frac{1}{\epsilon}$  (esempio  $O(n^{\frac{1}{\epsilon}})$ )
- la complessità di un PTAS può crescere 'drammaticamente' quando  $\epsilon$  decresce
- **nota:** per ogni valore fisso di  $\epsilon$ , un PTAS corrisponde ad un algoritmo di  $(1+\epsilon)$ -approssimazione di tempo polinomiale (o  $(1-\epsilon)$ )

### problema: min multiprocessor scheduling (già definito precedentemente)

- INPUT:
  - insieme di  $n$  jobs  $P$
  - numero di processori  $h$
  - tempo di esecuzione  $t_j$ ,  $\forall p_j \in P$
- SOLUZIONE:
  - una schedule per  $P$ , ovvero una funzione
- MISURA:
  - *makespan* o tempo di completamento di  $f$ , ovvero

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

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

### richiamiamo l'algoritmo Greedy-Graham

- scelta greedy:
  - ad ogni step assegna un job ad uno dei processori correntemente meno carichi
- richiamiamo rapidamente gli step base della dimostrazione del rapporto di approssimazione di Greedy-Graham
  - $T_i(j)$  = tempo di completamento del processore  $i$  alla fine del tempo  $j$  ( $h$  = numero di processori)
- abbiamo utilizzato i seguenti *lower bounds* per  $m^*$ :
  - $m^* \geq \frac{T}{h}$ , 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  $p_j$ , in qualsiasi soluzione uno dei processori deve eseguire  $p_j$
- abbiamo utilizzato il seguente *upper bound* per il valore della soluzione restituita:

- per limitare superiormente il valore della soluzione restituita, se  $k$  é uno dei processori piú carichi e  $p_l$  é 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 piú 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  $\epsilon m^*$  ( $(1+\epsilon)$ -approssimante), cioè un PTAS

- mostriamo ora come far diventare  $t_l$  arbitrariamente piccolo, cioè  $\epsilon m^*$ , ottenendo una  $(1+\epsilon)$ -approssimazione, ovvero un PTAS

## ottenere un PTAS...

- proviamo a far diventare  $t_l$  piú piccolo possibile
- possiamo sfruttare il seguente lemma:

**lemma:**  $t_i \leq \frac{T}{i}$

se  $t_1, t_2, \dots, t_n$  sono ordinati in maniera decrescente e  $t_1 + t_2 + \dots + t_n = T$ , allora:

$$\forall i, 1 \leq i \leq n \quad t_i \leq \frac{T}{i}$$

**dimostrazione:**

- assumiamo per contraddizione che  $t_i > \frac{T}{i}$ , allora:

$$t_1 + t_2 + \dots + t_n \geq i \cdot t_i > i \cdot \frac{T}{i} = T$$

- contraddizione, poiché  $T = \sum_{i=1}^n t_i$

□

## PTAS: idea sottostante

- calcola la soluzione ottima per i primi  $q$  jobs
- completa assegnando in maniera greedy i restanti jobs
- se otteniamo  $t_l \leq \epsilon \cdot m^*$ , allora:

$$m \leq \frac{T}{h} + \left(\frac{h-1}{h}\right) \cdot t_l < \frac{T}{h} + t_l \leq m^* + \epsilon \cdot m^* = (1 + \epsilon)m^*$$

- partendo dal lemma precedente, é sufficiente assegnare  $q$  in modo tale che, poiché  $l > q$
- questo vale per

$$q = \left\lceil \frac{h}{\epsilon} \right\rceil$$

- anzi, le disuguaglianze

$$t_l \leq \frac{T}{l} \leq \frac{T}{q+1} \leq \epsilon \cdot \frac{T}{h} \leq \epsilon \cdot m^*$$

sono vere se consideriamo

$$q \geq \frac{h}{\epsilon} - 1$$

## algoritmo: PTAS-Scheduling

---

### Algorithm 14 PTAS-Scheduling

---

```
ordina i jobs in modo decrescente rispetto ai tempi di esecuzione  $t_i$ 
sia  $p_1, p_2, \dots, p_n$  la risultante sequenza ordinata con  $t_1 \geq t_2 \geq \dots \geq t_n$ 
calcola lo schedule ottimo  $f$  per i primi  $q = \lceil \frac{h}{\epsilon} \rceil$  jobs
for  $j = q+1$  to  $n$  do
  assegna  $p_j$  al processore  $i$  con minimo  $T_i(j-1)$  // ovvero  $f(p_j) = i$ 
end for
return schedule  $f$ 
```

---

## teorema: l'algoritmo PTAS-Scheduling ritorna sempre una soluzione $(1 + \epsilon)$ -approssimata

l'algoritmo PTAS-Scheduling ritorna sempre una soluzione  $(1 + \epsilon)$ -approssimata

### dimostrazione:

- sia  $t \leq m^*$  il tempo di completamento della soluzione ottima per i primi  $q$  jobs
  - se  $m \leq t$ , ovvero la fase greedy non ha incrementato il tempo di completamento, allora l'algoritmo ritorna la soluzione ottima
  - se  $m > t$ , allora nuovamente, denotando con  $k$  il processore più carico al termine dell'algoritmo e con  $p_l$  l'ultimo job assegnato a  $k$  nella fase greedy:

$$m = T_k(n) = T_k(l-1) + t_l \leq \frac{T-t_l}{h} + t_l = \frac{T}{h} + \frac{h-1}{h} t_l < \frac{T}{h} + t_l \leq m^* + \epsilon \cdot m^* = (1 + \epsilon)m^*$$

- ovvero

$$\frac{m}{m^*} \leq 1 + \epsilon$$

□

- quindi l'algoritmo soddisfa il requisito di approssimazione per PTAS, ma è un PTAS?
- complessità:
  - l'ordinamento iniziale richiede  $O(n \cdot \log n)$  step temporali
  - la ricerca esaustiva di una soluzione ottima per i primi  $q$  jobs richiede al massimo  $O(h^{\frac{1}{\epsilon}})$ , poiché vi sono al massimo  $h^q \approx h^{\frac{1}{\epsilon}}$  possibili soluzioni ( $h$  possibili scelte per ciascuno dei  $q$  jobs)
  - l'ultimo *for* esegue al massimo  $n$  iterazioni, ciascuna delle quali richiede  $O(h)$
  - quindi, la complessità temporale generale è:

$$O(n \cdot \log n + h^{\frac{1}{\epsilon}} + n \cdot h)$$

- tale complessità è esponenziale in  $h$ , e dunque può essere esponenziale nella dimensione dell'input (per esempio se  $h = n$ )
- quindi **non abbiamo un PTAS**, a meno che non fissiamo  $h$  in modo tale che sia un valore costante dato (ad esempio:  $h = 2, h = 3, \dots, h = 100, \dots$ )
- fissare  $h$  costante è equivalente a dire che  $h$  non dipende dall'istanza di input, o analogamente che non è parte dell'input
- in altre parole esso corrisponde a considerare il seguente problema:

### problema: min $h$ -processor scheduling

- INPUT:
  - insieme di  $n$  jobs  $P$
  - tempo di esecuzione  $t_j$ ,  $\forall p_j \in P$
- SOLUZIONE:
  - una schedule per  $P$ , ovvero una funzione
- MISURA:
  - *makespan* o tempo di completamento di  $f$ , ovvero

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

### teorema: l'algoritmo PTAS-Scheduling è un PTAS per il problema min $h$ -processor scheduling

l'algoritmo PTAS-Scheduling è un PTAS per il problema min  $h$ -processor scheduling

#### dimostrazione:

- come già visto, la complessità temporale di PTAS-Scheduling è:

$$O(n \cdot \log n + h^{\frac{1}{\epsilon}} + n \cdot h)$$

- che è polinomiale nella dimensione dell'input (ma esponenziale in  $\frac{1}{\epsilon}$ )
- approssimazione:  $1 + \epsilon$

□

- **nota:** min  $h$ -processor scheduling con  $h = 2$  coincide con il famoso min partition problem, che a sua volta ammette un PTAS

### **problema: min partition**

- INPUT:

- insieme di oggetti  $X$
- peso intero positivo  $a_i, \forall o_i \in X$

- SOLUZIONE:

- una partizione di  $X$  in 2 sottoinsiemi  $X_1$  e  $X_2$ , tale che

$$X_1 \cap X_2 = \emptyset \text{ e } X_1 \cup X_2 = X$$

- MISURA:

$$\max\left\{\sum_{o_i \in X_1} a_i, \sum_{o_i \in X_2} a_i\right\}$$

### **ottenere un PTAS per il problema min partition**

- chiaramente, gli oggetti corrispondono ai jobs, i pesi ai tempi di esecuzione e i 2 sottoinsiemi ai 2 processori
- é possibile estendere il precedente risultato per ottenere un PTAS per il problema min multiprocessor scheduling, ovvero per ogni (non costante) numero  $h$  di processori
- richiamiamo che nella precedente dimostrazione di approssimazione di PTAS, denotando con  $t$  il tempo di completamento dello schedule ottimo ottenuto per i primi  $q$  jobs, abbiamo 2 casi:
  1.  $m \leq t$ , ovvero la fase greedy non accresce il tempo di completamento, allora l'algoritmo restituisce la soluzione ottima poiché  $t \leq m^*$
  2.  $m > t$ , in tal caso vengono applicati gli step abituali per la parte greedy
- idea: la dimostrazione dell'approssimazione continua ad essere valida se, invece di determinare l'ottimo per i primi  $q$  jobs, determiniamo una soluzione approssimata per questi ultimi, ovvero tale che  $t \leq (1 + \epsilon)m^*$

### **lemma: algoritmo di programmazione dinamica polinomiale (schedule approssimato per i primi $q$ jobs)**

esiste un algoritmo di programmazione dinamica che determina in tempo polinomiale uno scheduling per i primi  $q$  jobs avente tempo di completamento  $t \leq (1 + \epsilon)m^*$

### **teorema: esiste un PTAS per il problema min multiprocessor scheduling**

esiste un PTAS per il problema min multiprocessor scheduling

# approximation schemes: fully polynomial time approximation scheme (FPTAS)

## definizione: FPTAS

- un algoritmo  $A$  per un problema di ottimizzazione  $\pi \in NPO$ , è un fully polynomial time approximation scheme per  $\pi$  se, data un'istanza in input  $x \in I_\pi$  e un numero razionale  $\epsilon > 0$ , esso ritorna una soluzione  $(1+\epsilon)$ -approssimata (per min) o  $(1-\epsilon)$ -approssimata (per max) in tempo polinomiale, entrambe rispetto alla dimensione dell'istanza  $x$  e in  $\frac{1}{\epsilon}$
- **nota:** FPTAS mantiene una buona complessità temporale quando  $\epsilon$  decresce
- esempio:
  - $O(n \cdot \log n + 2^{\frac{1}{\epsilon}}) \rightarrow$  non è un FPTAS
  - $O(\frac{n \cdot \log n}{\epsilon^2}) \rightarrow$  è un FPTAS
- nella pratica dobbiamo stare attenti a far sí che  $\frac{1}{\epsilon}$  (o una funzione piú logaritmica) non compaia in un esponente

## FPTAS-Knapsack

- sappiamo che Progr-Dyn-Knapsack-Dual ha una complessità temporale pseudo-polinomiale  $O(n^2 \cdot p_{max})$
- dove  $p_{max} = \max p_j$  è il massimo profitto di un oggetto
- se possiamo approssimare i profitti originali con profitti piú piccoli allora la complessità viene ridotta, ottenendo comunque una buona approssimazione
- diamo un'occhiata piú ravvicinata a quest'idea:
- **idea:**
  1. approssimiamo i profitti di multipli di  $k$  per un parametro intero adatto  $k > 0$ 
$$p'_i = \lfloor \frac{p_i}{k} \rfloor \cdot k$$
  2. se  $k$  è sufficiente piccolo, i nuovi profitti  $p'_i$  approssimano sufficientemente i profitti originali  $p_i$ , quindi una soluzione ottima per  $p'_i$  approssima bene la soluzione ottima per i profitti originali  $p_i$
  3. scalando tutti i profitti  $p'_i$ , dividendoli per  $k$ , ovvero
$$p'_i = \frac{\lfloor \frac{p_i}{k} \rfloor \cdot k}{k} = \lfloor \frac{p_i}{k} \rfloor$$
otteniamo un'istanza equivalente con profitti piú piccoli
  4. **complessità:** l'algoritmo di programmazione dinamica duale applicato a questa istanza produce quindi una complessità minore

$$O(n^2 \cdot p'_{max}) = O(n^2 \cdot \frac{p_{max}}{k})$$

5. **approssimazione:**
  - errore al massimo  $k$  per ogni oggetto scelto, ovvero al massimo  $n \cdot k$  in totale
  - dunque,  $m \geq m^* - n \cdot k$
6. **idea:** scegli  $k$  sufficientemente grande da ottenere una complessità polinomiale e sufficientemente piccolo da fornire una buona approssimazione

$$k = \lfloor \frac{\epsilon \cdot p_{max}}{n} \rfloor$$

## algoritmo: FPTAS-Knapsack

---

**Algorithm 15** FPTAS-Knapsack

---

$k = \lfloor \frac{\epsilon \cdot p_{max}}{n} \rfloor$   
trova la soluzione ottima per l'istanza con profitti scalati  $p'_i = \lfloor \frac{p_i}{k} \rfloor$  utilizzando l'algoritmo di programmazione dinamica pseudo-polinomiale Progr-Dyn-Knapsack-Dual  
sia  $S$  l'insieme di oggetti ritornati  
**return**  $S$

---

- **complessità:**

$$O(n^2 \cdot p'_{max}) = O(n^2 \cdot \frac{p_{max}}{k}) = O(\frac{n^2 \cdot p_{max}}{\frac{\epsilon \cdot p_{max}}{n}}) = O(\frac{n^3}{\epsilon})$$

- **approssimazione o errore:**

$$\frac{m}{m^*} \geq \frac{m^* - n \cdot k}{m^*} = 1 - \frac{n \cdot k}{m^*} \geq 1 - \frac{n \cdot k}{p_{max}} \geq 1 - \frac{\frac{n \cdot \epsilon \cdot p_{max}}{n}}{p_{max}} = 1 - \epsilon$$

dunque:

$$\frac{m}{m^*} \geq 1 - \epsilon$$

- **nota:** non abbiamo scalato i pesi, perché non è garantito che tornando ai pesi originali la soluzione resti ammissibile (potrebbe superare il peso totale  $b$ )

### lemma: $m \geq m^* - n \cdot k$ (FPTAS-Knapsack)

$$m \geq m^* - n \cdot k$$

#### dimostrazione:

- sia  $S = \{o_{j1}, o_{j2}, \dots, o_{jh}\}$  e sia  $S^* = \{o_{i1}, o_{i2}, \dots, o_{il}\}$  la soluzione ottima, allora:

$$m = p_{j1} + p_{j2} + \dots + p_{jh} \text{ e } m^* = p_{i1} + p_{i2} + \dots + p_{il}$$

- assumiamo per contraddizione che  $m < m^* - n \cdot k$

$$\begin{aligned} \lfloor \frac{p_{i1}}{k} \rfloor + \dots + \lfloor \frac{p_{il}}{k} \rfloor &\geq (\frac{p_{i1}}{k} - 1) + \dots + (\frac{p_{il}}{k} - 1) = \\ &= \frac{p_{i1} + \dots + p_{il}}{k} - l \geq \frac{p_{i1} + \dots + p_{il}}{k} - n = \frac{m^*}{k} - n > \dots \end{aligned}$$

- (dall'ipotesi  $m < m^* - n \cdot k$ )

$$\dots > \frac{m + n \cdot k}{k} - n = \frac{m}{k} = \frac{p_{j1} + \dots + p_{jh}}{k} \geq \lfloor \frac{p_{j1}}{k} \rfloor + \dots + \lfloor \frac{p_{jh}}{k} \rfloor$$

- **contraddizione:** poiché  $S^*$  sarebbe una soluzione strettamente migliore di  $S$  per l'istanza con profitti scalati, quindi contraddicendo l'ottimalità di Progr-Dyn-Knapsack-Dual per l'istanza con profitti scalati

### teorema: FPTAS-Knapsack è un FPTAS per il problema max 0-1 knapsack

- **complessità:**

$$O(\frac{n^2 \cdot p_{max}}{k}) = O(\frac{n^2 \cdot p_{max}}{\frac{\epsilon \cdot p_{max}}{n}}) = O(\frac{n^3}{\epsilon})$$

- **approssimazione:** (dal lemma precedente)

$$\frac{m}{m^*} \geq \frac{m^* - n \cdot k}{m^*} = 1 - \frac{n \cdot k}{m^*} \geq 1 - \frac{n \cdot k}{p_{max}} \geq 1 - \frac{n \cdot \frac{\epsilon \cdot p_{max}}{n}}{p_{max}} = 1 - \epsilon$$

□



## é possibile ridurre la complessità temporale?

- complessità temporale della programmazione dinamica:

$$O(n \cdot \sum_j p_j) = O(n \cdot n \cdot p_{max}) = O(n^2 \cdot p_{max})$$

- complessità con i profitti scalati:

$$O(n \cdot \sum_j \frac{p_j}{k}) = O(n \cdot \frac{n \cdot p_{max}}{k}) = O(\frac{n^2 \cdot p_{max}}{k})$$

- approssimazione:

$$1 - \frac{n \cdot k}{m^*} \geq 1 - \frac{n \cdot k}{p_{max}}$$

- **nota:**  $p_{max}$  e  $n \cdot p_{max}$  rispettivamente sono il *lower bound* e l'*upper bound* per  $m^*$ , ovvero  $p_{max} \leq m^* \leq n \cdot p_{max}$
- **nota:**  $p_{max}$  può essere più piccolo di  $m^*$  e  $n \cdot p_{max}$  può essere più grande
- **domanda:** é possibile fornire dei *bound* migliori per  $m^*$ ?
- **osservazioni:**

- utilizzando un *lower bound* migliore per  $m^*$  e mantenendo lo stesso  $k$  otteniamo una migliore approssimazione e analogamente incrementando  $k$  (riducendo la complessità) manteniamo la stessa approssimazione
- utilizzando un *upper bound* migliore per  $m^*$  e mantenendo lo stesso  $k$  otteniamo una migliore approssimazione e analogamente decrementando  $k$  (migliorando l'approssimazione) manteniamo la stessa complessità

## come migliorare i bound per $m^*$

- richiamiamo l'algoritmo Modified-Greedy  $\frac{1}{2}$ -approssimante e sia  $M_{mod}$  la misura della soluzione da esso ritornata
- per definizione di approssimazione e poiché  $M_{mod} \leq m^*$

$$M_{mod} \leq m^* \leq 2 \cdot M_{mod}$$

- quindi, considerando

$$P' = \lceil \frac{2 \cdot M_{mod}}{k} \rceil$$

come massimo profitto in Progr-Dyn-Knapsack-Dual (ovvero  $P'$  é il numero di colonne nella tabella di Progr-Dyn-Knapsack-Dual), otteniamo una migliore complessità...

- impostando il parametro

$$k = \lfloor \frac{\epsilon \cdot M_{mod}}{n} \rfloor$$

ed eseguendo Progr-Dyn-Knapsack-Dual Modificato sull'istanza con profitti scalati

$$p'_i = \lfloor \frac{p_i}{k} \rfloor$$

otteniamo la seguente approssimazione e complessità temporale

- **complessità:**

$$O(\frac{n \cdot M_{mod}}{k}) = O(\frac{n \cdot M_{mod}}{\frac{\epsilon \cdot M_{mod}}{n}}) = O(\frac{n^2}{\epsilon})$$

- **approssimazione:**

$$\frac{m}{m^*} \geq 1 - \frac{n \cdot k}{m^*} \geq 1 - \frac{n \cdot k}{M_{mod}} \geq \frac{n \cdot \frac{\epsilon \cdot M_{mod}}{n}}{M_{mod}} = 1 - \epsilon$$

## algoritmo: New-FPTAS-Knapsack

---

**Algorithm 16** New-FPTAS-Knapsack

---

calcola  $M_{mod}$  eseguendo l'algoritmo Modified-Greedy e sia  $k = \lfloor \frac{\epsilon \cdot M_{mod}}{n} \rfloor$   
trova la soluzione ottima per l'istanza con profitti scalati  $p'_i = \lfloor \frac{p_i}{k} \rfloor$  utilizzando  
l'algoritmo Progr-Dyn-Knapsack-Dual Modificato  
sia  $S$  l'insieme di oggetti ritornati  
**return**  $S$

---

domanda: possiamo evitare il calcolo di  $P'$  all'interno di  
Progr-Dyn-Knapsack-Dual Modificato?

## **alternative approaches**

### **performance garantite**

- finora abbiamo considerato approcci con performance garantite
- pro:
  - approssimazione e tempo di esecuzione garantiti per ogni istanza in input
  - prende in considerazione il caso peggiore
- contro:
  - alcuni problemi non ammettono algoritmi con performance garantite
  - per alcuni problemi non sono noti algoritmi con performance garantite
  - a volte, cattivi comportamenti nella pratica

### **restrizione dell'insieme delle istanze**

- performance garantite nel sottoinsieme delle istanze in input che sono significative o di interesse
- pro:
  - permette di applicare nuovamente l'approccio con performance garantite
- contro:
  - performance garantite solo per il sottoinsieme scelto di istanze o per un caso particolare
- esempio: metric TSP (con disequazioni triangolari)

### **media o analisi probabilistica**

- in generale, assumendo una distribuzione di probabilità dell'istanza, essa eguaglia la media o la performance attesa, alcune volte con alta probabilità
- pro:
  - può accorgersi di buoni comportamenti pratici dell'algoritmo
  - è un metodo analitico, ovvero basato su dimostrazioni matematiche
- contro:
  - non ha performance garantite
  - l'analisi è spesso complessa
  - spesso la distribuzione delle istanze in input è sconosciuta

### **euristiche**

- algoritmi con un buon comportamento pratico ma solitamente con performance non dimostrabili
- pro:
  - buon comportamento pratico
- contro:
  - performance spesso non dimostrabili

## algoritmi randomizzati

- effettuano scelte randomiche durante la computazione
- le soluzioni ritornate possono essere differenti per esecuzioni differenti sullo stesso input
- esse sono infatti variabili random (per ogni istanza vi sono diverse soluzioni, ciascuna restituita con una certa probabilità determinata in accordo con le scelte randomiche dell'algoritmo)
- é mostrato che, fissata una qualsiasi istanza, il valore atteso delle performance é buono o la performance é buona con alta probabilità (sempre in accordo con le scelte randomiche)
- pro:
  - sono generalmente semplici
  - sono veloci (sia analiticamente che in pratica)
- contro:
  - incertezza del risultato per ogni istanza fissata
  - impossibilit  di fare scelte realmente randomiche (sebbene esse possano essere simulate)

## algoritmi randomizzati

- $m \rightarrow$  é una variabile randomica
- $E(m) \rightarrow$  é il valore atteso di  $m$  calcolato in accordo con le scelte randomiche dell'algoritmo

## definizione: algoritmi randomizzati e $r$ -approssimazione

- un algoritmo randomizzato  $A$  é  $r$ -approssimante se:

$$\frac{E(m)}{m^*} \leq r \quad (\text{per min})$$

$$\frac{E(m)}{m^*} \geq r \quad (\text{per max})$$

## problema: max weighted cut

- INPUT:
  - grafo  $G = (V, E)$
  - peso non-negativo  $w_{ij} > 0, \forall \{v_i, v_j\} \in E$
- SOLUZIONE: partizione di  $V$  in 2 sottoinsiemi  $V_1$  e  $V_2$  tale che

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

- MISURA: peso del taglio, ovvero

$$\sum_{\{v_i, v_j\} \in E | v_i \in V_1 \wedge v_j \in V_2} w_{ij}$$

## algoritmo: Random-Cut

---

**Algorithm 17** Random-Cut

---

```
 $V_1 = \emptyset$   
 $V_2 = \emptyset$   
for  $i = 1$  to  $n$  do  
  inserisci  $v_i$  in  $V_1$  con probabilità  $\frac{1}{2}$  indipendentemente dagli altri nodi (oppure  
  in  $V_2$ )  
end for  
return  $V_1$  e  $V \setminus V_1$  ( $\equiv V_2$ )
```

---

- chiaramente, l'algoritmo é polinomale

## teorema: Random-Cut é $\frac{1}{2}$ -approssimante

Random-Cut é  $\frac{1}{2}$ -approssimante

### dimostrazione:

- sia  $x_{ij}$  la variabile randomica "l'arco  $\{v_i, v_j\}$ " é nel taglio"
- allora

$$m = \sum_{\{v_i, v_j\} \in E} w_{ij} \cdot x_{ij}$$

- quindi

$$\begin{aligned} E(m) &= E\left(\sum_{\{v_i, v_j\} \in E} w_{ij} \cdot x_{ij}\right) = \sum_{\{v_i, v_j\} \in E} w_{ij} \cdot E(x_{ij}) = \\ &= \sum_{\{v_i, v_j\} \in E} w_{ij} \cdot P(x_{ij} = 1) = \\ &= \sum_{\{v_i, v_j\} \in E} w_{ij} \cdot P((v_i \in V_1 \wedge v_j \in V_2) \vee (v_i \in V_2 \wedge v_j \in V_1)) = \\ &= \sum_{\{v_i, v_j\} \in E} w_{ij} \cdot P\left(\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2}\right) = \frac{1}{2} \cdot \sum_{\{v_i, v_j\} \in E} w_{ij} \geq \frac{m^*}{2} \end{aligned}$$

- dunque

$$\frac{E(m)}{m^*} \geq \frac{1}{2}$$

□

## problema: min weighted set cover (giá definito precedentemente)

- INPUT:
  - un universo  $U = \{o_1, o_2, \dots, o_n\}$  di  $n$  oggetti
  - una famiglia  $\hat{S} = \{S_1, S_2, \dots, S_h\}$  di  $h$  sottoinsiemi di  $U$
  - un costo intero  $c_j$  associato ad ogni  $S_j \in \hat{S}$
- SOLUZIONE: un cover di  $U$ , ovvero una sottofamiglia  $\hat{C} \subseteq \hat{S}$  tale che:

$$\cup_{S_j \in \hat{C}} S_j = U$$

- MISURA: costo totale del cover, ovvero

$$\sum_{S_j \in \hat{C}} c_j$$

- $f$  = frequenza massima di un oggetto nel sottoinsieme  $\hat{S}$ , ovvero ciascun oggetto occorre in al massimo  $f$  sottoinsiemi
- dato un insieme di  $n$  elementi  $\{1, 2, \dots, n\}$  (chiamato universo) e una collezione  $S$  di  $m$  insiemi, la cui unione eguaglia l'universo, il problema del set cover consiste nell'identificare il più piccolo sottoinsieme di  $S$  la cui unione eguaglia l'universo

### algoritmo greedy per il problema min weighted set cover

- **nota:** nella scelta dei sottoinsiemi da inserire nel cover:
  - non possiamo considerare solo i costi, perché in proporzione non potremmo coprire abbastanza elementi di  $U$
  - non possiamo considerare solo il numero di oggetti coperti, poiché potremmo incorrere in un costo eccessivo
- **scelta greedy:** ad ogni step scegli il sottoinsieme avente costo minimo per il nuovo oggetto coperto

### scelta greedy

- ad un dato step  $j$  in cui nell'ordine l'algoritmo ha selezionato  $j-1$  sottoinsiemi  $S_1, \dots, S_{j-1}$ , l'efficacia di un sottoinsieme  $S_k$  non ancora scelto è definito come:

$$eff(S_k) = \frac{c_k}{|S_k \cap \overline{C_{j-1}}|}$$

- dove:
  - $c_k$  = costo di  $S_k$
  - $C_{j-1} = (S_1 \cup \dots \cup S_{j-1})$
  - $\overline{C_{j-1}} = U \setminus C_{j-1}$  (insieme degli elementi non ancora selezionati in  $U$ )
  - $|S_k \cap \overline{C_{j-1}}|$  = insieme degli elementi in  $S_k$  non ancora selezionati in  $U$ )
- nello step  $j$  scegli un sottoinsieme  $S_j$  di efficacia minima, ovvero tale che

$$eff(S_j) = \min\{eff(S_k) \mid S_k \text{ non è stato ancora scelto}\}$$

### algoritmo: Greedy-Min-Weighted-Set-Cover

---

#### Algorithm 18 Greedy-Min-Weighted-Set-Cover

---

```

// C = insieme degli oggetti coperti
C = ∅
// Ĉ = insieme dei sottoinsiemi scelti nel cover
Ĉ = ∅
j = 1
while C ≠ U do
  sia Sj il sottoinsieme di efficacia minima
  Ĉ = Ĉ ∪ Sj
  per ogni oggetto oi ∈ Sj ∩ C̄, sia price(oi) = eff(Sj)
  C = C ∪ Sj
  j = j + 1
end while
return Ĉ

```

---

**lemma:**  $m = \sum_{S_j \in \hat{C}} c_j = \sum_{i=1}^n price(o_i)$

$$m = \sum_{S_j \in \hat{C}} c_j = \sum_{i=1}^n price(o_i)$$

**dimostrazione:**

- banale, inquanto la somma dei *price* degli oggetti coperti durante lo step  $j$  é proprio  $c_j$
- in altre parole, il costo totale  $c_k$  viene suddiviso tra gli oggetti coperti

□

**lemma:**  $price'(o_i) \geq eff(S'_j)$

- $\forall j$ , data una qualunque scelta di sottoinsiemi  $S'_j, \dots, S'_t$ , che forma un cover con i sottoinsiemi  $S_1, \dots, S_{j-1}$  scelti dall'algoritmo greedy all'inizio dello step  $j$
- $\forall o_i$  (oggetto) non ancora coperto all'inizio dello step  $j$

$$price'(o_i) \geq eff(S_j)$$

dove:

- $S_j$  é il sottoinsieme scelto dall'algoritmo greedy allo step  $j$
- $eff(S_j)$  é l'efficacia del sottoinsieme  $S_j$
- $price'(o_i)$  é l'efficacia del sottoinsieme  $S'_l$  che copre  $o_i$  assumendo che, partendo dallo step  $j$ , la scelta greedy é effettuata solo tra i sottoinsiemi  $S'_j, \dots, S'_t$

**dimostrazione:**

- é sufficiente osservare che  $eff(S_j)$  é il *min covering price* per l'oggetto allo step  $j$
- e che l'efficacia di un sottoinsieme non scelto può solo aumentare durante gli step successivi
- poiché il suo costo é fisso, ma alcuni altri oggetti possono essere coperti durante i successivi step (scelta di ulteriori sottoinsiemi)
- quindi il  $price'(o_i)$ , ovvero l'efficacia del sottoinsieme  $S'_l$  ( $eff(S'_l)$ ) con  $l \geq j$  tra  $S'_j, \dots, S'_t$  che lo copre é almeno uguale a ( $\geq$ )  $eff(S_j)$

□

**lemma:**  $price(o_i) \leq \frac{m^*}{n-i+1}$

- siano  $o_1, \dots, o_n$  gli oggetti elencati nell'ordine di covering dell'algoritmo greedy (ovvero tale che gli oggetti coperti durante lo step  $j$  vegono elencati dopo quelli coperti negli step precedenti e prima di quelli coperti negli step successivi)
- allora

$$price(o_i) \leq \frac{m^*}{n-i+1} \quad \forall i, 1 \leq i \leq n$$

**dimostrazione:**

- all'inizio dello step  $j$ , poiché gli insiemi non ancora scelti di una soluzione ottima possono coprire tutti gli oggetti non coperti con un costo generale al massimo pari a  $m^*$ :

- esiste un sottoinsieme  $S_k$  di efficacia al massimo pari a

$$\frac{m^*}{|\overline{C_{j-1}}|}$$

dove  $\overline{C_{j-1}}$  é i sottoinsieme di oggetti non ancora coperti all'inizio dello step  $j$

- infatti, se non é questo il caso, dato che il  $price(o_i) = eff(S_j)$  del sottoinsieme  $S_j$  scelto dall'algoritmo greedy
- dal precedente lemma, per ogni possibile scelta di sottoinsiemi restanti per completare il cover, ovvero per ogni possibile  $price$  degli oggetti rimanenti:

$$\sum_{o_i \in \overline{C_{j-1}}} price'(o_i) \geq \sum_{o_i \in \overline{C_{j-1}}} eff(S_j) > \sum_{o_i \in \overline{C_{j-1}}} \frac{m^*}{|\overline{C_{j-1}}|} = |\overline{C_{j-1}}| \cdot \frac{m^*}{|\overline{C_{j-1}}|} = m^*$$

- **contraddizione:** all'ipotesi che esiste una scelta di sottoinsiemi che copre i restanti oggetti con un costo al massimo pari a  $m^*$
- quindi:

$$price(o_i) \leq \frac{m^*}{|\overline{C_{j-1}}|}$$

- ma  $|\overline{C_{j-1}}| \geq n - i + 1$  poiché  $o_i, \dots, o_n \in \overline{C_{j-1}}$  e dunque:

$$price(o_i) \leq \frac{m^*}{n - i + 1}$$

□

### teorema: Greedy-Min-Weighted-Set-Cover é $H_n$ -approssimante

l'algoritmo Greedy-Min-Weighted-Set-Cover é  $H_n$ -approssimante, con  $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + 1$  **dimostrazione:**

$$m = \sum_{i=1}^n price(o_i) \leq \sum_{i=1}^n \frac{m^*}{n - i + 1} = m^* \left( \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + 1 \right) = m^* \cdot H_n$$

- da cui:

$$\frac{m}{m^*} \leq H_n$$

□

- **nota:**  $\forall n > 1 \rightarrow \ln(n+1) \leq H_n \leq \ln(n) + 1$
- quindi  $r$  ha una dipendenza logaritmica dalla dimensione dell'input

### esempio: $H_n \cdot m^*$ !

- il rapporto di approssimazione dell'algoritmo greedy é almeno  $H_n$
- considera infatti la seguente istanza:
  - $S_1$ : 1 oggetto, costo  $c_1 = \frac{1}{n}$
  - $S_2$ : 1 oggetto, costo  $c_2 = \frac{1}{n-1}$
  - $S_3$ : 1 oggetto, costo  $c_3 = \frac{1}{n-2}$
  - ...
  - $S_n$ : 1 oggetto, costo  $c_n = 1$



- $S_{n+1}$ :  $n$  oggetti, costo  $c_{n+1} = 1 + \epsilon$  (con  $\epsilon > 0$ , arbitrariamente piccolo)
- esecuzione:
  - **step 1**:
    - \*  $eff(S_1) < eff(S_{n+1}) = \frac{1}{n} < \frac{1+\epsilon}{n}$
    - \* scelgo l'insieme di efficacia minima  $S_1$
  - **step 2**:
    - \*  $eff(S_2) < eff(S_{n+1}) = \frac{1}{n-1} < \frac{1+\epsilon}{n-1}$
    - \* scelgo l'insieme di efficacia minima  $S_2$
  - ...
  - l'algoritmo **non** sceglierà mai il sottoinsieme  $S_{n+1}$

- quindi:

$$m = \sum_{j=1}^n \frac{1}{j} = H_n$$

$$m^* = 1 + \epsilon \quad (\text{con } \epsilon > 0, \text{ arbitrariamente piccolo})$$

- dunque:

$$\frac{m}{m^*} = \frac{H_n}{1 + \epsilon}$$

- allora, per  $\epsilon \rightarrow 0$ :

$$\frac{m}{m^*} \rightarrow H_n$$

- dunque la misura della soluzione restituita dall'algoritmo greedy, per alcune istanze, è  $H_n$  volte quella ottima!