Capitolo 3

Algoritmi di approssimazione

3.1 Introduzione

3.1.1 Problemi di ottimizzazione

I problemi di ottimizzazione sono definiti su insiemi di istanze e soluzioni $\mathcal{I}, \mathcal{S}$ per cui esiste una funzione di costo

$$c : \mathcal{S} \to \mathbb{R}$$

e, dato l'insieme di soluzioni ammissibili associate ad un'istanza

$$\mathcal{S}(i) = \{s \in \mathcal{S} : i \,\mathbf{\Pi}\, s\}$$

si vuole individuare la soluzione di costo massimo (o minimo)

$$s_i^* = \arg\max \{c(s) : s \in \mathcal{S}(i)\}$$

3.1.2 Approcci risolutivi

Risoluzioni esaustive

Si cerca la soluzione esatta, con una ricerca esaustiva efficiente, per esempio con le tecniche *Branch and Bound* o *Branch and Cut*.

Algoritmi pseudo polinomiali

Per alcuni problemi, se i dati fossero rappresentati con codifica unaria, l'algoritmo risolutivo sarebbe di complessità polinomiale.

I problemi Strong-NP-Hard restano NP-hard anche sotto encoding unario.

Subset Sum, per esempio, si può approcciare come problema di programmazione dinamica (ESERCIZIO). Si possono infatti definire i sottoproblemi $S_{i,j}$ dove S_i è un prefisso di S: $S_i = \{s_1, \ldots, s_i\}$ e dove $1 \leq j \leq t$. Questi sottoproblemi sono $n\,t$, ma t è il numero nell'istanza, che si rappresenta con solamente $\log t$ bit. Il numero di problemi è quindi esponenziale nella taglia dell'istanza, se $t = 2^{50}$, si genererebbe un numero enorme di problemi, ma il numero viene rappresento con 50 bit. La taglia è quindi logaritmica in t ma il numero di problemi è lineare in t. Il numero di problemi sarebbe polinomiale sotto la codifica unaria. In istanze ragionevoli dove t è piccolo (sul miliardo) si possono risolvere.

L'algoritmo di programmazione dinamica usa una proprietà di sottostruttura che permette di trovare la soluzione $S_{i,j}$ a partire da istanze precedenti $S_{i-1,j}$ con valori di j più piccoli.

Molte istanze piccole di problemi pseudo polinomiali vengono risolte usando la programmazione dinamica.

Rinunciare all'ottimalità

Si rinuncia all'ottimalità e si ottiene una soluzione che ha una relazione garantita con la soluzione ottima. Si può *quantificare* di quanto sia peggiore dell'ottimo.

3.2 Algoritmi di approssimazione

3.2.1 Definizione

Definizione 3.2.1 (Algoritmo di approssimazione)**.** Dato $\mathbf{\Pi}$ di ottimizzazione, $A_{\mathbf{\Pi}}$ è un algoritmo per $\mathbf{\Pi}$ che ritorna $A_{\mathbf{\Pi}}(i) \in \mathcal{S}(i)$ Si dice che $A_{\mathbf{\Pi}}$ è di $\rho(n)$-approssimazione per $\mathbf{\Pi}$ se $\forall i \in \mathcal{I}, |i| = n$, vale, per $\rho(n) \geq 1$

- problema di `minimo`: $\dfrac{c\left(A_{\mathbf{\Pi}}\left(i\right)\right)}{c\left(s^{*}\left(i\right)\right)} \leq \rho\left(n\right)$
- problema di `massimo`: $\dfrac{c\left(s^{*}\left(i\right)\right)}{c\left(A_{\mathbf{\Pi}}\left(i\right)\right)} \leq \rho\left(n\right)$

Dove $c\left(s^{*}\left(i\right)\right)$ è il costo della soluzione ottima. Nota: se $A_{\mathbf{\Pi}}$ risolve il problema, $\rho(n) = 1$. Si può riscrivere la maggiorazione in una singola espressione:

$$\max\left\{\frac{c\left(A_{\mathbf{\Pi}}\left(i\right)\right)}{c\left(s^{*}\left(i\right)\right)}, \frac{c\left(s^{*}\left(i\right)\right)}{c\left(A_{\mathbf{\Pi}}\left(i\right)\right)}\right\} \leq \rho\left(n\right)$$

□

Lower/Upper bound sul costo ottimo

È interessante notare che un algoritmo di approssimazione fornisca in tempo polinomiale un lower (upper) bound al costo della soluzione ottima, che non è conoscibile in tempo ragionevole.

- problema di `minimo`: $c\left(s^{*}\left(i\right)\right) \geq \dfrac{c\left(A_{\mathbf{\Pi}}\left(i\right)\right)}{\rho\left(n\right)}$
- problema di `massimo`: $c\left(s^{*}\left(i\right)\right) \leq \rho\left(n\right) \cdot c\left(A_{\mathbf{\Pi}}\left(i\right)\right)$

Tipologie di algoritmi

La $\rho(n)$-approssimazione è un concetto generale, legato alla taglia dell'istanza.

I problemi possono essere approssimati con qualità molto variabile.

Per esempio, per *Vertex Cover*, si trova un'approssimazione $\rho(n) = 2$ costante. Per *Set Cover*, la qualità dell'approssimazione è legata alla taglia $\rho(n) = \Theta(\log n)$. Per il *Travelling Salesman Problem* in versione generale, e per *Clique*, sotto l'ipotesi $\boldsymbol{P \neq NP}$, si trova un limite inferiore all'approssimazione $\rho(n) = \Omega(n^{1-\varepsilon})$.

3.2.2 Schemi di approssimazione

Definizione 3.2.2 (Schema di approssimazione)**.** L'algoritmo $A_{\mathbf{\Pi}}(i, \varepsilon)$ è uno schema di approssimazione per $\mathbf{\Pi}$ se $A_{\mathbf{\Pi}}(i, \varepsilon)$ è di $(1+\varepsilon)$-approssimazione per $\mathbf{\Pi}$ □

Definizione 3.2.3 (PTAS)**.** L'algoritmo $A_{\mathbf{\Pi}}(i, \varepsilon)$ è uno schema di approssimazione polinomiale (*polinomial time approximation scheme*) per $\mathbf{\Pi}$ se $A_{\mathbf{\Pi}}(i, \varepsilon)$ è di $(1+\varepsilon)$-approssimazione per $\mathbf{\Pi}$ e, fissato ε, ha complessità polinomiale. □

Definizione 3.2.4 (FPTAS)**.** L'algoritmo $A_{\mathbf{\Pi}}(i, \varepsilon)$ è uno schema di approssimazione pienamente polinomiale (*fully polinomial time approximation scheme*) per $\mathbf{\Pi}$ se $A_{\mathbf{\Pi}}(i, \varepsilon)$ è di $(1+\varepsilon)$-approssimazione per $\mathbf{\Pi}$ e, fissato ε, è polinomiale sia in n sia in $1/\varepsilon$. □

La complessità è quindi legata alla taglia e al fattore di approssimazione scelto, e può essere di varie forme, per esempio:
Algoritmi con complessità legata in maniera polinomiale ad ε e alla taglia n (FPTAS):

$$T_{A_{\mathbf{\Pi}}}(n, \varepsilon) = O\left(\frac{1}{\varepsilon^2}\, n^3\right)$$

se si varia l'errore relativo di un fattore k costante, la complessità cresce di un fattore costante

$$\rho = (1+\varepsilon) \rightsquigarrow \rho = \left(1 + \frac{\varepsilon}{k}\right)$$
$$T_{A_{\mathbf{\Pi}}}(n, \varepsilon) = O\left(\frac{1}{\varepsilon^2}\, n^3\right) \rightsquigarrow O\left(\frac{1}{\varepsilon^2}\, k^2\, n^3\right)$$

Altri algoritmi vedono comparire ε come esponente, in questo caso la complessità *è* polinomiale per ε fissato, ma cresce rapidamente variando ε (PTAS):

$$T_{A_{\mathbf{\Pi}}}(n, \varepsilon) = O\left(n^{1/\varepsilon}\right)$$

se si varia l'errore relativo di un fattore k

$$\rho = (1+\varepsilon) \rightsquigarrow \rho = \left(1 + \frac{\varepsilon}{k}\right)$$
$$T_{A_{\mathbf{\Pi}}}(n, \varepsilon) = O\left(n^{1/\varepsilon}\right) \rightsquigarrow O\left(n^{1/(\varepsilon/k)}\right) = O\left(\left(n^{1/\varepsilon}\right)^k\right)$$

Spesso si ottiene una complessità della forma

$$T_{A_{\mathbf{\Pi}}}(n, \varepsilon) = O\left(n^h\, 2^{(1/\varepsilon)^k}\right)$$

quando si generano in tempo polinomiale delle sottoistanze piccole (di dimensione legata a ε), che poi si risolvono esaustivamente.

3.3 Vertex cover

3.3.1 Introduzione

Dato un grafo $G = (V, E)$ non orientato, si vuole identificare il *Vertex Cover* di taglia minima, ovvero un sottoinsieme di vertici $V^* \subseteq V$ tale che ogni arco in E ha almeno un estremo in V^*.

Approccio Greedy

Un primo approccio greedy potrebbe essere quello di scegliere un nodo arbitrario, eliminare gli archi coperti e iterare fino a coprire tutti gli archi. Per provare che un algoritmo approssima male il problema, è sufficiente esibire un singolo controesempio. In questo caso se si considera una stella, l'algoritmo potrebbe selezionare tutti gli estremi e non in centro, generando un VC di taglia $n-1$, rispetto alla scelta ottima del singolo nodo centrale. Il fattore di approssimazione risulta $\rho(n) = (n-1)/1 = n-1$ che è quasi il peggiore possibile. Si può pensare di seguire una scelta greedy più astuta, per esempio scegliendo il nodo di grado massimo, ma in questo caso si può provare che risulta un algoritmo con fattore di approssimazione $\rho(n) = \log n$.

Approccio tramite costrutto polinomiale

Si segue quindi una strada differente: basandosi su qualche costrutto che si riesce a costruire in tempo polinomiale, si ottiene un *Vertex Cover*, e ne si studia la dimensione relativa all'ottimo.

Per un algoritmo di approssimazione vanno studiati

- correttezza
- complessità
- fattore di approssimazione

3.3.2 Approssimazione tramite *matching* massimale

Il *matching*, o *independent edge set*, è un insieme di archi senza vertici in comune. A partire da un *matching* massimale M (ovvero per cui se viene aggiungo un arco, non è più un *matching* valido), si costruisce un *Vertex Cover* che consiste in tutti gli estremi degli archi in M.

Algoritmo 3.11 Aprossimatore per Vertex Cover

```
1: procedure APPROX_VC(G = (V, E))
2:     V' ← ∅
3:     E' ← E
4:     while E' ≠ ∅ do
5:         * sceglie arco arbitrario {u, v} ∈ E *
6:         V' ← V' ∪ {u, v}
7:         E' ← E' − {e ∈ E' : ∃z ∈ V : (e = {u, z}) ∨ (e = {v, z})}
8:     return V'
```

Nota: alla riga 6, V' è un insieme di nodi a cui vengono aggiungi i due nodi nell'arco, che è visto come insieme di due nodi.

Complessità

La complessità di questo algoritmo è lineare, $O(|V| + |E|)$.

Correttezza

Per la correttezza, va argomentato che $V' = VC$. Questo è vero, infatti si esce dal ciclo solo quando E' è vuoto, quindi ogni arco $e = \{u, v\}$ viene eliminato, per uno di due motivi:

- è l'arco scelto arbitrariamente, quindi i suoi nodi sono inseriti in V' (riga 6)
- uno dei suoi estremi è parte dell'arco preso in considerazione, ed e viene rimosso nel clean up (riga 7)

In entrambi i casi, almeno uno degli estremi di ogni arco è in V', che è quindi un *Vertex Cover*.

Fattore di approssimazione

Il fattore di approssimazione è dato da

$$\rho = \frac{|V'|}{|V^*|} = \frac{\text{costo soluzione ammissibile ritornata}}{\text{costo soluzione ottima}}$$

di cui si vuole trovare un limite superiore.

Per quanto riguarda $|V'|$: sia $M \subseteq E$ l'insieme degli archi selezionati da $APPROX\_VC$. M è un *matching*, per cui nessuna coppia di archi selezionati condivide estremi (gli archi sono insiemi di nodi):

$$\forall e_1, e_2 \in M \rightarrow e_1 \cap e_2 = \emptyset$$

Gli archi selezionati sono quindi tutti indipendenti, e la taglia del VC selezionato V' è esattamente il doppio della taglia del *matching* (vengono scelti entrambi gli estremi di ogni arco in M)

$$|V'| = 2|M|$$

Nota: questo è un valore molto preciso del valore della funzione obiettivo che l'algoritmo ritorna, è il costo della soluzione ammissibile ritornata.

Per quanto riguarda $|V^*|$: va trovato un *lower bound* alla taglia del VC ottimo $|V^*|$ rispetto a $|M|$ (si cerca un *upper bound* di ρ). Un *matching* qualsiasi è un insieme di archi che non ha estremi in comune. Un qualsiasi *Vertex Cover* per un grafo con questo *matching*, tutti gli archi devono essere controllati. Per controllare $|M|$ archi disgiunti, serve almeno un nodo per arco. Questo vale per ogni VC costruito su un qualsiasi *matching*, e in particolare vale per il VC ottimo.

$$|V^*| \geq |M|$$

Combinando i due risultati

$$\rho = \frac{|V'|}{|V^*|} \leq \frac{2|M|}{|M|} = 2$$

In più, il *matching* trovato è massimale, per cui se gli venisse aggiunto un arco, non sarebbe un *matching* valido:

$$\nexists e \in E - M : M \cup \{e\} \text{ è un } matching$$

ossia è uno dei più grandi *matching* possibili. Se l'arco e esistesse, l'algoritmo non sarebbe terminato, perché per eliminare tutti gli archi, devono avere almeno un estremo che appartiene al *matching*.

Qualsiasi *matching* avrebbe dato un *lower bound*, il *matching* massimale serve per provare l'upper bound. Se non fosse massimale, l'unione dei nodi degli archi che compongono il *matching* non sarebbe un *Vertex Cover*.

Analisi *slack* o *tight*

Questa analisi è *slack* o *tight*? È un'analisi che non è la migliore possibile, o meglio di così questo algoritmo non può fare? Ovvero, *l'analisi* dell'algoritmo può essere migliorata, rendendo per esempio più stretti i *bound* trovati?

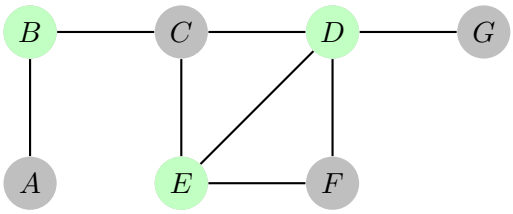
Si determina un grafo sotto cui l'algoritmo di approssimazione performa il peggio possibile e ottiene proprio il fattore di approssimazione.

Per esempio il grafo in figura 3.1a ha un *Vertex Cover* di taglia minima 3, che si può verificare con una ricerca esaustiva data la piccola taglia.

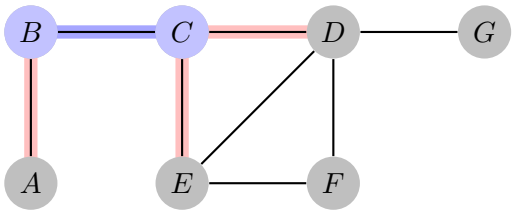
Applicando l'algoritmo di approssimazione, e scegliendo volontariamente gli archi in un ordine sfortunato, però, risulta un *Vertex Cover* da $6 = 2 \cdot 3$ elementi, per cui l'approssimazione non è migliorabile (figura 3.1).

Figura 3.1: Esempio di esecuzione pessima dell'algoritmo di approssimazione. Gli archi blu sono quelli presi in esame all'iterazione corrente, gli archi rossi sono quelli coperti dai nodi selezionati. I nodi blu sono quelli selezionati in V'.

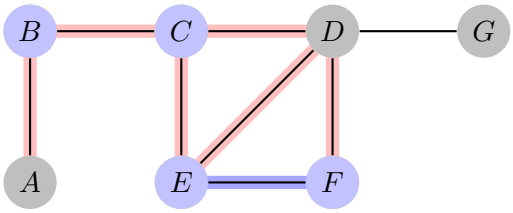
(a) VC ottimo di taglia 3 (nodi verdi)



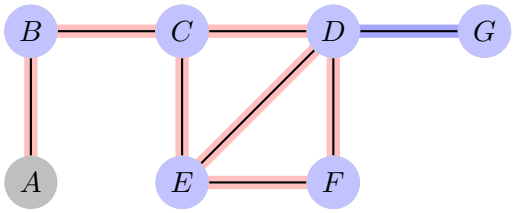
(b) Primo arco: $V' = \{B, C\}$



(c) Secondo arco: $V' = \{B, C, E, F\}$



(d) Terzo arco: $V' = \{B, C, E, F, D, G\}$



Approssimazione di $CLIQUE$

Seguendo la funzione trovata nella riduzione da $CLIQUE$ a VC, si potrebbe pensare di utilizzare il risultato ottenuto per identificare una Clique nel grafo.

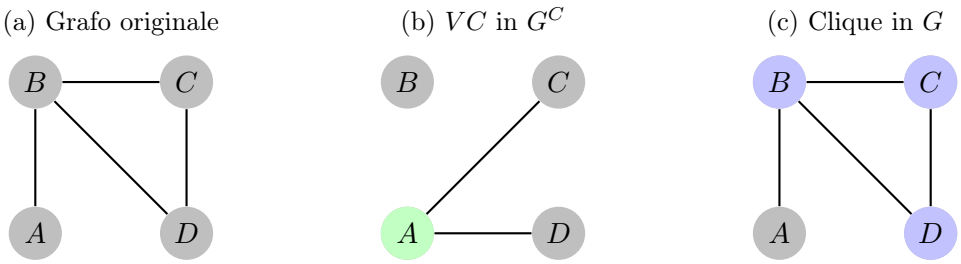
Algoritmo 3.12 Approssimatore per Clique

```
1: procedure APPROX_C(G = (V, E))
2:     G^C = (V, E^C)
3:     V' ← APPROX_VC(G^C)
4:     return V − V'
```

Questo vale se il *Vertex Cover* trovato è di dimesione massima. Se però si considera un'istanza di $CLIQUE$ $G = (V, E)$, per cui la clique massima ha dimensione $|V^*| = n/2 + 1$, il VC nel complementato avrà taglia $n - (n/2 + 1) = n/2 - 1$. L'algoritmo di approssimazione sbaglia al massimo di un fattore 2, per cui $|V'| = 2\ (n/2 - 1) = n - 2$. La clique trovata ha taglia $|V - V'| = n - (n - 2) = 2$. Il fattore di approssimazione risulta allora

$$\rho_{AC} \geq \frac{n/2 + 1}{2} \approx \frac{n}{4}$$

Figura 3.2: Trasformazione da *Vertex Cover* a Clique



Questa cattiva prestazione dell'algoritmo deriva dal fatto che la trasformazione non preserva l'approssimazione.

3.4 Travelling Salesman Problem

Si ricordano le definizioni dei problemi di $HAMILTON$ e TSP:

Hamilton:
`istanza:` $\langle G = (V, E) \rangle$
`domanda:` G contiene un ciclo semplice che tocca tutti i nodi (*tour*)?

TSP:
`istanza:` $\langle G_C = (V, E_C), c, k \rangle$
dove G_C grafo completo non orientato
$c : V \times V = E \to \mathbb{N}$
$k \in \mathbb{N}$
`domanda:` G contiene un ciclo *Hamiltoniano* di costo $\leq k$?

dove $c(\langle v_1, \cdots, v_{|V|+1} \rangle) = \sum_{i=1}^{|V|} c(v_i, v_{i+1})$

3.4.1 Inapprossimabilità del TSP generale

Se $\boldsymbol{P} = \boldsymbol{NP}$ sarebbe disponibile la soluzione esatta.
Sotto l'ipotesi $\boldsymbol{P} \neq \boldsymbol{NP}$, si mostra che se TSP fosse $\rho(n)$-approssimabile in tempo polinomiale, si potrebbe risolvere un problema $\boldsymbol{NPC}$ in tempo polinomiale, che è assurdo.

Teorema 3.4.1 (Inapprossimabilità del TSP)**.** *Sotto l'ipotesi $\boldsymbol{P} \neq \boldsymbol{NP}$, data un'istanza di TSP $\langle G = (V, E), c \rangle$, per qualsiasi $\rho(|V|)$ calcolabile in tempo polinomiale, allora il TSP non è $\rho(|V|)$-approssimabile in tempo polinomiale.*

Nota: per alleggerire la notazione, $\rho \equiv \rho(|V|)$

Dimostrazione. Per assurdo, esista un algoritmo $A_{TSP}(\langle G = c \rangle)$ di ρ-approssimazione, poliomiale. Si può usare questo algoritmo per risolvere $HAMILTON$. Si trasforma un'istanza di $HAMILTON$ in una di TSP, che può essere mandata in input ad A_{TSP}, in maniera simile alla riduzione già vista.

$$f : \langle G = (V, E) \rangle \to \langle G^K = (V, E^K), c \rangle$$

dove il grafo G^K è il grafo completato, e i costi sono associati agli archi nella maniera seguente:

$$c(e) = \begin{cases} 1 & e \in E \\ \rho \cdot |V| & e \notin E \end{cases}$$

Questa funzione è calcolabile in tempo polinomiale, infatti

- la taglia della nuova istanza è polinomiale, il grafo completo ha numero di archi quadratico
- i costi sono legati a ρ, che è polinomiale per ipotesi, $|\rho| = poly(|V|)$

"$\Rightarrow$" Se $\langle G = (V, E)\rangle \in HAMILTON$, esiste in G un cammino Hamiltoniano, che esiste anche in G^K e ha costo $|V|$, perché tutti gli archi utilizzati sono originali.
$A_\rho(\langle G^K, c\rangle)$ deve ritornare un tour che non costi più di $\rho \cdot |V|$, perché è di ρ-approssimazione. Questo costo è inferiore al costo di ogni arco spurio, quindi l'algoritmo non può selezionarne nessuno.
$A_\rho(\langle G^K, c\rangle)$ ritorna quindi un circuito Hamiltoniano di G, di costo $\leq \rho \cdot |V| + 1$.

"$\Leftarrow$" L'algoritmo deve anche riconoscere quando un'istanza è negativa.
$\langle G = (V, E)\rangle \notin HAMILTON$ implica che ogni tour in G^K deve usare almeno un arco in $E^K - E$. Allora, per ogni tour π, $c(\pi) \geq \rho \cdot |V| + 1$. Ossia $A_\rho(\langle G^K, c\rangle)$ ritorna un tour di costo $\geq \rho \cdot |V| + 1$, e quindi, combinando i due risultati:

$$\langle G\rangle \in HAMILTON \Leftrightarrow A_\rho(f(\langle G\rangle)) \text{ ritorna un tour di costo } \leq \rho \cdot |V| + 1$$

Sia A_ρ sia f sono polinomiali, quindi la loro composizione è polinomiale, e si è esibito un decisore per $HAMILTON$ polinomiale, che è in contrasto con l'ipotesi $\boldsymbol{P \neq NP}$. □

3.4.2 Risultati di inapprossimabilità

L'idea del procedimento da seguire per provare l'inapprossimabilità di un problema è quella di creare un *gap* tra il costo ottimo di istanze *associate* a istanze positive e negative di un problema $\boldsymbol{NPC}$.

Sia $\boldsymbol{\Pi}_m \subseteq \mathcal{I} \times \mathcal{S}$ un problema di minimo di cui si vuole provare l'inapprossimabilità e $\boldsymbol{\Pi}_d \in \boldsymbol{NPC}$ un problema decisionale.
Si deve trovare una funzione calcolabile in tempo polinomiale $f(x)$ che trasforma istanze di $\boldsymbol{\Pi}_d$ in istanze di $\boldsymbol{\Pi}_m$, tale che esiste una funzione $k(n)$ per cui

- se $x \in L_{\boldsymbol{\Pi}_d} \Rightarrow c\left(s^*\left(f(x)\right)\right) \leq k\left(|f(x)|\right)$
- se $x \notin L_{\boldsymbol{\Pi}_d} \Rightarrow c\left(s^*\left(f(x)\right)\right) > k\left(|f(x)|\right)$

Studiando il costo associato alla soluzione ottima dell'istanza di $\boldsymbol{\Pi}_m$, $f(x) \in L_{\boldsymbol{\Pi}_m}$ si riuscirebbe a decidere un problema $\boldsymbol{NPC}$ in tempo polinomiale.

3.5 Triangle TSP

3.5.1 Disuguaglianza triangolare

Il problema generale del TSP non è di grande utilità pratica, spesso un grafo è immerso in una metrica euclidea per cui gli archi soddisfano la disuguaglianza triangolare, ovvero i costi degli archi non possono essere arbitrari.

Definizione 3.5.1 (Disuguaglianza triangolare)**.** Una funzione di costo del tipo $c(u, v)$ intera, posivita, simmetrica, soddisfa la disuguaglianza triangolare se

$$\forall u, v, z \in V \Rightarrow c(u, v) \leq c(u, z) + c(z, v)$$

□

Anche altri spazi metrici hanno questa proprietà, per esempio lo spazio metrico indotto dai cammini minimi in un grafo.

Definizione 3.5.2 (Shortcutting)**.** In un grafo completo dove vale la disuguaglianza triangolare, si può modificare un cammino

$$\pi = \langle \cdots, u, z, v, \cdots \rangle \rightsquigarrow \pi' = \langle \cdots, u, v, \cdots \rangle$$

e vale

$$c(\pi') \leq c(\pi)$$

Questa proprietà si dice *shortcutting* (scorciatoia). □

3.5.2 Riduzione da TSP a TRIANGLE_TSP

Si modifica il problema del TSP nel problema $TRIANGLE\_TSP$:

TRIANGLE_TSP:

`istanza:`	$\langle G = (V, E^K), c, k \rangle$
dove	G grafo completo non orientato
	$c : V \times V \to \mathbb{N}$ soddisfa la disuguaglianza triangolare
	$k \in \mathbb{N}$
`domanda:`	G contiene un ciclo *Hamiltoniano* di costo $\leq k$?

Questa è una restrizione di un problema $\boldsymbol{NPC}$, è stato ristretto così tanto da diventare polinomiale?

Dimostrazione $TSP <_P TRIANGLE\_TSP$. La funzione f trasforma istanze del TSP generale in istanze in cui la funzione costo rispetta la disuguaglianza triangolare scalando tutti i costi di una costante additiva.

$$f : \langle G = (V, E^K), c, k \rangle \to \langle G = (V, E^K), c', k' \rangle$$
$$W = \max_{e \in E^K} \{c(e)\}, \quad \forall e \in E^K \quad c'(e) = c(e) + W$$

Questa funzione

- è calcolabile in tempo polinomiale: W è parte dell'istanza (che è polinomiale), e si somma il valore su n^2 archi.
- soddisfa la disuguaglianza triangolare:

$$\begin{aligned} \forall u, v, z \quad c'(u, v) &= c(u, v) + W \\ &\leq W + W \\ &\leq c(u, z) + W + c(z, v) + W \\ &\leq c'(u, z) + c'(z, v) \end{aligned}$$

- è una riduzione: ogni tour di costo T in $G = (V, E^K)$ sotto c diventa un tour di costo $T + |V|W$ in $G = (V, E^K)$ sotto c' e viceversa. Ogni tour infatti ha esattamente $|V|$ archi, quindi a prescindere dai loro costi singoli il costo del tour varia di $|V|W$. G ha un tour di costo $\leq k$ sotto c se e solo se G ha un tour di costo $\leq k + |V|W$ sotto c'. La funzione viene specificata completamente definendo $k' = k + |V|W$.

  $$f : \langle G = (V, E^K), c, k\rangle \rightarrow \langle G = (V, E^K), c', k + |V|W\rangle$$

  E si è mostrato che

  $$\langle G, c, k\rangle \in TSP \Leftrightarrow \langle G, c', k + |V|W\rangle \in T\_TSP$$

  Per cui $T\_TSP \in \boldsymbol{NPC}$.

□

3.5.3 Algoritmo di 2-approssimazione per TRIANGLE_TSP

Albero di copertura minimo

Definizione 3.5.3 (Albero di copertura)**.** Dato un grafo $G = (V, E)$ non orientato, connesso, un albero di copertura è un suo sottografo $T_G = (V, E_G)$ con $|E_G| = |V| - 1$ e T_G connesso. □

Definizione 3.5.4 (Albero di copertura minimo)**.** Sotto la funzione di costo

$$c\left(T_G\right) = \sum_{e \in E_T} c(e)$$

L'albero di copertura minimo (*Minimum Spanning Tree*) è l'albero di copertura di G di costo minimo. □

L'albero di copertura minimo si trova in tempo polinomiale, per esempio usando l'algoritmo di *Prim* [3], con complessità $O\left(|E| \log |V|\right)$, quasi lineare nel numero di archi.

1. si seleziona un nodo a caso
2. si trovano gli archi uscenti
3. si seleziona l'arco di peso minore
4. si itera dal punto 2 fino a toccare tutti i nodi

(Full) Preorder Walk

La visita in preordine dell'albero è chiaramente lineare nel numero di nodi

Algoritmo 3.13 Preorder Walk

```
1: procedure PREORDER_VISIT(T, r)
2:     PRINT(r)
3:     if not (leaf (r)) then                    ▷ il nodo corrente ha figli
4:         for all v ∈ r.children do
5:             PREORDER_VISIT(T, v)
```

La visita in preordine completa (*Full Preorder Visit*) si effettua stampando di nuovo il nodo quando si è conclusa l'esplorazione del sottoalbero corrispondente.

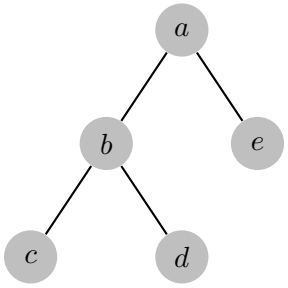
Algoritmo 3.14 Full Preorder Walk

```
1: procedure FULL_PREORDER_WALK(T, r)
2:     PRINT(r)
3:     if not (leaf (r)) then
4:         for all v ∈ r.children do
5:             FULL_PREORDER_WALK(T, v)
6:             PRINT(r)
```

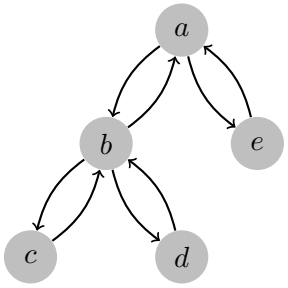
Se si considera l'albero in figura 3.3a, la visita in preordine risulta $\pi = \langle a, b, c, d, e \rangle$, mentre la visita in preordine completa è $\pi = \langle a, b, c, b, d, b, a, e, a \rangle$.

Figura 3.3: Esempio di visita in preordine completa

(a) Albero d'esempio



(b) FPW



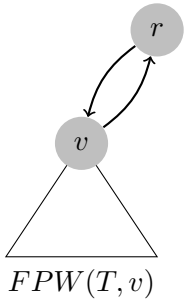
La visita in preordine completa forma un ciclo (non semplice) all'interno dell'albero (figura 3.3b).

Proposizione 3.5.1. *Il costo della FPW è due volte il costo del MST*

$$c(FPW) = 2\,c(MST)$$

Dimostrazione. La FPW utilizza ogni arco del MST esattamente due volte. La prima volta mentre sta scendendo ricorsivamente l'albero, seleziona $\{r, v\}$, e la seconda mentre risale, selezionando $\{v, r\}$, come mostrato in figura 3.4. □

Figura 3.4: Costo FPW



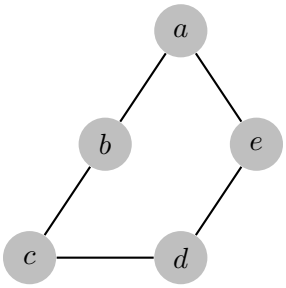
Algoritmo di approssimazione

Algoritmo 3.15 Approssimatore per Triangle TSP

```
1: procedure APPROX_T_TSP(i)
2:     T_G ← PRIM(⟨G, c⟩)
3:     π ← PREORDER_VISIT(T_G)
4:     * sia π = ⟨v_1, v_2, ··· , v_|V|⟩ *
5:     return ⟨π, v_1⟩
```

Se si considera di nuovo l'albero in figura 3.3a, l'algoritmo ritorna $\pi = \langle a, b, c, d, e, a \rangle$.

Figura 3.5: Risultato dell'algoritmo



Riguardo al fattore di approssimazione di questo algoritmo, vanno trovati *bound* per

$$\rho = \frac{|H'|}{|H^*|} = \frac{\text{costo soluzione ammissibile ritornata}}{\text{costo soluzione ottima}}$$

Dove si indica H' la soluzione trovata al problema T_TSP, H^* la soluzione ottima al problema, T^* il *minimum spanning tree* del grafo.

- Per $|H'|$: il ciclo ritornato è una sotto sequenza della *full preorder walk*, dove vengono selezionati i nodi la prima volta che vengono visti (ed aggiunta la radice).

  $$H' = \langle a, b, c, d, e, a \rangle \quad FPW = \langle \underline{a}, \underline{b}, \underline{c}, b, \underline{d}, b, a, \underline{e}, a \rangle$$

  Per la proprietà dello *shortcutting*, vale $c(H') \leq c(FPW)$ e dalla proposizione 3.5.1 si conosce $c(FPW) = 2\,c(MST) = 2\,c(T^*)$. Componendo si ottiene

  $$c(H') \leq 2\,c(T^*)$$

- Per $|H^*|$: Se dal ciclo ottimo H^* si rimuove un arco e qualsiasi, si ottiene uno *spanning tree* generico , che è di sicuro non di costo inferiore al migliore albero di copertura trovato:

  $$c(H^*) \geq c(H^* - \{e\}) = c(ST) \geq c(T^*)$$

Componendo i risultati

$$\rho = \frac{|H'|}{|H^*|} \leq \frac{2\,c(T^*)}{|H^*|} \leq \frac{2\,c(T^*)}{c(T^*)} = 2$$

3.5.4 Algoritmo di Christofides

Multigrafi e circuiti Euleriani

Definizione 3.5.5 (Multigrafo)**.** Un multigrafo $G = (V, E)$ non orientato è costruito su un insieme di nodi e un multiinsieme di archi. Ogni arco $e \in E$ può essere presente in multiple copie:

$$\text{molteplicità:} \quad m(e) = \#\text{ copie di } e$$

Il grado di un nodo deve tenere conto di tutte le copie degli archi incidenti:

$$deg(v) = \sum_{e:v \in e} m(e)$$

Cammini e cicli sono definiti come per un grafo. □

Definizione 3.5.6 (Circuito Euleriano)**.** Un circuito Euleriano è un ciclo *non semplice* che tocca tutti gli archi del grafo. Un (multi)grafo è Euleriano se ammette un circuito Euleriano. □

Definizione 3.5.7 (Grafo Euleriano)**.** Un (multi)grafo è Euleriano se ammette un circuito Euleriano. □

Teorema 3.5.2. *Un multigrafo connesso è Euleriano se e solo se tutti i vertici del grafo hanno grado pari.*

Proposizione 3.5.3. *Dato un (multi)grafo non orientato, il numeri di vertici di grado dispari è sempre pari.*

Dimostrazione. Ogni arco incide su due vertici, per cui

$$2|E| = \sum_{v \in V} deg(v) = \sum_{v \in V_{EVEN}} deg(v) + \sum_{v \in V_{ODD}} deg(v)$$

La somma dei due termini deve essere pari, il primo è pari perché somma di contributi pari, quindi anche il secondo deve essere pari. Perché una somma di contributi dispari sia pari, deve essere un numero pari di contributi. □

Definizione 3.5.8 (Matching perfetto)**.** Dato un grafo G con $|V| = 2k$ un matching perfetto è un insieme di archi $M \subseteq E$ tale che

- $\forall e_1, e_2 \in M : e_1 \cap e_2 = \emptyset$
- $\forall v \in V, \exists e \in M : v \in e$

□

Il grafo deve avere un numero pari di nodi perché ogni arco ne copre due.
Se G è pesato esiste un matching perfetto di costo minimo, che può essere individuato con l'algoritmo di *Gabow* in tempo $O(|E||V|)$, che in caso di grafo denso ($|E| = \Theta(|V|^2)$) diventa $O(|V|^3)$.

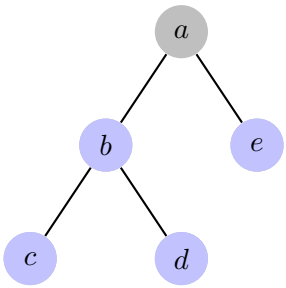
Christofides

Christofides nel 1976 [4] ha trovato un algoritmo di 3/2-approssimazione per TRIANGLE_TSP.

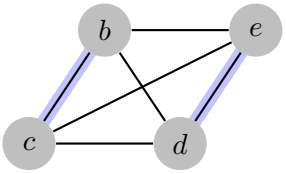
L'idea è di partire da un MST ed aggiungere archi in modo da rendere pari ogni nodo, e poi semplificare il cammino Euleriano trovato usando lo *shortcutting*, come mostrato in figura 3.6.

Figura 3.6: Algoritmo di Christofides

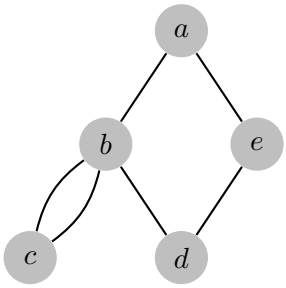
(a) T^* con i nodi dispari evidenziati



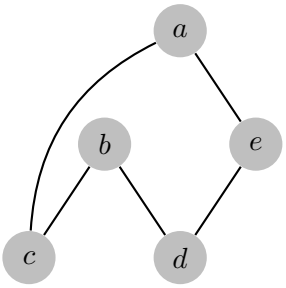
(b) Sottografo (completo) indotto dai nodi dispari, e matching perfetto di costo minimo evidenziato



(c) Ciclo Euleriano $\langle c, b, d, e, a, b, c\rangle$



(d) Dopo lo *shortcutting*: $\langle c, b, d, e, a, \not{b}, c\rangle$



Il grafo ottenuto (prima dello *shortcutting*), è

$$G' = (V,\, E_{T^*} \cup M^*)$$

È già stato mostrato nella sezione 3.5.3 che $c(T^*) \leq c(H^*)$. Resta da dimostrare che $c(M^*) \leq c\,(H^*)\,/2$. Si supponga di conoscere H^* e di aver calcolato T^*. Alcuni nodi in T^* saranno di grado pari *in* T^* (fig 3.7a). Usando lo *shortcutting*, si rimuovono questi nodi da H^* (fig 3.7c), ottenendo $H^{*'}$, con chiaramente $c(H^{*'}) \leq c(H^*)$. I nodi dispari in T^* erano in numero pari, quindi $H^{*'}$ ha un numero pari di vertici. Colorando in modo alternato gli archi del ciclo (fig 3.7d), gli archi di ciascun colore sono un matching perfetto (non ottimo): $H^{*'} = M_B \cup M_R$, il cui costo è pari al costo del ciclo $c(H^{*'}) = c(M_B) + c(M_R)$. Combinando i due vincoli sui costi si ottiene

$$c(M_B) + c(M_R) \leq c(H^*)$$

Se si indica il matching di costo inferiore con M_X, deve valere

$$c(M_X) \leq \frac{c(H^*)}{2}$$

$(x + y \leq z \rightarrow \min\{x, y\} \leq z/2)$. Il matching perfetto costruito sui nodi dispari di T^* dall'algoritmo di Christofides è quello ottimo M^*, di costo $c(M^*) \leq c(M_X)$. Combinando,

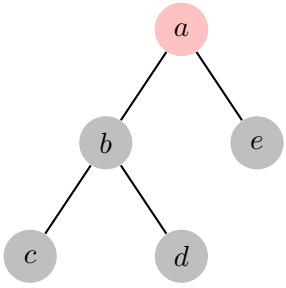
$$c(M^*) \leq \frac{c\,(H^*)}{2}$$

In conclusione

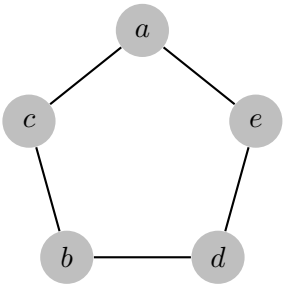
$$c(H') \leq c(E_{T^*} \cup M^*) \leq c\,(H^*) + \frac{c\,(H^*)}{2} = \frac{3}{2}\,c\,(H^*)$$

Figura 3.7: Costo del matching perfetto ottimo

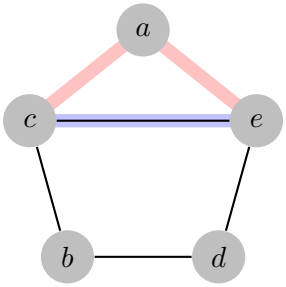
(a) T^* con i nodi pari evidenziati



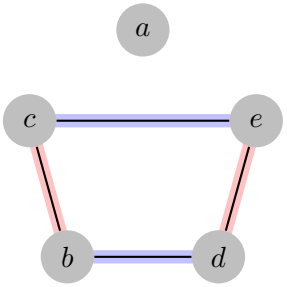
(b) Tour ottimo H^*



(c) Tour ridotto $H^{*'}$



(d) Matching M_R e M_B



3.5.5 Altri risultati sull'approssimazione di TSP

Arora ha mostrato nel 1998 [5] che $EUCLIDEAN\_TSP$ (in cui la distanza per esempio è la norma 2) ammette un algoritmo PTAS, per cui $\forall \varepsilon, \rho \leq 1+\varepsilon$ e la complessità è $O\left(n^{1/\varepsilon}\right)$.

Papadimitriou e Vempala hanno mostrato nel 2006 [6] che il $TRIANGLE\_TSP$ con archi simmetrici non può essere approssimato con un fattore migliore di $220/219$.

3.6 Set Cover

Definizione

Dato un insieme X e una famiglia di sottoinsiemi $\mathcal{F} \subseteq \{S : S \subseteq X\}$, un *set cover* C di X è un insieme di sottoinsiemi S di X, tali per cui ogni elemento di X è in almeno un sottoinsieme selezionato (chiaramente ogni elemento $x \in X$ deve appartenere ad almeno un sottoinsieme $S \in \mathcal{F}$).

$$C \subseteq \mathcal{F} : \forall x \in X, \exists S \in C : x \in S$$

Si cerca il *set cover* di taglia minima C^*.

Il problema viene posto come problema decisionale:

$$SC :$$
$$\texttt{istanza:} \quad \langle X, \mathcal{F}, k \rangle$$

$$\begin{aligned} \text{dove} \quad & \mathcal{F} \subseteq \{S : S \subseteq X\} = \mathcal{B}(X) \\ & \forall x \in X : \exists S \in \mathcal{F} : x \in S \\ \texttt{domanda:} \quad & \exists\, C \subseteq \mathcal{F} \text{ con } |C| \leq k \end{aligned}$$

Il problema è simile al *vertex cover*, ma molto più generale, si può quindi ridurre da VC a SC.

Dimostrazione. $VC <_P SC$: si esibisce una trasformazione da un problema di VC in uno di SC, rappresentando un grafo come insieme x e $\mathcal{F}$.

$$\begin{aligned} & f : \langle G = (V, E), k \rangle \in VC \to \langle X, \mathcal{F}, k \rangle \in SC \\ & X = E \\ & \mathcal{F} = \{N_v = \{e \in E : v \in e\}\, \forall v \in V\} \end{aligned}$$

Dove $\mathcal{F}$ è definito come la capacità di copertura di ogni vertice. Se si conosce un VC, è un insieme di nodi che copre ogni arco. Si selezionano i corrispettivi insiemi N_v, a ciascuno dei quali appartengono tutti gli archi coperti da v. In questo modo ogni elemento di X appartiene ad almeno un insieme N_v.
La trasformazione è lineare nella taglia dell'istanza, ed è da notare che *preserva* l'approssimazione. □

3.6.1 Approccio *Greedy*

L'approccio più intuitivo si rivela estremamente efficace: si seleziona ad ogni iterazione l'insieme $T \in \mathcal{F}$ che contiene il numero massimo di elementi ancora da coprire.

Algoritmo 3.16 Approssimatore per *Set Cover*

```
1: procedure APPROX_SET_COVER(⟨X, F⟩)
2:     U ← X                                   ▷ elementi ancora da coprire
3:     C ← ∅
4:     while U ≠ ∅ do
5:         S ← arg max {|T ∩ U| : T ∈ F}
6:         U ← U − S                           ▷ si rimuovono da U gli elementi in U ∩ S
7:         C ← C ∪ {S}
8:     return C
```

L'algoritmo è terminante, perché $\mathcal{F}$ è una famiglia di copertura, ossia ogni elemento di X appartiene ad almeno uno degli insiemi S. Almeno un elemento del residuo U è quindi nell'insieme T selezionato, e quindi almeno un elemento viene rimosso da U a ciascuna iterazione; $|U|$ decresce monotonicamente, e quando si svuota l'algoritmo termina.

L'algoritmo è corretto, perché termina proprio quando tutti gli elementi sono coperti.

L'algoritmo ha complessità ragionevole. Con un implementazione naive, il numero di iterazioni è pari a $\min\{|X|, |\mathcal{F}|\}$, perché si copre almeno un elemento per iterazione ($|X|$) e se venissero selezionati tutti i sottoinsiemi l'algoritmo terminerebbe di sicuro, perché la famiglia è di copertura ($|\mathcal{F}|$). Il calcolo dell'arg max costa $|\mathcal{F}||X|$ al caso peggiore, bisogna verificare se ogni elemento appartiene ad ogni insieme della famiglia. La complessità risulta quindi $O(\min\{|X|, |\mathcal{F}|\}|\mathcal{F}||X|)$, ovvero cubica nella taglia dell'istanza.

Con una struttura dati più sofisticata, si ottiene un'implementazione di complessità lineare dell'algoritmo $\Theta\left(\sum_{s \in S} |S|\right)$.

Fattore di approssimazione

La gloriosa analisi di Pucci risulta assai più intuitiva delle paginate di conti del Cormen.

Si definiscono:

- $n = |X|$: la taglia di X
- U_t: l'insieme di elementi ancora da coprire all'inizio dell'iterazione t (vale $U_1 = X$)
- S_t: il sottoinsieme selezionato all'iterazione t: $S \leftarrow \arg\max \{|T \cap U_t| : T \in \mathcal{F}\}$

Se si eseguono h iterazioni, allora $|C| = h$. Si studia la dinamica di $|U_t|$, che deve essere veloce, se si vuole coprire con pochi insiemi gli elementi. L'insieme X è coperto ottimamente dall'insieme ignoto $C^* = \{S_1^*, S_2^*, \cdots, S_k^*\}$, di taglia $|C^*| = k$. All'iterazione generica $t > 0$, ci sono ancora degli elementi scoperti in U_t. Esistono k sottoinsiemi di $\mathcal{F}$ che coprono U_t, e per *pigeon hole*, almeno uno di questi deve essere grande:

$$\exists S_j^* : |S_j^* \cap U_t| \geq \frac{|U_t|}{k}$$

Se tutti gli insiemi fossero di taglia minore, anche immaginandoli composti di elementi distinti, non coprirebbero U_t. Si è così trovato un rate rispetto allo stato attuale della taglia di U_t, che comporta una disequazione di ricorrenza:

$$\begin{aligned} |U_{t+1}| \leq |U_t| - \frac{|U_t|}{k} &= \left(1 - \frac{1}{k}\right)|U_t| \\ &\leq \left(1 - \frac{1}{k}\right)^2 |U_{t-1}| \leq \left(1 - \frac{1}{k}\right)^3 |U_{t-2}| \leq \dots \\ \text{(per } \textit{unfolding}) \qquad &\leq \left(1 - \frac{1}{k}\right)^{i+1} |U_{t-i}| \end{aligned}$$

Il caso base si raggiunge per $i = t - 1$

$$\begin{aligned} |U_{t+1}| &\leq \left(1 - \frac{1}{k}\right)^t |U_1| \\ \text{(vale } |U_1| = |X| = n) \qquad &= n\left(1 - \frac{1}{k}\right)^t \end{aligned}$$

Vale $1 - x \leq e^{-x}$. Per $x > 0$ la disuguaglianza è stretta. $(1 - x)^t \leq (e^{-x})^t$

$$\begin{aligned} (x = 1/k) \qquad &< ne^{-t/k} \\ \text{(Scegliendo } t = k \log n) \qquad &= ne^{-(k \log n)/k} = ne^{-n} = 1 \\ |U_{t+1}| < 1 \Rightarrow |U_{t+1}| &= 0 \end{aligned}$$

Per cui la $(t + 1)$-esima iterazione non avviene. L'insieme C ha taglia pari al numero di iterazioni, per cui

$$\begin{aligned} |C| = t \leq k \log n &= |C^*| \log n \\ \rho(n) = \frac{|C|}{|C^*|} &\leq \log n \end{aligned}$$

Fattore di approssimazione più stretto

Il fattore di approssimazione ricavato è relativo al caso peggiore. In alcune istanze particolari, però, si riesce a ricavare un *bound* molto più stretto, addirittura costante, legato ai numeri armonici.

Capitolo 5

Algoritmi randomizzati

5.1 Uso della probabilità negli algoritmi

5.1.1 Nell'analisi

Nell'analisi probabilistica lo spazio di probabilità Ω che sottende l'analisi di probabilità non sono le scelte casuali fatte dall'algoritmo, perché l'algoritmo non ne fa, ma è uno spazio di probabilità sugli ingressi dell'algoritmo.

Se gli spazi di probabilità sono suddivisi rispetto alla taglia dell'input, Ω_n è uno spazio di probabilità su $\mathcal{I}_n$, ovvero l'insieme delle istanze di taglia n.

$$\Omega_n = \mathcal{I}_n = \{i \in \mathcal{I} : |i| = n\}$$

Lo spazio delle probabilità sono tutte le possibili istanze, e ogni istanza ha una certa probabilità di accadere. Si studia l'algoritmo deterministico su una particolare distribuzione degli ingressi. Nel caso più semplice, la distribuzione è uniforme.

Se si suppone di dover studiare l'algoritmo su un input estratto a caso, ci sono parecchie variabili aleatorie rilevanti. In particolare, la probabilità di errore viene studiata in base all'istanza d'ingresso:

$$Pr\left(A \text{ sia corretto su } \boldsymbol{i}, |\boldsymbol{i}| = n\right)$$

E per esempio grazie a Pomerance, per il test di primalità vale $\lim_{n\to\infty} p_n = 1$.

Ci possono essere di casi in cui l'analisi probabilistica viene fatta sul tempo di esecuzione dell'algoritmo: la performance al caso peggiore cerca l'istanza critica per cui l'algoritmo impiega più tempo, ma si può supporre che gli input siano estratti uniformemente dallo spazio delle istanze e si vuole studiare l'algoritmo deterministico e la sua performance media su un input estratto casualmente dallo spazio.
Anche la complessità diventa allora una variabile aleatoria, e si studia il comportamento di $\boldsymbol{T}_A$ su una certa istanza (anch'essa aleatoria) $\boldsymbol{i}$:

$$Pr\left(\boldsymbol{T}_A\left(\boldsymbol{i}\right) \leq f(n),\ |\boldsymbol{i}| = n\right)$$

Che è una sorta di proprietà distribuzionale di $\boldsymbol{T}_A\left(\boldsymbol{i}\right)$.
Per esempio, quicksort è un algoritmo deterministico, ma se lo si studia su sequenze scelte a caso, si dimostra che al caso medio ha complessità $O\left(n \log n\right)$ invece di $O\left(n^2\right)$ che risulta avere al caso peggiore.

L'analisi al caso medio è proprio l'analisi probabilistica di un algoritmo deterministico. Se $\boldsymbol{i}_n$ è un'istanza di taglia n in $\mathcal{I}_n$, la media del tempo di esecuzione su istanze $\boldsymbol{i}_n$ estratte a caso.

$$\mathbb{E}\left[T_A\left(\boldsymbol{i}_n\right)\right]$$

Questa analisi fa un'assunzione molto forte, dovendo conoscere lo spazio di probabilità e la probabilità degli eventi elementari, che molto spesso si assume uniforme. Se si deve estrarre un numero primo casuale questo è ragionevole, ma se per esempio si devono ordinare dati, l'assunzione sul loro ordinamento casuale è molto forte, e in genere non accurata, dato che in pratica i dati sono spesso già ordinati a gruppi. La permutazione di ingressi quindi non è casuale, e in più quicksort (nella versione che sceglie il primo elemento come pivot) performa molto male su dati ordinati, rendendo l'analisi particolarmente inefficace.

5.1.2 Nell'algoritmo

Lo spazio di probabilità viene creato dall'algoritmo stesso: l'insieme Ω_n è formato delle scelte casuali fatte dall'algoritmo. Lo spazio non è basato sugli ingressi, perché in un algoritmo randomizzato si considera un ingresso fissato.

$$Pr\left(A \text{ sia corretto su un input } \textit{fissato}\right)$$

Scelto un input, si studia la probabilità della corettezza al caso peggiore dell'algoritmo. Si cerca quindi l'input su cui l'algoritmo sbaglia di più.

Anche in questo caso la complessità può diventare una variabile aleatoria.

$$Pr\left(\boldsymbol{T}_A\left(n\right) \geq c \cdot f(n)\right) \leq d(n)$$
$$Pr\left(\boldsymbol{T}_A\left(n\right) \leq c \cdot f(n)\right) \geq d'(n)$$

Per cui si studia la distribuzione del tempo di esecuzione.

Algoritmi Las Vegas

Gli algoritmi di tipo *Las Vegas* sono algoritmi randomizzati *sempre* corretti. Dato un algoritmo $A_{\boldsymbol{\Pi}}\left(i\right) \to s$, questo ritorna sempre la soluzione esatta.

$$Pr\left(i \,\boldsymbol{\Pi}\, s\right) = 1$$

La randomizzazione viene utilizzata per rendere aleatorio il tempo di esecuzione

$$Pr\left(\boldsymbol{T}_A\left(n\right) \leq c \cdot f(n)\right) \geq d(n)$$

Esempi di algoritmi di questo tipo sono il quicksort randomizzato o gli algoritmi di approssimazione. Quest'ultimi ritornano una soluzione che può non essere ottima, ma è sempre ammissibile.

Algoritmi Monte Carlo

Gli algoritmi di tipo *Monte Carlo* sono algoritmi randomizzati che *possono* sbagliare. Dato un algoritmo $A_{\boldsymbol{\Pi}}\left(i\right) \to s$, si studia la sua probabilità di correttezza $p_c(n)$:

$$Pr\left(i \,\boldsymbol{\Pi}\, s\right) = p_c(n)$$

Anche per algoritmi di questo tipo si analizza il tempo di esecuzione

$$Pr\left(\boldsymbol{T}_A\left(n\right) \leq c \cdot f(n)\right) \geq d(n)$$

e ci sono casi in cui questo è deterministico, perché il numero di iterazioni non è legato alle scelte casuali effettuate e vale $d(n) = 1$.

Algoritmi Monte Carlo decisionali

Gli algoritmi *Monte Carlo* decisionali possono commettere errori di due tipi:

- *one-sided*: possono compiere errori solo su uno dei due *outcome*
- *two-sided*: possono compiere errori su entrambi gli *outcome*

5.1.3 Bound in alta probabilità

Limite alla complessità in alta probabilità

Definizione 5.1.1 (Complessità in alta probabilità)**.** Dato $\mathbf{\Pi} \subseteq \mathcal{I} \times \mathcal{S}$, un algoritmo randomizzato $A_{\mathbf{\Pi}}$ ha complessità $O(f(n))$ con alta probabilità se $\exists c, d > 0$ costanti, $\exists n_0$ tali che $\forall n \geq n_0$, $\forall i \in \mathcal{I}_n$ vale

$$Pr\left(T\left(A_{\mathbf{\Pi}}(i)\right) \geq cf(n)\right) \leq \frac{1}{n^d}$$

o alternativamente

$$Pr\left(T\left(A_{\mathbf{\Pi}}(i)\right) \leq cf(n)\right) \geq 1 - \frac{1}{n^d}$$

□

Note:
d può essere una costante qualsiasi, anche minore di 1, perché è comunque sufficiente a indicare che c'è una sostanziale diminuzione della probabilità all'aumentare della taglia. Entrambi i metodi per definire l'alta probabilità delimitano la massa della probabilità della coda dell'istanza
L'analisi è al caso peggiore: l'istanza non è scelta casualmente ma è fissata. La probabilità invece è la probabilità sullo spazio Ω_n delle scelte fatte dall'algoritmo.

Correttezza in alta probabilità

Definizione 5.1.2 (Correttezza in alta probabilità)**.** Dato $\mathbf{\Pi} \subseteq \mathcal{I} \times \mathcal{S}$, un algoritmo randomizzato $A_{\mathbf{\Pi}}$ è corretto in alta probabilità se $\exists d > 0$ costante, $\exists n_0$ tali che $\forall n \geq n_0$, $\forall i \in \mathcal{I}_n$ vale

$$Pr\left(A_{\mathbf{\Pi}}(i) \not\mathbf{\Pi} i\right) \leq \frac{1}{n^d}$$

□

Note:
È un'analisi di probabilità che mostra che l'algoritmo è corretto quasi sempre.
È una caratterizzazione molto più forte della media: la massa di probabilità della coda della distribuzione ha misura che tende a 0.
Il valore di d può spesso essere aumentato molto con facilità: nel caso per esempio di Miller-Rabin, è sufficiente un valore delle iterazioni pari a

$$s = \log_2 n \rightarrow Pr\left(errore\right) = \frac{1}{2^s} = \frac{1}{n}$$

per ottenere la correttezza in alta probabilità. Aumentando di un fattore costante il numero di iterazioni la probabilità scende però esponenzialmente

$$s = k \log_2 n \rightarrow Pr\left(errore\right) = \frac{1}{2^s} = \frac{1}{n^k}$$

5.2 Concentration bound per variabili aleatorie

5.2.1 Disuguaglianze di Markov

Per ogni variabile aleatoria intera positiva $\boldsymbol{T}$ vale $\forall t$:

$$Pr\left(\boldsymbol{T} \geq t\right) \leq \frac{\mathbb{E}\left[\boldsymbol{T}\right]}{t}$$

Se in più $\boldsymbol{T}$ è limitata superiormente da b con probabilità 1: $Pr\left(\boldsymbol{T} > b\right) = 0$ vale

$$\mathbb{E}\left[\boldsymbol{T}\right] \leq t + (b - t)\, Pr\left(\boldsymbol{T} \geq t\right)$$

Confronto tra analisi al caso medio e in alta probabilità

Se, come è ragionevole supporre in molti casi, l'algoritmo randomizzato ha limite superiore deterministico alla complessità al caso peggiore

$$Pr\left(T_A(n) \leq cn^{\alpha}\right) = 1$$

(per esempio per quicksort randomizzato, anche nel caso peggiore di scelte più sfortunate, la complessità è al massimo quadratica. Questa analisi lasca mette comunque un limite all'esecuzione, che non può diventare esponenziale).
e si può scegliere $d = d_c$ come funzione crescente di c (si può aumentare il grado del polinomio aumentando la constante moltiplicativa davanti alla complessità),

$$Pr\left(T_A(n) \geq cf(n)\right) \leq \frac{1}{n^d}$$

sotto queste condizioni si dimostra che l'analisi al caso medio è meno informativa dell'analisi in alta probabilità.

Usando la seconda disuguaglianza di Markov con $b = n^{\alpha}$ e $t = cf(n)$, con $c : d \geq \alpha$:

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{T}_A(n)\right] &\leq cf(n) + \left(n^{\alpha} - cf(n)\right)\frac{1}{n^d} \\
&\leq cf(n) + \frac{n^{\alpha}}{n^d} \\
&\leq cf(n) + 1
\end{aligned}$$

5.2.2 Bound di Chernoff

Data una generica famiglia di variabili indicatrici

$$X_i = \begin{cases} 1 & \text{con probabilità } p_i \\ 0 & \text{con probabilità } 1 - p_i \end{cases}$$

Le variabili sono mutualmente indipendenti, e vale $\mathbb{E}\left[X_i\right] = p_i$. La variabile di interesse è la somma delle variabili indicatrici

$$X = \sum_{i=1}^{n} X_i$$

Per cui vale

$$\mathbb{E}\left[X\right] = \sum_{i=1}^{n} p_i = \mu$$

Lemma 5.2.1 (Chernoff). *Siano $X_1, \cdots, X_n$ variabili indicatrici con $p_i = Pr(X_i = 1)$. Data $X = \sum_{i=1}^{n} X_i$ e $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} p_i$, $\forall \delta > 0$ vale*

$$Pr(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Note:
Questo lemma fornisce un bound molto più stretto di Markov, infatti la frazione è un numero minore di 1 ($(1+\delta)^{(1+\delta)}$ cresce molto più velocemente di e^{δ}) e μ è legato al numero di variabili indicatrici, quindi c'è un bound esponenziale, mentre la prima disuguaglianza di Markov risulta

$$Pr(X > (1+\delta)\mu) < \frac{\mu}{(1+\delta)\mu} = \frac{1}{1+\delta}$$

che non ha alcun legame con la taglia.

Dimostrazione. Si considera la famiglia di variabili aleatorie

$$Y_t = e^{tX}$$

e ne si studia la media per trovarne una correlazione con la media di X. Per prima cosa si mostra che

$$\begin{aligned}
\mathbb{E}\left[e^{tX}\right] &= \mathbb{E}\left[e^{t\sum_{i=1}^{n} X_i}\right] \\
&= \mathbb{E}\left[e^{\sum_{i=1}^{n} tX_i}\right] \\
&= \mathbb{E}\left[\prod_{i=1}^{n} e^{tX_i}\right]
\end{aligned}$$

le variabili e^{tX_i} sono indipendenti, perché le X_i sono indipendenti e la funzione è invertibile, per cui la media del prodotto è il prodotto delle medie

$$= \prod_{i=1}^{n} \mathbb{E}\left[e^{tX_i}\right]$$

vale per una singola variabile $\mathbb{E}\left[e^{tX_i}\right] = e^{t\cdot 0}(1-p_i) + e^{t\cdot 1}p_i = 1 + p_i\left(e^t - 1\right)$

$$= \prod_{i=1}^{n} \left(1 + p_i\left(e^t - 1\right)\right)$$

considerando la serie di Taylor troncata vale $e^y > 1 + y$

$$\begin{aligned}
&< \prod_{i=1}^{n} e^{p_i\left(e^t-1\right)} \\
&= e^{\sum_{i=1}^{n} p_i\left(e^t-1\right)} \\
&= e^{\left(e^t-1\right)\sum_{i=1}^{n} p_i} \\
&= e^{\left(e^t-1\right)\mu}
\end{aligned}$$

Si sfrutta questo risultato per provare il lemma

$$Pr(X > (1+\delta)\mu) = Pr(tX > t(1+\delta)\mu)$$

$$= Pr\left(e^{tX} > e^{t(1+\delta)\mu}\right)$$
$$= Pr\left(Y_t > e^{t(1+\delta)\mu}\right)$$

e per la prima disuguaglianza di Markov

$$< \frac{\mathbb{E}\left[Y_t\right]}{e^{t(1+\delta)\mu}}$$
$$= \frac{\mathbb{E}\left[e^{tX}\right]}{e^{t(1+\delta)\mu}}$$

e per il risultato precedente

$$< \frac{e^{\left(e^t-1\right)\mu}}{e^{t(1+\delta)\mu}}$$

questa è una famiglia di limiti superiori, valida $\forall t > 0$, di cui si seleziona il più stretto che risulta: $t_{min} = \ln(1+\delta)$

$$< \frac{e^{\left(e^{\ln(1+\delta)}-1\right)\mu}}{e^{\ln(1+\delta)(1+\delta)\mu}}$$

chiaramente $e^{\ln(1+\delta)} = (1+\delta)$

$$= \frac{e^{(1+\delta-1)\mu}}{(1+\delta)^{(1+\delta)\mu}}$$

numeratore e denominatore sono elevati alla μ, e si conclude

$$= \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

□

Corollario 5.2.1.1. *Per $0 < \delta < 1$ valgono i seguenti bound*

$$Pr\left(X > (1+\delta)\mu\right) < e^{-\delta^2\mu/3}$$
$$Pr\left(X < (1-\delta)\mu\right) < e^{-\delta^2\mu/2}$$

Per cui i valori di X sono estremamente concentrati intorno alla media. Se si ricava un valore di d per cui $\delta^2\mu/2 > d\log n$ si ottiene l'alta probabilità

$$Pr\left(X < (1-\delta)\mu\right) < e^{-\delta^2\mu/2} < \frac{1}{n^d}$$

Se quello scostamento dalla media rappresenta l'evento "cattivo", in questo modo si riesce a limitare la probabilità della sua occorrenza con un inverso di un polinomio in n.

5.3 Analisi di Quicksort randomizzato

5.3.1 Implementazione

L'algoritmo di quicksort viene definito su un insieme S di elementi distinti. Questo non comporta perdita di generalità, perché eventuali chiavi ripetute si possono arricchire con la posizione iniziale nella lista per sciogliere i conflitti.

Si definiscono gli elementi ordinati come *order statistics*:

$$\textsc{Sort}(S) = \langle x_1, \cdots, x_n \rangle$$

L'algoritmo è quello standard:

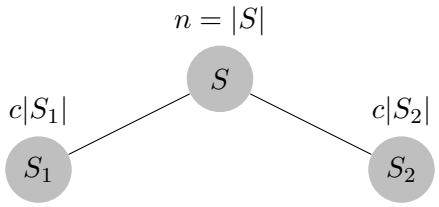
Algoritmo 5.36 Template per Quicksort

```
1: procedure QUICKSORT(S)
2:     if |S| ≤ 1 then                                  ▷ BASE
3:         return S
4:     y ← CHOOSE_PIVOT(S)                              ▷ DIVIDE
5:     S_1 ← {s ∈ S : s < y}
6:     S_2 ← {s ∈ S : s > y}
7:     X_1 ← QUICKSORT(S_1)                             ▷ RECURSE
8:     X_2 ← QUICKSORT(S_2)
9:     return ⟨X_1, y, X_2⟩                             ▷ CONQUER
```

Alla riga 4 si effettua la scelta del pivot, che comporta versioni diverse dell'algoritmo. Il costo del divide non è trascurabile, in particolare alcuni tipi di scelta sono molto costosi, ma in ogni caso lineari nella taglia. Anche la divisione in insiemi è lineare. Il conquer ha costo fisso.

L'albero delle ricorsioni ha struttura simile a prescindere dalla scelta. Ad ogni livello,

Figura 5.1: Albero delle ricorsioni di Quicksort



S_1 e S_2 sono una *sottopartizione* di S, quindi la somma delle taglie è minore di n ad ogni livello: $\sum |S_j^l| \leq n$, e ogni livello non può eseguire più di cn lavoro. Allora, se h è l'altezza massima dell'albero delle chiamate, il tempo di esecuzione è limitato da

$$T_{QS}(n) \leq cnh$$

L'altezza h dipende dalla scelta del pivot eseguita.

Scelta del primo elemento come pivot

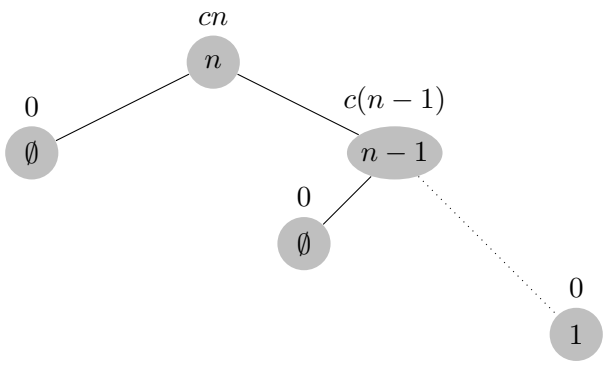
Se la scelta del pivot viene implementata come

```
1: procedure CHOOSE_PIVOT(S)
2:     return s_1
```

il caso peggiore avviene quando l'insieme è già ordinato. In questa situazione, $S_1 = \emptyset$ e $S_2 = S - \{s_1\}$. Sommando tutti i work eseguiti nell'albero in Figura 5.2 si ottiene

$$T_{QSfirst}(n) = c\frac{n(n-1)}{2} = \Theta\left(cn^2\right)$$

Figura 5.2: Albero delle ricorsioni nel caso peggiore



Scelta della mediana come pivot

Se la scelta del pivot viene implementata come

1: **procedure** CHOOSE_PIVOT(S)
2: return $x_{\lfloor n/2 \rfloor + 1}$

Dove viene ritornata l'order statistic centrale, si ottiene l'albero più bilanciato possibile, dove la taglia dell'insieme dimezza ad ogni livello. Ci sono quindi $h = \log n$ livelli, per una complessità pari a

$$T_{QSmedian}(n) \leq cnh = O(n \log n)$$

Selezionare la mediana però è molto costoso, esiste un algoritmo deterministico lineare per calcolarla, ma la costante è molto alta (circa $80n$), ed è quindi inutilizzabile in pratica.

5.3.2 Scelta casuale del Pivot

Profondità dell'albero

Supponendo di riuscire a garantire una taglia degli insiemi generati limitata da

$$|S_1|, |S_2| \leq \frac{3}{4}t$$

e ricordando il legame tra le taglie, un limite superiore implica anche un limite inferiore:

$$|S_1| = n - 1 - |S_2|$$

In questo caso, la taglia del primo livello è

$$|S_0| = n$$
$$|S_{i+1}| \leq \frac{3}{4}|S_i|$$

per unfolding:

$$|S_i| \leq \left(\frac{3}{4}\right)^i |S| = \left(\frac{3}{4}\right)^i n$$

La taglia allora decresce esponenzialmente, e il numero di livelli risulta

$$h = O\left(\log_{4/3} n\right)$$

Scelta casuale del pivot

La scelta casuale del pivot riesce a garantire questa proprietà delle taglie dei sottoinsiemi in alta probabilità.

```
1: procedure CHOOSE_PIVOT(S)
2:     return RANDOM(S)
```

Si consideri l'insieme degli elementi "fortunati"

$$F = \left\{ X_i : i \in \left\{ \left\lfloor \frac{n}{4} + 1 \right\rfloor, \cdots, \left\lceil \frac{3}{4}n + 1 \right\rceil \right\} \right\}$$

Questo insieme contiene circa la metà degli elementi, per cui la probabilità di selezionare un elemento in questo insieme è circa un mezzo.
Se viene scelto come pivot un elemento di questo insieme, entrambi i sottoinsiemi hanno taglia inferiore ai 3/4 dell'insieme originale, infatti, il primo insieme deve contenere almeno il primo quarto degli elementi

$$|S_1| \geq \left\lfloor \frac{n}{4} + 1 \right\rfloor \quad \rightarrow \quad |S_2| \leq n - 1 - \left\lfloor \frac{n}{4} + 1 \right\rfloor \leq \frac{3}{4}n$$

e analogamente, il secondo insieme deve contenere almeno l'ultimo quarto di elementi

$$|S_2| \geq n - \left\lceil \frac{3}{4}n \right\rceil \quad \rightarrow \quad |S_1| \leq n - 1 - \left(n - \left\lceil \frac{3}{4}n \right\rceil \right) \leq \frac{3}{4}n$$

Per cui nel corso delle scelte del pivot, circa una volta su due viene effettuata una scelta fortunata. L'intuizione è che in media, un cammino non potrà essere lungo più di due volte il cammino più corto possibile. Lungo un cammino da radice a foglia, un nodo su due garantisce una riduzione di 3/4 della taglia dell'insieme. Questo garantisce un numero logaritmico di livelli, che corrisponde a un algoritmo $n \log n$ con alta probabilità, con una scelta del pivot che si può realizzare in tempo costante.

Analisi della profondità

Fissato $t = a \log_{4/3} n$ come target di altezza dell'albero, si vuole determinare la probabilità dell'evento negativo *any bad path*

$$ABP = \text{un arbitrario cammino è lungo più di } a \log_{4/3} n$$

Riguardo al numero di cammini distinti, se nell'albero delle ricorsioni si eliminano gli insiemi vuoti, ne esiste uno per foglia dell'albero, ed è presente una foglia per ogni elemento. Ci sono allora n cammini radice-foglia nell'albero, ciascuno individuato da un valore x_i. Si studia allora la probabilità dell'evento *fixed bad path*, per un cammino fissato:

$$\begin{aligned} FBP = &\text{ un cammino } \textit{fissato} \text{ da radice a foglia nell'albero} \\ &\text{della ricorsione ha profondità maggiore di } a \log_{4/3} n \end{aligned}$$

Se si conosce la probabilità per un cammino fissato, si limita facilmente la probabilità che *nessun* cammino sia più lungo di t sfruttando l'*union bound*. Infatti, una volta nota la probabilità su un singolo evento negativo, la probabilità che un *qualche* cammino sia molto lungo è minore o uguale alla somma delle probabilità associate a ogni singolo cammino. Con lo *union bound* si ricava quindi la probabilità che l'evento negativo *Any Bad Path* accada.

Perché capiti l'evento "cammino più lungo di t", nei primi $a \log_{4/3} n$ nodi del cammino, devono verificarsi *meno* di $\log_{4/3} n$ scelte fortunate del pivot. Infatti, ogni scelta fortunata abbassa di 3/4 gli elementi, per cui dopo $\log_{4/3} n$ si sarebbe arrivati alla foglia. Ma il cammino in considerazione *non* era terminato, quindi devono essere capitate meno di $\log_{4/3} n$ scelte fortunate.
Questo si formalizza utilizzando le variabili indicatrici

$$X_i = \begin{cases} 1 & \text{scelta fortunata al livello } i \\ 0 & \text{altrimenti} \end{cases}$$

Le scelte ad ogni livello sono chiaramente indipendenti, e vale

$$Pr\left(X_i = 1\right) \geq \frac{1}{2}$$

Si può allora studiare la somma nei primi t livelli dell'albero, che rappresenta il numero di scelte fortunate compiute

$$X = \sum_{i=1}^{a \log_{4/3} n} X_i \quad \rightarrow \quad \mu = \mathbb{E}\left[X\right] = \frac{1}{2} a \log_{4/3} n = \frac{t}{2}$$

Per applicare il bound di Chernoff, va riscritto il vincolo su X in funzione di $\delta < 1$ e μ

$$Pr\left(X < \log_{4/3} n\right) \quad \leftrightarrow \quad Pr\left(X < (1-\delta)\,\mu\right)$$

I valori corretti sono $a = 8$ e $\delta = 3/4$, ottenuti portando avanti l'analisi simbolica, e ricavando il valore necessario in modo che

$$\begin{aligned} t = 8 \log_{4/3} n \quad &\rightarrow \quad \mu = 4 \log_{4/3} n \\ \delta = \frac{3}{4} \quad &\rightarrow \quad 1 - \delta = \frac{1}{4} \\ (1-\delta)\,\mu &= \log_{4/3} n \end{aligned}$$

Si può quindi procedere applicando il corollario del lemma di Chernoff

$$\begin{aligned} Pr\left(X < \log_{4/3} n\right) &= Pr\left(X < \left(1 - \frac{3}{4}\right) 4 \log_{4/3} n\right) \\ &\leq e^{-\left(\frac{3}{4}\right)^2 4 \log_{4/3} n / 2} \\ &= e^{-\frac{9}{8} \log_{4/3} n} \\ &< e^{-\log_{4/3} n} \\ &= e^{-\ln n / \ln(4/3)} \\ &= \frac{1}{n^{\frac{1}{\ln(4/3)}}} \\ &\approx \frac{1}{n^{3{,}47}} \\ &< \frac{1}{n^3} \end{aligned}$$

Riassumendo, l'evento "cattivo" per cui l'albero ha altezza maggiore di $a \log_{4/3} n$ accade con probabilità

$$Pr\left(\text{l'albero ha altezza} > 8 \log_{4/3} n\right) \leq$$

$$
\begin{aligned}
&= Pr\left(\text{esiste } \textit{un} \text{ cammino di lunghezza} > 8\log_{4/3} n\right) \\
&\leq nPr\left(\text{un cammino } \textit{fissato} \text{ ha lunghezza} > 8\log_{4/3} n\right) \\
&\leq nPr\Big(\text{ci sono meno di } \log_{4/3} n \text{ scelte fortunate} \\
&\qquad\qquad \text{nei primi } a\log_{4/3} n \text{ nodi del cammino}\Big) \\
&\leq n\frac{1}{n^3} = \frac{1}{n^2}
\end{aligned}
$$

Per cui in alta probabilità l'albero della ricorsione di quicksort randomizzato ha un numero logaritmico di livelli, e ogni livello contribuisce al lavoro complessivo $O(n)$, per cui in alta probabilità quicksort esegue in tempo $O(n \log n)$.

Per gli algoritmi randomizzati si scompone l'analisi in una serie di eventi che possono accadere o meno, e poi si studia l'evento buono (o cattivo) in funzione di questi singoli eventi elementari che catturano le varie scelte fatte dall'algoritmo. L'analisi deterministica è simile, si scompone l'algoritmo in passi elementari e si trova qual è la sequenza peggiore. Piuttosto che il caso peggiore, qua si studia la probabilità che questo avvenga, scomponendo l'intera computasione in una serie di eventi da comporre tra di loro in maniera da poter usare i bound noti.

Analisi al caso medio

Utilizzando la disuguaglianza di Markov si passa facilmente dall'alta probabilità al caso medio.

La variabile $T(n)$ che rappresenta il tempo di quicksort randomizzato è limitata

$$Pr\left(T(n) \leq n^2\right) = 1$$

perché anche nel caso di scelte sempre sfortunate, termina in tempo quadratico. Il bound in alta probabilità è noto

$$Pr\left(T(n) \geq 8n\log_{4/3} n\right) < \frac{1}{n^2}$$

per cui si può applicare la disuguaglianza di Markov per variabili limitate con $b = n^2$ e $t = 8n\log_{4/3} n$

$$
\begin{aligned}
\mathbb{E}[T] &< t + (b-t)\,Pr(T > t) \\
&= 8n\log_{4/3} n + \left(n^2 - 8n\log_{4/3} n\right)\frac{1}{n^2} \\
&< 8n\log_{4/3} n + 1
\end{aligned}
$$

L'analisi randomizzata si può stringere molto, e risulta quasi lineare in $n \log n$.

In pratica quicksort è uno degli algoritmi di sorting più efficienti: la scelta del pivot è molto semplice, e anche la partizione può essere effettuata in maniera veloce. Non si usa solo nei sistemi con cache molto profonde, dove *k-way merge sort* riesce a sfruttare meglio la cache, generando meno miss.

5.4 Amplificazione della probabilità

Si consideri un algoritmo *Monte Carlo* decisionale $A_{\mathbf{\Pi}}$, che compie *two-sided error*, caratterizzato da

$$Pr\left(i\,\mathbf{\Pi}\,A_{\mathbf{\Pi}}\right) = \frac{1}{2} + \varepsilon \quad \text{con} \quad 0 < \varepsilon \leq \frac{1}{2}$$

Questa minima deviazione dalla scelta casuale è sufficiente per costruire un algoritmo decisionale di precisione arbitraria: secondo il *majority protocol*, si possono compiere un numero dispari di esecuzioni dell'algoritmo, e scegliere come corretta la risposta che viene restituita il numero maggiore di volte. Si applica quindi un algoritmo deterministico su una *black box* randomizzata.

Algoritmo 5.37 Majority protocol per amplificare la probabilità

```
1: procedure M_A(i, k)
2:     ANS[0] ← 0
3:     ANS[1] ← 0
4:     for i ← 1 to 2k − 1 do
5:         r ← A_Π(i)
6:         ANS[r] ← ANS[r] + 1
7:     if ANS[0] ≥ k then
8:         return ANS[0]
9:     else
10:        return ANS[1]
```

La probabilità di errore $Pr\left(i\not\Pi MA_{\mathbf{\Pi}}\right)$ si studia con la famiglia di variabili indicatrici

$$X_i = \begin{cases} 1 & \text{se } i\text{-esima iterazione ha risultato corretto} \\ 0 & \text{altrimenti} \end{cases}$$

con chiaramente probabilità di successo della singola iterazione pari a

$$Pr\left(X_i = 1\right) = \frac{1}{2} + \varepsilon$$

di cui si studia la somma

$$X = \sum_{i=1}^{2k-1} X_i = \#\text{ esecuzioni corrette di } A_{\mathbf{\Pi}} \text{ all'interno di } MA_{\mathbf{\Pi}}$$

e l'algoritmo $MA_{\mathbf{\Pi}}$ è corretto quando $X \geq k$. La media di X vale

$$\mu = \mathbb{E}\left[X\right] = (2k-1)\left(\frac{1}{2} + \varepsilon\right) = k - \frac{1}{2} + 2k\varepsilon - \varepsilon$$

Per poter studiare $Pr\left(X < k\right)$ con i lemmi di Chernoff, occorre riscrivere k in funzione di μ e δ, in modo che $Pr\left(X < (1-\delta)\,\mu\right)$.

$$k = \mu + \frac{1}{2} + \varepsilon - 2k\varepsilon$$

vale $2k\varepsilon \geq \varepsilon\mu$:

$$\begin{aligned} \varepsilon\mu &= \varepsilon\,(2k-1)\left(\frac{1}{2} + \varepsilon\right) \\ &= (2k-1)\left(\frac{\varepsilon}{2} + \varepsilon^2\right) \\ &\text{per } 0 < \varepsilon \leq 1/2 \text{ vale } \varepsilon^2 \leq \varepsilon/2 \\ &\leq 2k\left(\frac{\varepsilon}{2} + \frac{\varepsilon}{2}\right) = 2k\varepsilon \end{aligned}$$

$$\leq \mu + \frac{1}{2} + \varepsilon - \varepsilon\mu$$
$$\leq \mu - \varepsilon\mu + 1$$
$$\leq (1-\varepsilon)\,\mu + 1$$

Per cui risulta

$$Pr\left(i \not\in MA_{\mathbf{\Pi}}\right) = Pr\left(X < k\right)$$
$$= Pr\left(X \leq k-1\right)$$
$$\leq Pr\left(X \leq (1-\varepsilon)\,\mu\right)$$

applicando il lemma di Chernoff

$$\leq e^{-\frac{\varepsilon^2}{2}\mu}$$

si può esprimere come bound su k

$$\mu = (2k-1)\left(\frac{1}{2} + \varepsilon\right)$$
$$\geq (2k-1)\,\frac{1}{2}$$
$$= k - \frac{1}{2}$$
$$> k - 1$$
$$\text{e per } k \geq 2 \text{ vale}$$
$$\geq \frac{k}{2}$$
$$< e^{-\frac{\varepsilon^2}{4}k}$$

definendo $c_\varepsilon = \varepsilon^2/4$ la probabilità di errore vale

$$< e^{-c_\varepsilon k}$$

È allora sufficiente scegliere il numero di iterazioni pari a

$$k = \frac{\ln|i| \cdot d}{c_\varepsilon}$$

per ottenere il bound in alta probabilità sull'errore dell'algoritmo

$$Pr\left(i \not\in MA_{\mathbf{\Pi}}\right) < e^{-c_\varepsilon k} = \frac{1}{(|i|)^d}$$

È da notare che il numero di iterazioni necessarie è solamente logaritmico nella taglia dell'istanza, ed è quadratico (tramite c_ε) con ε. Se l'algoritmo $A_{\mathbf{\Pi}}$ è poco affidabile, sono necessarie molte iterazioni.

5.5 Problema del min-cut

5.5.1 Multigrafi e tagli

Si indica un multigrafo con $\mathscr{G} = (V, E)$, dove V è l'insieme dei vertici ed E è il multiinsieme degli archi, che si indicano anche come $V(\mathscr{G})$ ed $E(\mathscr{G})$.
Il grado di un nodo è pari al numero di archi incidenti sul nodo, contati con la loro molteplicità:

$$d(v) = |\{\!\{\{v, x\} \in E\}\!\}|$$

Definizione 5.5.1 (Edge cut)**.** Un taglio (sugli archi) in un multigrafo connesso $\mathscr{G}$ è un multiinsieme di archi C tali che $\mathscr{G}' = (V, E - C)$ non è connesso. □

Note:

- Un node cut è sempre anche un edge cut, mentre un edge cut potrebbe non essere un node cut.
- Un edge cut minimo è anche un node cut minimo

Il problema del *min-cut* richiede di determinare un edge cut di minima cardinalità in $\mathscr{G}$ connesso.

5.5.2 Contrazioni

Definizione

La contrazione di un (multi)grafo rispetto a un arco viene definita come

Definizione 5.5.2 (Contrazione)**.** Dato $\mathscr{G} = (V, E)$ e $e \in E$, la contrazione di $\mathscr{G}$ rispetto ad e è il multigrafo $\mathscr{G}/e = (V', E')$, dove, posto $e = \{u, v\}$, valgono

$$\begin{aligned} V' &= V - \{u, v\} \cup \{z_{uv}\} \\ E' &= E - \{\!\{\{x, y\} : (x = u) \vee (y = v)\}\!\} \\ &\quad \cup \{\!\{\{z_{uv}, x\} : (\{u, x\} \in E) \vee (\{u, x\} \in E) : (x \neq u) \wedge (x \neq v)\}\!\} \end{aligned}$$

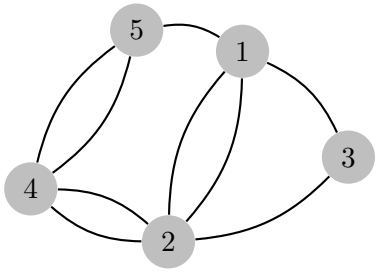
I nodi u e v sono sostituiti da z_{uv}. Vengono eliminati tutti gli archi incidenti su u e v, e inseriti, con la loro molteplicità, come archi incidenti su z_{uv}. Non vengono reinseriti gli archi tra u e v. □

La contrazione rimpicciolisce il grafo, e le cardinalità del grafo ridotto risultano:

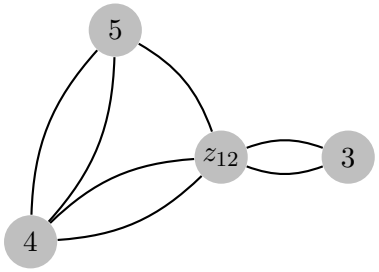
$$|V(\mathscr{G}/e)| = |V(\mathscr{G})| - 1 \qquad |E(\mathscr{G}/e)| = |E(\mathscr{G})| - molt(e)$$

Figura 5.3: Esempio di contrazione

(a) Grafo originale



(b) Grafo dopo la contrazione di $\{1, 2\}$



Effetti della contrazione su un taglio

Ogni taglio C' sul grafo contratto ha un suo corrispettivo nel grafo originale con la stessa taglia. Questo implica che il taglio minimo in $\mathscr{G}/e$ può solo essere più grande. Il taglio minimo deve corrispondere ad un taglio con la stessa cardinalità in $\mathscr{G}$ e quindi il taglio minimo può solo crescere.

Teorema 5.5.1. *Per ogni taglio C' di $\mathscr{G}/e = \{u, v\}$, esiste C taglio di $\mathscr{G}$ con $|C| = |C'|$.*

Dimostrazione. Sia $\mathscr{G}/e = (V', E')$, e $C' \subseteq E'$ un taglio, per cui C' disconnette $\mathscr{G}/e$. Si consideri l'insieme di archi $C \in E(\mathscr{G})$ ottenuto da C' sostituendo ogni arco $\{z_{uv}, x\} \in C'$ con il corrispettivo arco $\{u, x\}$ o $\{v, x\}$ in $E(\mathscr{G})$. Riguardo alla cardinalità, si sostituisce un arco con un altro, quindi sono uguali. Va dimostrato che C è un taglio.
Se C' non contiene nessun arco che abbia z_{uv} come estremo, C' è già un taglio in $\mathscr{G}/e$ e disconnette anche $\mathscr{G}$.
Si sa che C' disconnette $\mathscr{G}/e$, e disconnette z_{uv} da qualche nodo x. La rimozione del taglio crea almeno due componenti connesse, una componente connessa che contiene z_{uv} e un'altra componente connessa che contiene x. Questo significa che ogni cammino da z_{uv} a x in $\mathscr{G}/e$ deve contenere un arco di C'.
Si supponga che in $\mathscr{G}$, C non sia un taglio, ovvero in $\mathscr{G}$ sono tutti nella stessa componente connessa, ovvero C non ha disconnesso il grafo. Per esempio esiste un cammino tra u e x che usa solo archi in $E - C$, ovvero non usa archi in C. Se non usa archi in C, diventa un cammino tra z_{uv} e x che non usa archi in C'. Di conseguenza, si contraddice l'ipotesi che z_{uv} fosse disconnesso da x in $\mathscr{G}/e$ a cui si è rimosso C', perché c'è un cammino tra z_{uv} e x che non usa archi in C'. □

Corollario 5.5.1.1. *Vale $|\text{min-cut}(\mathscr{G}/e)| \geq |\text{min-cut}(\mathscr{G})|$*

Dimostrazione. Tutte le cardinalità dei tagli in $\mathscr{G}/e$ sono realizzate anche in $\mathscr{G}$. □

Corollario 5.5.1.2. *Se C è un taglio di $\mathscr{G}$ con $e \notin C$, allora l'insieme C' corrispondente a C in $\mathscr{G}/e$ è un taglio in $\mathscr{G}/e$.*

Si possono identificare in questo modo i tagli che vengono distrutti. La contrazione infatti mantiene i tagli che sopravvivono nel grafo $\mathscr{G}/e$ e distrugge gli altri tagli. Questo caratterizza tutti i tagli che scompaiono, che sono tutti i tagli che contengono e. Se *non* si contrae rispetto a un arco del taglio minimo, quel taglio sopravvive.

Dimostrazione. Sia $e = \{u, v\}$, e sia C un taglio che non contiene e. Nella rimozione di C da E, si creano almeno due componenti connesse e u e v devono risultare nella stessa componente connessa: e non è parte del taglio, per cui u e v devono essere connessi. Un cammino tra due nodi in due componenti connesse diverse dovrebbe utilizzare un arco del taglio. Considerando x in una componente connessa diversa, si considerano in $\mathscr{G}/e$ i nodi z_{uv} e x. Il cammino tra z_{uv} e x era originariamente da u a x o da v a x. Deve quindi utilizzare un arco del taglio originario (modificato rispetto alla contrazione). Perché altrimenti sopravviverebbe in $\mathscr{G}/e$ un cammino che non usa nessun arco nel taglio C' corrispettivo a C, ma quel cammino esisterebbe anche nel grafo originario, e C non sarebbe un taglio. □

5.5.3 Algoritmo di Karger

Implementazione

Si è dimostrato che se si contrae su un arco $e \notin C$, se C era un taglio in $\mathscr{G}$, allora C' è un taglio in $\mathscr{G}/e$. Rimangono allora tutti e soli i tagli che non contengono e. Gli archi cambiano nomenclatura, ma rimane lo stesso taglio.

L'idea di Karger è sperare di evitare di contrarre su un arco del taglio minimo, e in questo modo conservare il taglio minimo. Nel fascio di archi che si restituisce alla fine ci sarà proprio il taglio minimo.

Va studiato con che probabilità si riesce ad evitare di contrarre il taglio minimo. Questo capita con una probabilità non bassissima: il taglio è minimo per definizione, e ha taglia molto piccola rispetto al numero totale di archi. Selezionare un arco di uno specifico taglio minimo avviene allora con probabilità moderata.

Algoritmo 5.38 Full contraction

```
1: procedure FULL_CONTRACTION(𝒢)
2:     n ← |V(𝒢)|
3:     for i ← 1 to n − 2 do
4:         e ← RANDOM(E(𝒢))
5:         𝒢 ← 𝒢/e
6:     return |E(𝒢)|
```

L'algoritmo ha complessità lineare nel numero di archi di $\mathscr{G}$: $T_{FC}(n, m) = \Theta(m)$.
La FULL_CONTRACTION($\mathscr{G}$) genera un qualche taglio (che è anche un node-cut). L'algoritmo di *Karger* amplifica la probabilità di essere corretto: si vuole ottenere il taglio minimo, quindi si fa girare più volte la contrazione e si preserva il taglio più piccolo trovato. Anche se l'algoritmo non è decisionale, vale l'amplificazione, con la speranza che se il taglio minimo si individua con una certa probabilità, prima o poi lo si individua.

Algoritmo 5.39 Karger

```
1: procedure KARGER(𝒢, k)
2:     min ← |E(𝒢)|
3:     repeat k times
4:         t ← FULL_CONTRACTION(𝒢)
5:         min ← MIN(min, t)
6:     return min
```

Rispetto all'algoritmo deterministico, che deve costruire strutture complesse per sconfiggere il caso peggiore (cammini aumentanti...), l'algoritmo randomizzato supera il caso peggiore con l'idea di schivare alcuni elementi, sfruttando proprio la randomizzazione della scelta.

Rapido recap di probabilità

Dati due eventi indipendenti E_1 e E_2, la probabilità che avvengano entrambi vale

$$Pr(E_1 \cap E_2) = Pr(E_1)\, Pr(E_2)$$

Se invece non sono indipendenti, la probabilità che avvenga il secondo dato che il primo si è verificato vale, per $Pr(E_1) > 0$

$$Pr(E_2 \mid E_1) = \frac{Pr(E_1 \cap E_2)}{Pr(E_1)} \quad \leftrightarrow \quad Pr(E_1 \cap E_2) = Pr(E_1)\, Pr(E_2 \mid E_1)$$

Questo si estende per n eventi

$$Pr\left(\bigcap_{i=1}^{h} E_i\right) = Pr(E_1) \prod_{i=2}^{h} Pr(E_i \mid E_1 \cap \cdots \cap E_{i-1})$$

Analisi dell'algoritmo

Dato un multigrafo $\mathscr{G} = (V, E)$, il grado di ogni nodo è $d(v) = |\{\{v, x\} \in E\}|$. Togliendo tutti gli archi incidenti su un nodo, si ottiene un edge-cut valido. Se il taglio minimo è C^* di taglia $|C^*| = t$, deve allora valere $d(v) \geq t$. Si può allora mostrare che il numero di archi è grande rispetto alla cardinalità del taglio minimo, infatti

$$\begin{aligned} |E| &= \sum_{v \in V} d(v) \Big/ 2 \\ &\geq \sum_{v \in V} t \Big/ 2 \\ &= \frac{nt}{2} \end{aligned}$$

per cui il numero di archi è circa n volte t, e il numero di nodi può facilmente essere molto grande. Si vuole studiare la probabilità che un certo taglio minimo *fissato* sopravviva alla contrazione

$$Pr\left(\text{un taglio minimo fissato } C^* \text{ sopravvive a full contraction}\right)$$

questa probabilità è di sicuro inferiore a quella relativa a un taglio minimo *generico*, di cui si trova così un lower bound.
Seguendo le operazioni dell'algoritmo, si studia per primo l'evento

$$E_1 = \text{la prima contrazione evita } C^*$$

Perché questo avvenga, non si devono selezionare archi del taglio, per cui

$$\begin{aligned} Pr\left(E_1\right) &= 1 - \frac{|C^*|}{|E|} \\ &\geq 1 - \frac{t}{nt/2} \\ &= 1 - \frac{2}{n} \end{aligned}$$

e questo è un bound in alta probabilità. Il secondo evento da analizzare è

$$E_2 = \text{la seconda contrazione evita } C^*$$

ma si deve considerare questo evento condizionato dal successo di E_1: se C^* è sopravvissuto, si trova un taglio trasformato (con nomenclatura degli archi diversa) di taglia uguale $|C^*| = |C^{*'}| = t$

$$\begin{aligned} Pr\left(E_2 \mid E_1\right) &= 1 - \frac{C^{*'}}{|E\left(\mathscr{G}/e_1\right)|} \\ &\geq 1 - \frac{t}{(n-1)t/2} \\ &= 1 - \frac{2}{n-1} \end{aligned}$$

La probabilità inizia quindi a degradare, la cardinalità del taglio cercato resta costante, mentre il numero di archi tra cui scegliere diminuisce. Al passo generico i

$$Pr\left(E_i \mid E_1 \cap \cdots \cap E_{i-1}\right) \geq 1 - \frac{t}{(n-i+1)t/2}$$

$$= 1 - \frac{2}{n-i+1}$$

Se in tutte le $n-2$ iterazioni il taglio minimo viene evitato, l'algoritmo ritorna il taglio corretto.

$$\begin{aligned} Pr\left(\bigcap_{i=1}^{n-2} E_i\right) &= \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) \\ &= \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\ &= \frac{\cancel{n-2}}{n}\frac{\cancel{n-3}}{n-1}\frac{\cancel{n-4}}{\cancel{n-2}}\cdots\frac{\cancel{3}}{\cancel{5}}\frac{2}{\cancel{4}}\frac{1}{\cancel{3}} \\ &= \frac{2}{n(n-1)} \end{aligned}$$

La probabilità che un taglio minimo sopravviva è quindi molto bassa, ma non trascurabile. Ripetendo k volte la contrazione

$$k = d\frac{n^2}{2}\ln n$$

si riesce ad amplificare la probabilità, sfruttando il fatto che le esecuzioni della contrazione sono indipendenti

$$\begin{aligned} Pr\left(k \text{ full contraction falliscono}\right) &\leq \left(1 - \frac{2}{n(n-1)}\right)^k \\ &\leq \left(1 - \frac{2}{n^2}\right)^k \\ &\leq \left(\left(1 - \frac{2}{n^2}\right)^{n^2/2}\right)^{\ln n^d} \\ &< \left(e^{-1}\right)^{\ln n^d} \\ &= \frac{1}{n^d} \end{aligned}$$

La complessità è molto elevata: una singola contrazione è lineare nel numero di archi, e viene ripetuta k volte, per cui

$$T_{Karger}(n) = \Theta\left(m \cdot n^2 \log n\right)$$

che per grafi densi risulta quartica in n.

5.5.4 Variante Karger-Stein

Implementazione

Nel corso dell'algoritmo di *Karger*, c'è una progressiva degradazione della probabilità di evitare il taglio minimo cercato: nelle prime iterazioni, ci sono tantissimi archi tra cui scegliere, mentre nelle ultime parecchi archi rimasti sono parte del taglio minimo.

$$\text{Prob di successo:} \quad 1 - \frac{2}{n},\ 1 - \frac{2}{n-1},\ 1 - \frac{2}{n-2} \quad \cdots \quad 1 - \frac{2}{4},\ 1 - \frac{2}{3}$$

In questo modo si stanno sfruttando male le prime contrazioni, che hanno qualità migliore delle finali. L'idea di *Stein* è proprio quella di fermare la contrazione dopo k iterazioni