Algoritmi e complessità

Francesco Tomaselli

18 marzo 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:

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

□

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.

$$\begin{aligned} \rho(C) &\leq 2r && \text{per il discorso precedente} \\ \frac{\rho(C)}{\rho^*(C)} &\leq 2r && \text{rapporto di approssimazione} \end{aligned}$$

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:

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

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.*

1.4.3 Disjoint Paths

Dato un grafo orientato, esistono K sorgenti e target, l'obiettivo è trovare K cammini per le coppie di sorgenti e destinazioni. Ogni arco può essere usato al più C volte.

Formalmente:
Input: $G = (V, E)$ orientato, $s_1, \ldots, s_k$ sorgenti, $t_1, \ldots, t_k$ destinazioni $\in V$
$C \in \mathbb{N}^{>0}$
Output: $I = \{1, \ldots, K\}, \forall i \in I$, cammino Π_i, tale che nessun arco è utilizzato più di C volte
Funzione obiettivo: $|I|$
Tipo: max

L'idea su cui si basa l'algoritmo è quella di allungare un arco tanto quanto è utilizzato, in modo da scoraggiare il suo utilizzo.
È quindi definita una funzione di lunghezza:

$$l : E \longrightarrow \mathbb{R}^{>0}$$

Dato un cammino come sequenza di nodi, la lunghezza del cammino sarà data dalla lunghezza su ogni coppia di consecutivi.

Greedy Paths with capacity L'algoritmo considera come input, oltre a quello per Disjoint Paths, un parametro $\beta > 0$, che indica il fattore di allungamento degli archi.

Algorithm 8: GreedyDisjointPaths

Input: $G = (V, E), s_1, \ldots, s_k, t_1, \ldots, t_k, \beta > 0$
Result: Cammini da sorgenti a destinazioni
$I \leftarrow \emptyset$
$P \leftarrow 0$
$l(xy) \leftarrow 1, \forall xy \in E$
while $\Pi \leftarrow$ *shortest path from* s_i *to* t_i, $i \notin I$ **do**
  $I \leftarrow I \cup i$
  $P \leftarrow P \cup \Pi$
  for $e \in \Pi$ **do**
    $l(e) \leftarrow l(e) * \beta$ **if** e *used by* C *paths* **then**
      remove(e)
return I, P

Consideriamo ora l'evolversi dei cammini nell'algoritmo, in particolare un cammino può essere:

- Inutile, se unisce una sorgente e destinazione già collegate
- Utile ma ostruito, ovvero collega una sorgente e destinazione sconnesse, ma passa per un arco utlizzato da più di C cammini

Definizione 4. *Un cammino* Π *è corto rispetto a una certa lunghezza* l *sse* $l(\Pi) < \beta^c$*, lungo altrimenti.*

Osservazione 13. *Se sono presenti cammini utili non ostruiti e corti, l'algoritmo sceglie un cammino di questa natura, inoltre, nelle prime fasi dell'algoritmo, ovvero quando sono presenti cammini di questo tipo, si può evitare di eliminare archi.*

Dimostrazione. Se esite un cammino corto, la sua lunghezza è per forza minore di β^c quindi nessun arco è stato utilizzato C volte. □

Tenendo presente quanto detto nell'osservazione 13, teniamo ora in considerazione la funzione di lunghezza $\bar{l}$ nell'esatto istante in cui finiscono i cammini corti. Inoltre, $\bar{I}, \bar{P}$ saranno le sorgenti, destinazioni collegate e i rispettivi cammini in quell'istante.

Lemma 3. *Se* $i \in I^* \setminus I$*, allora* $\bar{l}(\Pi_i^*) \geq \beta^c$

Dimostrazione. Se valesse $\bar{l}(\Pi_i^*) < \beta^c$, il cammino Π_i^* sarebbe corto e utile, in quanto collega una coppia ancora non collegata, dato che non appartiene a I. Quindi, non esisterebbe motivo per non selezionarlo. □

Lemma 4. *Vale che* $\sum_{e \in E} \bar{l}(e) \leq \beta^{c+1}|\bar{I}| + m$.

Dimostrazione. All'inizio dell'algoritmo, $\sum_{e\in E} l(e) = m$. Consideriamo ora le due funzioni di lunghezza dopo la selezione di un cammino

$$l_1 \longrightarrow \Pi \longrightarrow l_2$$

$$l_2(e) = \begin{cases} \beta l_1(e) & se\ e \in \Pi \\ l_1(e) & se\ e \notin \Pi \end{cases}$$

Consideriamo ora la differenza delle lunghezze

$$\begin{aligned} &\sum_{e\in E} l_1(e) - \sum_{e\in E} l_2(e) = \sum_{e\in E}(l_1(e) - l_2(e)) \\ &= \sum_{e\in E}(l_1(e) - l_2(e)) = \sum_{e\in \Pi}(\beta l_1(e) - l_2(e)) \\ &= \sum_{e\in \Pi}(\beta - 1) l_1(e) = (\beta - 1)\sum_{e\in \Pi} l_1(e) \\ &= (\beta - 1) l(\Pi) \leq (\beta - 1)\beta^c \leq \beta^{c+1} \end{aligned}$$

Considerando l'inizio della dimostrazione, visto che ad ogni passo la lunghezza aumenta di al massimo β^{c+1}, il lemma vale. □

Osservazione 14. *Valgono:*

$$\sum_{i\in I^*\setminus I} \bar{l}(\Pi_i^*) \geq \beta^c |I^* \setminus I| \qquad \textit{dal lemma 3}$$

$$\sum_{i\in I^*\setminus I} \bar{l}(\Pi_i^*) \leq c \sum_{e\in E} \bar{l}(e) \leq c(\beta^{c+1}|\bar{I} + m|) \quad \textit{per il lemma 4}$$

Mettendo tutto insieme:

$$\begin{aligned} \beta^c|I^*| &= \beta^c|I^* \setminus I| + \beta^c|I^* \cap I| \\ &\leq \sum_{i\in I^*\setminus I} \bar{\Pi}_i^* + \beta^c && \text{per il punto 1 di oss. 14, e maggioro} \\ &\leq c(\beta^{c+1}|\bar{I}| + m) + \beta^c|I| && \text{per il punto 2 di oss. 14} \\ &\leq c(\beta^{c+1}|I| + m) + \beta^c|I| && \text{I è sovrainsieme per I barra} \\ |I^*| &\leq c\beta|I| + c\beta^{-c}m + |I| && \text{divido per beta elevato alla c} \\ |I^*| &\leq c\beta|I| + c\beta^{-c}m|I| + |I| && \text{moltiplico per la cardinalità quello in mezzo} \\ \frac{|I^*|}{|I|} &\leq c\beta + c\beta^{-c}m + 1 && \text{divido per cardinalità di I} \end{aligned}$$

Poniamo ora $\beta = m^{\frac{1}{c+1}}$ si ottiene:

$$\frac{|I^*|}{|I|} \leq c(m^{\frac{1}{c+1}} + m^{\frac{-c+c+1}{c+1}}) + 1$$

$$\frac{|I^*|}{|I|} \leq 2cm^{\frac{1}{c+1}} + 1$$

Al variare di C varia il grado di approssimazione dell'algoritmo.

C	Bound approssimazione
1	$2\sqrt[2]{m}+1$
2	$4\sqrt[3]{m}+1$
3	$6\sqrt[4]{m}+1$
...	...

Teorema 17. *Greedy Disjoint Paths con* $\beta = m^{\frac{1}{c+1}}$ *fornisce una* $(2cm^{\frac{1}{c+1}} + 1)$*-approssimazione.*

Osservazione 15. *L'algoritmo non è buono, il fattore di approssimazione è pessimo, infatti, funziona decentemente solo se il numero di coppie da collegare* $K >> 2\sqrt[2]{m}$.
Si fa notare infine che l'algoritmo funziona anche se una coppia è da collegare più volte.

1.5 Tecniche di arrotondamento

In questa sezione si presenta il problema di programmazione lineare e si ricorre alla tecnica dell'arrotondamento di una soluzione reale per individurarne una intera.

Essa si rivela possibile soluzione per il problema di copertura dei vertici.

1.5.1 Linear programming

Un problema di programmazione lineare è formato da una funzione obiettivo e dei vincoli.

Esempio Funzione obiettivo:

$$min \;\; 3x_1 + |7x_2 - 4x_3|$$

Vincoli:

$$\begin{cases} x_1 + x_2 \leq 3 \\ 3x_3 - x1 \geq 7 \\ x1 \geq 0 \\ x2 \geq 0 \\ x3 \geq 0 \end{cases}$$

Più formalmente:
Input: $A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$
Output: $x \in \mathbb{R}^n, Ax \geq b$
Funzione obiettivo: $C^T \cdot x$, valore della funzione obiettivo
Tipo: max, min

Osservazione 16. *Il problema di LP* $\in$ *PO, spesso però si utilizzano algoritmi worst case esponenziali.*

Integer linear programming Cambiando il vincolo della soluzione ai soli interi, il problema diventa NPO completo.

Ovvero:
Input: $A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$
Output: $x \in \mathbb{Z}^n, Ax \geq b$
Funzione obiettivo: $C^T \cdot x$, valore della funzione obiettivo
Tipo: max, min

Teorema 18. $\hat{Vertex Cover} \leq_p \hat{ILP}$

Dimostrazione. Consideriamo un input per il problema di $\hat{Vertex Cover}$, ovvero

$$(G = (E, V), w_i \forall i \in V, \theta)$$

Ci si chiede se:

$$\exists X \subseteq V | \forall xy \in E, x \in X \, or \, y \in X, \sum_{i \in X} w_i \leq \theta$$

Costruisco ora il problema di programmazione linerare in questo modo:

$$x_i \forall i \in V$$

Vincoli:

$$2(n+m) \text{ volte} \begin{cases} x_i \geq 0 \ \forall i \in V \\ x_i \leq 1 \ \forall i \in V \\ x_i + x_j \geq 1 \ \forall ij \in E \end{cases}$$

Obiettivo,:

$$\min \sum_{i \in V} w_i x_i$$

Assumiamo ora $\bar{x_i}$ soluzione ammissibile per ILP. Sia $X = \{i | \bar{x_i} = 1\}$. Visto che c'è il vincolo sugli archi, X è soluzione per $\hat{Vertex Cover}$.

Quindi, risolvendo in tempo polinomiale ILP si risolverebbe VertexCover, ma visto che $\hat{Vertex Cover}$ è NP completo, anche ILP è NP completo. □

1.5.2 Vertex Cover con arrotondamento

Consideriamo ora la trasformazione mostrata al teorema precedente, ma invece che un problema di programmazione lineare intera consideriamone uno reale, Π'.

Esiste il problema di fare il passo inverso. Con il problema intero infatti bastava considerare le variabili con valore 1, ora sono reali, bisogna quindi trovare un modo di capire se una variabile (nodo), fa o no parte del cover.

Si sta di fatto rilassando il problema, chiamiamo Π'_{INT} il problema intero, rispetto a quello reale si avrà che:[3]

$$x_i^*, w^*, \tilde{x_i^*}, \tilde{w^*} \quad \text{Soluzioni rispettivamente reale e intera}$$
$$\tilde{w^*} \geq w^* \quad \text{Poichè il dominio di ricerca reale è maggiore}$$

Si definisce ora la seguente trasformazione:

$$r_i = \begin{cases} 1 \; se \; x_i^* \geq \frac{1}{2} \\ 0 \; se \; x_i^* < \frac{1}{2} \end{cases}$$

Osservazione 17. *La trasformazione fornisce una soluzione ammissibile per* Π'_{INT}.

Dimostrazione. Bisogna dimostrare che:

$$\forall ij \in E, r_i + r_j \geq 1$$

Sapendo che:

$$\forall ij \in E, x_i^* + x_j^* \geq 1$$

Supponiamo per assurdo che valgano entrambi 0, ma allora:

$$x_i^* < \frac{1}{2}, x_j^* < \frac{1}{2}$$

che contraddice la seconda assunzione. □

Teorema 19. *Arrotondando la soluzione reale con la trasformazione mostrata, si fornisce una 2-approssimazione per* $\hat{Vertex Cover}$.

Dimostrazione. Valgono:

$$r_i \leq 2x_i^* \quad \text{Per definizione della trasformazione}$$
$$\sum_{i \in V} w_i r_i \leq 2 \sum_{i \in V} w_i x_i^* = 2w^* \leq 2\tilde{w} \quad \text{La soluzione arrotondata è al più il doppio}$$

□

Osservazione 18. *La soluzione non migliora Price Vertex Cover, anch'esso 2-approssimante, è però interessante l'approccio utilizzato.*

1.6 Altri esempi

In questa sezione si considera un problema che non ricade nelle techine viste in precedenza.

[3] Si sta considerando un problema di minimo

1.6.1 Traveling Salesman Problem

Prima di introdurre il problema, si introducono alcuni concetti preliminari tramite un secondo problema, quello dei ponti di Konisberg.

Problema dei ponti di Konisberg Il problema considera una città, costruita a cavallo di un fiume, e risponde alla domanda, è possibile effettuare un cammino che passi una e una sola volta su tutti i ponti della città?

La specifica istanza del problema si modella su un multigrafo [4] e la risposta è no.

Il problema generale consiste nell'individuare un *circuito Euleriano* nel grafo.

Teorema 20. *Un multigrafo ammette un circuito Euleriano sse tutti i vertici hanno grado*[5] *pari.*

Dimostrazione. Si sta dimostrando l'implicazione da sinistra a destra, ovvero se i nodi hanno tutti grado pari esiste un circuito Euleriano.

Concettualmente in un cammino, non posso rimanere bloccato, visto che tutti i nodi hanno grado pari. Nello specifico, in un cammino può succedere:

1. Creo un ciclo con la partenza, in questo caso riparto
2. Incido su un nodo del cammino, posso continuare, visto che ha grado pari

È facile intuire che è possibile continuare cosi finchè non si visitano tutti gli archi del multigrafo. □

Lemma 5. *In ogni grafo non orientato il numero di nodi con grado dispari è pari.*

Dimostrazione. Consideriamo la somma dei gradi:

$$\sum_{x \in V} d(x) = 2m$$

Entrambi i membri sono pari. Nella somma dei gradi, se considero un grado pari, la parità della somma non cambia, mentre, se aggiungo un grado dispari, la somma diventa dispari.

È facile intuire che serve necessariamente un numero pari di gradi dispari per ottenere un numero pari nel primo membro dell'equazione. □

Terminata la digressione sul problema dei ponti, si introduce ora formalmente il problema del commesso viaggiatore.

[4] Grafo in cui possono esistere piu archi per la stessa coppia sorgente destinazione
[5] Numero di archi incidenti

Input: $G = (V, E)$ non orientato, $\delta_e \in \mathbb{Q}^{>0} \forall e \in E$
Output: ciclo che passa per tutti i vertici, ovvero un ciclo Hamiltoniano
Funzione obiettivo: $\delta(\Pi) = \sum_{e \in E} \delta_e$
Tipo: min

1.6.2 Traveling Salesman Problem su cricche

Il problema del commesso viaggiatore spesso si considera su una cricca[6].

È possibile trasformare un grafo normale in una cricca aggiungendo tutti gli archi che mancano con peso equivalente alla somma di tutti i vertici più uno. La soluzione ottima non varia.

1.6.3 Traveling Salesman Problem metrico

Il problema in questa versione considera come input una cricca, inoltre vale la distanza triangolare, ovvero:

$$\forall i, j, k \in V, \delta_{ij} \leq \delta_{ik} + \delta_{kj}$$

Teorema 21. *TSP metrico è NPO completo.*

Algoritmo di Christofides L'algoritmo procede trovando un albero di copertura minimo e successivamente lavorando attorno a quello per individuare una soluzione.

Algorithm 9: Christofides

Input: $G = (V, \binom{V}{2}), \delta_e \in \mathbb{Q}^{>0} \forall e \in E$ che soddisfa la triangolare
Result: Soluzione per TSP metrico
$T \leftarrow \mathit{MinimumSpanningTree}(G)$
$D \leftarrow \{v \in T, d(v) \text{ } \mathit{dispari}\}$
$M \leftarrow \mathit{MinimumPerfectMatching}(D)$
$H \leftarrow T \cup M$
$\Pi \leftarrow \mathit{EulerianCircuit}(H)$
$\Pi \leftarrow \mathit{Hamiltonian}(\Pi)$
return Π

Osservazione 19. *Vale che:*

1. *D ha cardinalità pari per il lemma 5*
2. *H è un multigrafo con tutti gradi pari, quindi esiste un circuito euleriano*
3. *Il circuito euleriano si costruisce come nella dimostrazione del teorema 20*
4. *Per trovare un ciclo hamiltoniano da* Π *basta procedere nel cammino saltando i nodi già visitati, tanto è una cricca e ci sono tutti gli archi*

[6]Grafo con tutti gli archi possibili $K_V = (N, \binom{n}{2})$

Lemma 6. *Il peso dello spanning tree è al più uguale al peso della soluzione ottima.*

$$\delta(T) \leq \delta^*$$

Dimostrazione. Sia Π^* la soluzione ottima. Rimuovendo un qualunque lato, si ottiene un albero. Vale che, visto che T è il minimo albero di copertura:

$$\delta(T) \leq \delta(\Pi^* - e) \leq \delta(\Pi^*) = \delta^*$$

□

Lemma 7. *Il costo del perfect matching è al massimo la metà della soluzione ottima:*

$$\delta(M) \leq \frac{1}{2}\delta^*$$

Dimostrazione. Sia Π^* la soluzione ottima. Siano D i vertici con grado dispari considerati nell'algoritmo, che ovviamente appartengono alla soluzione ottima.

Si considera ora il ciclo costruito unendo i nodi di D e siano M_1 e M_2 due gruppi di lati, che contengono in maniera alternata i lati del ciclo. Sia M_1 che M_2 sono matching.
Vale che:

$$\begin{aligned} \delta(M) &\leq \delta(M_1) \\ \delta(M) &\leq \delta(M_2) \\ 2\delta(M) &\leq \delta(M_1) + \delta(M_2) \leq \delta^* \end{aligned}$$

L'ultima disequazione segue dal fatto che, collegando i nodi che sono in D si sta cortocircuitando il cammino, ovvero, due nodi n_1 e $n_2 \in D$ sono collegati direttamente nel matching ma potrebbero non esserlo nella soluzione ottima, segue quindi che per la triangolare, la distanza tra i due è al piu uguale a quella nel cammino ottimo.
Proseguendo con la disequazione si ottiene che

$$\delta(M) \leq \frac{1}{2}\delta^*$$

□

Teorema 22. *L'algoritmo di Christofides è $\frac{3}{2}$-approssimante per TSP metrico.*

Dimostrazione. Sia $\tilde{\Pi}$ il cammino ottimo. Vale che:

$$\begin{aligned} \delta(\tilde{\Pi}) \leq \delta(\Pi) &= \sum_{e \in M} \delta_e + \sum_{e \in T} \delta_e \\ \delta(\tilde{\Pi}) &\leq \frac{1}{2}\delta^* + \delta^* \leq \frac{3}{2}\delta^* \quad \text{Per i lemmi 6 e 7} \end{aligned}$$

□

Teorema 23. *L'analisi di Christofides è stretta.*

Dimostrazione. Si considera un grafo con n nodi disposti in linea. I nodi sono collegati in sequenza con archi lunghi 1. Poi, si collegano i nodi a distanza 2, partendo dal primo, andando verso destra con archi $1+\epsilon$, stesso discorso partendo dall'ultimo tornando indietro.

Si aggiungono poi tutti gli archi per creare una cricca, tali archi hanno lunghezza equivalente a quella del cammino minimo che unisce sorgente e destinazione.

Il minimum spanning tree pesca solo gli archi lunghi 1.
Il perfect matching utilizza l' arco dal nodo più a sinistra a quello più a destra, lungo

$$(1+\epsilon)(\frac{n}{2}-1)+1 = \frac{n}{2}+\epsilon\frac{n}{2}-\epsilon$$

Il cammino individuato da Christofides è quindi:

$$(n-1)+\frac{n}{2}+\epsilon\frac{n}{2}-\epsilon = \frac{3}{2}n+\epsilon\frac{n}{2}-(\epsilon+1)$$

Esiste però la soluzione ottima che consiste nel muoversi da sinistra a destra seguendo gli archi che portano a distanza 2, seguire l'arco dal penultimo all'ultimo nodo lungo 1, e ripetere il procedimento tornando indietro.
Tale approccio ha costo:

$$\delta(\Pi^*) = 2((1+\epsilon)(\frac{n}{2}-1)+1) = n+\epsilon n-2\epsilon$$

Calcoliamo il rapporto di approssimazione:

$$\frac{\delta(\Pi)}{\delta(\Pi^*)} = \frac{\frac{3}{2}n+\epsilon\frac{n}{2}-(\epsilon+1)}{n+\epsilon n-2\epsilon}$$

$$\lim_{n\to\infty,\epsilon\to 0}\frac{\delta(\Pi)}{\delta(\Pi^*)} = \frac{3}{2}$$

□

Teorema 24. *Non esiste un $\alpha > 1$ tale che TSP sia α-approssiamante in tempo polinomiale.*

Dimostrazione. Dato un grafo supponiamo di dover decidere se ammette un ciclo Hamiltoniano (NP completo).

Sia il grafo $G' = (V, \binom{V}{2})$ costruico come segue:

$$d(x,y) = \begin{cases} 1 \text{ se } xy \in E \\ \lceil \alpha n \rceil + 1 \text{ se } xy \notin E \end{cases}$$

Applichiamo l'algoritmo ottimo su G', la soluzione equivale a:

$$\delta^*(\Pi) = \begin{cases} n \text{ se esiste un ciclo hamiltoniano} \\ \lceil \alpha n \rceil + 1 \text{ altrimenti} \end{cases}$$

Usando l'algoritmo α-approssimante che si assume esistere, si otterrebbe una soluzione:

$$\begin{aligned} n \leq \delta(\Pi) \leq \alpha n \quad & \text{con ciclo hamiltoniano} \\ \lceil \alpha n \rceil + 1 \leq \delta(\Pi) \leq \alpha \lceil \alpha n \rceil + 1 \quad & \text{altrimenti} \end{aligned}$$

Significa che si può decidere se il grafo ammette un ciclo hamiltoniano in tempo polinomiale.
Deve perciò valere che:

$$\begin{aligned} \alpha n &\geq \lceil \alpha n \rceil + 1 \\ \alpha n &\geq \lceil \alpha n \rceil + 1 \\ \alpha &\geq \alpha + \frac{1}{n} \end{aligned}$$

Non esiste un α maggiore di uno che soddisfa l'equazione. □

1.7 Problemi PTAS e FPTAS

I due problemi a seguire ammettono algoritmi che appartengono alla classe PTAS, sono quidni approssimabili in maniera arbitraria.

1.7.1 Minimum partition

Il problema del minium partition equivale a suddividere un insieme in due sottoinsiemi tali che il massimo delle somme degli elementi dei due insiemi è minimizzato. Si può vedere come *2-load balancing*, visto a sezione 1.3.1.

Formalmente:
Input: $w_1, \ldots, w_n \in \mathbb{Q}^{>0}$
Output: $y_1 \subseteq \{1, \ldots, n\}, y_2 = \{1, \ldots, n\} \setminus y_1$
Funzione obiettivo: $max(\sum_{i \in y_1} w_i, \sum_{j \in y_2} w_j)$
Tipo: min

PTAS 2-loadbalance L'algoritmo è simile a Sorted Balance, assegna in modo ottimale alcuni task.

Algorithm 10: PTAS 2-loadbalance

Input: $w_1, \ldots, w_n \in \mathbb{Q}^{>0}$, $\epsilon > 0$
Result: Soluzione per load balance
if $\epsilon \geq 1$ **then**
  Assegno tutto alla prima macchina
$T \leftarrow$ *reverse*(*sorted*$(w_1, \ldots, w_n)$)
$k \leftarrow \lceil \frac{1}{\epsilon} - 1 \rceil$
Assegna i primi k task in modo ottimale
GreedyBalance$(w_{k+1}, \ldots, w_n)$
return $\max(W(y_1), W(y_2))$

Teorema 25. *L'algoritmo proposto fornisce una* $(1+\epsilon)$*-approssimazone per 2-loadbalance.*

Dimostrazione. Per ogni $X \subseteq \{1,\dots,n\}$, $w(X) = \sum_{i\in X} w_i$.

Sia $L = \frac{w(\{1,\dots,n\})}{2}$ il peso complessivo dei task, diviso 2.

Vale che:

$$w^* \geq L$$

Se così non fosse, significa che sia la prima che la seconda macchina hanno task per un peso inferiore alla metà del peso complessivo delle task. Ma ciò è assurdo, visto che la somma dei carichi delle macchine è $2L$.

Procediamo ora per casi rispetto ad ϵ:

Caso 1: $\epsilon \geq 1$
In questo caso:

$$w(y_1) = 2L \leq 2w^* \leq (1+\epsilon)w^*$$

Ovvero il peso della prima macchina rispetta l'approssimazione di $1+\epsilon$.

Caso 2: $\epsilon < 1$
Supponiamo che la prima macchina abbia peso maggiore:

$$w(y_1) \geq w(y_2)$$

Sia h l'indice dell'ultimo task assegnato alla prima macchina: Caso 2A: $h \leq k$
Ho trovato la soluzione ottima, visto che il peso della prima macchina è maggiore di quello della seconda.

Caso 2B $h > k$
Sia il peso subito prima dell'ultimo task, vale che:

$$\begin{aligned}
w(y_1) - w_h \leq w(y_2)' \leq w(y_2) && \text{peso della macchina 2 in quel momento}\\
w(y_1) + w(y_1) - w_h \leq w(y_2) + w(y_1) && \text{sommiamo il peso della macchina 1}\\
2w(y_1) - w_h \leq 2L &&\\
w(y_1) \leq \frac{w_h}{2} + L &&
\end{aligned}$$

Inoltre:

$$\begin{aligned}
2L = \sum_{i=1}^{n} w_i &= w_1 + \cdots + w_h + \cdots + w_n\\
&\geq w_1 + \cdots + w_h \geq w_h(k+1)\\
&\implies L \geq \frac{w_h}{2}(k+1)
\end{aligned}$$

Calcoliamo ora il rapporto di approssimazione:

$$\begin{aligned}
\frac{w(y_1)}{w^*} &\leq \frac{w(y_1)}{L} && \text{per la proprietà vista all'inizio} \\
&\leq \frac{\frac{w_h}{2} + L}{L} = 1 + \frac{w_h}{2L} && \text{per la seconda proprietà} \\
&\leq 1 + \frac{w_h}{2\frac{w_h}{2}(k+1)} = 1 + \frac{1}{k+1} && \text{per la terza proprietà} \\
&= \frac{1}{\lceil \frac{1}{\epsilon} - 1 \rceil + 1} \leq \frac{1}{\frac{1}{\epsilon} - 1 + 1} = 1 + \epsilon && \text{per la definizione di k}
\end{aligned}$$

□

Teorema 26. *L'algoritmo impiega un tempo pari a*

$$O(2^{\frac{1}{\epsilon}} + n \log n)$$

Ovvero è polinoimiale in n, ma esponenziale in ϵ

Corollario 4. *Minimum Partition $\in$ PTAS*

1.7.2 Knapsack FPTAS

Il problmea dello zaino consiste nel selezionare un insieme di elementi che massimizzino il valore, il cui ingombro non superi la capacità dello zaino.

Formalmente:
Input: $(w_1, v_1), \ldots, (w_n, v_n)$ $w_i, v_i, W \in \mathbb{N}$ dove W è la capacità dello zaino
Output: $X \subseteq \{1, \ldots, n\}, \sum_{i \in X} w_i \leq W$
Funzione obiettivo: $\sum_{i \in X} v_i$
Tipo: max

La versione di decisione del problema è NP completa.

Knapsack in programmazione dinamica È possibile risolvere il problema dello zaino in programmazione dinamica, si puo considerare la seguenze equazione di ricorrenza.

$$DP[i, w] = \begin{cases} 0 \textit{ se } i = 0 \textit{ or } w = 0 \\ DP[i-1, w] \textit{ se } w_i > w \\ \max(DP[i-1, w], DP[i-1, w - w_i] + v_i) \textit{ altrimenti} \end{cases}$$

L'algoritmo di facile implementazione segue l'equazione appena introdotta, riempendo una matrice iterativamente o ricorsivamente.

Osservazione 20. *Iterativamente si possono mantenere solo le ultime due righe, lo spazio si abbatte.*

Teorema 27. *Knapsack in programmazione dinamica ha complessità in tempo e spazio (a meno di ottimizzazioni) equivalente a $O(nW)$, dove n è il numero di oggetti e W la capacità dello zaino.*

Dimostrazione. Per mantenere il parametro W in memoria sono necessari $O(\log W)$ bit, quindi $W = 2^{|W|}$.

L'algoritmo si dice essere pseudopolinomiale. □

Knapsack programmazione dinamica rivisitato Questa volta cambia il contenuto della matrice. In particolare, in $DP[i][v]$ è contenuto il minimo peso, che consente di portare a casa $\geq v$ usando i primi i oggetti.

I casi base sono:

1. $v = 0$, in quel caso il valore della cella è nullo
2. $i = 0$, in quel caso si considera $+\infty$

L'equazione di ricorrenza diventa:

$$DP[i, v] = \begin{cases} 0 \text{ } se \text{ } v = 0 \\ +\infty \text{ } se \text{ } i = 0 \\ \min(DP[i-1, v], DP[i-1, v - v_i] + w_i) \text{ } altrimenti \end{cases}$$

Teorema 28. *Il problema riformulato ha complessità in tempo e spazio (a meno di ottimizzazioni) equivalente a $O(n \sum_i v_i) = O(n^2 v_{max})$. L'algoritmo rimane pseudopolinomiale, per il secondo termine.*

Osservazione 21. *La matrice si può restringere in larghezza, arrotondando i valori, in modo da introdurre approssimazione nel valore della soluzione ma non nella validità di essa.*

Knapsack rescaling Preso un input per Knapsack, si considera il valore:

$$\theta = \frac{\epsilon v_m ax}{2n}, 0 < \epsilon \leq 1$$

L'input del problema originale diventa:

$$\bar{\Pi} = ((w_i, \bar{v_i}), W), \bar{v_i} = \lceil \frac{v_i}{\theta} \rceil \theta$$

Si considera poi una terza versione:

$$\hat{\Pi} = ((w_i, \hat{v_i}), W), \hat{v_i} = \lceil \frac{v_i}{\theta} \rceil$$

La formulazione $\bar{\Pi}$ è utile nello studio del problema, quella versione non si può risolvere in programmazione dinamica, visto che i valori non sono interi.

Osservazione 22. *Le soluzioni ammissibili dei problemi sono equivalenti.*

Lemma 8. *Le soluzioni ottime $\hat{S}^*, \bar{S}^*$ di $\hat{\Pi}^*, \bar{\Pi}^*$ sono equivalenti, sono semplicemente scalati per il parametro θ.*

Lemma 9. *Sia S una soluione ammissibile di Π, allora:*

$$\sum_{i \in S} v_i \leq (1+\epsilon) \sum_{i \in \hat{S^*}} v_i$$

Dimostrazione.

$$\begin{aligned}
\sum_{i \in S} v_i &\leq \sum_{i \in S} \bar{v_i} && \text{Per l'operazione di ceil} \\
&\leq \sum_{i \in \bar{S^*}} \bar{v_i} && \text{S ammissibile, S barra ottima} \\
&= \sum_{i \in \hat{S^*}} \bar{v_i} && \text{per la prima osservazione} \\
\leq \sum_{i \in \hat{S^*}} (v_i + \theta) &= \sum_{i \in \hat{S^*}} v_i + |\hat{S^*}|\theta \\
\leq \sum_{i \in \hat{S^*}} v_i + n\theta &= \sum_{i \in \hat{S^*}} v_i + n\frac{\epsilon v_{max}}{2n} \\
&= \sum_{i \in \hat{S^*}} v_i + \frac{\epsilon v_{max}}{2}
\end{aligned}$$

Siccome questo vale per ogni soluzione ammissibile, è vera anche per $S = \{i_{max}\}$, ovvero l'indice che dà valore massimo. Continuando con l'equazione precedente:

$$\begin{aligned}
v_{max} \leq \sum_{i \in \hat{S^*}} v_i + \frac{\epsilon v_{max}}{2} \leq \sum_{i \in \hat{S^*}} v_i + \frac{v_{max}}{2} && \text{epsilon minore uguale a 1} \\
\sum_{i \in \hat{S^*}} v_i \geq \frac{\epsilon v_{max}}{2}
\end{aligned}$$

Applicando ora l'equazione appena ottenuta alla prima equazione della dimostrazione:

$$\begin{aligned}
\sum_{i \in S} v_i &\leq \sum_{i \in \hat{S^*}} v_i + \frac{\epsilon v_{max}}{2} \\
\sum_{i \in S} v_i \leq \sum_{i \in \hat{S^*}} v_i + \epsilon \sum_{i \in \hat{S^*}} v_i &= (\sum_{i \in \hat{S^*}} v_i)(1+\epsilon)
\end{aligned}$$

□

Osservazione 23. *Vale che:*

$$\hat{v_{max}} = \lceil \frac{v_{max}}{\frac{\epsilon v_{max}}{2n}} \rceil = \lceil \frac{2n}{\epsilon} \rceil \leq \frac{2n}{\epsilon} + 1$$

Corollario 5. *L'algoritmo in programmazione dinamica rivisitato, applicato a $\hat{\Pi}$, fornisce una $(1=\epsilon)$-approssimazione di Knapsack in tempo e spazio $O(\frac{n^3}{\epsilon})$.*

Dimostrazione. L'algoritmo individua $\hat{S^*}$, vale che:

$$\begin{aligned}
(1-\epsilon) \sum_{i \in \hat{S^*}} v_i \geq \sum_{i \in S} v_i \geq v^* && \text{Per il lemma 9} \\
1 \leq \frac{v^*}{\sum_{i \in \hat{S^*}} v_i} \leq (1+\epsilon)
\end{aligned}$$

L'algoritmo ha complessità

$$O(n^2 \hat{v_{max}}) = O(n^2(\frac{2n}{\epsilon} + 1)) = O(\frac{n^3}{\epsilon})$$

□

Osservazione 24. *Al variare di ϵ si aumenta o abbassa il tempo necessario al completamento dell'algoritmo, in particolare.*

Corollario 6. *Knapsack risolto in questo modo appartiene alla classe FPTAS.*

2 Algoritmi probabilistici

In questa sezione si vedono approcci probabilistici a problemi di ottimizzazione, in particolare si vedranno tre scenari, il primo in cui si individua la soluzione ottima con una certa probabilità, un secondo caso in cui con probabilità stimabile si individua una soluzione che non si discosta troppo da quella ottima e infine un'ultimo scenario in cui si sfruttano concetti probabilistici per comporre un algoritmo completamente deterministico.

2.1 Problemi di decisione

Per quanto riguarda i problemi di decisione, esistono sostanzialmente due categorie di algoritmi probabilistici.

Algoritmi Monte-Carlo Algoritmi che sono caratterizzati da un tempo fissato, e una correttezza probabilistica della soluzione.
Essi possono essere:

- *One-sided*: possono sbagliare solo in un verso
- *Two-sided*: possono sbagliare in entrambe le risposte

Algoritmi Las Vegas Algoritmi per cui la soluzione individuata è corretta, ma per cui il tempo è probabilistico.

Osservazione 25. *Un algoritmo Monte-Carlo può diventare quasi Las Vegas, se si esegue per molte volte, e si prende la soluzione più probabile. Si abbassa la probabilità di errore della decisione, ma impiega più tempo.*

2.2 Problemi di ottimizzazione

Approcciare in maniera probabilistica un problema di ottimizzazione, può significare individuare con una certa probabilità la soluzione ottima, oppure individuare probabilisticamente una soluzione che è abbastanza buona.

2.2.1 Min cut globale

Il problema di min cut globale cerca di individuare un taglio in un grafo, in modo da minimizzare il numero di archi che incidono sulle due partizioni create.

Formalmente:
Input: $G = (V, E)$
Output: $V = V_1 \cup V_2, V_1 \neq \emptyset, V_2 \neq \emptyset$
Costo: $|\{xy \in E, x \in V_1, y \in V_2\}|$
Tipo: min

Lemma 10. *Se esiste un vertice di grado d, esiste un taglio minimo E^* con $|E^*| \leq d$.*

Dimostrazione. Basta considerare come uno dei due insiemi il singolo vertice di grado d. □

Contrazione Operazione che consiste nell'eliminare un particolare arco e unire le due estremità in un unico nodo. Nel caso si un multigrafo, si cancellano tutti gli archi paralleli a quello che si sta considerando. L'operazione sarà indicata con $G \downarrow e$. Due nodi che vengono contratti formano un nuovo nodo, che li contiene.

Algoritmo di Karger L'algoritmo sceglie a caso tra gli archi e contrae. Restituisce i due nodi finali.

Algorithm 11: Karger

Input: $G = (V, E)$
Result: Taglio per il grafo
while $|G.V| > 2$ **do**
  $e \leftarrow random(E)$
  $G \leftarrow G \downarrow e$
return V_1, V_2

Durante l'esecuzione dell'algoritmo si può pensare al grafo come: $G = G_0, G_1, \ldots$

Osservazione 26. *Sia ora E_{S^*} un taglio ottimo e $k^* = |E_{S^*}|$.*
Si osserva che:

1. *G_i ha $n - i + 1$ vertici e numero di lati $\leq m - i + 1$*
2. *Ogni taglio di G_i corrisponde a un taglio di G*
3. *Il grado minimo di un vertice in G_i è sempre $\geq k^*$, se così non fosse, G_i avrebbe un taglio $< k^*$, quindi anche G, per il punto 2*
4. *Il punto 1 e 3 fanno in modo che il numero di lati sia $\geq \frac{(n-i+1)k^*}{2}$*

Lemma 11. *Vale che:*

$$m - i + 1 \geq \frac{k^*(n - i + 1)}{2}$$

Dimostrazione. Mettendo insieme le osservazioni fatte in precedenza, si ottiene che:

$$\begin{aligned} 2(m-i+1) \geq 2 * \textit{lati di } G_i = \sum_{v \in V_i} d_{G_i}(v) & \quad \text{Per osservazione 1} \\ \geq \sum_{v \in V_i} k^* & \quad \text{Per osservazione 3} \\ \geq k^*(n-i+1) & \quad \text{Per osservazione 1} \end{aligned}$$

□

Sia E_i l'evento che indica che all'iesima iterazione dell'algoritmo di Karger non contraggo nessun lato di E_s^*.

Lemma 12. *Vale che,per ogni i:*

$$P(E_i|E_1, \ldots, E_{i-1}) \geq \frac{n-i-1}{n-1+1}$$

Dimostrazione.

$$\begin{aligned} P(E_i|E_1, \ldots, E_{i-1}) &= 1 - P(\bar{E}_i|E_1, \ldots, E_{i-1}) \\ &= 1 - \frac{k^*}{\#\textit{ lati } G_i} \geq 1 - \frac{2k^*}{(n-i+1)k^*} \quad \text{Vale per oss. 26 punto 4} \\ &\geq 1 - \frac{2k^*}{k^*(n-i+1)} \\ &= \frac{n-i+1-2}{n-i+1} = \frac{n-i-1}{n-i+1} \end{aligned}$$

□

Teorema 29. *L'algoritmo di Karger trova l'ottimo con probabilità* $\geq \frac{1}{\binom{n}{2}}$

Dimostrazione. Trovare una soluzione ottima coincide con la probabilità:

$$P[E_1 \cap E_2 \cap, \ldots, \cap, E_{n-2}]$$

Sviluppando si ottiene

$$\begin{aligned} P[E_1 \cap E_2 \cap, \ldots, \cap, E_{n-2}] &= P[E_1] \cdot P[E_2|E_1] \cdot P[E_3|E_1, E_2] \ldots \\ &\geq \frac{n-1-1}{n-1+1} \cdots \frac{n-(n-2)-1}{n-(n-2)+1} = \frac{\prod_{i=1}^{n-2} i}{\prod_{i=3}^{n} i} \\ &= \frac{1 \cdot 2}{n(n-1)} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \end{aligned}$$

□

Corollario 7. *L'esecuzione dell'algoritmo di Karger* $\binom{n}{2} \ln n$ *volte e prendendo il taglio minimo, l'ottimo di ottiene con probabilità* $\geq 1 - \frac{1}{n}$

Dimostrazione. Ad ogni iterazioe non troviamo l'ottimo con probabilità $\leq 1 - \frac{1}{\binom{n}{2}}$

La probabilità di non trovarlo in nessuna esecuzione è

$$\leq (1 - \frac{1}{\binom{n}{2}})^{\binom{n}{2}\ln n} \leq (\frac{1}{e})^{lnn} = \frac{1}{e^{\ln n}} = \frac{1}{n}$$

□

Osservazione 27. *L'esecuzione dell'algoritmo costa $O(n \log n)$ con un MFset, quindi per ottenere quella probabilità la complessità finale è $O(n^3 \log n)$.*

2.2.2 Set Cover probabilitstico

Il problema è equivalente a quello presentato a sezione 1.4.1, cambia l'approccio seguito.

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

Trasformazione programmazione lineare Per affrontare il problema si passa a una formulazione in programmazione lineare, Π_{LP}. Si aggiunge una variabile per ogni insieme.

$$x_1, \ldots, x_m \; 0 \leq x_i \leq 1$$

La funzione da minimizzare equivale a:

$$w_1 x_1 +, \ldots, w_m x_m$$

Tutto l'universo deve essere coperto, quindi:

$$\sum_{j, i \in s_j} x_j \geq 1, \; \forall i \in U$$

Sia ora x^* soluzione ottima della versione intera Π_{LP}, con valore v^*.

Se si considera il dominio reale, esisterà una soluzione $\hat{x}, \hat{v}$, tale che

$$\hat{v} \leq v^*$$

Si formula ora un algoritmo per Set Cover, dove si trova una soluzione per la riformulazione in programmazione lineare reale e poi si considera il valore

della variabile $\hat{x_i}$ come probabilità di scelta. Si immagina di iterare il processo $\lceil k + \ln n \rceil$ volte.

Algorithm 12: SetCoverProbabilistico

```
Input: S_i, w_i, i ∈ {1, ..., m}
Result: Indici degli insiemi da coprire
Î ← solve Π_LP
I ← ∅
for i = 1, ..., m do
    P_i ← 1 − (1 − x̂_1)^⌈k+ln n⌉
    if random() < P_i then
        I ← I ∪ i
return I
```

Nel codice P_i è la probabilità di inserire un elemento, ovvero 1 meno la probabilità di non inserirlo $\lceil k + \ln n \rceil$ volte.

Per l'analisi dell'algoritmo servono alcuni concetti preliminari.

Disuguaglianza di Boole Vale che la probabilità dell'unione di eventi è al più la somma delle probabilità singole.

$$P[A_1 \cup \ldots, \cup A_n] \leq \sum_{i=1}^{n} P[A_i]$$

Dimostrazione. Si prova per induzione.

Per $n = 1$ è banale.
Per $n + 1$ vale che:

$$\begin{aligned} P[A_1 \cup \ldots, \cup A_{n+1}] &= P[(A_1 \cup \ldots, A_n) \cup A_{n+1}] \\ P[A_1 \cup \ldots, \cup A_n] + P[a_{n+1}] - P[A_1 \cup \ldots, \cup A_n] \cap P[a_{n+1}] \\ &\leq \sum_{i=1}^{n} P[A_i] + P[A_{n+1}] = \sum_{i=1}^{n+1} P[A_i] \end{aligned}$$

□

Disuguaglianza di Markov Se X è una variabile aleatoria non negativa con media finita, e sia $\alpha > 0$, vale:

$$P[X > \alpha] \leq \frac{E[X]}{\alpha}$$

Dimostrazione. Sia I una nuova variabile aleatoria che indica l'evento $X \geq \alpha$.

$$I_{X \geq \alpha} = \begin{cases} 1 \textit{ se } X \geq \alpha \\ 0 \textit{ altrimenti} \end{cases}$$

Inoltre

$$\alpha I_{X\geq\alpha} = \begin{cases} \alpha \text{ se } X \geq \alpha \\ 0 \text{ altrimenti} \end{cases}$$

Vale che

$$\begin{aligned} \alpha I_{X\geq\alpha} &\leq X && \text{X è sempre maggiore di zero} \\ E[\alpha I_{X\geq\alpha}] &\leq E[X] && \text{monotonia valore atteso} \\ \alpha E[I_{X\geq\alpha}] &\leq E[X] && \text{linearità valore atteso} \\ \alpha(1 \cdot P[X \geq \alpha] + 0 \cdot P[X < \alpha]) &\leq E[X] && \text{definizione di I} \\ \alpha P[X \geq \alpha] \leq E[X] &\implies \frac{E[X]}{\alpha} \geq P[X \geq \alpha] \end{aligned}$$

□

Teorema 30. *L'algoritmo proposto produce una soluzione ammissibile con probabilità* $\geq 1 - e^{-k}$.

Dimostrazione. Sia

$$\hat{v} = \sum_{i=1}^{m} v_i \hat{x_i} \leq v^*$$

Vale che :

$$\begin{aligned} P[\textit{soluzione ammissibile}] &= 1 - P[\textit{soluzione errata}] \\ \geq 1 - \sum_{u\in U} P[u \textit{ non coperto}] &= 1 - \sum_{u\in U} \prod_{i,u\in S_i} P[i \notin I] \quad \text{Per union bound} \\ = 1 - \sum_{u\in U} \prod_{i,u\in S_i} (1-\hat{x_i})^{k+\ln n} &\geq 1 - \sum_{u\in U} \prod_{i,u\in S_i} e^{-(k+\ln n)} \\ = 1 - \sum_{u\in U} e^{-(k+\ln n)\sum_{i,u\in S_i}\hat{x_i}} &\geq 1 - \sum_{u\in U} e^{-(k+\ln n)} \\ = 1 - \sum_{u\in U} e^{-k} \cdot e^{-\ln n} &= 1 - \frac{1}{n}\sum_{u\in U} e^{-k} \\ = 1 - \frac{e^{-k}}{n} \cdot n &= 1 - e^{-k} \end{aligned}$$

□

Teorema 31. *Per ogni* $\alpha > 0$*, sia* R *il fattore di approssimazione dell'algoritmo.*

$$P[R \geq \alpha(k + \ln n)] \leq \frac{1}{\alpha}$$

Ovvero è poco probabile avere un fattore di approssimazione brutto.

Dimostrazione. La probabilità che S_i venga scelto è $\leq (k+\ln n)\hat{x_i}$, per lo union bound,

Il valore atteso del costo equivale a:

$$E[v] = \sum_i w_i P[S_i \text{ sia scelto}] \leq \sum_i w_i(k + \ln n)\hat{x_i}$$
$$= \hat{v}(k + \ln n) \leq v^*(k + \ln n)$$

Ragionando sul rapporto di approssimazione

$$P[\frac{v}{v^*} \geq \alpha(k + \ln n)] \leq \frac{E[\frac{v}{v^*}]}{\alpha(k + \ln n)} \quad \text{Per disuguaglianza di Markov}$$
$$= \frac{E[v]}{\alpha v^*(k + \ln n)} \leq \frac{v^*(k + \ln n)}{\alpha v^*(k + \ln n)} = \frac{1}{\alpha}$$

□

Corollario 8. *Fissando il parametro $k = 3$:*

1. *il teorema 30 dice che la probabilità di ottenere una soluzione ammissibile $\geq 1 - e^{-k} = 95\%$*
2. *il teorema 31 invece, dice che, fissando ad esempio $\alpha = 2$, con R fattore di approssimazione, $P[R \geq 2(3 + \ln n)] \leq \frac{1}{2}$, quindi la probabilità di ottenere un fattore di approssimazione peggiore di $2(3 + \ln n)$ è inferiore a $\frac{1}{2}$*

Corollario 9. *Con $k = 3$ c'è almeno il 45% di probabilità che l'algoritmo emetta una soluzione ammissibile con fattore di approssimazione al più $2(3 + \ln n)$*

Dimostrazione. Considerando i due punti del corollario 8 appena scritto come eventi indipendenti, si calcola ora la probabilità che si verifichino insieme, che sarebbe quello che si vuole ottenere dall'algoritmo.

$$\begin{aligned} P[E_1 \wedge E_2] &= 1 - P[\bar{E_1} \vee \bar{E_2}] \\ &\leq 1 - (P[\bar{E_1}] + P[\bar{E_2}]) && \text{Per union bound} \\ &\leq 1 - (0.05 + 0.5) && \text{Per il corollario 8} \\ &= 1 - 0.55 = 0.45 \end{aligned}$$

□

2.2.3 MaxEkSat

Questa riformulazione del problema aggiunge un costraint per quanto riguarda il numero di variabili per ogni condizione di sat.

Formalmente: *Input*: Formula CNF con t clausole, ognuna di k letterali, ogni variabile al massimo una volta per clausola
Output: Assegnamento che rende vera la CNF
Funzione obiettivo: Numero di clausole risolte
Tipo: max

Osservazione 28. *Il problema è NP completo per ogni $k \geq 3$*

Teorema 32. *Un assegnamento casuale delle variabili, porta ad una soluzione che soddisfa in valore atteso* $\geq \frac{2^k-1}{2^k}t$ *clausole, ovvero, sia* T *la variabile aleatoria che rappresenta il numero di clausole soddisfatte:*

$$E[T] = \frac{2^k - 1}{2^k}t$$

Dimostrazione. Siano $x_1, \ldots, x_m$ le variabili che compaiono nella formula e $c_1, \ldots, c_t$ le clausole.

Si definiscono poi delle variabili aleatorie:

1. $X_1, \ldots, X_m \sim \mathit{Unif}(\{0,1\})$,dove X_i è il valore assegnato a x_i
2. $C_1, \ldots, C_t$ ognuna delle quali vale 1 sse la clausola c_i è soddisfatta
3. T numero di clausole soffisfatte, $T = \sum_{i=1}^{t} C_i$

Si calcola ora il valore atteso di T.

Vale che:

$$\begin{aligned}
E[T] &= \sum_{E} E[T|E] \cdot P[E] \\
&= \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} P[X_1 = b_1, \ldots, X_n = b_n] \\
&\quad E[T|X_1 = b_1, \ldots, X_n = b_n] \\
&= \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} P[X_1 = b_1] \ldots P[X_n = b_n] && \text{var indipendenti} \\
&\quad E[C_1 + \cdots + C_t | X_1 = b_1, \ldots, X_n = b_n] && \text{definizione di T} \\
&= \frac{1}{2^n} \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} \sum_{i=1}^{t} E[C_i | X_1 = b_1, \ldots, X_n = b_n] && \text{valore di X, linearità v.a.}
\end{aligned}$$

Ora si nota che quel valore atteso finale, vale 0 oppure 1, in base al soddisfacimento o meno della clausola C_i. Ogni clausola è composta da k variabili in disgiunzione, quindi, esiste un solo assegnamento delle k che la rende falsa.

Visto che si hanno n variabili totali, esistono esattamente 2^{n-k} assegnamenti che rendono falsa la clausola, ovvero, si individua un assegnamento delle k variabili che rendono falsa la clausola C_i e si considerano tutti i possibili assegnamenti delle restanti $(n-k)$ variabili.

Vale quindi che gli assegnamenti che rendono vera la clausola C_i sono $2^n - 2^{n-k}$, ovvero il totale delle combinazioni meno quelle che la rendono falsa.

Proseguendo con l'equazione quindi, sostituendo quella sommatoria con quanto appena discusso:

$$= \frac{1}{2^n} \sum_{b_1 \in 2} \cdots \sum_{b_n \in 2} \sum_{i=1}^{t} E[C_i | X_1 = b_1, \ldots, X_n = b_n]$$
$$= \frac{1}{2^n} t(2^n - 2^{n-k}) = \frac{t(2^n - 2^{n-k})}{2^n}$$
$$= \frac{t(2^n - 2^{n-k})}{2^n} * \frac{2^{k-n}}{2^{k-n}} = t\frac{2^k - 1}{2^k}$$

□

Teorema 33. *Dato un input per MaxEkSat con n variabili, e t clausole.*

$$\forall j = 0, \ldots, n, \ \exists b_1, \ldots, b_j \in 2, t.c$$
$$E[T | X_1 = b_1, \ldots, X_j = b_j] \geq t\frac{2^k - 1}{2^k}$$

Dimostrazione. Si dimostra per induzione per j.
$j = 0$, dimostrato dal teorema precendente.

Passo induttivo:

$$E[T | X_1 = b_1, \ldots, X_j = b_j] =$$
$$E[T | X_1 = b_1, \ldots, X_j = b_j, X_{j+1} = 0] \cdot P[x_{j+1} = 0]$$
$$+ E[T | X_1 = b_1, \ldots, X_j = b_j, X_{j+1} = 1] \cdot P[x_{j+1} = 1] \quad \text{Eventi disgiunti}$$
$$= \frac{1}{2} \cdot E[T | X_1 = b_1, \ldots, X_j = b_j, X_{j+1} = 0]$$
$$+ \frac{1}{2} \cdot E[T | X_1 = b_1, \ldots, X_j = b_j, X_{j+1} = 1]$$

Per ipotesi induttiva, la media dei due valori attesi è almeno $t\frac{2^k-1}{2^k}$, questo implica che uno dei due valori attesi è maggiore di quella quantitità, ovvero, per qualche $b \in 2$:

$$E[T | X_1 = b_1, \ldots, X_j = b_j, X_{j+1} = b] \geq t\frac{2^k - 1}{2^k}$$

□

Osservazione 29. *La dimostrazione appena scritta suggerisce un approccio da seguire per l'algoritmo finale, si deciderà iterativamente quale valore assegnare alla variabile j in base al valore atteso di T.*

MaxEkSat derandomizzato Seguendo le proprietà osservate in precedenza si formula il seguente algoritmo.

Algorithm 13: MaxEkSat Derandomizzato

Input: $C_1 \wedge \cdots \wedge C_t$, ognuna con k letterali $\in x_1, \ldots, x_n$
Result: Assegnamento di variabili
$D \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  $\Delta_0 \leftarrow 0$
  $\Delta_1 \leftarrow 0$
  $\Delta D_0 \leftarrow \emptyset$
  $\Delta D_1 \leftarrow \emptyset$
  for $j = 1, \ldots, t$ **do**
    if $j \in D \vee x_i \notin C_j$ **then**
      continue
    $h \leftarrow |x_k \in C_j, k > i|$
    if x_i *positiva in* C_j **then**
      $\Delta_0 \leftarrow \Delta_0 - \frac{1}{2^h}$
      $\Delta_1 \leftarrow \Delta_1 + \frac{1}{2^h}$
      $\Delta D_1 \leftarrow \Delta D_1 \cup \{j\}$
    else
      $\Delta_0 \leftarrow \Delta_0 + \frac{1}{2^h}$
      $\Delta_1 \leftarrow \Delta_1 - \frac{1}{2^h}$
      $\Delta D_0 \leftarrow \Delta D_0 \cup \{j\}$
  $b = (\Delta_0 \leq \Delta_1)$
  $X_i = b$
  $D \leftarrow D \cup \Delta D_b$
return *Assegnamento*

L'algoritmo implementa quanto osservato nel teorema precedente. In particolare si cerca di mantenere al passo i vera questa proprietà

$$E[T|X_1 = b_1, \ldots, X_{i-1} = b_{i-1}] \geq t\frac{2^k - 1}{2^k}$$

Il valore atteso si può scomporre

$$\begin{aligned} E[T|X_1 = b_1, \ldots, X_{i-1} = b_{i-1}] = \\ E[C_1|X_1 = b_1, \ldots, X_{i-1} = b_{i-1}] + \\ E[C_2|X_1 = b_1, \ldots, X_{i-1} = b_{i-1}] + \\ \ldots \\ + E[C_t|X_1 = b_1, \ldots, X_{i-1} = b_{i-1}] \end{aligned}$$

Quello che si fa ora è confrontare i singoli valori attesi con gli stessi ottenuti dopo l'assegnamento della iesima variabile.

Si fa notare che, se una clausola C_j appartiene a D, il valore atteso dopo l'assegnamento della iesima variabile non cambia, ovvero quella clausola è già decisa.

Il valore può cambiare per quelle che contengono la variabile X_i, consideriamone una che contiene quella variabile, per esempio:

$$x_1 \vee \bar{x_4} \vee \cdots \vee x_i \vee x_{i+1} \cdots \vee x_n$$

Consideriamo h variabili da i a n comprese, ci sono $\frac{2^h-1}{2^h}$ assegnamenti delle h variabili che rendono vera la clausola.

Il valore atteso della clausola vale, prima dell'assegnamento di x_i

$$E[C_j] = \frac{2^h - 1}{2^h}$$

dopo l'assegnamento varia in questo modo

$$E[C_j] = \begin{cases} 1 \ se \ x_i = 1 \\ \frac{2^{h-1}-1}{2^{h-1}} \ se \ x_i = 0 \end{cases}$$

il cambiamento di valore atteso vale

$$\Delta \ valore \ atteso = \begin{cases} 1 - \frac{2^h-1}{2^h} = \frac{2^h-2^h+1}{2^h} = +\frac{1}{2^h} \ se \ x_i = 1 \\ \frac{2^{h-1}-1}{2^{h-1}} - \frac{2^h-1}{2^h} = \frac{2^h-2-2^h+1}{2^h} = -\frac{1}{2^h} \ se \ x_i = 0 \end{cases}$$

Nell'algoritmo non si sta facendo altro che calcolare il delta di valore atteso di T in base al valore assegnato alla variabile x_i.
Per la dimostrazione del teorema 33 uno degli assegnamenti di x_i produrrà un valore atteso maggiore, quindi si sceglierà quello.

Si sta quindi procedendo nell'algoritmo accertandosi di mantenere vera la proprietà enunciata dal teorema 33.

Teorema 34. *Al termine dell'algoritmo almeno* $\frac{2^k-1}{2^k}t$ *clausole saranno vere.*

Corollario 10. *L'algoritmo fornisce una* $\frac{2^k-1}{2^k}$ *'approssimazione per MaxEkSat.*

Dimostrazione. Sia t^* la soluzione ottima e sia $\bar{t}$ la soluzione prodotta dall'algoritmo.

$$\bar{t} \geq \frac{2^k - 1}{2^k} t \implies t \leq \bar{t} \frac{2^k}{2^k - 1} \quad \text{Per il teorema 33}$$

$$\frac{t^*}{\bar{t}} \leq \frac{t}{\bar{t}} \leq \frac{\bar{t}\frac{2^k}{2^k-1}}{\bar{t}} \leq \frac{2^k}{2^k - 1}$$

□

Corollario 11. *MaxE3Sat è* $\frac{7}{8}$*-approssimante.*

3 Teoria della complessità di approssimazione

In questa sezione si affrontano alcune tematiche, in particolare il teorema PCP, con il quale di riuscirà a dimostrare l'inapprossimabilità di MaxEkSat e Indipendent set.

3.1 Verificatori

Il concetto di verificatore assume diverse sfaccettature nel corso delle sezioni che seguono, ma sostanzialmente è una macchina che si occupa di verificare la decisione di un certo problema per un certo input.

3.1.1 NP attravero testimoni

Una Macchina di Turing verificatore per un problema di decisione $L \subseteq 2^*$, riceve due input, x e w, testimone, e si comporta come segue:

1. se $x \notin L$ risponde no qualunque sia w
2. se $x \in L$, allora $\exists w \in 2^*$ tale che la macchina risponde sì

Devono valere però:

1. $|w| \leq P(|x|)$ ovvero la lunghezza di w è polinomiale nell'input
2. la macchina lavora in tempo polinomiale

Osservazione 30. *Si può pensare alla classe NP come una classe di problemi facilmente verificabili, ma difficilmente risolvibili.*

Osservazione 31. *Esistono problemi nè risolvibili nè verificabili facilmente.*

Teorema 35. *Un problema $L \subseteq 2^* \in$NP se e solo se esiste una Macchina di Turing deterministica V, verificatore, e un polinomio $P()$, tali che:*

1. *$V(x, w)$ lavora in tempo polinomiale in $|x|$*
2. *$\forall x \in 2^*$:*
   (a) *se $x \in L$, $\exists w \in 2^*$ tale che $|w| \leq P(|x|)$ e $V(x, w) = yes$*
   (b) *se $x \notin L$, $\forall w \in 2^*$ $|w| \leq P(|x|)$, $V(x, w) = no$*

3.1.2 NP con oracolo

L'idea di una Macchina di Turing con oracolo presuppone l'esistenza, oltre al nastro classico, di un nastro dell'oracolo.

La Macchina può entrare in uno stato di query all'oracolo, in cui lo interroga e l'oracolo risponderà, facendole cambiare stato di conseguenza.

Teorema 36. *Un problmema $L \subseteq 2^* \in$ NP se esiste una Macchina di Turing con oracolo V e un polinomip $P()$, tale che:*

1. *$V(x)$ lavora in tempo polinomiale in $|x|$*
2. *$V(x)$ effettua al più $P(|x|)$ query all'oracolo*

3. $\forall x \in 2^*$*:*

(a) se $x \notin L$, $V(x) =$ *no qualunque sia l'oracolo*

(b) se $x \in L$, $V(x) =$ *yes per una qualche stringa dell'oracolo*

3.1.3 Verificatore probabilistico

In questo caso si estende il concetto di verificatore con oracolo aggiungendo un nastro di numero casuali che la Macchina può leggere.

Date due funzioni

$$q, r : \mathbb{N} \longrightarrow \mathbb{N}$$

$PCP[r, q]$ è la classe di linguaggi $L \subseteq 2^*$ per i quali esiste un verificatore probabilistico con le seguenti proprietà:

1. $V(x)$ lavora in tempo polinomiale
2. $V(x)$ fa al massimo $q(|x|)$ query all'oracolo
3. $V(x)$ usa al più $r(|x|)$ bit casuali
4. $\forall x \in 2^*$:
   (a) Se $x \notin L$, $V(x) = no$ con probabilità $\geq \frac{1}{2}$
   (b) Se $x \in L$, $V(x) = yes$ con probabilità 1 per almeno un $w \in 2^*$

Osservazione 32. *Vale che:*

- $NP = PCP[0, Poly] = \cup_{p:polinomio} PCP[0, p]$
- $P = PCP[0, 0]$

3.1.4 Teorema PCP

Definiti i verificatori probabilistici, si passa ora al teorema PCP, che in un certo senso dice che il non determinismo può essere portato ad una costante, che fa quindi un numero costante di interrogazioni all'oracolo, a patto che si facciano una quantità logaritmica di scelte casuali.

Teorema 37. $PCP[O(\log n), O(1)] = NP$

3.1.5 Verificatore NP in forma canonica

Si considera ora la classe

$$PCP[r(n), q], r(n) = O(\log n), q \in \mathbb{N}$$

Un verificatore facente parte di questa classe, può leggere fino a $r(|x|)$ bit random, e q query all'oracolo.

Il processo seguito da un verificatore del genere è quello di interrogare l'oracolo e agire di conseguenza. In particolare, si può pensare a un albero contenente le varie strade che può prendere il verificatore in base alla risposta dell'oracolo. Ha quindi un comportamento adattivo.

Quello che può fare però il verificatore, invece che chiedere di volta in volta all'oracolo, è simulare tutti i rami di decisione, e segnarsi quando dovrebbe interrogarlo, in modo da poi richiedere in una volta sola tutto quello di cui ha bisogno.

Un verificatore in forma canonica è proprio questo, ovvero un verificatore che prima simula tutti i rami di computazione possibili, che variano in base all'interrogazione all'oracolo, per poi interrogarlo su tutte le posizioni di cui ha bisogno. Il verificatore così ottenuto può essere considerato come una semplice Macchina di Turing, infatti non deve più interrogare l'oracolo.

3.2 Problemi inapprossimabili

In questa sezione si affrontano due dimostrazioni di inapprossimabilità, esse sfruttano i concetti introdotti a sezione precedente.

3.2.1 Inapprossimabilità MaxEeSat

Il problema è stato introdotto a sezione 2.2.3, e si era individuato un algoritmo deterministico che sfruttava concetti probabilistici.

Teorema 38. *Esiste un algoritmo deterministico che approssima MaxE3Set con tasso* $\frac{8}{7}$

Teorema 39. *Esiste un* $\bar{\epsilon} > 0$ *tale che MaxE3Sat non è* $(\frac{8}{7} - \bar{\epsilon})$*-approssimabile.*

Un versione rilassata del teorema appena introdotto è la seguente.

Teorema 40. *Esiste un* $\bar{\epsilon} > 0$ *tale che MaxE3Sat non è* $(1 + \bar{\epsilon})$*-approssimabile.*

Osservazione 33. *Il teorema 39 dimostra che l'algoritmo individuato a sezione 2.2.3 è ottimo, mentre quello successivo che MaxE3Sat non fa parte di PTAS. La dimostrazione di quest'ultimo è molto più semplice e usa lo stesso principio.*

Lemma 13. *Esiste un* $\epsilon > 0$ *tale che MaxSat non è* $(1 + \bar{\epsilon})$*-approssimabile in tempo polinomiale.*

Dimostrazione. Considerando quanto detto su PCP a sezione precedente:

$$L \in NP \implies \exists r(n) \in O(\log n), \exists w \in \mathbb{N}, L \in PCP[r(n), q]$$

Significa quindi che esiste un verificatore probabilistico V che opera come segue:

$$x \in 2^* \longrightarrow V(R \in 2^{r(|x|)}, (w_{i_1}, \ldots, w_{i_q})) \longrightarrow yes/no$$

Dove $(w_{i_1}, \ldots, w_{i_q})$ sono funzione di x e di R.

Fissando x e R si può immaginare una funzione che opera così:

$$f^R(w^R_{i_1}, \ldots, w^R_{i_q}) = \textit{true sse } V \textit{ accetta}$$

La funzione può essre espressa come formula sat, e ridotta a 3-sat:

$$\Phi_x = \wedge_{R \in 2^{r(|x|)}} \phi^R_z$$

Dove ϕ^R_z è la traduzione di f^R in una formula logica dove le variabili rappresentano il seguente evento:

$$x_i = \begin{cases} \textit{true se } w_i = 1 \\ \textit{false se } w_i = 0 \end{cases}$$

Φ_x ha $2^{r(|x|)} \cdot 2^q$ clausole, ovvero una quantità polinomiale perch $r(|x|) \in O(\log n)$.

Distinguiamo ora due casi:

1. se $x \in L$ esiste un w per cui V accetta con probabilità 1, ovvero per ogni $R \in 2^{r(|x|)} \implies \Phi_x$ ha $2^q \cdot 2^{r(|x|)}$ clausole soddisfatte da qualche assegnamento (da quello indotto da w).
2. se $x \notin L$, V accetta con probabilità $< \frac{1}{2}$ qualunque sia w

Quindi, si considera ora un certo problema L facente parte di NP, e si sfrutta la costruzione della formula appena descritta, ottenendo una certa Φ_x, questo è possibile perchè se L è in NP esiste un verificatore PCP.

Fissiamo $\bar{\epsilon} = \frac{1}{2^{q+1}}$. Supponiamo che esiste un algoritmo per MaxSat con tasso di approssimazione $(1 + \bar{\epsilon})$, sia $t^*(\Phi_x)$ la soluzione ottima dell'istanza di Sat.

Distinguiuamo ora i due casi come prima:

1. se $x \in L \implies t^*(\Phi_x) = 2^q \cdot 2^{r(|x|)}$
2. se $x \notin L \implies t^*(\Phi_x) \leq \frac{2^{r(|x|)}}{2} \cdot 2^q + \frac{2^{r(|x|)}}{2} \cdot 2^{q-1}$, ovvero, in metà dei casi risponde sì, mentre nell'altra metà sicuramente avrà un numero di clausole risolte al più di 2^{q-1}, visto che deve rispondere no, che equivale a $\leq 2^{r(|x|)} \cdot 2^q - \frac{2^{r(|x|)}}{2}$

Considerando l'algoritmo $(1+\bar{\epsilon})$-approssimante, nel primo caso, $x \in L$, individua una soluzione che corrisponde a

$$A = t(\Phi_x) \geq \frac{t^*(\Phi_x)}{1 + \bar{\epsilon}} = \frac{2^q \cdot 2^{r(|x|)}}{\frac{1}{2^{q+1}}}$$

Nel secondo caso invece, $x \notin L$

$$B = t(\Phi_x) \leq t^*(\Phi_x) \leq 2^{r(|x|)} \cdot 2^q - \frac{2^{r(|x|)}}{2}$$

Vale che $A < B$:

$$A - B = \frac{2^q \cdot 2^{r(|x|)}}{\frac{1}{2^{q+1}}} - 2^{r(|x|)} \cdot 2^q - \frac{2^{r(|x|)}}{2}$$
$$= 2^{r(|x|)} \cdot \frac{2^{q+1} - 2^{q+1}(1 + \frac{1}{2^{q+1}}) + 1 + \frac{1}{2^{q+1}}}{2(1 + \frac{1}{2^{q+1}})} > 0$$

Questo significa che l'output emesso nel caso $x \in L$, è strettamente maggiore del caso in cui $x \notin L$, questo implica che, calcolando A, si può capire, osservando l'output dell'algoritmo, a capire se $x \in L$. Ovvero in tempo polinomiale si è deciso L, assurdo. □

Osservazione 34. *La dimostrazione del lemma si può estendere a MaxE3Sat.*

3.2.2 Inapprossimabilità Max Indipendent Set

Il problema di Max Indipendent Set consiste in:

Input: $G = (V, E)$
Output: $X \subseteq V, \binom{X}{2} \cap E = \emptyset$
Costo: $|X|$
Tipo: max

Teorema 41. *Non è possibile approssimare in tempo polinomiale Max Indipendet Set a meno di* $(2 - \epsilon)$ *per* $\epsilon > 0$.

Dimostrazione. Si parte da un $L \subseteq 2^*, L \in NP$.

Per il teorema PCP, esiste un verificatore

$$V \in PCP[r(n), q], r(n) \in O(\log n), q \in \mathbb{N}$$

tale che V accetta L.

Fissato un $x \in 2^*$ simulo V per ogni stringa $R \in 2^{r(|x|)}$ e per ogni sequenza $\in 2^q$, per un totale di $2^{r(|x|)} \cdot 2^q$ simulazioni, che sono in quantità polinomiale.

Siano ora tutte le simulazioni che ritornano sì, così fatte:

$$(R, \{i_1^R = v_1, \ldots, i_q^R = v_q\})$$

Queste coppie diventano nodi di un grafo, esiste un lato tra due nodi, sse, dato

$$(R', \{i_1^R = v'_1, \ldots, i_q^R = v'_q\})$$

Non creo l'arco sse $R \neq R'$ oppure se i secondi membri sono compatibili, sarebbero incompatibili se in una stessa posizione ci fossero due v_i differenti.

Per il grafo, vale che:

1. Se $x \in L$, G_x ha un insieme indipendente di cardinalità $\geq 2^{r(|x|)}$: questo perchè se $x \in L$ esiste un $w \in 2^q$ che accetta qualunque sia R. Ma allora ci sono $2^{r(|x|)}$ coppie compatibili con w, visto che w fa accettare tutti, quindi non ci sono archi tra loro e di conseguenza vale l'osservazione.

2. Se $x \notin L$, gli insiemi indipendenti di G_x hanno cardinalità $\leq 2^{r(|x|)-1}$: supponendo che ne esista uno con cardinalità maggiore, esisterebbe un w_s tale che tutte le simulazioni rappresentate dall'insieme si verificherebbero. Questo non è corretto poichè avremmo più della metà delle simulazioni verificate.

Se ora esistesse un algoritmo A con approssimazione α, che produce una soluzione di cardinalità $|S_{G_x}| = \frac{|S^*_{G_x}|}{\alpha}$

1. $x \in L \implies |S^*_{G_x}| \geq 2^{r(|x|)} \implies |S_{G_x}| \geq \frac{2^{r(|x|)}}{\alpha}$
2. $x \notin L \implies |S^*_{G_x}| \leq \frac{2^{r(|x|)}}{2} \implies |S_{G_x}| \leq \frac{2^{r(|x|)}}{2}$

Se α fosse < 2, i due casi sarebbero disgiunti, quindi deciderei in tempo polinomiale l'appartenenza di x a L. □

4 Strutture dati

L'attenzione passa ora alle strutture dati, in particolare a quelle *space-conscious*, si studiano quindi strutture che implementano le primitive in maniera efficiente ma che occupano poco spazio.

4.1 Information theoretical lower bound

L'information theoretical lower bound indica il numero di bit minimi necessari a una struttura dati che assume particolari tipi di valori.

Teorema 42. *Una struttura dati con v valori possibili, che usi $x_1, \ldots, x_v$ bit per rappresentare tali valori, soddisfa:*

$$\frac{\sum_{i=1}^{v} x_i}{v} \geq \log_2 v$$

Dimostrazione. Se la struttura usasse meno bit, due dei suoi valori possibili avrebbero la stessa rappresentazione binaria, assurdo. □

Strutture dati space-consious Dato un abstract data type zdove Z_n è l'information theoretical lower bound per strutture di taglia n, sia D_n lo spazio medio occupato da tale struttura, ovviamente vale $D_n \geq Z_n$, tale struttura si dice:

- *Implicita* se $D_n = Z_n + O(1)$
- *Succinta* se $D_n = Z_n + o(Z_n)$
- *Compatta* se $D_n = O(Z_n)$

a patto che le primitive abbiano la stessa complessità di quelle su una struttura dati non compressa.

4.2 Rango e selezione

Dato un $b \in 2^n$, si definiscono due funzioni, di rango e selezione,

$$rank_b, select_b : \mathbb{N} \longrightarrow \mathbb{N}$$

Dove:

$$rank_b(p) = |\{i \mid i < p, b_i = 1\}|$$
$$select_b(k) = \max\{p \mid rank_b(p) \leq k\}$$

Implementazione naïve Una prima implementazione potrebbe essere quella che risponde con costo lineare alle richieste, mantenendo solo il vettore b, tale struttura è troppo inefficiente, ma sarebbe ottima per quanto riguarda la memoria, infatti equivale al theoretical lower bound.

Implementazione non space-consious Per migliorare il primo approccio, si può mantenere in memoria la funzione di rank e select associata ad ogni posizione. Tale soluzione offre primitive costanti, ma occupa troppa memoria, ovvero $O(2n \log n)$ bit.

Desiderata L'implementazione ideale consiste nell'avere una struttura succinta, ovvero che occupi $n + o(n)$ e che risponda alle query in $O(1)$.

4.2.1 Struttura di Jacobson per il rango

L'idea di base è quella di suddividere il vettore b superblocchi, lunghi rispettivamente $(\log n)^2$.
Tali superblocchi si suddividono in blocchi, lunghi $\frac{1}{2} \log n$.

Four Russian Trick Soffermandosi ora sul numero di tipi di blocchi, ne esistono rispettivamente $2^{\frac{1}{2} \log n}$.

Consideriamo ora un certo blocco, memorizzare la sua tabella di rank significherebbe avere $\frac{1}{2} \log n$ righe, ognuna con un valore che occupa $\log(\frac{1}{2} \log n)$.

Se volessimo mantenere la tabella di rank per ogni tipo di blocco, si avrebbe

$$2^{\frac{1}{2} \log n} \cdot \frac{1}{2} \log n \cdot \log(\frac{1}{2} \log n)$$
$$\leq \sqrt{n} \frac{1}{2} \log n \log \log n = o(n)$$

L'idea di mantenere in memoria tutte queste informazioni è chiamata Four Russian Trick, sostanzialmente si spezza un problema in sottoproblemi talmente piccoli, da poterli mantenere esplicitamente in memoria.

Costruzione Per costruire la struttura, memorizzo:

1. per ogni superblocco, il numero di uni prima del superblocco, indicato con $S[i]$
2. per ogni blocco, il numero di uni tra l'inizio del superblocco che lo contiene, e l'inizio del blocco, indicato con $B[i]$
3. la tabella discussa nel paragrafo precedente, indicata con TAB

Calcolo del rango Il calcolo del rank per una certa posizione, equivale a

$$rank_b(p) = S\left[\frac{p}{(\log n)^2}\right] + B\left[\frac{p}{\frac{1}{2}\log n}\right] + TAB_t\left[p \ mod \ \frac{1}{2}\log n\right]$$

dove t equivale al tipo del blocco in cui si trova p. Non è nulla di sofisticato, visto che i blocchi sono talmente piccoli che si possono usare come indici della tabella TAB.
La primitiva così ottenuta ha tempo costante.

Spazio occupato Lo spazio utilizzato dalla struttura, equivale allo spazio occupato dalle strutture ausiliarie e il vettore b, le strutture occupano:

1. S: $\frac{n}{(\log n)^2} \cdot \log n = o(n)$
2. B: $\frac{n}{\frac{1}{2}\log n} \log((\log n)^2) = o(n)$, visto che si stanno memorizzando al massimo $(\log n)^2$ uni, ovvero la dimensione del superblocco.
3. TAB: $o(n)$ per i calcoli fatti in precedenza.

Lo spazio totale è $O(n) + o(n)$, il tempo della primitiva di rango è $O(1)$. La truttura è quindi succinta.

4.2.2 Struttura di Clarke per la selezione

La creazione di una struttura efficacie per la selezione è complessa. La struttura di Clarke è divisa in livelli.

Primo livello Al primo livello, memorizzato in P si mantengono le posizioni degli uni in posizioni multiple di $\log n \log\log n$, quindi

$$P[i] = select_b(i \cdot \log n \log\log n)$$

Il vettore così creato occupa

$$\frac{n}{\log n \log\log n} \cdot \log n = o(n)$$

Secondo livello Si crea ora un secondo livello, S. Definiamo la quantità:

$$\forall i, r_i = P[i+1] - P[i] \geq \log n \log\log n$$

Si distinguono ora due livelli, in base alla densità di uni:

- Caso sparso: $r_i \leq (\log n \log\log n)^2$, in questo caso $S[i]$ contiene la lista esplicita delle posizioni intermedie, ovvero le posizioni degli uni compresi tra $P[i]$ e $P[i+1]$, memorizzate in modo relativo da $P[i]$. In questo caso la memoria occupata è

$$\log n \log\log n \cdot \log r_i \leq \frac{(\log n \log\log n)^2}{\log n \log\log n} \cdot \log r_i$$
$$\leq \frac{r_i}{\log n \log\log n} \cdot \log r_i \leq \frac{r_i}{\log n \log\log n} \cdot \log n \leq \frac{r_i}{\log\log n}$$

- Caso denso: $\log n \log\log n \leq r_i \leq (\log n \log\log n)^2$ in questo caso memorizzo le posizioni degli uni multiple di $\log r_i \log\log n$, ottengo che

$$S[i][j] = \mathit{select}_b(i \log n \log\log n + j \log r_i \log\log n)$$

  In questo caso la memoria occupata sarà:

$$\frac{\log n \log\log n}{\log r_i \log\log n} \cdot \log r_i \leq \frac{r_i}{\log r_i \log\log n} \cdot \log r_i$$
$$\leq \frac{r_i}{\log r_i \log\log n} \cdot \log r_i = \frac{r_i}{\log\log n}$$

Valutando ora lo spazio occupato da S, si ottiene che per la memoria occupata vale:

$$\leq \sum_i \frac{r_i}{\log\log n} = \sum_i \frac{P[i+1] - P[i]}{\log\log n}$$
$$= \frac{P[n] - P[0]}{\log\log n} = \frac{n}{\log\log n} = o(n)$$

Terzo livello Per gli uni del caso denso del secondo livello, ovvero quando $r_i \leq (\log n \log\log n)^2$, si crea un terzo livello. $S[i]$ contiene le posizioni multiple di $\log r_i \log\log n$. Definiamo ora

$$\forall j, \bar{r}^i_j = S[i][j+1] - S[i][j]$$

Vale che:

$$(\log n \log\log n)^2 > \bar{r}^i_j \geq \log r_i \log\log n$$

Esistono ora due casi, come in precendenza:

- Caso sparso: $\bar{r}^i_j \geq \log \bar{r}^i_j \log r_i (\log\log n)^2$.
  Memorizzo in $T[i][j]$ la lista delle posizioni degli uni, come differenza da $S[i][j]$. La memoria occupata da un elemento è:

$$(\log r_i \log\log n) \cdot \log \bar{r}^i_j = \frac{\log r_i (\log\log n)^2 \cdot \log \bar{r}^i_j}{\log\log n}$$
$$\leq \frac{\bar{r}^i_j}{\log\log n} = \frac{S[i][j+1] - S[i][j]}{\log\log n}$$

Considerando tutti gli elementi:

$$\sum_j \frac{S[i][j+1] - S[i][j]}{\log\log n} = \frac{P[i+1] - P[i]}{\log\log n} \leq \frac{n}{\log\log n}$$

- Caso denso: $\bar{r}^i_j < \log \bar{r}^i_j \log r_i (\log\log n)^2$.
  In questo caso tra $S[i][j+1]$ e $S[i][j]$ ci sono pochi zeri, si utilizza il Four-Russian trick, vale che:
  - visto che $r_i \leq (\log n \log\log n)^2$, essendo nel caso denso del secondo livello:
    $$\log \bar{r}^i_j \leq \log r_i \leq \log((\log n \log\log n)^2) =$$
    $$2\log\log n + 2\log\log\log n \leq 4\log\log n$$
  - ricordiamo che vale $\bar{r}^i_j < \log \bar{r}^i_j \log r_i (\log\log n)^2$. Per il punto precedente, $\log \bar{r}^i_j$ e $\log r_i$ sono entrambi $\leq 4\log\log n$. quindi, mettendo tutto insieme:
    $$\bar{r}^i_j \leq 16(\log\log n)^4$$

Lo spazio utilizzato dal Four-Russian trick è minore uguale di

$$\leq 2^{\bar{r}^i_j} \cdot \bar{r}^i_j \cdot \log \bar{r}^i_j$$

Ovvero il numero di tabelle, per il numero di righe per i bit per riga. Sfruttando le osservazioni precedenti:

$$\begin{aligned} 2^{\bar{r}^i_j} \cdot \bar{r}^i_j \cdot \log \bar{r}^i_j &\leq 2^{16(\log\log n)^4} \cdot 16(\log\log n)^4 \cdot \log(16(\log\log n)^4) \\ &= (\log\log n)^{4\cdot 16} \cdot 16(\log\log n)^4 \cdot \log(16(\log\log n)^4) = o(n) \end{aligned}$$

Osservazione 35. *Nonostante la struttura sia teoricamente succinta, bisogna notare come la quantità* $(\log\log n)^{4\cdot 16}$ *sia nella pratica enorme.*
La soluzione più semplice per adottare il design di Clark è saltare il Four-Russian trick e utilizzare sempre la memorizzazione esplicita, ovvero il primo punto del terzo livello.

4.3 Alberi binari

Un albero binario è definito ricorsivamente, ovvero, un albero può essere:

1. Un nodo singolo
2. Un nodo con due figli che sono alberi binari

I nodi senza figli sono detti *nodi esterni* o foglie, gli altri sono *nodi interni*.

Teorema 43. *Il numero di nodi esterni è uguale al numero di nodi interni aumentato di uno.*

$$|E| = |I| + 1$$

Dimostrazione. Su un unico nodo, il teorema vale, infatti i nodi esterni sono uno e zero quelli interni.

Su un albero T con figli, indicati con T_1 e T_2:

$$|E(T)| = |E(T_1)| + |E(T_2)| = (|I(T_1)| + 1 + |I(T_2)|) + 1$$

La quantità tra parentesi sono i nodi interni di T, ovvero quelli dei figli più se stesso, vale quindi il teorema. □

In generale si può dire che il numero di nodi totali di un albero binario è equivalente a

$$n = |E| + |I| + 2|E| - 1 = 2|I| + 1$$

Teorema 44. *Il numero di alberi binari con n nodi interni equivale a*

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Information theoretical Lower Bound Utilizzando l'*approssimazione di Stirling*

$$x! \approx \sqrt{2\pi x}\left(\frac{x}{e}\right)^x$$

Si ottiene che il numero di alberi binari con n nodi interi equivale a

$$\begin{aligned} C_n = \frac{1}{n+1}\binom{2n}{n} &= \frac{1}{n+1}\cdot\frac{(2n)!}{n!(2n-n)!} = \frac{1}{n+1}\cdot\frac{(2n)!}{(n!)^2} \\ &\approx \frac{1}{n+1}\cdot\frac{\sqrt{2\pi n}(\frac{2n}{e})^{2n}}{(\sqrt{2\pi n}(\frac{n}{e})^n)^2} = \frac{1}{n+1}\cdot\frac{\sqrt{4\pi n}(\frac{2n}{e})^{2n}}{2\pi n(\frac{n}{e})^{2n}} \\ = \frac{1}{n+1}\cdot\frac{1}{\sqrt{\pi n}}\cdot\left(\frac{2n}{n}\right)^{2n} &= \frac{1}{n+1}\cdot\frac{1}{\sqrt{\pi n}}\cdot 2^{2n} \approx \frac{4^n}{\sqrt{\pi n^3}} \end{aligned}$$

Segue che

$$Z_n = \log(4^n) = n\log 4 = 2n$$

Teorema 45. *Per rappresentare un albero binario con n nodi interni sono necessari $2n - O(\log n)$ bit.*

Numerazione nodi I nodi dell'albero si considerano numerati dalla radice per livelli, si mantiene poi un vettore b lungo $2n+1$ per un albero con n nodi interni dove una componente a uno significa che il nodo i è interno, zero che è esterno.

Definiamo ora delle primitive sull'albero:

- $isInternal(k) = b_k$
- $leftChild(k) = 2rank_b(k) + 1$
- $rightChild(k) = 2rank_b(k) + 2$
- $parent(k) = select_b(\lfloor\frac{k-1}{2}\rfloor)$

Spazio Lo spazio occupato equivale a quello delle strutture più il vettore

$$2n + o(2n)$$

Si possono considerare anche i dati per i nodi, nel caso in cui i dati siano solo in quelli interni, si possono memorizzare in un vettore e accedervi tramite indice $rank_b(k)$.

Tempo Il tempo delle primitive è costante.

Segue che la struttura per gli alberi definita in questo modo è succinta.

4.4 Sequenze monotone di Elias-Fano