Algoritmi e complessità

Francesco Tomaselli

22 febbraio 2021

Indice

1 Algoritmi di approssimazione

In questa parte si introdurranno gli algoritmi di approssimazione, dopo aver accennato ad alcuni concetti preliminari legati alla complessità. Si defineranno in particolari classi di problemi, e si definiranno alcune tecniche note nella risoluzione di problemi di ottimizzazione.

1.1 Classi di complessità

Partiamo dalla definizione di algoritmo, per poi arrivare a definire la complessità algoritmica e strutturale.

Algoritmo Un algoritmo per un problema Π può essere visto come una *balck-box* che verrà indicata con A, che opera come segue:
dato un input $x \in I_\Pi$, l'algoritmo A produrrà un output $y \in O_\Pi$, tale che $y \in Sol_\Pi(x)$.

1.1.1 Complessità algoritmica

La complessità algoritmica è lo studio del dispendio di risorse di un algoritmo. Un esempio è il tempo: $T_a : I_\Pi \to \mathbb{N}$, possiamo passare in una notazione $t_a : \mathbb{N} \to \mathbb{N}$ dove il dominio è la lunghezza di input, in quel caso si avrà che $t_a(n) : \max\{T_a(n),\ x \in I_\Pi,\ |x| = n\}$

Date due soluzioni, è preferibile quella asintoticamente minima, più formalmente, quella a numeratore tale che $\lim_{x\to\infty} \frac{t_1}{t_2} = \infty$.

1.1.2 Complessità strutturale

Si definiscono ora due classi di problemi, P e NP, per poi introdurre i concetti di *riducibilità polinomiale* e di *NP-completezza.*

Classe P La classe P corrisponde ai problemi decisionali per cui esiste un algoritmo che opera in tempo polinomiale, ovvero:

$$P = \{\Pi \mid \Pi\ decisionale,\ \exists\ A\ per\ \Pi,\ t.c.\ t_a(n)\ =\ O(Polinomio)\}$$

Classe NP La classe NP corrisponde ai problemi decisionali per cui esiste un algoritmo che opera in tempo polinomiale su una macchina non deterministica, ovvero:

$$NP = \{\Pi \mid \Pi\ decisionale,\ \exists\ A\ per\ \Pi,\ t.c.\ t_a(n)\ =\ O(Polinomio)\ su\ una\ macchina\ non\ deterministica\}$$

Riducibilità polinomiale Un problema si dice riducibile polinomialmente se esiste un mapping del suo input in un input per un algoritmo polinomiale, ovvero:

$$\Pi_1 \leqslant_p \Pi_2 \; sse \; \exists f : 2^* \to 2^* \; t.c$$

1. f è calcolabile in tempo polinomiale
2. $\forall x \in I_{\Pi 1},\ f(x) \in I_{\Pi 2},\ Sol_{\Pi 1}(x) = Sol_{\Pi 2}(f(x))$

NP completezza Un problema Π è NP completo sse $\forall\ \Pi' \in NP,\ \Pi' \leqslant_p \Pi,\ \Pi \in NP$. Ovvero se ogni problema in NP è riducibile polinomialmente al problema che si sta considerando.

Teorema 1. *SAT è NP completo.*

Corollario 1. *Se* $\Pi_1 \leqslant_p \Pi_2$ *e* Π_1 *è NP completo, allora* Π_2 *è NP completo.*

Dimostrazione. $\Pi' \in NP, \Pi' \leqslant_p \Pi_1 \leqslant_p \Pi_2$, quindi Π_2 è NP completo. □

Osservazione 1. *Se trovassi un problema* Π *NP completo t.c,* $\Pi' \leqslant_p \Pi$*, allora* P=NP.

1.2 Problemi di ottimizzazione

Nella definizione di un problema di ottimizzazione bisogna tenere conto dei seguenti parametri:

1. Insieme di input I_Π
2. Insieme di output O_Π
3. $F_\Pi : I_\Pi \to 2^{O_\Pi} \setminus \{\emptyset\}$, $F_\Pi(x)$ indica le soluzioni accettabili per l'input x
4. $C_\Pi : I_\Pi \times O_\Pi \to \mathbb{Q}^{>0}$, funzione obiettivo, con $C_\Pi(x, y)$ si indica il valore della funzione obiettivo per l'input x, con soluzione $y \in O_\Pi$
5. $t_\Pi \in \{min,\ max\}$, ovvero il criterio del problema.

Osservazione 2. $Sol(x) = \{y^* \in O_\Pi | y^* \in F_\Pi, \forall y' \in F_\Pi, c(x, y^*) \leqslant (\geq) c(x, y')\}$

Problema di decisione associato Dato un problema di ottimizzazione Π esiste un problema di decisione associato $\hat{\Pi}$.
L'idea è quella di considerare un input del problema originale e un costo alla soluzione, e rispondere in base all'esistenza di una soluzione con quel costo.
In particolare:

$$I_{\hat{\Pi}} = I_\Pi \times \mathbb{Q}^{>0}$$

$$(x, \theta) = C_\Pi(x, y^*(x)) \leqslant (\geq)\theta$$

1.2.1 Classi di ottimizzazione

Data una definizione di problema di ottimizzazione e individuato il problema di decisione assocciato, si passa ora alla definizione delle classi di problemi di ottimizzazione, *PO* e *NPO*.

PO La classe PO equivale, nel mondo dei problemi di ottimizzazione, alla classe P, più formalmente:

$$PO = \{\Pi \mid \Pi \textit{ di ottimizzazione e polinomiale}\}$$

NPO Fanno parte della classe NPO quei problemi di ottimizzazione non risolvibili in tempo polinomiale, o meglio:

1. $I_\Pi, O_\Pi \in P$
2. Esiste un polinomio Q tale che:
   - $\forall x \in I_\Pi, \forall y \in F_\Pi, |y| \leq Q(|x|)$
   - $\forall x \in I_\Pi, \forall y \in 2^*, se |y| \leq Q(|x|)$ decidere se $t \in F_\Pi$ è polinomiale
3. c_π è calcolabile in tempo polinomiale

L'idea consiste poi nel generare tutte le soluzioni ammissibili, su una macchina non deterministica. La valutazione delle soluzioni avviene poi in tempo polinomiale, per i vincoli espressi sopra.

Teorema 2. *$PO \subseteq NPO$*

Teorema 3. *Se $\Pi \in PO, \hat{\Pi} \in P$, se $\Pi \in NPO, \hat{\Pi} \in NP$*

Dimostrazione. Dato un problema $\Pi \in P, \exists A$ polinomiale, tale che

$$x \in I_\Pi \longrightarrow A \longrightarrow y^*(x)$$

Dato il problema di decisione associato, esiste $\hat{A}$ tale che:

$$(x, \theta) \in I_\Pi \times \mathbb{Q}^{>0} \longrightarrow \hat{A} \longrightarrow yes/no$$

L'output sarà yes sse $c^*(x) \geq (\leq)\theta$, con $c^*(x) = c_\Pi(x, y^*(x))$.

L'algoritmo $\hat{A}$ sarà polinomiale, infatti, basta applicare A ad x per individuare $y^*(x)$ per poi confrontarlo con θ.

Similmente, nel caso in cui il problema appartenga a *NP*, sia A che $\hat{A}$ saranno non deterministiche. □

NPO completezza Un problema si dice NPO completo sse il problema di decisione associato è NP completo.

$$NPOcomp = \{\Pi \in NPO | \hat{\Pi} \in mathitNP\}$$

Teorema 4. *Se* $\Pi \in NPOcomp$, $\Pi \notin PO$, *a meno che* $P = NP$

Dimostrazione. Supponiamo di avere $\Pi \in NPOcomp$ tale che $\Pi \in PO$.
Allora avremmo che per la prima affermazione $\hat{\Pi} \in NPcomp$ e per la seconda, $\hat{\Pi} \in P$, assurdo. □

Rapporto di approssimazione Dato Π problema di ottimizzazione e siano $x \in I_\Pi, y \in F_\Pi(x)$ definiamo rapporto di approssimazione:

$$R_\Pi(x, y) = \max \left\{ \frac{c(x,y)}{c^*(x)}, \frac{c^*(x)}{c(x,y)} \right\} \geq 1$$

In un problema di massimo dominerà il secondo termine, mentre in uno di minimo il primo, si può perciò esprimere il rapporto senza dipendere dal tipo di problema.

APX I problemi in APX hanno rapporto di approssimazione limitato da una certa quantità, formalmente:

$$\begin{aligned} APX = \{\Pi | \Pi \textit{ di ottimizzazione } t.c.\ \exists \rho \geq 1, A, \\ t.c\ x \to A \to y(x) \textit{ con } R_\Pi(x,y) \leq \rho\} \end{aligned}$$

La definizione di classe APX permette una stratificazione per ρ, si fa notare poi che APX con valore 1 equivale alla classe PO.

PTAS I problemi in PTAS hanno rapporto di approssimazione limitato da una certa quantità, che si può decidere in modo arbitrario, formalmente:

$$\begin{aligned} PTAS = \{\Pi | \Pi \textit{ di ottimizzazione } t.c.\ \exists A, \\ t.c\ (x,r) \in I_\Pi \times \mathbb{Q}^{>1} \to A \to y \\ y \in F_\Pi, R_\Pi(x,y) \leq r, \\ \textit{polinomiale in } |x|\} \end{aligned}$$

Si fa notare che r può essere molto vicino ad 1, ma il tempo polinomiale potrebbe aumentare esponenzialmente.

1.2.2 BiMaxMatching

Si presenta ora un problema della classe PO, ovvero un problema di ottimizzazione per cui essite un algoritmo esatto in tempo polinomiale.

Input: grafo non orientato $G = (V, E)$
Output: matching $M \subseteq E$ tale che $\forall x\ \exists!\ xy \in M$

Costo: $|M|$
Tipo: max

A seguire l'algoritmo esatto polinomiale per i grafi bipartiti, esiste anche per grafi generici.

Cammino aumentante per M Nel grafo in cui si sta cercando il matching esistono due tipi di vertici:

1. Occupati: incide un lato di M
2. Liberi: non usati nel matching

Un cammino aumentante è un cammino semplice che:

1. Parte e arriva su un vertice libero
2. Alterna lati che appartengono a M e ad $E \setminus M$

Teorema 5. *Se $\exists$ un cammino aumentante per M, M non è massimo.*

Dimostrazione. Individuato un cammino aumentante, ci saranno lati che appartengono al matching, X e lati che non ci appartengono, Y. Questi ultimi sono in quantità maggiore per la definizione di cammino aumentante. Posso quindi rimuovere da M i lati in X e aggiungere quelli in Y. □

Teorema 6. *Se M non è massimo, esiste un cammino aumentante.*

Dimostrazione. Se M non è massimo significa che $\exists M', |M'| \geq |M|$.
Sia $X = M' \Delta M = (M' \setminus M) \cup (M \setminus M')$

Si osserva che, su ogni vertice incidono al massimo due lati di X, poichè nei due matching c'è al massimo un lato che indice su ogni vertice.

Rappresentando graficamente X si noterebbero solo cammini cicli o nodi singoli, visto che ogni nodo ha grado 0,1 oppure 2.

Considerando un ciclo in X si nota che:

- due lati consecutivi fanno parte di matching differenti, altrimenti avrei più lati incidenti su uno stesso vertice per matching
- i cicli hanno lunghezza pari

Si nota poi che:

$$|M'| \geq |M| \implies (M' \setminus M) \geq (M \setminus M')$$

quindi, dato che i cicli in X hanno in egual quantità lati di M' e M, deve esitere un cammino. Tale cammino avrà per forza più lati in M' che M, inoltre i lati si alternano tra i due matching e iniziano e finiscono con lati che non appartengono a M, quindi è un cammino aumentante per M. □

Teorema 7. *Siano $G = (V, E)$ un grafo bipartito e sia $M \subseteq E$ un suo matching, sono equivalenti:*

1. *M è massimo*
2. *Non esiste un cammino aumentante per M*

Algoritmo di risoluzione L'algoritmo di risoluzione è abbastanza semplice e si basa sulla ricerca di un cammino aumentante.

Algorithm 1: BiMaxMatching

Input: $G = (V, E)$
Result: Matching M per G
$M \leftarrow \emptyset$
while $\Pi = findAugmenting(G)$ **do**
  $M.update(\Pi)$
return M

Siano v_1 e v_2 le parti destra e sinistra rispettivamente di un grafo bipartito. Un'idea per sviluppare la funzione di cammino aumentante su grafi bipartiti è quella di considerare solo archi che non appartengono a M mentre si va da v_1 a v_2 e solo quelli che appartengono a M nella direzione opposta. Se trovo un cammino del genere aggiorno M.

Osservazione 3. *Il problema può anche essere risolto con una rete di flusso, aggiungendo un nodo sorgente collegato alla partizione sinistra dei nodi e un nodo pozzo alla partizione destra. Si applica poi uno dei tanti algoritmi per il flusso.*

Osservazione 4. *Una variante del problema è quella del perfect matching, che capita quando la cardinalità di v_1 e v_2 equivale a $\frac{n}{2}$. Successivamente si verifica se $M = BiMaxMatching(G)$ ha cardinalità $\frac{n}{2}$.*

1.3 Tecniche greedy

Nella sezione a seguire si presentano problemi di ottimizzazione per cui tecniche greedy funzionano abbastanza bene. I problemi affrontati sono quelli di *Load Balancing*, *Center Selection* e *Set Cover*.

1.3.1 Load balancing

Il problema di Load Balancing può essere visto come il compito di assegnare a macchine dei lavori da compiere, che richiedono del tempo, in modo da minimizzare il tempo totale.

Input: $t_0, \ldots, t_n \in \mathbb{N}^{>0}$ task, $m \in \mathbb{N}^{>0}$ macchine
Output: $\alpha : n \to m$
Costo: $L = \max_{i \in n}(L_i)$, $L_i = \sum_{j \in \alpha^{-1}(i)} t_j$
Tipo: min

Teorema 8. *Load Balancing è NPO completo*

Bisogna perciò trovare un modo di approssimare una soluzione.

Greedy balance Il primo approccio alla risoluzione del problema è quello di assegnare la prossima task alla macchina più scarica in questo momento.

Algorithm 2: GreedyBalance

Input: M numero di macchine, $t_0, \ldots, t_n$ task
Result: Assegnamento delle task alle macchine
$L_i \leftarrow 0 \; \forall i \in M$
$\alpha \leftarrow \emptyset$
for $j = 0, \ldots, n$ **do**
  $\hat{i} = \min(L_i)$
  $\alpha(j) = \hat{i}$
  $L_{\hat{i}} \mathrel{+}= t_j$
return α

Il tempo è polinomiale, si ottiene una complessità $O(nm)$, implementando la ricerca del minimo con coda di priorità si ottiene $O(n \log(m))$.

Teorema 9. *Greedy balance è 2-approssimante per load balancing, ovvero la soluzione individuata è al massimo il doppio di quella ottima.*

Dimostrazione. Sia L^* il costo della soluzione ottima, il suo costo non è minore della somma delle task diviso il numero di macchine:

$$L^* \geq \frac{1}{m} \sum_{i \in n} t_i$$

Infatti, sommando i carichi, si ottiene la somma dei task:

$$\sum_{i \in m} L_i^* = \sum_{i \in n} t_i$$

$$\frac{1}{m} \sum_{i \in m} L_i^* = \frac{1}{m} \sum_{i \in n} t_i \implies \exists L_i^* \geq \frac{1}{m} \sum_{i \in n} t_i \implies L^* \geq \frac{1}{m} \sum_{i \in n} t_i$$

Si osserva poi che:

$$L^* \geq \max_{i \in n}(t_i)$$

Sia ora L la soluzione individuata da Greedy Balance, e $\hat{i}$ la macchina con carico massimo $L_{\hat{i}} = L$.
Sia $\hat{j}$ l'ultimo task assegnato a $\hat{i}$. Il carico che la macchina aveva prima dell'ultimo task era:

$$L'_{\hat{i}} = L_{\hat{i}} - t_{\hat{j}}$$

La scelta greedy dell'algoritmo implica che:

$$L_{\hat{i}} - t_{\hat{j}} \leq L'_i \forall i \implies L_{\hat{i}} - t_{\hat{j}} \leq L_i \forall i$$

Dove L'_i indica il carico della macchina i prima di assegnare $\hat{j}$. Ottengo m disequazioni che posso sommare tra loro:

$$\begin{aligned} m(L_{\hat{i}} - t_{\hat{j}}) &\leq \sum_{i \in m} L_i = \sum_{i \in n} t_i && \text{divido per m} \\ L_{\hat{i}} - t_{\hat{j}} &\leq \frac{1}{m} \sum_{i \in n} t_i \leq L^* && \text{dall'osservazione iniziale} \\ L = L_{\hat{i}} &= (L_{\hat{i}} - t_{\hat{j}}) + t_{\hat{j}} && \text{uso la riga sopra e la seconda osservazione} \\ L &\leq L^* + L^* \leq 2L^* \end{aligned}$$

□

Corollario 2. *Load balancing appartiene a APX*

Teorema 10. $\forall \epsilon > 0$ *esiste un input su cui Greedy Balance produce una soluzione L t.c.*

$$L - \epsilon \leq \frac{L}{L^*} \leq 2$$

Dimostrazione. Si considerano $m > \frac{1}{\epsilon}$ macchine e $n = m(m-1) + 1$ task.
Tutti i task ad eccezione dell'ultimo hanno lunghezza 1, mentre l'ulitmo ha lunghezza m.

L'assegnamento di Greedy Balance assegna tutte le task grandi 1 a tutte le macchine, infine si assegna l'ulitma task lunga m a una macchina causale, visto che sono tutte piene allo stesso modo. Il tempo totale è $L = 2m - 1$.

Esiste però un assegnamento migliore, ovvero, alla prima macchina si assegna la task lunga m, mentre alle altre tutte quelle lunghe 1 in modo uniforme. In questo caso si ottiene $L^* = m$.

$$\frac{L}{L^*} = \frac{2m-1}{2} = 2 - \frac{1}{m} \geq 2 - \epsilon$$

□

Sorted Balance L'algoritmo dapprima ordina i task in ordine inverso di lunghezza e poi effettua la scelta greedy vista in precendeza.

Algorithm 3: SortedBalance

Input: M numero di macchine, $t_0, \ldots, t_n$ task
Result: Assegnamento delle task alle macchine
$tasks \leftarrow rev(sorted(t_i, \ldots t_n))$
return $GreedyBalance(M, tasks)$

Teorema 11. *Sorted Balance fornisce una* $\frac{3}{2}$*-approssimazione a Load Balancing*

Dimostrazione. Se $N \leq M$ l'algoritmo trova la soluzione ottima, si considera quindi il caso di avere più task che macchine.

Si osserva che:

$$L^* \geq 2t_m$$

questo vale poichè dei primi $m+1$ task almeno 2 sono assegnati ad una stessa macchina (principio delle camicie e dei cassetti) e quei due task sono $\geq t_m$ (task ordinati in ordine decrescente).

Sia poi $\hat{i}$ la macchina tale che $L_{\hat{i}} = L$ e sia $\hat{j}$ l'ultimo task assegnato alla macchina $\hat{i}$[1]. Si osserva che:

$$\begin{aligned} \hat{j} &\geq m \\ t_{\hat{j}} \leq t_m \leq \frac{1}{2}L^* & \qquad \text{dall'osservazione iniziale} \\ L = L_{\hat{i}} = (L_{\hat{i}} - t_{\hat{j}}) + t_{\hat{j}} & \quad \text{vale la dimostrazione precedente} \\ L \leq L^* + \frac{1}{2}L^* \leq \frac{3}{2}L^* & \end{aligned}$$

□

Osservazione 5. *In realtà Sorted Balance è $\frac{4}{3}$-approssimante*

Osservazione 6. *Load Balancing $\in$ PTAS*

1.3.2 Center selection

Il problema della selezione dei centri può essere visto come il compito di eleggere alcune città come centri in un insieme di città.
L'obiettivo è quello di minimizzare la distanza della città più sfortunata dal proprio centro. Si sta lavorando in uno spazio metrico.

Input: $S \subseteq \Omega$ dove (Ω, d) è uno spazio metrico, K centri da selezionare
Output: $C \subseteq S$, $|C| \leq K$, $C : S \longrightarrow C$ funzione che manda un punto in S nel centro che minimizza la distanza da esso
Costo: $\rho(C) = \max_{s \in S} d(s, C(s))$
Tipo: min

Definizione 1. *Uno spazio si dice metrico se esiste il concetto di distanza in esso. Una funzione di distanza è:*

$$d : \Omega \times \Omega \longrightarrow \mathbb{R}$$

E rispetta le seguenti proprietà

- $d(x, y) \geq 0$, $d(x, y) = 0 \implies x = y$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(z, y)$

[1] La macchina ha almeno 2 task, altrimenti abbiamo trovato la soluzione ottima

Center selection plus L'algoritmo che segue ha un input aggiuntivo r, idealmente dovrebbe equivalere a ρ^*, ovvero la soluzione ottima.

Algorithm 4: CenterSelectionPlus

Input: $S \subseteq \Omega$, $K \in \mathbb{N}^{>0}$, $r \in \mathbb{R}^{>0}$
Result: Selezione dei centri
$C \leftarrow \emptyset$
while $S \neq \emptyset$ **do**
  $\bar{s} = random(S)$
  $C.add(\bar{s})$
  forall $e \in S$ **do**
    if $d(e, \bar{s}) \leq 2r$ **then**
      $S.pop(e)$
return $(|C| \leq K)?\ C\ :\ impossible$

Teorema 12. *Considerando l'algoritmo proposto, valgono:*

1. *se l'algoritmo emette un C, l'approssimazione è $\leq \frac{2r}{\rho^*}$*
2. *se $r \geq \rho^*$, l'algoritmo emette un output*
3. *se il risultato è impossibile, allora $r \leq \rho^*$*

Dimostrazione. Partendo dal punto 1, si osserva che: $\forall s \in S$, la cancellazione di s implica che la sua distanza da uno dei centri fosse $\leq 2r$, segue che, ogni punto è al massimo distante $2r$ da un centro.

$$\rho(C) \leq 2r \qquad \text{per il discorso precedente}$$
$$\frac{\rho(C)}{\rho^*(C)} \leq 2r \quad \text{rapporto di approssimazione}$$

Per il punto 2, si considera un elemento $\bar{s}$ appena aggiunto a C. Nella soluzione ottima, esso si rivolge a qualche centro $C^*(\bar{s})$. Si considerano ora i punti che si riferiscono a quel centro:

$$X = \{s \in S | C^*(\bar{s}) = C^*(s)\}$$
$$\begin{aligned} \forall s \in X d(s, \bar{s}) &\leq d(s, C^*(s)) + d(C^*(s), \bar{s}) \quad \text{triangolare} \\ &\leq d(s, C^*(s)) + d(C^*(\bar{s}), \bar{s}) \\ &\leq \rho^* + \rho^* = 2\rho^* \leq 2r \end{aligned}$$

Questo implica che dopo aver inserito $\bar{s}$ tutti i punti in X sono eliminati.
Visto che $|C^*| \leq K$, dopo K iterazioni l'insieme S sarà vuoto.

Il punto 3 segue dalla dimostrazione del punto 2, $a \implies b, !a \implies !b$ □

Osservazione 7. *La dimostrazione fornisce un'idea per l'algoritmo, si potrebbe sfruttare una ricerca binaria su r.*

Greedy Center Selection L'idea dell'algoritmo è prendere un punto causale, e di volta in volta aggiungere a C il punto piu sfortunato, ovvero quello con distanza dai centri massima.

Algorithm 5: GreedyCenterSelection

Input: $S \subseteq \Omega$, $K \in \mathbb{N}^{>0}$
Result: Selezione dei centri
if $|S| \leq K$ **then**
  return S
$s = random(S)$
$C = \{s\}$
while $|C| < K$ **do**
  $\bar{s} = \max_{d(s,C)}(S)$
  $C.add(\bar{s})$
return $(|C| \leq K)?\ C\ :\ impossible$

Teorema 13. *Greedy Center Selection è 2 approssimante per Center Selection*

Dimostrazione. Supponiamo esista un input (S, K) tale che $\rho(c) > 2\rho^*$. Questo implica che $\exists \hat{s} \in S$ tale che $d(\hat{s}, C) > 2\rho^*$

Nelle prime K iterazioni dell'algoritmo:

$$\begin{aligned} \bar{s}_1, \ldots, \bar{s}_k &\quad \text{centri aggiunti} \\ \bar{C}_1, \ldots, \bar{C}_k &\quad \text{centri prima di aggiungere } s_i \end{aligned}$$

Considerando la distanza del centro s_i:

$$\begin{aligned} d(\bar{s}_i, \bar{C}_i) &\geq d(\hat{s}, \bar{C}_i) \quad \text{per massimizzazione} \\ d(\hat{s}, \bar{C}_i) &\leq d(\hat{s}, C) \leq 2\rho^* \end{aligned}$$

Questo caso coincide con le prime k iterazioni di Center Selection Plus con $r = \rho^*$. Ma quindi avrei $S \neq \emptyset$, quindi output impossibile e $r < \rho^*$ □

Teorema 14. *Non esiste* $\Pi \in P$ *α-approssimante per Center Selection con* $\alpha < 2$

Dimostrazione. Riduco a Dominating set che appartiene ad NP. Il problema individua un insieme di nodi in un grafo tale che, ogni nodo o fa parte di tale insieme, o è vicino di un nodo che ne fa parte.

Per tale problema:
Input: $G = (V, E)$, $K \in \mathbb{N}^{>0}$
Output: $\exists D \subseteq V$ tale che $|D| \leq K$ tale che $\forall x \in V \setminus D, \exists y \in D, xy \in E$

Considerando l'input di Dominating Set, i vertici costituiscono i punti di Center Selection. Il mapping per le distanze avviene come segue:

$$d(x, y) = \begin{cases} 0 & se\ x = y \\ 1 & se\ xy \in E \\ 2 & se\ xy \notin E \end{cases}$$

La distanza così definita è simmetrica e rispetta la disuguaglianza triangolare:

$$\begin{aligned} & x, y, z \in S \\ d(x,y) \leq d(x,z) + d(z,y) & \quad \text{le distanze sono 1,2, o 3} \\ 1,2 \leq 2,3,4 & \quad \text{1 oppure 2 minore di 2, 3 o 4} \end{aligned}$$

Definita la distanza, supponiamo ora di conoscere $\rho^*(S,K)$ che vale 1 oppure 2. Se $\rho^*(S,K) = 1$, allora:

$$\exists C \subseteq S, |C| \leq K, \ t.c., \ \forall x \in S \setminus C, \exists c \in C, d(x,y) = 1$$

Ovvero, se esiste un C tale che ogni elemento al di fuori dei centri ha distanza 1 da uno dei punti in C. Se due punti x, y hanno distanza 1, $xy \in E$.
In breve, $\rho^*(S,K) = 1$ sse esiste una soluzione per Dominating Set.

Per assurdo supponiamo esista un algoritmo Π che α-approssima Center Selection con $\alpha < 2$.

$$(G,K) \longrightarrow (S,K) \longrightarrow \Pi \longrightarrow \rho(S,K)$$

La soluzione individuata da Π sarà:

$$\rho^*(S,K) \leq \rho(S,K) < 2\rho^*(S,K)$$

Esistono due casi:

- $\rho^*(S,K) = 1$, allora $1 \leq \rho(S,K) < 2$, ovvero esiste una soluzione per Dominating Set
- $\rho^*(S,K) = 2$, allora $2 \leq \rho(S,K) < 4$, quindi non esiste una soluzione per Dominating Set

Questo significa che, se quell'algoritmo Π esisitesse, riuscirei a decidere, guardando il risultato $\rho(S,K)$, Dominating Set, il tutto in tempo polinomiale, assurdo. □

1.4 Tecniche di pricing

L'idea è quella di attribuire ad ogni elemento da inserire in una soluzione un costo. I costi permettono di scegliere gli elementi più vantaggiosi e di analizzare il tasso di approssimazione di questi algoritmi.

1.4.1 Minimum Set Cover

Nel problema di Set Cover l'obiettivo è quello di coprire l'universo U, utilizzando seubset di esso che hanno un costo. Si vuole ottenere il costo minimo, dato come somma dei costi dei set che si sono scelti.

Formalmente:
Input: $s_1, \ldots, s_m$, $\bigcup_{i=1}^{m} s_i = U$, $|U| = n$

Output: $C = \{s_1, \ldots, s_n\}$, tali che, $\bigcup_{s_i \in C} s_i = U$
Costo: $w = \sum_{s_i \in C} w_i$
Tipo: min

Funzione armonica La funzione armonica è definita come:

$$H : \mathbb{N}^{>0} \to \mathbb{R}$$
$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

Vale la seguente proprietà:

$$\ln(n+1) \leq H(n) \leq 1 + \ln n$$

Greedy Set Cover L'algoritmo effettua ad ogni iterazione una scelta greedy, si sceglie l'insieme che minimizza il rapporto tra prezzo e copertura dell'universo.

Algorithm 6: GreedySetCover

Input: $s_1, \ldots, s_m$, $w_0, \ldots, w_n$
Result: Scleta di sottoinsiemi che copre l'universo
$R \leftarrow U$
$C \leftarrow \emptyset$
while $R \neq \emptyset$ **do**
  $S_i \leftarrow \min(s_1, \ldots, s_m, \frac{w_i}{|S \cap R|})$
  $C.add(S_i)$
  $R \leftarrow R \setminus S_i$
return C

Osservazione 8. *Il costo della soluzione equivale a*

$$w = \sum_{s \in U} c_s$$

ovvero la somma dei costi degli insiemi scelti.

Osservazione 9. *Per ogni k, il costo degli elementi in s_k, ottengo*

$$\sum_{s \in S_k} C_s \leq H(|S_k|) \cdot W_k$$

Dimostrazione. Sia $S_k = \{s_1, \ldots, s_d\}$ un insieme tra quelli da scegliere, e siano i suoi elementi elencati in ordine di copertura[2].

Consideriamo ora l'istante in cui si copre S_h tramite un quale insieme S_h. Si può notare che, visto che gli elementi sono in ordine di copertura:

$$R \supseteq \{S_j, \ldots, S_d\}$$

[2]Per chiarezza, l'insieme S_k non verrà scelto, ma i suoi elementi saranno coperti da altri insiemi che intersecano con esso.

Inoltre, visto che gli elementi di S_k sono in ordine di copertura:

$$|S_k \cap R| \geq d - j + 1$$

Riguardo al costo dell'elemento j, e in geenerale per tutti i j, vale:

$$\begin{aligned} C_{s_j} = \frac{W_h}{|S_h \cap R|} &\leq \frac{W_k}{|S_k \cap R|} \quad h \text{ minimizza quel rapporto} \\ &\leq \frac{W_k}{d - j + 1} \qquad \text{equazione precedente} \end{aligned}$$

Considerando ora tutti gli elementi di S_k:

$$\sum_{s \in S_k} C_s \leq \sum_{j=1}^{d} \frac{W_k}{d - j + 1} = \frac{W_k}{d} + \frac{W_k}{d-1} \dots \qquad \text{La relazione vale per tutti i j}$$

$$= W_k(1 + \frac{1}{2}, \dots, \frac{1}{d}) = H(d)W_k = H(|S_k|)W_k \quad \text{Sviluppo e raccolgo, ottengo l'oss.}$$

□

Teorema 15. *Greedy Set Cover fornisce una $H(M)$-approssimazione per Set Cover, dove $M = \max_i |S_i|$*

Dimostrazione. Sia il peso della soluzione ottima

$$w^* = \sum_{S_i \in C^*} w_i$$

Per l'osservazione 9 vale che:

$$w_i \geq \frac{\sum_{s \in S_i} C_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_i} C_s}{H(M)}$$

Visto che gli $s_i \in C^*$ sono una copertura, per l'osservazione 8:

$$\sum_{S_i \in C^*} \sum_{s \in S_i} C_s \geq \sum_{s \in U} C_s = w$$

Inoltre, vale che, sfruttando le due disequazioni appena scritte:

$$\begin{aligned} w^* = \sum_{S_i \in C^*} w_i \geq \sum_{S_i \in C^*} \frac{\sum_{s \in S_i} C_s}{H(M)} &\geq \frac{w}{H(M)} \\ \implies \frac{w}{w^*} &= H(M) \end{aligned}$$

□

Corollario 3. *Greedy Set Cover fornisce una $O(\log n)$-approssimazione, non quindi costante come gli algoritmi precedenti.*

Osservazione 10. *L'analisi è tight, non esiste un algoritmo migliore, quindi Greedy Set Cover $\notin$ APX, bensì, $\in \log(n)$-APX, una classe in cui si accetta un'approssimazione che peggiora logaritmicamente nell'input. Esistono varie f-APX.*

Dimostrazione. Per dimostrare la tightness, ecco un esempio in cui Greedy Set Cover va male.

Consideriamo l'insieme S di set tra cui scegliere così formato:

1. Due insiemi grandi $\frac{n}{2}$ che uniti coprono tutti gli elementi, di costo $1+\epsilon$
2. Un insieme che copre $\frac{n}{4}$ elementi degli insiemi 1 e 2 al punto 1, di costo 1
3. Un insieme che copre $\frac{n}{8}$ elementi degli insiemi 1 e 2 al punto 1, di costo 1
4. Un insieme che copre $\frac{n}{16}$ elementi degli insiemi 1 e 2 al punto 1, di costo 1

   ...

Al primo passo, si sceglie l'insieme del punto 2, visto che il suo costo equivale a $\frac{1}{\frac{n}{2}} = \frac{2}{n}$, mentre il costo di entrambi gli insimemi al punto 1, $\frac{1+\epsilon}{\frac{n}{2}} = \frac{2+2\epsilon}{n}$

Al secondo passo, si preferisce ai primi due l'insieme al punto 3, il calcolo è simile al punto precedente.
...

Il costo che si ottiene è $w = \log n$ La soluzione ottima sarebbe quella di prendere al passo 1 i primi due insiemi, in modo da coprire l'intero universo, ovvero $w^* = 2 + 2\epsilon$. □

1.4.2 Vertex Cover

Il problema di Vertex Cover consiste nel trovare una copertura di vertici in un grafo tale per cui per ogni vertice, una delle due estremità è contenuta nella copertura.

Formalmente:
Input: $G = (V, E), w_i \in \mathbb{Q}^{>0}, \forall i \in V$
Output: $X \subseteq V, \forall xy \in E, x \in X \vee y \in X$
Costo: $w = \sum_{i \in X} w_i$
Tipo: min

Consideriamo la versione di decisione del problema, ovvero $\hat{Vertex Cover}$, cioè, posso trovare una copertura che ha peso minore di θ?

Vale che $\hat{Vertex Cover} \leq_p \hat{Set Cover}$ Per effettuare il passaggio, bisogna passare dall'input del primo al secondo

$$G = (V, E), w_i \in \mathbb{Q}^{>0}, \theta \longrightarrow f \longrightarrow S_1, \ldots, S_m, \bar{W}_1, \ldots, \bar{W}_m, \bar{\theta}$$

La funzione f funziona così:

$$S_i = \{e \in E, i \in e\} \quad \text{Per ogni vertice considero i suoi vicini}$$
$$U = E, \bar{W}_i = W_i, \bar{\theta} = \theta$$

Osservazione 11. *La funzione f può essere utilizzata per mappare anche VertexCover in SetCover di ottimizzazione, visto che non stravolge il problema. La trasformazione appena discussa quindi può essere utilizzata per fornire una $\log n$-approssimazione per VertexCover di ottimizzazione.*

Price Vertex Cover Prima di definire l'algoritmo vero e proprio, sono necessari alcuni passaggi preliminari. L'idea si basa sul fatto che gli archi sono intenzionati a comprare uno dei due estremi, ad un certo prezzo. Un nodo si vende, se la somma delle offerte degli archi incidenti arriva al suo W_i.

Definizione 2. *Un insieme di offerte si dice equo per un vertice sse:*

$$\sum_{e \in E, i \in e} P_e \leq W_i$$

Ovvero se le offerte P_e degli archi che incidono sul vertice i non superano il suo costo, ovviamente devono raggiungerlo per comprare il vertice.

Lemma 1. *Se P_e è equo, allora*

$$\sum_{e \in E} P_e \leq w^*$$

Dimostrazione. La definizione di equità implica che

$$\forall i \in V \sum_{e \in E, i \in e} P_e \leq W_i$$

Consideriamo ora la somma delle disequazioni per la soluzione ottima

$$\sum_{i \in S^*} \sum_{e \in E, i \in e} P_e \leq \sum_{i \in S^*} W_i$$

$$\sum_{e \in E} P_e \leq \sum_{i \in S^*} \sum_{e \in E, i \in e} P_e \leq W^*$$

Questo vale perchè la seconda parte della disequazione considera potenzialmente più volte alcuni lati. □

Definizione 3. *P_e è stretto sul vertice i sse*

$$\sum_{e \in E, i \in e} P_e = W_i$$

Ecco ora l'algoritmo.

Algorithm 7: PriceVertexCover

Input: $G = (V, E), W_i \forall i \in V$
Result: Cover per il grafo
$P_e \leftarrow 0, \forall e \in E$
while $\exists ij \in E, P_e$ *non stretto su i o su j* **do**
  $\bar{e} \leftarrow \bar{ij}$
  $\Delta \leftarrow \min(W_{\bar{i}} - \sum_{e \in E, \bar{i} \in e} P_e, W_{\bar{j}} - \sum_{e \in E, \bar{j} \in e} P_e,)$
  $P_e \leftarrow P_e + \Delta$
return $\{i | P_e$ *stretto su* $i\}$

L'idea su cui si basa l'algoritmo è quella che, se un arco non sta offrendo abbastanza per i vertici, allora aumenta la sua offerta, del minimo per soddisfare uno dei due nodi.

Lemma 2. *Price Set Cover crea un peso*

$$W \leq 2 \sum_{e \in E} P_e$$

Dimostrazione. Il costo finale di PSC è $W = \sum_{i \in S} W_i$, ovvero il costo dei vertici in output. Vale che, per la definizione di strettezza:

$$W = \sum_{i \in S} W_i = \sum_{i \in S^*} \sum_{e \in E, i \in e} P_e$$

Nella seconda sommatoria un lato compare una o due volte, se rispettivamente una o due estremità appartengono ad S. □

Teorema 16. *Price Set Cover è 2-approssimante per Set Cover.*

Dimostrazione.

$$\frac{w}{w^*} \leq \frac{2 \sum_{e \in E} P_e}{w^*} \quad \text{Per il lemma 2}$$
$$\frac{w}{w^*} \leq \frac{2 \sum_{e \in E} P_e}{\sum_{e \in E} P_e} \leq 2 \quad \text{Per il lemma 1}$$

□

Osservazione 12. *Non esistono ad oggi algoritmi migliori per Price Set Cover.*