Algoritmi e Complessità

Massimo Perego

Contents

1 Notazione

Definire una notazione che verrà usata da qui in avanti.

Verranno usati tanto gli **insiemi**: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ e anche le versioni con **solo i positivi** di ognuno di questi $\mathbb{N}^+$, ...

Un **magma** è un **insieme più un'operazione binaria** definita su di esso. Esempio

$$(A, \circledast)$$

Se il magma ha la **proprietà associativa** sull'operazione è un **semigruppo**

$$(y \circledast z) \circledast x = y \circledast (z \circledast x)$$

Se **esiste elemento neutro** rispetto all'operazione si parla di **monoide**. Se esiste elemento neutro è unico

$$x \circledast \overline{\epsilon} = \overline{\epsilon} \circledast x = x$$

Se l'**operazione è anche commutativa** allora **monoide abelliano**. L'esempio più semplice sono i naturali con l'operazione somma.

Se per ogni elemento c'è **anche un inverso** allora è un **gruppo**.

Definendo un **alfabeto come un insieme finito non vuoto** di elementi.

Monoide libero: Prendendo un alfabeto Σ e considerando tutte le sequenze finite costituite da elementi dell'alfabeto, si nota con

$$w \in \Sigma^* \quad w = w_1, w_2, w_3, ..., w_n \quad n \geq 0$$

$$w_i \in \Sigma \qquad |w| = n$$

La concatenazione è un'operazione interna a Σ^*, due parole se le concateno le metto una dopo l'altra

$$w = w_1, w_2, w_3, ..., w_n$$

$$w' = w'_1, w'_2, w'_3, ..., w'_m$$

$$w \cdot w' = w_1, w_2, w_3, ..., w_n, w'_1, w'_2, w'_3, ..., w'_m$$

$(\Sigma^*, \cdot)$ è un monoide, chiamato **monoide libero** su Σ (abelliano solo quando nell'insieme è presente solo una lettera).

In generale un monoide libero è un monoide con una base, ovvero un insieme di elementi che generano il monoide e che non possono essere generati da altri elementi della base.

Se A e B sono due insiemi allora B^A è l'**insieme delle funzioni da A a B**

$$B^A = \{f | f : A \to B\}$$

La cardinalità finale è quella di $|B|^{|A|}$, per questo la notazione è in quest'ordine.

Con l'**insieme** K si intende

$$K = \{0, 1, \ldots, k-1\}$$

Con 0 che significa l'insieme vuoto.

Sostanzialmente l'insieme di tutti i valori inferiori a K.
L'insieme dei valori dei bit si chiama $2 = \{0, 1\}$.

Considerando

$$2^A = \{f | f : A \to 2\} = \{f | f : A \to \{0, 1\}\}$$

Questa è una funzione che **mappa ogni elemento di A in 0 o in 1**, in biezione con $\cong \{x | x$ è sottoinsieme di $A\} = P(A)$ (insieme delle parti di A).

A^2, **insieme delle coppie di elementi di** A, ma sarebbe

$$A^2 = \{f | f : 2 \to A\} = \{f | f : \{0, 1\} \to A\}$$

che è in biezione con

$$A \times A$$

Quindi con il prodotto cartesiano di A con A.

2^* è un monoide libero sull'alfabeto binario, **tutte le combinazioni binarie**. Quindi una qualsiasi stringa in binario sarà parte di questo monoide.

Mentre 2^{2^*} rappresenta la **famiglia di tutti i linguaggi binari** (su $\{0, 1\}$). Ogni linguaggio che consiste di numeri binari è membro di 2^{2^*}.

2 Algoritmi 101

2.1 Cos'è un problema?

Un **problema** π è **costituito da**:

- Un **insieme di input** possibili I_π
- Un **insieme di output** possibili O_π
- Funzione $Sol_\pi : I_\pi \to 2^{O_\pi} \setminus \{\emptyset\}$ **mappa ogni input a un insieme di output corretti possibili**, non è detto che dobbiamo avere un solo output corretto, possono esserci più output validi. Molti problemi considerati potranno avere più output corretti.

Esempio di problema: Descrizione informale: decidere se un numero è primo.

Supponendo che si tratta, in input, di numeri naturali scritti in base 2 (non specificato, da specificare, questo sono assunzioni).
Mentre in output si può avere solo s o no, si tratta di un problema di decisione

$$O_\pi = \{no, yes\}$$

Anche se non esplicitamente detti, **input e output dovrebbero essere specificati come su un linguaggio binario**, sono **sequenze di bit**.

Altro esempio: Emettere il MCD tra due interi positivi x e y. Non è più un problema di decisione.

Come distinguo i due valori in input?

Soluzioni: raddoppio tutti i bit e 01 diventa il separatore.

Altra soluzione: Elias γ_x sapendo la lunghezza l in bit dell'input x, scrivo l in unario, quindi l bit e uno 0 prima dell'input, così in base a quanti 1 leggo so quanti bit leggere dopo.

I problemi generalmente saranno scritti "approssimativamente", quindi sono necessarie diverse assunzioni per specificarli formalmente.

Nella pratica quindi i dettagli di implementazione dell'algoritmo possono cambiare, ma tra n^2 e $n^2 \log n$ la differenza non è importante, in quanto sempre dominato da un polinomio (assunzione solo teorica, nella realtà non è proprio così, ma sticazzi).

2.2 Cos'è un algoritmo?

Senza definirlo esattamente in modo formale, si presume che a questo punto tu ne abbia un'idea.

Ma se proprio dobbiamo: tutti gli algoritmi sono Macchine di Turing (programmi eseguibili da una MdT).

Tesi di Church-Turing: Storicamente, hanno preso tutte le definizioni di algoritmi presenti, hanno provato che sono tutte equivalenti e ogni definizione riporta alla stessa famiglia di problemi risolvibili: problemi che con una MdT sono risolvibili allora sono risolvibili anche secondo qualsiasi altra definizione.

Al giorno d'oggi è importante perché si traspone in "ogni linguaggio di programmazione (Turing-completo) può risolvere gli stessi problemi risolvibili da tutti gli altri".

In passato è stato discusso come sia impossibile concepire una definizione realistica di algoritmo non equivalente alla MdT.

2.3 Cos'è un algoritmo per un problema π?

Dando $x \in I_\pi$ a un algoritmo A eseguito su una MdT restituirà $y \in O_\pi$ tale che $y \in Sol_\pi(x)$, ovvero **dato un input restituisce una soluzione tra quelle ammesse**.

Molti problemi non sono risolvibili. Considerando i problemi di decisione, data una stringa binaria bisogna rispondere *sì* o *no*.

Se ne può avere uno per ogni possibile insieme di stringhe binarie, gli elementi di 2^{2^*} sono tutti problemi di decisione, e possiamo vedere che

$$|2^{2^*}| \cong |2^{\mathbb{N}}| \cong |\mathbb{R}|$$

ha la potenza del continuo, quindi il numero possibile di problemi di decisione è un infinito non numerabile.

Ma i programmi (algoritmi per risolvere questi problemi) sono infiniti? Sì, ma hanno cardinalità $|\mathbb{N}|$, essendo questa la dimensione dell'insieme di risposte possibili ($|2^*|$).

Quindi **i problemi sono di più dei possibili programmi per risolverli**. Per forza di cose la maggior parte dei problemi non sono risolvibili.

2.4 Teoria della complessità

Tutti i problemi che verranno considerati sono risolvibili, ma **quanto efficientemente**? La teoria della complessità stabilisce la **quantità di risorse consumate da un problema**.

Si può parlare di teoria della complessità in due termini:

- **Algoritmica**
- **Strutturale**

2.4.1 Algoritmica

Dato un problema π

1. Stabilire **se è risolvibile** (per noi sempre sì); c'è un algoritmo che lo risolve?
2. **Risolvibile con che costo?** Ci sono più **misure** possibili:
   - **Tempo** (solitamente sarà usata questa)
   - Spazio
   - Numero di processori utilizzati/somma dei tempi di tutte le CPU
   - Energia dissipata

Considerando il tempo, vogliamo stabilire **dato un input quanto tempo impiega**, ovvero **quanti passi** deve svolgere

$$T_A : I_\pi \to \mathbb{N}$$

Invece di calcolare la **complessità** input per input si considera "per taglia", **per ogni dimensione**, prendendo l'input che ci impiega il tempo massimo per taglia

$$t_a : \mathbb{N} \to \mathbb{N}$$

$$t_a(n) = \max\{T_A(x) \,|\, x \in I_\pi \text{ t.c. } |x| = n\}$$

Si tratta di un'assunzione worst-case, dati input di stessa dimensione prendo il peggiore. Potrebbe essere utile fare una media al posto di un'assunzione pessimistica, ma noi teniamo così perché è comodo, inoltre nella realtà possiamo solo metterci di meno.

Se dobbiamo valutare quale algoritmo usare il **confronto va fatto asintoticamente**, i casi piccoli sono "facili" (di solito, ma andrebbero considerati i casi d'uso effettivi? In estremo si potrebbe compilare una tabella). Quindi consideriamo come le **performance dell'algoritmo scalano con la dimensione dell'input**.

Quindi **due assunzioni**: worst-case e asintotico.

Quindi, generalmente, tra

$$t_a = O(n^2) \qquad t_a = O(n^7)$$

è tendenzialmente meglio il primo.

Bisogna anche provare che non ci possono essere casi migliori/peggiori, dimostrare un upper/lower bound.

2.4.2 Strutturale

Fissato un problema π, qual è la sua complessità? Non vogliamo cercare la complessità di un algoritmo, ma la **complessità del miglior algoritmo che risolve un problema** π. Questo permette di eliminare facilmente problemi che non hanno soluzioni efficienti.

Con la notazione worst case asintotica sopracitata possiamo avere un **upper bound** per il **tempo impiegato** per la risoluzione di un problema. Quindi nel caso peggiore avrà asintoticamente quel comportamento. $O(n^2)$ vuol dire che nel caso peggiore ci mette tempo quadratico rispetto all'input.

Esistono anche **lower bounds**, ovvero limiti inferiori per il **tempo impiegato** da un algoritmo e dimostrarlo implica **dimostrare che per come è fatto un problema, questo non può impiegare meno di un certo tempo**. $\Omega(n)$ vuol dire che non ci può mettere meno di tempo lineare.

Un problema viene "*chiuso*" quando upper e lower bound coincidono, vuol dire che si ha l'algoritmo ottimo e si conosce esattamente la complessità del problema. Per esempio, l'ordinamento di array basato su confronti e scambi ci mette esattamente $n \log n$, quindi si dice $\Theta(n \log n)$.

Le casistiche problematiche sono quando gli **upper bound sono esponenziali e i lower bound polinomiali**, quindi le possibilità sono:

- esiste un upper bound da migliorare, quindi esistono algoritmi polinomiale non ancora scoperti
- esiste un lower bound esponenziale, quindi il problema non è risolvibile in tempo polinomiale

In entrambi i casi saremmo contenti, il problema è non avere informazioni a riguardo.

2.4.3 Classi di complessità:

Per la complessità strutturale sono state inventate delle classi di problemi per **dividerli in base alla difficoltà di risoluzione**:

- $\mathcal{P}$: problemi di decisione risolvibili in tempo polinomiale; la domanda diventa "il mio π è in $\mathcal{P}$?"
- $\mathcal{NP}$: problemi di decisione risolvibili in tempo polinomiale **su macchine non deterministiche**

Sostanzialmente macchine non deterministiche vuol dire che: una macchina su cui il programma può essere eseguito più volte in parallelo con determinate variabili settate a 0 e 1, permettendo di eseguire in parallelo tante "versioni" e ottenere una soluzione per ogni esecuzione parallela. La soluzione alla fine è *yes* se esiste almeno una esecuzione che ha dato *yes* e *no* se tutte le esecuzioni hanno riportato *no*.

CNF-Sat problem: L'input è una espressione logica in forma CNF (Conjunctive Normal Form), quindi un `and` tra clausole, ognuna delle quali contiene `or` di letterali, i quali possono essere variabili o negazioni di variabili.

L'output è se l'espressione logica φ è soddisfacibile, ovvero, esiste $\exists$ assegnazione che rende φ *true*?
Attualmente questo problema si pensa sia in $\mathcal{NP}$, non sappiamo se possa stare in $\mathcal{P}$.

Sappiamo rientra in $\mathcal{NP}$ perché è facile pensare a un algoritmo non deterministico per questo problema, basta avere una esecuzione parallela per ogni possibile valore di ognuna delle variabili, e se almeno un output è *sì* allora è soddisfacibile.

Relazione tra $\mathcal{P}$ e $\mathcal{NP}$: $\mathcal{P}$ è ovviamente **incluso in $\mathcal{NP}$**, dato che se posso risolvere un problema in modo polinomiale, posso risolverlo anche con il parallelismo aggiunto delle macchine non deterministiche.

La vera domanda è se $\mathcal{P} = \mathcal{NP}$, quindi **se le due classi coincidono**, le possibilità sono:

- Le due **classi sono uguali** $\mathcal{P} = \mathcal{NP}$: sarebbe molto bello, vorrebbe dire che **tutti i problemi sono risolvibili polinomialmente**.
- $\mathcal{P}$ **contenuta in** $\mathcal{NP}$, $P \subset NP$: meno emozionante ma vorrebbe dire che **esistono dei problemi non risolvibili in tempo polinomiale**.

Ma è uno dei millennium problem, quindi penso rimarrà lì.

2.4.4 Riduzione in tempo polinomiale

Un problema π_1 e polinomialmente riducibile a π_2 iff:

$$\pi_1 \leq_p \pi_2 \text{ iff:}$$

$$\exists f : I_{\pi_1} \to I_{\pi_2} \text{ t.c. } x \in I_{\pi_1} \quad Sol_{\pi_1}(x) = yes \quad Sol_{\pi_2}(f(x)) = yes$$

Esiste una funzione che a partire da un input di pi_1 arriva a un input di π_2 in tempo polinomiale e se x è un input di π_1 possiamo avere lo stesso output che x fornisce su π_1 per π_2 con input $f(x)$.

Sostanzialmente vuol dire che possiamo **trasformare input di π_1 in input di π_2 in tempo polinomiale**, allora **se $\pi 2$ è risolvibile in tempo polinomiale lo è anche π_1**.

Teorema: Se $\pi_1 \leq_p \pi_2$ e $\pi_2 \in \mathcal{P}$ allora $\pi_1 \in \mathcal{P}$. Se un problema può essere polinomialmente ridotto a un altro all'interno di $\mathcal{P}$, allora anch'esso è in $\mathcal{P}$. Sostanzialmente $\leq_p$ può essere letto come "non è più difficile di".

Teorema di Cook: Dato che CNF-Sat $\in \mathcal{NP}$ e $\forall \pi \in \mathcal{NP}$ allora $\pi \leq_p$ CNF-Sat. Esistono **problemi all'interno di $\mathcal{NP}$ a cui tutti gli altri problemi si possono ridurre** (provato per la prima volta da Cook con il CNF-Sat).

Questi problemi sono **$\mathcal{NP}$-completi** ($\mathcal{NP}c$)

$$(\mathcal{NP}c) = \{\pi \in \mathcal{NP} | \, \forall \pi' \in \mathcal{NP} \pi' \leq_p \pi\}$$

$\mathcal{NP}$ completi vuol dire **tutti i problemi in $\mathcal{NP}$ si possono ridurre a un altro problema in $\mathcal{NP}$**, quindi alla complessità di questi altri. Vuol dire sostanzialmente che sono tutti circa "difficili uguali" e che "cambia poco" (secondo la riduzione polinomiale).

Se noi sapessimo che un qualunque problema $\mathcal{NP}c$ è risolvibile in tempo polinomiale, l'implicazione di questo teorema è che tutti i problemi in $\mathcal{NP}$ sarebbero risolvibili in tempo polinomiale, facendo collassare $\mathcal{P}$ e $\mathcal{NP}$.

Corollario: Se $\pi \in \mathcal{NP}c$ e $\pi \in \mathcal{P}$ allora $\mathcal{P} = \mathcal{NP}$.

Nello scenario in cui $\mathcal{P} \neq \mathcal{NP}$ i problemi $\mathcal{NP}c$ sono sottoinsieme di $\mathcal{NP}$, ma completamente staccati dall'insieme di $\mathcal{P}$.

Dire che un **problema è $\mathcal{NP}c$** vuol dire che è **risolvibile esattamente solo in tempo esponenziale**.

3 Problemi di ottimizzazione

Sono un caso particolare dei problemi come precedentemente definiti.

Un **problema di ottimizzazione** π:

1. **Input** $I_\pi \subseteq 2^*$
2. Una **funzione** $Amm_\pi : I_\pi \to 2^{2^*} \setminus \{\emptyset\}$ **mappa** ogni **input** in un **insieme di soluzioni ammissibili**
3. $c_\pi : 2^* \times 2^* \to \mathbb{N}$, $\forall x \in I_\pi$, $\forall y \in Amm_\pi(y)$, $c_\pi(x, y)$, ovvero un costo/guadagno, **funzione obiettivo**
4. $T_\pi \in \{\max, \min\}$: "tipo del problema", voglio massimizzare o minimizzare il mio costo/guadagno

Esempio: Max-Sat

- **Input:** formule logiche in forma CNF, esempio:

$$(x_7 \vee x_2) \wedge (x_4 \vee \neg x_5 \vee \neg x_7) \wedge (x_9 \neg x_2)$$

- Le **soluzioni ammissibili** per una data formula φ sono tutti gli assegnamenti delle variabili di φ, quindi per l'esempio tutte le 2^5 possibili combinazioni di assegnamenti delle 5 variabili
- $c_\pi(\varphi, ass)$: il costo per ogni formula con un dato assegnamento è il numero di clausole rese vere dall'assegnamento
- $T_\pi = \max$: problema di **massimizzazione**, voglio un assegnamento che massimizzi il numero di clausole soddisfatte

Ovviamente non si può avere un algoritmo polinomiale che risolve questo problema in quanto sarebbe equivalente al CNF-SAT, avere il massimo numero di clausole soddisfacibili vorrebbe dire anche sapere se la formula è soddisfacibile.

3.1 Problema di decisione associato

A **ogni problema di ottimizzazione π si può associare un problema di decisione $\hat{\pi}$**:

- $I_{\hat{\pi}} = (x, k)$, $x \in I_\pi$, $k \in \mathbb{N}$.
- La risposta per l'input (x, k) è sì iff $\exists y \in Amm_\pi(x)$ esiste una soluzione ammissibile tale che $c_\pi(x, y) \geq k$ per i problemi di massimizzazione, oppure $c_\pi(x, y) \leq k$ per problemi di minimizzazione (rientro nel costo k?).

Sostanzialmente, "trovare il minimo" diventa "esiste una soluzione con costo minore di k?".

Esempio: Per $\hat{\text{Max}}$-Sat

- $I_{\hat{\text{Max}}\text{-Sat}} = \{(\varphi, k) | \, \varphi \text{ formula CNF e } k \in \mathbb{N}\}$.
- (φ, k) è *sì* iff $\exists$ assegnamento che rende vere almeno k clausole di φ.

3.2 Classe di complessità $\mathcal{PO}$

Si definisce $\mathcal{PO}$ l'insieme dei problemi di ottimizzazione π tali che π è risolvibile in tempo polinomiale

$$\mathcal{PO} = \{\pi \text{ problemi di ott. } | \, \pi \text{ risolvibile in t polinomiale } \}$$

Teorema: Se $\pi \in \mathcal{PO}$, allora il suo problema di decisione associato sta in $\mathcal{P}$, i.e., $\hat{\pi} \in \mathcal{P}$.

Corollario: Al contrario, se il problema di decisione associato $\hat{\pi}$ è $\mathcal{NP}c$, allora il problema non può essere risolto in tempo polinomiale $\pi \notin \mathcal{PO}$ (assumendo $\mathcal{P} \neq \mathcal{NP}$).

Esempio di problemi in $\mathcal{PO}$: problemi di programmazione lineare. Mentre restringere il campo a solamente a soluzioni intere fa diventare il problema $\mathcal{NP}c$.

3.3 Rapporto di prestazione

Verranno trattati principalmente problemi che rientrano in $\mathcal{NP}c$, quindi non saranno risolvibili in modo esatto, ma solamente in modo approssimato. Ma come si **quantifica l'approssimazione**?

Dato un problema di ottimizzazione π, chiamiamo $\text{opt}_\pi(x)$ il **valore ottimo della funzione obiettivo su input** x.
Dato un **algoritmo approssimato** A (in input prende un possibile input e in output fornisce una soluzione ammissibile, ma non necessariamente ottima) per π, definisco il **rapporto di approssimazione**

$$R_\pi(x) = \max\left\{\frac{c_\pi(x,y)}{opt_p i(x)}, \frac{opt_\pi(x)}{c_\pi(x,y)}\right\}$$

Il max serve a prendere un valore sempre ≥ 1, sia per problemi di minimizzazione che massimizzazione, in modo da avere una definizione unica.

Si dice che A è una **α-approssimazione** per π iff

$$\forall x \in I_\pi : R_\pi(x) \leq \alpha$$

Per ogni input $R_\pi(x)$ è minore o uguale di α, sostanzialmente comunque vada quell'algoritmo non può essere peggio di α sul problema π. Ovviamente con $\alpha = 1$ l'algoritmo è esatto e il problema sta in $\mathcal{PO}$.

3.4 Altre classi

Si hanno altre possibili classi di problemi:

- Fuori da $\mathcal{PO}$ ($\mathcal{PO} \subset$), ci sono le classi n-**APX**, quindi 2-APX ad esempio sono gli algoritmi che risolvono con α fino a 2 volte, 3 fino a 3 volte, ...; sostanzialmente classi sempre più grandi fino ad avere approssimazioni arbitrariamente grandi.

- **APX**: problemi risolvibili a meno di una costante di approssimazione di qualche tipo arbitrariamente grande, ma costante. Sempre algoritmi polinomiali, ma approssimati.

- Poi $\log n$-**APX**: approssimati ma l'approssimazione dipende anche dalla dimensione dell'input, fuori da APX (APX $\subset$).

- Tutte queste classi sono contenute in $\mathcal{NPO}$, ovvero l'equivalente di ottimizzazione di $\mathcal{NP}$.

- Esistono anche problemi $\mathcal{NPO}$-completi $\mathcal{NPO}c$, e interseca tutti i sottoinsiemi fin'ora (tranne $\mathcal{PO}$ ovviamente).

- **PTAS** (Polinomial Time Approximation Scheme), (appena fuori $\mathcal{PO}$): problemi approssimabili "quanto voglio", quindi posso decidere il tasso di approssimazione ma il tempo peggiora di conseguenza, anche in modo anche esponenziale.

- **FPTAS** (Fully PTAS): dentro PTAS, fuori $\mathcal{PO}$, come PTAS, ma il tempo peggiora esclusivamente in modo polinomiale.

Dovrei mettere un disegnino con i cerchietti ma non ho assolutamente voglia, si capisce anche così dai (spero).

3.5 Problema di MaxMatching

Caratteristiche del problema:

- **Input:** un **grafo non orientato** $G = (V, E)$ e bipartito (due gruppi di nodi, i nodi all'interno di un gruppo non hanno collegamenti tra loro)
- **Soluzioni ammissibili:** chiamate **matching**, insieme M, una selezione di lati per cui nessun vertice incide su > 1 lato, scegliere lati in modo da coprire il maggior numero di vertici possibili senza coprire nessun vertice più di una volta
- **Funzione obiettivo:** numero di lati scelti, ovvero **cardinalità di** M
- **Tipo:** problema di **massimizzazione** = max, massimizzare il numero di vertici coperti

In questo caso la **soluzione ottima** si può trovare in **tempo polinomiale**, questo problema rientra in $\mathcal{PO}$.

Algoritmo del cammino aumentante: Un cammino aumentante viene definito su un grafo su cui è già presente un matching, anche parziale (solo in realtà, altrimenti non aumento nulla). Un **vertice** viene definito "**esposto**" quando su quel vertice **non incide nessun lato del matching**.

Un **augmenting path** è un **cammino** che **parte e arriva su un vertice esposto** ed **alterna lati** "liberi" (**non nel matching**) **e lati** "presi" (**nel matching**). Sequenza di lati alternata che inizia e termina su un vertice esposto. Partendo da vertici esposti il primo lato è per forza non preso.

Sapendo che c'è un cammino aumentante, è possibile scambiare i lati presi e non presi, si può fare uno **switch** sul cammino aumentante trovato. Partendo e arrivando su nodi esposti per il cammino, **facendo lo switch** ci sarà sempre un **lato in più** "preso" **nel matching** e di conseguenza due nodi in più.

Se **ho un cammino aumentante posso migliorare il matching**, quindi quest'ultimo non è ottimo. Per migliorare il matching basta trovare un cammino aumentante, di conseguenza se lo trovo posso migliorare il matching, altrimenti il matching è massimo.

Lemma: Se **esiste un cammino aumentante** per il matching M, allora M **non è massimo**.

Lemma 2: Se M **non è massimo** allora **esiste un cammino aumentante**.

Proof. Sia M' un altro matching con $|M'| > |M|$ (M non è massimo, ne esiste uno con cardinalità maggiore). Di conseguenza, considerando gli insiemi dei lati presi da ognuno dei matching, abbiamo tre zone tra i due insiemi:

- lati in comune $M \cap M'$
- lati solo nell'insieme più piccolo $M \setminus M'$
- lati solo nell'insieme più grande $M' \setminus M$

Prendendo la differenza simmetrica tra i due

$$M \Delta M' = (M \setminus M') \cup (M' \setminus M)$$

Considerazione: Nessun vertice può avere più di 2 lati coincidenti in $M \Delta M'$; se in ogni insieme possono avere un solo lato incidente ed ho due insiemi non possono essere più di 2 lati totali, il grado di ogni vertice sarà ≤ 2, inoltre se ne ha 2 uno viene da un insieme e uno dall'altro.
Di conseguenza sul grafo generato dai lati in $M \Delta M'$ esiste $\exists$ almeno un cammino, composto da lati presi alternativamente uno da un insieme e uno dall'altro, non possono essere di fila altrimenti vorrebbe dire che un vertice è stato considerato due volte all'interno dello stesso matching.
Sapendo che $|M'| > |M|$ ci deve essere almeno un cammino che inizia e finisce con un lato di $M' \setminus M$ dato che sono di più e non può essere un ciclo (altrimenti sarebbero due lati dello stesso matching sullo stesso vertice).

Essendo questo cammino alternanza di lati presenti in $M \setminus M'$ e lati in $M' \setminus M$ questo vuol dire che è un cammino aumentante per M.

□

Visite di grafi

Un po' estemporaneo ma serve spiegarlo.

Con "visite di grafi" si intende **metodi sistematici per "scoprire" un grafo**. Durante una visita i **nodi** possono rientrare in **3 categorie**:

- sconosciuti, bianchi W
- conosciuti ma non visitati, grigi G, detti anche frontiera di visita
- visitati, neri B

La visita funziona iniziando **mettendo nella frontiera un solo nodo**, chiamato seed della visita.

$$F \leftarrow \{x_{seed}\}$$

Quindi in questo momento il nodo seme è grigio, tutti gli altri bianchi.

Algorithm 1 VisitaGrafo (*Graph*)

$\forall x\ c(x) \leftarrow W$
$F \leftarrow \{x_{seed}\}$
$c(_{seed}) \leftarrow G$
while $F \neq \emptyset$ **do**
  $x \leftarrow \text{pick}(F)$
  visit x
  $c(x) \leftarrow B$
  for $i \in \text{neighbor}(x)$ **do**
    if $c(y) == W$ **then**
      $F \leftarrow F \cup \{y\}$
      $c(y) = G$
    end if
  end for
end while

Quando la frontiera è non-vuota, prendo un nodo dalla frontiera e visito questo nodo (qualunque cosa significhi). Coloro questo nodo di nero. Guardo tutti i vicini di questo nodo e:

- se sono neri, già visitati, oppure grigi, già conosciuti, li lascio lì
- se sono bianchi li metto nella frontiera e li coloro di grigio

Per un grafo orientato l'unica cosa che cambia è scegliere se visitare i vicini uscenti o entranti (collegati con archi uscenti o entranti).

Se il **grafo è connesso** questo **garantisce di visitare ogni vertice una volta sola**. Se non è connesso verrà visitata solo la parte connessa del seme, per continuare serve un altro seed.

In base a **come viene scelto il nodo da prendere nella frontiera cambia il tipo di visita**, ad esempio con uno stack viene una DFS, con una coda viene una BFS, ma in generale, in base a come viene implementata la scelta del nodo si ha un tipo di visita diversa.

3.5.1 Trovare un cammino aumentante

Un cammino aumentante deve **iniziare e terminare su un nodo esposto**, partendo da un grafico e un matching.

Come **trovo nodi esposti**? Faccio, partendo da ogni nodo **una BFS che alterna lati all'interno di M e lati al di fuori di M**.

Se **una BFS** a un certo punto **trova altri nodi esposti** allora abbiamo un **cammino aumentante**, altrimenti si passa alle BFS sul prossimo nodo, se **non viene trovato nessun altro nodo esposto** in nessun nodo allora **non ci sono cammini aumentanti**.

Algorithm 2 FindAugmenting(G,M)

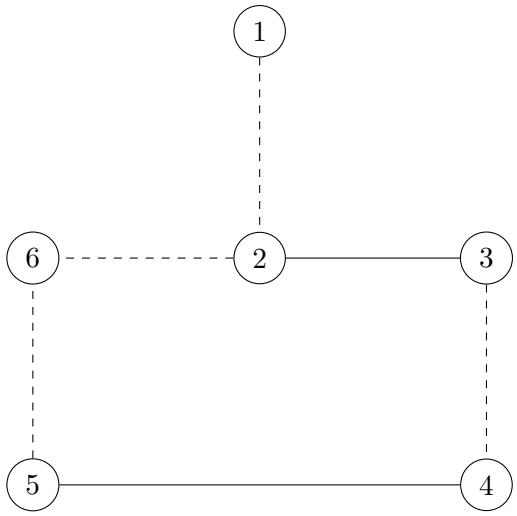
```
X ← vertici esposti in M
for all x ∈ X do
    BFS(x) con alternanza di lati ∉ M e ∈ M
end for
```

Tempo: La BFS richiede tempo proporzionale al numero di lati $O(N \cdot M)$, una BFS per ogni vertice esposto, quindi al massimo $O(n^2)$. Quante volte dobbiamo eseguire FindAugmenting? Al limite $n/2$, quindi $O(n/2 \cdot n^2) \subset O(n^3)$.

Teorema: BipartiteMaxMatching $\in \mathcal{PO}$.

Corollario: Il problema di decisione PerfectMatching (il grafo ammette un matching massimale) è $\in \mathcal{P}$.

Perché la BFS non funziona su grafi non bipartiti?



Partendo da qualsiasi nodo esposto in questo grafo non si possono trovare cammini aumentanti (prova a farci la BFS alternata se non mi credi, (non ho voglia di scriverla)).

4 Tecniche greedy

4.1 Problema di LoadBalancing

Non ammette una soluzione polinomiale ottima, quindi $\in \mathcal{NPO}$.

Problema in cui si ha **un numero di task** di una certa **durata** e bisogna **dividerli su varie macchine** in modo da avere la massima **durata di ogni macchina al minimo possibile**:

- **Input:** $m > 0$ numero di macchine, $n > 0$ numero di task, $(t_i)_{i \in n}$, durate dei task, $t_i > 0$
- **Output:** $\alpha : n \to m$, assegnare ogni task ad una macchina. Il carico della macchina j è definito come

  $$L_j := \sum_{i:\alpha(i)=j} t_i$$

  La somma di tutte le durate dei task assegnati a quella macchina. Il carico massimo è

  $$L := \max_j \{L_j\}$$

  il valore più alto di una singola macchina
- **Funzione obiettivo:** L
- **Tipo:** problema di minimizzazione min, devo minimizzare il costo della macchina con il costo più alto

Teorema: LoadBalancing $\in \mathcal{NPO}c$.

Definizione di $\mathcal{NPO}c$: un problema di ottimizzazione π è $\in \mathcal{NPO}c$ iff $\pi \in \mathcal{NPO}$ e $\hat{\pi} \in \mathcal{NP}c$, il problema di decisione associato è $\mathcal{NP}c$.

4.1.1 Algoritmo greedy per LoadBalancing

Fornisce una **soluzione approssimata**. Sostanzialmente l'algoritmo consiste nel **prendere i task** ed associarli sempre alla macchina più "scarica", con la attuale durata complessiva delle task minore.

Chiamando A_i l'insieme dei **task associati alla macchina** i.

All'inizio sia task associati che carichi totali partono a zero. Si itera su tutti i task, si sceglie la macchina con il carico minore e si aggiunge alla sua lista di task il task j considerato (ed il carico relativo).
Ovviamente è polinomiale: $O(n)$ per cercare tra i task e $O(m)$ per cercare le macchine, quindi il totale $O(nm)$

Algorithm 3 GreedyLoadBalancing(n, m)

$A_i \leftarrow \emptyset \ \forall i \in n$
$L_i \leftarrow \emptyset \ \forall i \in n$
for $j = 0, ..., m-1$ **do**
  $\bar{i} \leftarrow \arg\min_i L_i$
  $A_{\bar{i}} \leftarrow A_{\bar{i}} \cup \{j\}$
  $L_{\bar{i}} \rightarrow L_{\bar{i}} + t_j$
end for

Teorema: GreedyBalance è una 2-Approssimazione per Loadbalancing

Proof. Chiamando L^* il valore della funzione obiettivo della soluzione ottima.

Osservazione 1: posso dire che

$$L^* \geq \frac{1}{m} \sum_j t_j$$

Ovviamente se potessi dividere il totale di tutti i task in modo perfetto su m macchine otterrei una soluzione ideale (1/m del tempo totale per ogni macchina), quindi la soluzione ottima deve essere per forza $\geq$ di questa divisione.

Osservazione 2

$$L^* \geq \max_j t_j$$

Il compito più grande qualcuno lo deve fare, quindi il tempo massimo non può essere inferiore al tempo del task più lungo.

Sia $\hat{i}$ tale che $L_{\hat{i}} = L$ e sia $\hat{j}$ l'ultimo compito che le è stato assegnato.

Prima di assegnare alla macchina $\hat{i}$ l'ultimo carico, questa doveva essere quella con il carico minore di tutte

$$L_{\hat{i}} - t_{\hat{j}} \leq L'_i \quad \forall i \in m$$

$$m(L_{\hat{i}} - t_{\hat{j}}) \leq \sum_i L_i = \sum_j t_j$$

$$\implies L_{\hat{i}} - t_{\hat{i}} \leq \frac{1}{m} \sum_j t_j \leq L^*$$

L'ultima parte della disequazione possiamo aggiungerla essendo la stessa cosa della osservazione 1.

$$L = L_{\hat{i}} = L_{\hat{i}} - t_{\hat{j}} + t_{\hat{j}} \leq 2 \cdot L^*$$

Per la osservazione 2, posso far vedere che $t_{\hat{j}}$ è $\leq L^*$, quindi il totale è tutto minore di $2 \cdot L^*$. Di conseguenza possiamo evincere il rapporto di approssimazione

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

L'algoritmo quindi è 2-Approssimato. □

Dimostrazione di tightness: Questa dimostrazione serve a stabilire che L/L^* è davvero 2, ovvero che **il bound non può diminuire ulteriormente** e quel fattore di approssimazione dipende intrinsecamente dall'algoritmo e non da qualche errore nella dimostrazione.

Teorema: per ogni $\epsilon > 0$, esiste un input per il problema LoadBalancing tale che GreedyLoadBalancing produce un output con

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

(ovvero schiaccio quanto voglio il bound).

Proof. Dobbiamo **costruire un input** che fa "andare male" l'algoritmo:

- numero di macchine $m > 1/\epsilon$
- numero di task $n = m(m-1) + 1$
- ci sono $m(m-1)$ task lunghi 1 e 1 task lungo m, presentati in quest'ordine

I primi m verranno dati ciascuno ad una macchina, e si ripete $m-1$ volte.
Alla fine di questi tutte le macchine hanno lo stesso carico.
Poi arriva big task che verrà messo sulla macchina meno carica (sono tutte uguali, quindi una a caso praticamente).

Di conseguenza la funzione obiettivo finale sarà:

$$L = m - 1 + m = 2m - 1$$

Non sapendo che alla fine arriva il taskone non possiamo prepararci al suo arrivo. La soluzione ottima sarebbe dare il Titanic ad una macchina e dividere gli altri $m(m-1)$ alle altre macchine; alla fine tutte le macchine avranno lo stesso carico, quindi è sicuramente la soluzione ottima, con valore:

$$L = m$$

Il rapporto di prestazione nel primo caso diventa:

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

Essendo un **algoritmo greedy**, **l'ordine** dei task è **importante** ed in questo caso è la differenza tra un input ottimale ed il peggiore.

Comunque, magari gli input reali non sono così male, questo algoritmo su input reali potrebbe avere un tasso di approssimazione più basso, potrebbero essere input più favorevoli rispetto alle casistiche considerate in questo modo.
□

4.1.2 SortedGreedyBalance

Sempre sul problema di LoadBalancing, sempre stesso input e output dell'algoritmo precedente.

L'algoritmo funziona **ordinando** in ordine **decrescente i task** e poi **chiamando l'algoritmo greedy**:

Algorithm 4 SortedGreedyBalance(n, m)

Sort t_i in Non-increasing order
Run GreedyLoadBalancing(n, m)

Ma è davvero meglio?

Teorema: SortedGreedyBalance produce una $\frac{3}{2}$-Approssimazione

Proof. **Osservazione 1**: se $n < m$, la soluzione prodotta è ottima (tutti i task a macchine distinte, meglio di così non si può).

Da qui in poi viene considerato $n > m$ (almeno una macchina deve avere 2 task assegnati).

Osservazione 2: $L^* \geq 2 \cdot t_m$. Considerando i task

$$t_0 \geq t_1 \geq \ ... \ \geq t_m \geq t_{m+1} \geq \ ... \ \geq t_{n-1}$$

Almeno una macchina dovrà avere 2 task, uno dai primi m task ed uno dai restanti.

Sia $\hat{i}$ tale che $L_{\hat{i}} = L$:

- Se $\hat{i}$ ha un task solo, la soluzione è ottima
- Se $\hat{i}$ ha più di un compito. Sia $\hat{j}$ l'ultimo compito (ovvero l'indice del task) assegnato a $\hat{i}$

Sicuramente $\hat{j} \geq m$ in quanto i primi m task sono sicuramente dati a macchine distinte (non può esserci una macchina meno carica di una vuota).

Quindi

$$L = L_{\hat{i}} = \left(L_{\hat{i}} - t_{\hat{j}}\right) + t_{\hat{j}}$$

Di conseguenza, sapendo che

$$t_{\hat{j}} \leq t_m \leq \frac{1}{2}L^*$$

(una macchina con due task, uno dei quali $t_{\hat{j}}$ non può superare metà della soluzione ottima). Possiamo dire

$$\begin{matrix} \left(L_{\hat{i}} - t_{\hat{j}}\right) & \leq L^* \\ t_{\hat{j}} & \leq \frac{1}{2}L^* \end{matrix} \implies \left(L_{\hat{i}} - t_{\hat{j}}\right) + t_{\hat{j}} \leq L^* + \frac{1}{2}L^* = \frac{3}{2}L^*$$

In conclusione

$$L = L_{\hat{i}} \leq \frac{3}{2}L^*$$

□

Ci vorrebbe una dimostrazione di tightness, ma questa analisi non è tight, si può dimostrare che l'algoritmo in realtà è una $4/3$-Approssimazione (dimostrazione di `Graham 1969`).

Hochbaum-Shmoys 1988 hanno dimostrato che in realtà LoadBalancing apparitene $\in PTAS$, ovvero si può approssimare con una precisione ϵ qualunque, e scala esponenzialmente il tempo necessario per l'approssimazione, quindi si sa anche che $\notin FPTAS$.

4.2 Problema della selezione dei centri CenterSelection

Avendo vari posti dove smistare pacchi e n soldi per costruire un numero limitato di magazzini, bisogna decidere dove piazzarli. Una volta piazzati tutti gli altri centri di smistamento si rivolgono al magazzino più vicino. Si creano delle celle di Voronoi attorno ai magazzini. Di conseguenza c'è una distanza massima che un centro di smistamento deve coprire per arrivare al magazzino, ovvero un raggio di copertura, tra tutte le soluzioni si vuole minimizzare questo raggio.

Hai dei **punti in uno spazio metrico** e si vogliono **scegliere** k **punti** in questo spazio in modo che il **raggio di copertura** per tutti sia il **minore possibile**.

Spazio metrico: Un insieme con una funzione distanza (Ω, d)

$$d : \Omega \times \Omega \to \mathbb{R}^+$$

Con le seguenti proprietà:

- $d(x, y) = d(y, x)$ (simmetria)
- $d(x, y) = 0$ iff $x = y$
- $d(x, y) \leq d(x, z) + d(z, y)$ (disuguaglianza triangolare)

L'input per la selezione dei centri sarà immerso in uno spazio metrico.

Definizioni: nel mio insieme di punti, qual'è il punto scelto (magazzino) più vicino?

$$dist(x, A) := \min_{y \in A} d(x, y)$$

$$\forall s \in S \quad \delta_c(s) := dist(s, C)$$

$$\rho_c := \max_{s \in S} \delta_c(s)$$

Definizione del problema: Fissiamo uno spazio metrico (Ω, d)

- **Input:** un **insieme** $S \subseteq \Omega$ di n **punti immersi in uno spazio metrico** fissato e un **budget** $k > 0$
- **Soluzione accettabile:** è un **sottoinsieme** $C \subseteq S$ con $|C| \leq k$ ($\leq k$ punti nello spazio)
- **Funzione obiettivo:** ρ_c
- **Tipo:** min

Vogliamo minimizzare il massimo dei percorsi.

4.2.1 Algoritmo CenterSelectionPlus

Facciamo finta di avere in input un parametro che non avremo. Stesso input del problema, ma aggiungiamo

$$r \in \mathbb{R}^+$$

Algorithm 5 CenterSelectionPlus(S, k)

```
C ← ∅
while S ≠ ∅ do
  take any s̄ ∈ S
  C ← C ∪ {s̄}
  Remove from S all s t.c. d(s, s̄) ≤ 2r
end while
if |C| ≤ k then
  Output C
else
  Output "Impossibile"
end if
```

Insieme di centri vuoto, poi finché l'insieme di punti non è vuoto:

- prende un punto in in S
- toglie da S tutti i punti con una distanza dal punto preso minore di $2r$

Se la cardinalità di C è $\leq k$ torna C, altrimenti dice che è impossibile.

Teorema 1: Se CenterSelectionPlus **emette un output**, esso è una $\frac{2r}{\rho^*}$-**Approssimazione**.

Proof. $\forall s \in S$ con s cancellato da S e sia $\overline{s}$ il centro scelto quando abbiamo cancellato s, di conseguenza $d(s, \overline{s}) \leq 2r$:

$$\rho_c \leq \delta_c(s) \leq d(s, \overline{s}) \leq 2r$$

Di conseguenza

$$\frac{\rho_c}{\rho^*} \leq \frac{2r}{\rho^*}$$

□

Teorema 2: se $r \geq \rho^*$, l'algoritmo **emette un output**.

Proof. Sia C^* una soluzione ottima, una distribuzione di centri che ha come raggio di copertura ρ^*.
Sia $\overline{s} \in C$. Chiamiamo $\overline{c}^* \in C^*$ un centro tale che nella soluzione ottima $\overline{s}$ si rivolge a $\overline{c}^*$.
Sia X l'insieme dei punti che nella soluzione ottima C^* si rivolgono a $\overline{c}^*$

$$\forall s \in X \quad d(s, \overline{s}) \leq d(s, \overline{c}^*) + d(\overline{c}^*, \overline{s}) \leq 2\rho^* \leq 2r$$

$\implies$ Quando seleziono s, cancello tutti i punti di X (come minimo), perché distano da s meno di $2r$.
$\implies$ Elimino da S un'intera cella di Voronoi di C^*.

C^* ha al massimo k celle di Voronoi (avendo k centri al suo interno).
$\implies$ Dopo $\leq k$ passi ho cancellato tutto.

□

Il rapporto di approssimazione abbiamo detto essere $\frac{2r}{\rho^*}$, di conseguenza l'insieme delle soluzioni ammissibili è quello dei valori $\geq \rho^*$ e la scelta di r **migliora all'avvicinarsi a** ρ^*, fino ad essere una 2-Approssimazione con $r = \rho^*$.

Con $\frac{1}{2}\rho^a sr < r < \rho^*$ non è ben definito che output si può ottenere (e se si ottiene un output), ma sicuramente con un $r \leq \frac{1}{2}\rho^*$ non si può avere output perché altrimenti porterebbe, secondo l'approssimazione detta sopra, ad un risultato migliore dell'ottimo.

Quindi voglio avere un r **pari a** ρ^*, ma è un dato che non conosciamo.

Al posto di scegliere un punto a caso potrei scegliere punti $\overline{s}$ che sono almeno a distanza $> 2r$ da C, facendo terminare l'algoritmo quando non posso sceglierne altri. Equivalente all'algoritmo detto, al posto di cancellare i punti quegli stessi non potranno essere presi in considerazione ma è la stessa cosa.

4.2.2 GreedyCenterSelection

Stesso input di CenterSelection ma NON abbiamo più il parametro r.

Algorithm 6 GreedyCenterSelection(S, k)

if $|S| \leq k$ **then**
  output S
end if
choose any $\overline{s} \in S$
$C \leftarrow \{\overline{s}\}$
while $|C| \leq k$ **do**
  select $\overline{s}$ maximizing $d(\overline{s}, C)$
  $C \leftarrow C \cup \{\overline{s}\}$
end while
Output C

Se l'insieme dei punti ha cardinalità $\leq k$, torna S (basta prendere tutti i punti). Altrimenti sceglie un qualsiasi $\overline{s} \in S$ e lo sceglie come centro.

Finché la cardinalità di C è $< k$ (quindi sceglierà esattamente k punti) **sceglie il punto che massimizza la distanza** $d(\overline{s}, C)$ e lo aggiunge a C. Alla fine restituisce C.

Cancellare i punti o scegliere il punto più lontano sono equivalenti, non cambia nulla.

Teorema: GreedyCenterSelection è una **2-Approssimazione** per CenterSelection.

Proof. Supponendo che, per assurdo, l'algoritmo GreedyCenterSelection emetta una soluzione con $\rho > 2\rho^*$ (soluzione fuori da 2 volte il raggio di copertura ottimale). Questo vuol dire che esiste un elemento dell'insieme S, $\exists \overline{s} \in S$, con distanza $d(\overline{s}, C) > 2\rho^*$.

Sia $\overline{s}_i$ l'i-esimo centro aggiunto e sia $\overline{C}_i$ l'insieme dei centri in quel momento. Il punto va scelto in modo che massimizzi la distanza dai centri attuali:

$$d(\overline{s}_i, \overline{C}_i) \geq d(\hat{s}_i, C_i) \geq d(\hat{s}, C) > 2\rho^*$$

Quindi sarà maggiore della distanza di uno dei centri precedenti, che sarà maggiore della distanza finale, a sua volta maggiore di $2\rho^*$.
Alla fine, uno dei punti deve essere distante $> 2\rho^*$ da tutti i punti scelti.

Ma allora l'esecuzione è una delle esecuzioni possibili di CenterSelectionPlus quando $r = \rho^*$, ovvero la scelta (prendere quello a distanza massima) è compatibile quella che farebbe l'altro algoritmo (sostanzialmente sta prendendo uno dei punti lontani più di $2r$, non importa quale). GreedyCenterSelection è una particolare esecuzione di CenterSelectionPlus con $r = \rho^*$.

Ma CenterSelectionPlus con $r = \rho^*$ produce un output:
$\implies$ termina entro k iterazioni
$\implies$ tutti gli $s \in S$ sono tali che $d(s, C) \leq 2\rho^*$
(per i teoremi dimostrati per CenterSelectionPlus).

Ma non è vero, dato che: $d(\hat{s}, C) > 2\rho^*$. Assurdo.

□

Si potrebbe anche dimostrare che l'algoritmo è tight, ma sappiamo anche che c'è un lower bound per l'approssimazione in tempo polinomiale, quindi possiamo avere una dimostrazione di inapprossimabilità per mostrare che non si può approssimare con un fattore di approssimazione migliore di 2.

Teorema: Se $\mathcal{P} \neq \mathcal{NP}$ **non esiste un algoritmo polinomiale** che α-**approssimi** CenterSelection per **qualche** $\alpha < 2$ (quindi l'algoritmo greedy è polinomialmente ottimo).

Proof. La dimostrazione si basa su un problema di decisione chiamato DominatingSet:

- Input: un grafo $G = (V, E)$ ed un $k > 0$
- Output: $\exists D \subseteq V$, sapere se esiste un insieme di vertici di cardinalità al massimo k, $|D| \leq k$, tale che $\forall x \in V$ allora $\exists y \in D$ con $xy \in E$. Sostanzialmente metto sui vertici k guardie e ogni nodo deve essere collegato ad almeno una guardia, ogni nodo è in D oppure è vicino di un nodo $\in D$

Si sa che DominatingSet $\in \mathcal{NP}c$.

Dati G, k, input di DominatingSet, dobbiamo costruire un'istanza di CenterSelection, quindi innanzitutto uno spazio metrico:

$$\Omega = V = S$$

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

Una cosa non ovvia da dimostrare è la disuguaglianza triangolare, quindi

$$d(x, y) \leq d(x, z) + d(z, y)$$

Quindi il valore di $d(x, y)$ può essere solo 1 o 2, mentre dall'altro lato della disuguaglianza può valere $2, 3$ o 4. Non devo dirtelo io che 2 è sempre ≤ 2.

Abbiamo la nostra istanza di CenterSelection(S, k), quindi come input insieme S e budget k.

Quanto vale il raggio di copertura?

$$\rho^*(S, k) \in \{1, 2\}$$

Tutte le distanze sono 1 o 2, quindi il raggio sarà 1 o 2.

Quando succede che la distanza è 1?

$$\rho^*(S,k) = 1$$

iff $\exists C^* \subseteq S$, con cardinalità $|C^*| \leq k$ e centri tali che $\forall s$, $d(s, C^*) \leq 1$

iff $\exists C^* \subseteq S$ con cardinalità $|C^*| \leq k$ tale che $\forall s$, $\min_{c \in C^*} d(s,c) = 1$.

iff $\exists C^* \subseteq S$ con cardinalità $|C^*| \leq k$ tale che $\forall s$, $\exists c \in C^*$ con $sc \in E$.

iff C^* è un dominating set, in quanto è esattamente la definizione di DominatingSet.

Un algoritmo α-approssimante per CenterSelection fornirà

$$\rho^* \leq \rho(S,k) \leq \alpha\rho^*(s,k)$$

Ed i casi possibili sono:

- $\rho^* = 1$, quindi
$$1 \leq \rho(S,k) \leq \alpha$$
Se la soluzione ottima è 1 allora mi produce α al limite.

- $\rho^* = 2$
$$2 \leq \rho(S,k) \leq 2\alpha$$

Sostanzialmente, se potessi risolvere ottimamente CenterSelection in tempo polinomiale, avrei anche una risposta in tempo polinomiale per DominatingSet, dato che, con CenterSelection, se viene 1 la risposta è "sì", mentre se viene 2 la risposta è "no".

□

Comunque si parla di un'istanza di CenterSelection abbastanza particolare, quindi questo non significa che non esistano altre versioni ristrette di CenterSelection con migliori fattori di approssimazione. Limitare le istanze solitamente permette di approssimare meglio. Sapere come sono fatti gli input permette di fare meglio.

Funzione Armonica

Serve per dopo, la lascio qui.

Una funzione che **mappa numeri naturali positivi in reali**:

$$H : \mathbb{N}^+ \to \mathbb{R}$$

Secondo la funzione:

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

Proprietà 1: può essere approssimata per eccesso

$$H(n) \leq 1 + \int_1^n \frac{1}{x} dx$$

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

Proprietà 2: possiamo dare anche un limite inferiore

$$\int_t^{t+1} \frac{1}{x} dx \leq \int_t^{t+1} \frac{1}{t} dx = \frac{1}{t}$$

$$H(n) = \frac{1}{1} + \frac{1}{2} + \ ... \ + \frac{1}{n} \geq \int_1^2 \frac{1}{x} dx + \int_2^3 \frac{1}{x} dx + \ ... \ + \int_n^{n+1} \frac{1}{x} dx =$$

$$= \int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$$

Quindi

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

Unendo le due cose:

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

4.3 Problema di SetCover

Definizione del problema:

- **Input:** degli **insiemi** $S_1, \ldots, S_m$ e la loro unione $\bigcup_{i \in m} S_i = U$ (universo), ognuno con dei pesi $w_0, \ldots w_{m-1}$
- **Soluzioni ammissibili:** scegliere un certo numero di insiemi in modo che siano coperti tutti i punti. $I \subseteq m$ tale che

$$\bigcup_{i \in I} S_i = U$$

- **Funzione obiettivo:** la somma dei pesi per gli insiemi scelti

$$w = \sum_{i \in I} w_i$$

- **Tipo:** min, problema di minimizzazione

4.3.1 GreedySetCover

Algorithm 7 GreedySetCover()

$R \leftarrow U$
$I \leftarrow \emptyset$
while $R \neq \emptyset$ **do**
  choose S_i minimizing $\frac{w_i}{|S_i \cap R|}$
  $I \leftarrow I \cup \{i\}$
  $R \leftarrow R \setminus S_i$
end while
output I

Metto da parte l'insieme universo e prendo un insieme vuoto per i punti. Finché rimangono punti in R scelgo l'insieme che minimizza il rapporto tra costo e punti ancora da coprire presenti in quell'insieme. Una volta scelto, aggiungo l'insieme ed i punti che copre.
Alla fine restituisco I.

Questo algoritmo è come se assegnasse ad ogni vertice da coprire un costo

$$\forall s \in S_i \cap R \qquad c(s) := \frac{w_i}{|S_i \cap R|}$$

Lemma 1: Alla fine dell'esecuzione $w = \sum_{s \in U} c(s)$, il **costo totale è la somma di tutti i costi singoli pagati**.

Proof. $w = \sum_{i \in I} w_i$, ma il suo costo è ripartito in tutti gli elementi al momento in cui li ho scelti, quindi:

$$w = \sum_{i \in I} w_i = \sum_{i \in I} \sum_{s \in S_i \cap R} c(s) = \sum_{s \in U} c(s)$$

Insomma, la somma dei costi è il costo complessivo, tutto abbastanza ovvio. □

Lemma 2: per ogni k

$$\sum_{s \in S_k} c(s) \leq H(|S_k|) w_k$$

Se sommiamo i costi attributi all'interno di ogni insieme, quello che si ottiene è minore uguale della funzione armonica calcolata nel punto pari alla cardinalità dell'insieme S_k, moltiplicato per il peso w_k. Sì insomma, si capisce meglio dalla disequazione.

Proof. Prendendo $S_k = \{s_1, \ldots, s_d\}$, con i vertici enumerati in ordine di copertura (mettendo per primi i vertici che sono stati coperti per primi, quindi a blocchi di elementi, un "blocco" per ogni volta che viene scelto un'insieme).

Consideriamo l'iterazione che copre s_j e di conseguenza il momento prima che s_j venga coperto tutti i vertici dopo sicuramente non sono coperti

$$\{s_j, s_{j+1}, \ldots, s_d\} \subseteq R$$

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

Per la definizione di costo sappiamo che al passo h prima che venga scelto il vertice s_j:

$$\implies c(s_j) = \frac{w_h}{|S_h \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$$

Ed il costo è sempre minore di quelli successivi essendo che l'algoritmo cerca di minimizzarlo.

Di conseguenza:

$$\sum_{s \in S_k} c(s) \leq \sum_{j=1}^{d} c(s_j) \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} =$$

$$= \frac{w_k}{d} + \frac{w_k}{d-1} + \ ... \ + \frac{w_k}{1} = w_k \left(\frac{1}{1} + \frac{1}{2} + \ ... \ + \frac{1}{d} \right) =$$

$$= w_k H(d) = w_k H(|S_k|)$$

□

Dai due lemmi segue il teorema.

Teorema: Sia $M = \max_i |S_i|$ (massima cardinalità di un insieme). Allora GreedySetCover è **una $H(M)$-approssimazione** per SetCover (l'**approssimazione è in funzione dell'input**, al crescere di quest'ultimo cresce il fattore di approssimazione).

Proof. Sia I^* una soluzione ottima (come insieme di indici) e di conseguenza

$$w^* = \sum_{i \in I^*} w_i$$

Sarà il minimo possibile.

Per il Lemma 2

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

H è una funzione monotona, quindi con un valore maggiore risultato maggiore, di conseguenza al denominatore diventa più piccolo.

Quindi

$$\implies w^* = \sum_{i \in I^*} w_i = \sum_{i \in I^*} \sum_{s \in S_i} c(s)$$

La somma dei pesi degli insiemi è pari alla somma del costo di tutti gli elementi di tutti gli insiemi. In questa sommatoria stiamo sommando tutti gli elementi dell'universo, quindi (per il Lemma 1)

$$\geq \sum_{s \in U} c(s) = w$$

Ed inoltre possiamo vedere che, applicando, in ordine, la prima e seconda uguaglianza vista sopra

$$w^* = \sum_{i \in I^*} w_i \geq \sum_{i \in I^*} \frac{\sum_{s \in S_i} c(s)}{H(M)} \geq \frac{w}{H(M)}$$

$$\implies \frac{w}{w^*} \leq H(M)$$

□

Osservazione: M è sicuramente minore della dimensione dell'input, $M \leq |U| = n$, quindi $H(M) \leq H(n) = O(\log n)$.

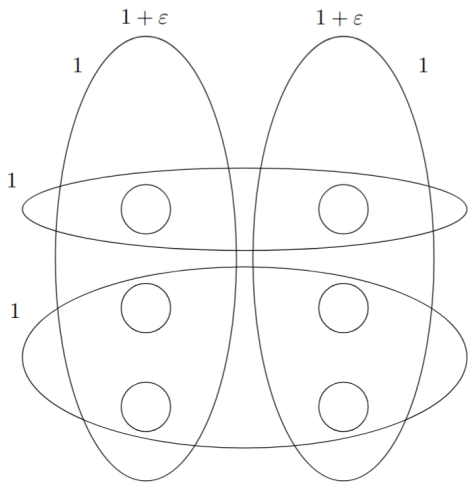
Corollario: GreedySetCover è **una** $O(\log n)$**-approssimazione**. L'approssimazione peggiora con $O(\log n)$. Non eccezionale come cosa ma ci accontentiamo.

Tightness: dobbiamo costruire un **input** su cui l'algoritmo va abbastanza male da **verificare** il nostro **upper bound** per l'approssimazione.

L'input (universo) di dimensione n è composto da due insiemi di costo $1+\epsilon$, ognuno con al suo interno $n/2$ punti.
Poi ci sono altri insiemi, tutti di costo 1:

- uno che contiene $n/4$ elementi da ognuno dei due insiemi iniziali (cardinalità totale $n/2$)
- uno che contiene $n/8$ elementi da ognuno dei due insiemi iniziali (cardinalità totale $n/4$)
- Puoi indovinare come continua

Esempio:



La soluzione ottima ovviamente è prendere i due insiemi più grandi per un costo di $2(1+\epsilon)$.

Ma GreedySetCover su questo input sceglierà per primo l'insieme che copre $n/2$ punti con costo 1 in quanto

$$\frac{1}{n/2} < \frac{1+\epsilon}{n/2}$$

Al passaggio dopo i due insiemi grossi conterranno $n/4$ nuovi elementi e si ripeterà:

$$\frac{1}{n/4} < \frac{1+\epsilon}{n/4}$$

Quindi non sceglierà mai i due insiemi grandi ma sempre le intersezioni che coprono sempre meno elementi. Di conseguenza:

$$w = \log n$$

$$w^* = 2 + 2\epsilon$$

$$\implies \frac{w}{w^*} = \frac{\log n}{2+2\epsilon} = \Omega(\log n)$$

Quindi questo algoritmo fa le scelte peggiori possibili, ed è l'input peggiore possibile.

Fun fact: A meno che $\mathcal{P} = \mathcal{NP}$, non esiste un algoritmo che approssimi SetCover meglio di $(1 - O(1)) \log n$. Questo pone SetCover nell'insieme dei problemi $\log n$-APX.

5 Tecnica di Pricing

5.1 Problema VertexCover

Definizione del problema:

- **Input:** un grafo non orientato $G = (V, E)$ con dei pesi/costi per ogni nodo $w_i \in \mathbb{R}^+$, $i \in V$
- **Soluzione ammissibile:** $X \subseteq V$ tale che $\forall e \in E$, $e \cap X \neq \emptyset$, ogni lato deve avere una delle due estremità dentro la soluzione
- **Funzione obiettivo:** somma dei pesi $\sum_{i \in X} w_i$
- **Tipo:** minimizzazione min

Pricing-based Solution: I problemi basati sul pricing funzionano basandosi sull'idea che ogni lato offre una cifra per farsi coprire, e i vertici devono decidere cosa fare, ovvero se entrare nella soluzione o meno, ottenendo la cifra di tutti i lati che copre. I vertici sono una gang che vuole massimizzare il costo dei lati. A ogni lato si associa un prezzo per "farsi coprire".
Quindi si ha un **pricing** $(P_e)_{e \in E}$ **per ogni lato**.

Un pricing $(P_e)_{e \in E}$ è **equo** se **iff** $\forall i$

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

Ovvero, i **lati**, in totale, **non offriranno più** di quanto "richiede" il vertice, ovvero **del costo del vertice**. Zero su tutti i lati soddisfa banalmente questa proprietà. Verranno considerati solo di pricing equi.

Lemma 1: Se $(P_e)_{e \in E}$ è un **pricing equo** allora se **sommo tutti i prezzi** di tutti i lati **ottengo** un **valore minore uguale** della **soluzione ottima**

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

Proof. Considerando $X^* \subseteq V$ soluzione ottima e il peso relativo

$$w^* = \sum_{i \in X^*} w_i$$

Per la definizione di equità: $\forall i$

$$w_i \geq \sum_{e \in E, i \in e} P_e$$

il peso di ogni nodo è maggiore della somma dei costi dei lati su di esso.

Quindi

$$w^* = \sum_{i \in X^*} w_i \geq \sum_{i \in X^*} \sum_{e \in E, i \in e} P_e \geq \sum_{e \in E} P_e$$

La soluzione ottima è per forza maggiore della somma di tutti i lati, quindi maggiore della somma del prezzo di tutti i lati (per ammissibilità).

□

Sostanzialmente, se il pricing è equo il costo del nodo deve essere $\geq$ della somma dei pricing dei suoi lati, quindi la soluzione ottima, in quanto composta dal peso dei nodi, non potrà essere minore della somma totale dei pricing.

Pricing stretto: Il pricing $(P_e)_{e \in E}$ è **stretto** su $\hat{i}$ (un vertice specifico) iff

$$\sum_{e \in E, \hat{i} \in e} P_e < w_{\hat{i}}$$

In breve, "**pricing stretto**" vuol dire che **non soddisfa quanto chiede il vertice**, somma dei pricing di quel vertice minore del costo del vertice.

5.1.1 PricingVertexCover

L'input e sempre $G = (V, E)$, $w_i \in \mathbb{R}^+$, $i \in V$.

Algorithm 8 PricingVertexCover

$P_e \leftarrow 0, \forall e \in E$
while $\exists \hat{e} = \{\hat{i}, \hat{j}\}$ t.c. (P_e) è stretto su $\hat{i}$ e $\hat{j}$ **do**
Sia $\hat{e} = \{\hat{i}, \hat{j}\}$ quello che minimizza

$$\Delta \leftarrow \min \left(w_{\hat{i}} - \sum_{e \in E, \hat{i} \in e} P_e, w_{\hat{j}} - \sum_{e \in E, \hat{j} \in e} P_e \right)$$

$P_{\hat{e}} \leftarrow P_{\hat{e}+\Delta}$
end while

Inizialmente attribuisce a tutti i lati un prezzo 0.

Finché **esiste un lato** tale che (P_e) è stretto su $\hat{i}$ e su $\hat{j}$, $\exists \hat{e} = \{\hat{i}, \hat{j}\}$, ovvero il **pricing è stretto su entrambe le estremità**, quindi per entrambi il costo del vertice è maggiore della somma dei pricing offerti, finché vale questa condizione si **viene scelto il lato** $\hat{e} = \{\hat{i}, \hat{j}\}$ tale che **minimizza** di quanto dovrebbe **aumentare la sua offerta** per **rendere non stretto** il costo di uno dei vertici adiacenti.

Alla fine **restituisce l'insieme dei vertici** i tali per cui P_e **su** i **non è stretto**.

Partono tutti a zero, guardo vertice per vertice quanto stanno ricevendo dai lati adiacenti, e all'inizio ovviamente non sarà stretto su nessun vertice.
Quanto dovrebbe offrire in più ogni lato per rendere stretto il pricing su uno dei due vertici a cui è collegato? Calcolo il valore per tutti e prendo il minimo.
Ripeto sui lati non ancora stretti per entrambe le estremità, finché sono presenti lati con questa caratteristica.

Alla fine restituisce i vertici "contenti", quindi i vertici che prendono quanto chiedono, ovvero i vertici con pricing non stretto.

Ricordando il lemma sul pricing equo.

Lemma 2: alla fine di PricingVertexCover il costo $w \leq 2 \sum_{e \in E} P_e$ è minore uguale al doppio della somma dei prezzi.

Proof.

$$w = \sum_{i \in output} w_i = \sum_{i : P_e \text{ non stretto su } i} w_i =$$

Emettiamo in output i vertici tali per cui il pricing su quel vertice non è stretto. Per la definizione di "stretto", possiamo continuare dicendo che:

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

Qua stiamo sommando per tutti i vertici nell'output i prezzi di alcuni lati, ma **ognuno dei lati può comparire al massimo due volte**, se entrambe le estremità di quel lato stanno nell'output. Quindi:

$$\sum_{i \in output} \sum_{e \in E, i \in e} P_e \leq 2 \sum_{e \in E} P_e$$

□

Teorema: PricingVertexCover è una 2-Approssimazione.

Proof.

$$\frac{w}{w^*} \leq \frac{2 \sum_{e \in E} P_e}{w^*} \leq \frac{2 \sum_{e \in E} P_e}{\sum_{e \in E} P_e} = 2$$

Per il Lemma 2, w è minore di 2 volte la somma totale dei prezzi, mentre la soluzione ottima w^* è per forza maggiore della somma totale dei prezzi.

□

Non si sa se esiste un'approssimazione migliore, non si conosce una γ-approssimazione con $\gamma < 2$. Si sa che non esiste un PTAS, quindi c'è per forza un minimo di approssimazione ma non si sa per certo quale sia.

5.2 Problema DisjointPaths

Problema dei **cammini disgiunti**. L'idea è, su un grafo orientato, ci sono k **sorgenti** e altrettante **destinazioni**, una lista di sorgenti e destinazioni. Vogliamo **collegare il maggior numero di sorgenti e destinazioni tramite un cammino**, con il vincolo di non usare ogni lato più di una volta. Verrà considerata una versione con un parametro c che determina la congestione, ovvero **ogni lato non può essere usato più di** c **volte**.

Definizione:

- **Input:** un grafo orientato $G = (N, A)$, la lista delle sorgenti $s_0, \ldots, s_k \in N$, la lista delle destinazioni $t_0, \ldots, t_k \in N$ e un parametro $c \in \mathbb{N}^+$
- **Output:** $I \subseteq k$ e $\forall i \in I$ un cammino $\pi_i : s_i \to t_i$ tale che nessun arco di G sia usato da più di c cammini
- **Funzione obiettivo:** la cardinalità $|I|$, ovvero il numero di cammini trovati
- **Tipo**: massimizzazione, max

Da notare che con i grafi orientati si parla di nodi (da qui N), non vertici, e archi (da qui A), non lati.

Per l'esecuzione dell'algoritmo **assoceremo** a **ogni arco** un **funzione lunghezza**:

$$\ell : A \to \mathbb{R}^+$$

Che può **variare nel tempo**, quindi va **specificato il momento** in cui viene considerata. Di conseguenza si avrà una lunghezza dei cammini: con $\pi =< x_1, \ldots, x_i >$

$$\ell(\pi) = \ell(x_1, x_2) + \ell(x_2, x_3) + \ldots + \ell(x_{i-1}, x_i)$$

5.2.1 GreedyPaths

Input come sopra, grafo, sorgenti, destinazioni, capacità. Si aggiunge un parametro $\beta > 1$.

Algorithm 9 GreedyPaths

$I \leftarrow \emptyset$
$P \leftarrow \emptyset$
$\ell(a) = 1, \forall a \in A$
while *true* **do**
  find shortest path $\pi_i : s_i \to t_i$ with $i \notin I$
  if such path does not exist **then**
    break
  end if
  $I \leftarrow I \cup \{i\}$
  $P \leftarrow P \cup \{\pi_i\}$
  for all $a \in \pi_i$ **do**
    $\ell(a) \leftarrow \ell(a) \cdot \beta$
    if $\ell(a) = \beta^c$ **then**
      remove a
    end if
  end for
end while
Output I, P

L'insieme I è l'insieme delle **coppie già collegate**, all'inizio vuoto, così come l'insieme di **cammini** P. La funzione **lunghezza all'inizio vale 1 per tutti** gli archi. Poi si ha un ciclo infinito in cui:

- trova il **percorso più breve** (secondo la lunghezza ℓ) **tra** una **coppia sorgente e destinazione non ancora collegata**
- **se** tale percorso **non esiste** per nessuna delle coppie rimaste, **esce** dal ciclo
- se ho trovato un percorso, **aggiungo** la coppia di **nodi collegati a** I e il relativo path in P
- per tutti gli archi nel cammino, **moltiplico** il **lunghezza per** β, rendendoli più costosi e meno appetibili per i prossimi path. Inoltre se un **arco è stato usato** c volte **viene rimosso**

Alla fine **restituisce l'insieme delle coppie collegate e i relativi path**.

Quando si dice "trovare il cammino minimo", bisogna usare più iterazioni di Dijkstra per trovarli tutti.

Definizione: In un certo istante dell'esecuzione un **cammino** π è definito **corto** iff la sua **lunghezza** è **minore di** β^c, quindi $\ell(\pi) < \beta^c$.

Definizione: In un certo istante, un **cammino** π è **utile** se **collega una coppia nuova** $i \notin I$. L'algoritmo considera solo cammini utili e il più corto tra quelli esistenti.

L'algoritmo avrà più **fasi**, una in cui seleziona solo cammini corti utili, poi una in cui sceglie cammini lunghi utili, e termina quando sono finiti tutti i cammini utili.

Considerando il programma nella fase in cui sono **finiti cammini corti utili**, chiameremo $\overline{\ell}, \overline{I}, \overline{P}$ rispettivamente lunghezza, insieme di coppie collegate e i relativi path in quel momento, ovvero subito dopo che è stato aggiunto l'ultimo cammino utile.
Alla fine dell'esecuzione chiameremo i parametri $\ell_{out}, I_{out}, P_{out}$.

Lemma 1: Sia $i \in I^* \backslash I_{out}$, una coppia **collegata dalla soluzione ottima ma non dalla soluzione dell'algoritmo**, allora

$$\overline{\ell}(\pi_i^*) \geq \beta^c$$

Ovvero il costo del path è superiore a β^c.

Proof. $i \in I^* \setminus I_{out}$, se fosse vera la condizione $\overline{\ell}(\pi_i^*) < \beta^c$ selezioneremmo quel path, ma noi stiamo guardando il momento in cui non sono più presenti cammini corti, quindi non è possibile.
Quindi se $\overline{\ell}(\pi_i^*) < \beta^c$ per costruzione dell'algoritmo verrebbe incluso, ma il presupposto è esattamente il contrario.

□

Lemma 2: Se **sommo la lunghezza di tutti gli archi** ottengo:

$$\sum_{a \in A} \overline{\ell}(a) \leq \beta^{c+1} |\overline{I}| + m$$

La somma della lunghezza è minore uguale di β^{c+1} per il numero di cammini corti $+m$, con m il numero totale di archi.

Proof. Per induzione. All'inizio:

$$\sum_{a \in A} \ell(a) = m \leq \beta^{c+1} |\overline{I}| + m$$

Supponendo che

$$\ell_1 \xrightarrow[i,\pi_i]{} \ell_2 \rightarrow \ ... \ \rightarrow \overline{\ell}$$

Quindi nel momento di transizione tra ℓ_1 e ℓ_2 vengono aggiunti i e π_i. Possiamo vedere che ℓ_2 diventa:

$$\ell_2(a) = \begin{cases} \ell_1(a) & a \notin \pi_i \\ \ell_1(a) \cdot \beta & a \in \pi_i \end{cases}$$

Quindi domandandoci di quanto sia aumentata la lunghezza degli archi possiamo vedere che:

$$\sum_{a \in A} \ell_2(a) - \sum_{a \in A} \ell_1(a) =$$

Ma possiamo considerare solo gli archi nel path π in quanto l'aumento per gli altri è 0:

$$= \sum_{a \in \pi} (\ell_2(a) - \ell_1(a)) =$$

Ma per passare da ℓ_1 a ℓ_2 abbiamo moltiplicato ℓ_1 per β, quindi

$$= \sum_{a \in \pi} (\beta \ell_1(a) - \ell_1(a)) =$$

Quindi possiamo raccogliere e dire che, per definizione di tutti i path scelti negli istanti precedenti $\overline{\ell}$

$$= (\beta - 1) \sum_{a \in A} \ell_1(a) = (\beta - 1) \ell_1(\pi) < (\beta - 1) \beta^c \leq \beta^{c+1}$$

A ognuno dei $|\overline{I}|$ passi (quindi per ognuno dei cammini trovati) possiamo dire che aumento la somma di al più β^{c+1}. Si aggiunge m in quanto valore iniziale di tutti gli archi, ovviamente presente.

□

Osservazione 1: Tutti i **cammini** nella **soluzione ottima** ma **non in quella dell'algoritmo** sono **maggiori di**

$$\sum_{i \in I^* \setminus I_{out}} \overline{\ell}(\pi_i^*) \geq \beta^c |I^* \setminus I_{out}|$$

Per il Lemma 1.

Osservazione 2: Nella **soluzione ottima nessun arco** è **usato più di** c **volte**, comunque è vincolata, quindi

$$\sum_{i \in I^* \setminus I_{out}} \overline{\ell}(\pi_i^*) \leq c \sum_{a \in A} \overline{\ell}(a) \leq$$

e per il lemma 2

$$\leq c \left(\beta^{c+1} |\overline{I}| + m \right)$$

Quindi, **usando le osservazioni**:

$$\beta^c |I^*| \leq \beta^c |I^* \setminus I_{out}| + \beta^c |I^* \cap I_{out}| \leq$$

La prima parte, per l'osservazione 1, è maggiorata da

$$\leq \sum_{i \in I^* \setminus I_{out}} \overline{\ell}(\pi_i^*) + \beta^c |I_{out}| \leq$$

Ma questo corrisponde all'osservazione 2:

$$\leq c \left(\beta^{c+1} |\overline{I}| + m \right) + \beta^c |I_{out}| \leq c \left(\beta^{c+1} |I_{out}| + m \right) + \beta^c |I_{out}|$$

Dividendo per β^c:

$$|I^*| \leq c \cdot \beta \cdot |I_{out}| + \frac{cm}{\beta^c} + |I_{out}| \leq$$

Aggiungendo un $|I_{out}|$ sicuramente il valore rimane maggiore, lo aggiungiamo per raccogliere:

$$\leq c \cdot \beta \cdot |I_{out}| + \frac{cm}{\beta^c}|I_{out}| + |I_{out}| = |I_{out}| \left(c\beta + \frac{cm}{\beta^c} + 1 \right)$$

Quindi

$$\frac{|I^*|}{|I_{out}|} \leq c \left(\beta + m\beta^{-c} \right) + 1$$

Ma bisogna trovare un β adeguato, e considereremo

$$\beta = m^{\frac{1}{c+1}}$$

Con questa scelta il rapporto visto può essere maggiorato da:

$$\frac{|I^*|}{|I_{out}|} \leq c \left(m^{\frac{1}{c+1}} + m^{1-\frac{1}{c+1}} \right) + 1 = 2cm^{\frac{1}{c+1}} + 1$$

Teorema: GreedyPath fornisce una $\left(2c \cdot m^{\frac{1}{c+1}} + 1\right)$**-approssimazione** per Disjoint paths.

Esempio di valori:

c	
1	$2\sqrt{m} + 1$
2	$4\sqrt[3]{m} + 1$
3	$6\sqrt[4]{m} + 1$

Dipende da c e dal numero di archi m. Si ha c che determina un parametro lineare e una radice che riduce il valore di m. Cresce al numero di archi, ma decresce con c.

L'approssimazione migliora nel caso di coppie ripetute.

6 Tecniche basate sull'arrotondamento

Programmazione Lineare

Serve per il prossimo algoritmo, bear with me.

La programmazione lineare (LP) può essere **descritta come problema di ottimizzazione**:

- **Input:** una **matrice di razionali** $A \in \mathbb{Q}^{m \times m}$, un **vettore** $b \in \mathbb{Q}^m$, **vettore** $c \in \mathbb{Q}^n$
- **Soluzioni ammissibili:** tutti i **vettori** $x \in \mathbb{Q}^n$ **tali che** $Ax \geq b$
- **Funzione obiettivo:** $c^T x$
- **Tipo:** minimizzazione min

Si sa che $LP \in \mathcal{PO}$ (il più famoso è algoritmo di Karmarkar).

PL Intera

Programmazione lineare intera (ILP):

- **Input:** matrice $A \in \mathbb{Q}^{m \times m}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$
- **Soluzioni ammissibili:** tutti i vettori $x \in \mathbb{Z}^n$ tali che $Ax \geq b$
- **Funzione obiettivo:** $c^T x$
- **Tipo:** minimizzazione min

Cercando una **soluzione intera al posto che razionale** il problema diventa $ILP \in \mathcal{NPO}c$.

6.1 VertexCover via Rounding

Soluzione per VertexCover **basata sull'arrotondamento**.

Programmazione lineare per VertexCover: Per il problema VertexCover π: Input: $G = (V, E)$, con pesi $w_i \in \mathbb{Q}^+$, $\forall i \in V$ (già definito prima, non riscritto).

Il problema di programmazione lineare intera associato a π diventa $ILP(\pi)$: $x \in \mathbb{Z}^n$, ed i vincoli sono:

$$\begin{cases} x_i + x_j \geq 1 & \forall \{i, j\} \in E \\ x_i \geq 0 & \forall i \in V \\ x_i \leq 1 & \forall i \in V \end{cases}$$

La funzione **obiettivo da minimizzare** è:

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

I vincoli sostanzialmente dicono che il nodo è o 0 o 1, possono essere solo numeri interi, quindi il nodo è preso oppure no, ed il vincolo ≥ 1 stabilisce che ogni arco è preso da almeno un nodo.
Ma questo rimane un problema $\in \mathcal{NPO}c$.

La versione "rilassata" di π è uguale ma **risolto con** $x \in \mathbb{Q}^n$ (LP non ILP). Stessi vincoli, stessa funzione obiettivo.

Quindi, partiamo dal **problema di VertexCover** π, lo abbiamo trasformato in un problema di **programmazione lineare intera** $ILP(\pi)$, ci dimentichiamo della parte intera , **diventa** $LP(\pi)$, ed **arrotondiamo** a partire dalla soluzione ottima x^* di $LP(\pi)$:

$$\pi = (V, E), w_i \rightarrow \underset{w^*_{ILP}}{ILP(\pi)} \rightarrow \underset{w^*_{LP}}{LP(\pi)} \xrightarrow{x^*} \text{Rounding} \rightarrow \hat{x}$$

$$\hat{x} = \begin{cases} 0 & \text{se } x^*_i < \frac{1}{2} \\ 1 & \text{se } x^*_i \geq \frac{1}{2} \end{cases}$$

Chiamando w^*_{ILP} la funzione obiettivo di $ILP(\pi)$ e w^*_{LP} la funzione obiettivo di $LP(\pi)$.

Lemma 1: $w^*_{LP} \leq w^*_{ILP}$.

Proof. Il problema rilassato ha un superinsieme di soluzioni ammissibili, quindi la soluzione ottima sicuramente non può peggiorare, può solo essere migliore.

□

Lemma 2: $\hat{x}$ è una **soluzione ammissibile** di $ILP(\pi)$.

Proof. Ricordando i vincoli

$$\begin{cases} \hat{x}_i + \hat{x}_j \geq 1 & \forall \{i,j\} \in E \\ 0 \leq \hat{x}_i \leq 1 & \forall i \in V \end{cases}$$

E la definizione di $\hat{x}_i$ (sopra).

Inoltre sappiamo che x^*_i rispetterà, in quanto soluzione ammissibile, gli stessi vincoli che sono presenti per $\hat{x}$.

L'unico vincolo importante da verificare è che la somma sia ≥ 1 e l'unico caso in cui può non succedere è quando entrambi sono $= 0$

$$\hat{x}_i + \hat{x}_j \not\geq 1 \implies \hat{x}_i + \hat{x}_j = 0 \implies \hat{x}_i = 0,\ \hat{x}_j = 0$$

Ma questo succede solo quando gli x^*_i erano entrambi $< 1/2$

$$x^*_i = 0,\ x^*_j = 0$$

Ma questo è impossibile in quanto vorrebbe dire che:

$$x^*_i + x^*_j < 1$$

Il che è impossibile per i vincoli del problema $LP(\pi)$, risolto in maniera ottima.

□

Lemma 3: $\forall i,\ \hat{x}_i \leq 2x_i^*$.

Proof. Guardando i possibili casi:

$$\hat{x}_i = 0 \implies x_i^* < 1/2 \implies 0 \leq 2 \cdot 1/2$$

$$\hat{x}_i = 1 \implies x_i^* \geq 1/2 \implies 2x_i^* \geq 1 = \hat{x}_i$$

□

Lemma 4:

$$w = \sum_i w_i \hat{x}_i \leq 2 \sum_i w_i x_i^* = 2w_{LP}^*$$

Questo dice che la soluzione ottenuta dall'algoritmo di arrotondamento è $\leq$ (per il Lemma 3) di 2 volte la soluzione ottima del problema $LP(\pi)$.

Teorema: RoundingVertexCover è una **2-Approssimazione** per VertexCover.

Proof. Per il Lemma 1 per il Lemma 4, rispettivamente, possiamo vedere che:

$$\frac{w}{w_{ILP}^*} \leq \frac{w}{w_{LP}^*} \leq \frac{2w_{LP}^*}{w_{LP}^*} = 2$$

Quindi l'algoritmo di arrotondamento è una **2-Approssimazione di VertexCover**.

□

Sperimentalmente, l'approssimazione peggiora, sia per rounding che pricing VertexCover, con grafi sparsi. In qualsiasi caso sempre abbastanza meglio della 2-Approssimazione, generare input pessimi è difficile, solitamente sono casi estremi.

7 Altri esempi

Nascita della teoria dei grafi

La città di Königsberg è attraversata dal fiume Pregel, nel quale sono presenti due isole collegate da 7 ponti in totale.

La domanda posta a Eulero fu: si può passare da tutti i ponti e tornare all'inizio? Ovvero, esiste un cammino che passa per tutti gli archi (ponti) del grafo per poi tornare al nodo iniziale?

Le due isole diventano 2 vertici e le sponde altri 2. Al giorno d'oggi sarebbe chiamato "multigrafo", avendo più lati incidenti sugli stessi due vertici, ma il problema rimane lo stesso.

Se volessimo formalizzare il problema nella terminologia moderna: dato un multigrafo, esiste un circuito che passa per tutti i lati? Si chiama "circuito Euleriano" (indovina perché).

Teorema di Eulero: Esiste un **circuito Euleriano iff** il grafo è connesso (ovviamente) e **tutti i vertici hanno grado pari**.

Come si **costruisce un circuito**, a partire **da un grafo con vertici di grado pari**?
Partendo da un qualsiasi vertice x_0 e si percorre un qualunque lato, arrivando ad x_1, il quale avrà almeno un altro lato uscente (avendo esso grado pari), che arriva ad x_2, e la cosa si ripete.
Nel caso si crei un ciclo prima di toccare tutti i lati vuol dire che questo ha almeno 4 lati incidenti e quindi si può continuare.
Se si torna ad x_0 prima di toccare tutti i lati, si ricomincia senza considerare i lati già visti.

Handshaking Lemma

"Lemma delle strette di mano", se un gruppo di persone si stringono la mano, il numero di persone che stringono la mano a un numero dispari di persone sono in quantità pari.

Teorema: In un grafo $G = (V, E)$, il numero di vertici di grado ($d()$) dispari è pari.

Proof. Sommando i gradi di tutti i vertici

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

In questo modo ogni lato viene contato 2 volte, quindi è pari 2 volte il numero di lati m. $2m$ è ovviamente pari, e i nodi con grado pari possono non essere considerati ai fini della parità (se tolgo o aggiungo un numero pari la parità non cambia), quindi i nodi con grado dispari devono essere pari, altrimenti la somma potrebbe non essere pari.

□

7.1 Problema del Traveling Salesman (TSP)

Avendo un insieme di città collegate da strade bidirezionali (grafo pesato) un commesso vuole passare per tutte le città una e una sola volta, facendo un percorso di lunghezza minima per poi tornare a casa (vuole fare il giro).

Definizione:

- **Input**: un **grafo non orientato** $G = (V, E)$ con delle **lunghezze** $[\delta_e]_{e \in E}$ per ogni **lato**
- **Soluzioni ammissibili**: circuiti che passano per ogni vertice esattamente una volta (**circuito hamiltoniano**)
- **Funzione obiettivo**: la **lunghezza** del circuito
- **Tipo**: minimizzazione, min

Non è detto che il circuito hamiltoniano esista.

Questo problema è **equivalente al TSP su clique** (quindi su un grafo fully connected). Per usare un algoritmo che funziona solo su clique per grafi generici basta aggiungere i lati mancanti con un peso molto alto (più di ogni possibile cammino tra lati tendenzialmente), in modo che non vengano mai scelti, se la soluzione passa per uno dei lati "finti" nel grafo originale non c'è un circuito hamiltoniano.

Per questo da qui in poi verrà **presupposto che i grafi siano clique** $G = K_n$ (notazione per una clique di n nodi).

TSP Metrico: Un TSP Metrico ha come proprietà:

- G è una clique
- $\forall x, y, z,\ \delta_{xy} \leq \delta_{xz} + \delta_{zy}$ (disuguaglianza triangolare)

Ingredienti mancanti: Due concetti che serviranno per l'algoritmo vero e proprio:

- **Minimum spanning tree**: dato un grafo pesato connesso $G = (V, E)$ trovare un albero di copertura (che tocca tutti i vertici) di peso totale minimo. Risolvibile in tempo polinomiale (es. Algoritmo di Kruskal).

- **Minimum weight perfect matching**: Data una clique pesata un numero pari di vertici trovare un matching perfetto (completo) di peso minimo. Risolvibile esattamente in tempo polinomiale (Blossom algorithm)

Ora possiamo cucinare.

7.1.1 Algoritmo di Cristophides

Algoritmo **per il TSP metrico**:

- Input: $G = (V, E)$ clique, con pesi $[\delta_e]_{e \in E}$ che sono una metrica

Passaggi dell'algoritmo:

1. **Troviamo** un **minimum spanning tree** T.

2. Sia D l'insieme dei **vertici di grado dispari in** T. Per Handshaking Lemma sappiamo che D ha **cardinalità pari**.

3. Scegliamo un **minimum weight perfect matching** M **sui nodi presenti in** D (sicuramente presente, dato che quei nodi fanno parte di una clique e sono pari).

4. Il perfect matching potrebbe riusare lati dell'albero, se il lato compare sia nel MST che nel matching bisognerà andare a considerare un multigrafo.
   Sia $H = T \dot{\cup} M$ (unione tenendo le ripetizioni). Tutti i vertici di H avranno grado pari, in quanto tutti quelli che avevano grado dispari hanno ricevuto un lato bonus dal matching.

5. Avendo tutti grado pari si può usare il teorema di Eulero, quindi **troviamo un circuito euleriano** π.

6. Ma questo può passare per lo stesso vertice più di una volta, si tratta di un circuito euleriano, noi lo vogliamo hamiltoniano. Quindi **trasformo** π **in un circuito hamiltoniano** $\tilde{\pi}$. Per farlo bisogna "strozzare i cappi", quando π passa per un nodo già preso, lo salto, tanto siamo in una clique, quindi al posto di fare il giro x, y, z, se ho già preso prima y, posso fare direttamente x, z.

Insomma, seguo il circuito euleriano ma salto i nodi visitati più volte, rendendolo un circuito hamiltoniano. Questo è l'output.

Lemma 1: $\delta(T) \leq \delta^*$, la somma dei pesi presenti nel MST è minore il peso del circuito hamiltoniano ottimo.

Proof. Sia π^* un TSP ottimo. Se da π^* togliamo un lato otteniamo un albero di copertura minimo (da un circuito, se tolgo un lato non ho più cicli, quindi ho un albero), quindi

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

□

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

Proof. Sia π^* is TSP ottimo. Dentro questo ci saranno anche i vertici di grado dispari presenti in D che deve coprire il matching M.

Prendendo un circuito tra elementi di D (vertici di grado dispari) e chiamandolo $\overline{\pi}^*$, ma questo ovviamente

$$\overline{\pi}^* \leq \delta(\pi^*)$$

Sto saltando dei pezzi dal circuito originale, sicuramente è meno.

Dividiamo i lati di $\overline{\pi}^*$ in due insiemi M_1 e M_2 alternati. Ognuno di questi due gruppi di lati è un perfect matching su D, ma sappiamo che M è il minimum perfect matching, quindi:

$$\delta(M) \leq \delta(M_1)$$

$$\delta(M) \leq \delta(M_2)$$

Di conseguenza

$$2\delta(M) \leq \delta(M_1) + \delta(M_2) = \delta(\overline{\pi}^*) \leq \delta(\pi^*)$$

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

□

Teorema: L'algoritmo di Cristophides fornisce una $3/2$**-approssimazione per il TSP metrico**.

Proof.

$$\delta(\pi) = \delta(T) + \delta(M)$$

Usando i due lemmi:

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

$$\implies \delta(\pi) \leq \frac{3}{2}\delta^*$$

Per la triangolare (durante la trasformazione prendo solo lati in meno, mai in più):

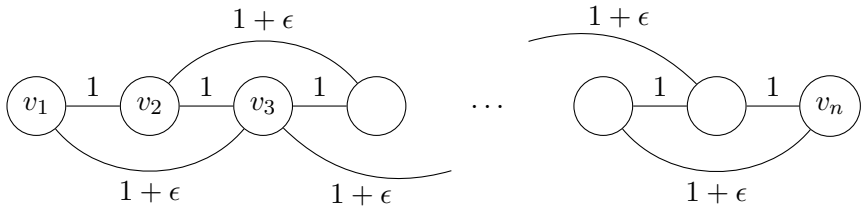
$$\delta(\tilde{\pi}) \leq \delta(\pi) \leq \frac{3}{2}\delta^*$$

□

Tightness: l'analisi è stretta, esistono input per cui l'algoritmo restituisce ${}^3\!/_2\delta^*$.

Proof. $n \in \mathbb{N}$ pari e $\epsilon \in (0, 1)$. Prendiamo n nodi, li colleghiamo in un cammino in fila e tutti questi lati hanno lunghezza 1.
Aggiungo lati da 1 a 3, da 2 a 4, da 3 a 5, da 5 a 7, tra tutti i pari e tra tutti i dispari, ognuno di lunghezza $1 + \epsilon$.

Sia $K_{n,\epsilon}$ la clique ottenuta aggiungendo tutti i lati mancanti, ognuno dei quali con lunghezza pari al cammino minimo tra i due vertici collegati. Ovviamente è metrica in quanto per costruzione rispetta la triangolare.



Cristophides sceglie il MST, che in questo caso saranno tutti i lati di lunghezza 1. T prende tutti gli $n-1$ lati di lunghezza 1, quindi

$$\delta(T) = n - 1$$

Gli unici vertici di grado dispari sono il primo e l'ultimo, quindi il minimum perfect matching sarà il lato tra v_1 e v_n, il quale sarà pari al cammino minimo tra i due ovvero tutti gli $1+\epsilon$ possibili, più l'ultimo lato:

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

Il cammino hamiltoniano coinciderà con quello euleriano quindi:

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

Ma il circuito ottimo è prendere due lati da 1 affianco a primo e ultimo nodo, poi tutti i lati da $1+\epsilon$, quindi il totale:

$$\delta^* = (1+\epsilon)n + 2$$

Quindi il rapporto di approssimazione diventa:

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

Tende a $3/2$ per $n \to +\infty$ e $\epsilon \to 0$.

□

7.1.2 Inapprossimabilità del TSP

TSP Metrico è approssimabile, **quello generale no**.

Teorema: Non esiste $\alpha > 1$ tale che TSP sia α-approssimabile (se $\mathcal{P} \neq \mathcal{NP}$). Il problema sta fuori da APX.

Proof. Fact: Il problema di **decidere se un grafo ammetta un circuito hamiltoniano** è $\mathcal{NP}$-Completo (problema di decisione).

Supponiamo per assurdo di **avere un algoritmo α-approssimante per TSP**.

Supponendo di avere un grafo $G = (V, E)$ e voler sapere se questo grafo ha un circuito hamiltoniano. Per **trasformarlo in un'istanza di TSP**, dobbiamo **aggiungere pesi** e **trasformarlo in una clique**

$$G' = \left(V, \binom{V}{2}, d\right)$$

Bisogna aggiungere le **distanze**, dove

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

Si tratta di una clique, quindi per forza ci deve essere un circuito hamiltoniano su G', ma se G **ammetteva** un circuito hamiltoniano a sua volta, allora G' ne ha uno con lunghezza $\leq n$.
Se G **non ha un circuito hamiltoniano**, qualunque circuito hamiltoniano viene trovato **su** G' ha **lunghezza** $\geq \lceil \alpha n \rceil + 1$.

Diamo G' come **input dell'algoritmo** α-approssimante per TSP, questo **emetterà** nei casi:

- G ha un c.H.: l'algoritmo emette un output $\leq \alpha n$ (dato che si tratta di una α-approssimazione)
- G non ha un c.H.: l'algoritmo emette un output $\geq \lceil \alpha n \rceil + 1$

Quindi possiamo avere output $\leq \alpha n$ e $\geq \lceil \alpha n \rceil + 1$, ma *è possibile che questi due intervalli si sovrappongano*?

$$\alpha n \geq \lceil \alpha n \rceil + 1$$

$$\implies \alpha \geq \frac{\lceil \alpha n \rceil + 1}{n} \geq \frac{\alpha n + 1}{n} = \alpha + \frac{1}{n}$$

$$\alpha \geq \alpha + \frac{1}{n}$$

Quindi sono **intervalli sempre disgiunti** e di conseguenza le soluzioni sono ben distinte.

Questo vorrebbe dire risolvere il problema di decidere se un grafo ammetta un circuito hamiltoniano efficientemente, che sappiano non essere possibile essendo in $\mathcal{NP}$.

□

8 Schemi di approssimazione PTAS e FPTAS

8.1 PTAS per 2-LoadBalancing

Ricordando cos'è un PTAS: problemi approssimabile a meno di una costante desiderata (arriva al tasso di approssimazione desiderato).

LoadBalancing già visto, quello delle macchine e task, bla bla bla. 2-LoadBalancing è la stessa cosa ma ha **solo 2 macchine**.

Definizione del problema:

- **Input**: sequenza di n task $t_0, \ldots, t_{n-1}$
- **Soluzioni ammissibili**: assegnamenti dei task su 2 macchine, una funzione $\alpha : n \to 2$
- **Funzione obiettivo**: massimo carico tra le due macchine

$$\max\left(\sum_{i:\alpha(i)=0} t_i, \sum_{i:\alpha(i)=0} t_a\right)$$

- **Tipo**: minimizzazione, min

Per l'**algoritmo PTAS**:

- **Input**: i task $t_0, \ldots, t_{n-1}$ e un **costante di approssimazione** (circa) $\epsilon > 0$ (per avere una $1+\epsilon$-approssimazione)

Passaggi:

- If $\epsilon \geq 1$:
  - Assegna tutti i task alla prima macchina
  - Termina

- Altrimenti, ordina i t_i in **ordine decrescente**.

- **Fase 1**:
  - **Calcola** $k \leftarrow \lceil \frac{1}{\epsilon} - 1 \rceil$
  - Si **cerca esaustivamente** l'assegnamento ottimo dei task $t_0, \ldots, t_{k-1}$ (primi k task, 2^k assegnamenti possibili)

- **Fase 2**:
  - I **restanti task** $t_k, \ldots, t_{n-1}$ sono **assegnati** in modo **greedy** (sempre alla macchina più scarica)

L'assegnamento dei task "grossi" (i primi) viene risolto a forza bruta, il resto greedy.

Teorema: l'algoritmo è una $(1+\epsilon)$-approssimazione di 2-LoadBalancing.

Proof. Chiamando la **somma dei task**

$$T := \sum_{i \in n} t_i$$

Se $\epsilon \geq 1$, otteniamo una 2-Approssimazione (butto tutto su una, di sicuro non può essere peggio di 2 volte la soluzione ottima).

$$L^* \geq \frac{T}{2}, \quad L = T$$

$$\implies \frac{L}{L^*} \leq \frac{T}{T/2}$$

Nel caso in cui $\epsilon < 1$: alla **fine della prima fase** abbiamo l'**assegnamento ottimo** per i primi k task e un carico sulle macchine 0 e 1, rispettivamente y_0 e y_1.

Rimangono **da assegnare** i task $t_k, \ldots, t_{n-1}$ per arrivare agli **assegnamenti finali** L_0 e L_1.

Assumiamo (senza perdita di generalità) che $L_0 \geq L_1$ (**alla fine** la macchina con il **carico maggiore** è **la prima**).

Due casi:

1. $L_0 = y_0$, dopo la prima fase tutti i restanti task sono stati attribuiti alla seconda macchina.
   $\implies$ Abbiamo **ottenuto l'assegnamento ottimo**, per costruzione dell'algoritmo.

2. Uno o più task di $t_k, \ldots, t_{n-1}$ sono stati assegnati alla prima macchina, sia t_h l'**ultimo assegnato** a suddetta macchina (il più piccolo). Di conseguenza
   $$L_0 - t_h \leq L_1' \leq L_1$$
   (dove L_1' è il carico della seconda macchina prima dell'assegnamento di t_h). Ovviamente prima di assegnare t_h il carico sulla prima macchina era minore del carico sulla seconda macchina
   $$\implies 2L_0 - t_h \leq L_0 + L_1$$
   $$T = L_0 + L_1$$
   $$\implies L_0 - \frac{t_h}{2} \leq \frac{T}{2}$$
   $$L_0 \leq \frac{T}{2} + \frac{t_h}{2}$$
   Ricordiamo che **i primi k task sono** $\geq (k+1)t_h$, in quanto ordinati in ordine decrescente
   $$T \geq (k+1)t_h$$
   I task dopo t_h li minoriamo con 0.
   $$\frac{T}{2} \geq (k+1)\frac{t_h}{2}$$

Quindi

$$\frac{L_0}{L^*} \leq \frac{L_0}{T/2} \leq \frac{\frac{T}{2} + \frac{t_h}{2}}{T/2} = 1 + \frac{t_h}{T} \leq 1 + \frac{t_h}{(k+1)t_h} = 1 + \frac{1}{(k+1)}$$

Ma **per definizione di** k

$$\frac{L_0}{L^*} \leq 1 + \frac{1}{(k+1)} \leq 1 + \frac{1}{\frac{1}{\epsilon} - 1 + 1} = 1 + \epsilon$$

$$\implies \frac{L_0}{L^*} \leq 1 + \epsilon$$

□

Teorema: Il PTAS **richiede tempo** $O\left(n \log n + 2^{\min\left(\frac{1}{\epsilon}, n\right)}\right)$.

Proof. $n \log n$ è il **tempo** per l'**ordinamento**.

L'**esponenziale** è la **fase di assegnamento esatto**.

Abbiamo 2^k assegnamenti ma

$$2^k = 2^{\lceil \frac{1}{\epsilon} - 1 \rceil} \sim 2^{\frac{1}{\epsilon}}$$

Ci sarebbe anche la parte di assegnamento greedy ma talmente breve che non conta.

□

Quindi abbiamo un algoritmo PTAS, con un **tasso di approssimazione arbitrariamente preciso** ma con **tempo che peggiora esponenzialmente** in base a quest'ultimo.

8.2 Knapsack Problem

I pirati arrivano in una grotta con degli oggetti, ognuno di questi n oggetti ha un valore

$$v_0, \ldots, v_{n-1}$$

ed i pirati vogliono portarsi a casa il maggior valore possibile, ma per portarli hanno solo uno zaino con una capacità W e ogni oggetto ha un peso

$$w_0, \ldots, w_{n-1}$$

I pirati vogliono portare a casa il maggior valore possibile con un peso che rientra nello zaino.

Definizione del problema:

- **Input**: $n > 0$ numero di oggetti, $v_i, w_i \in \mathbb{N}^+$ valori e pesi associati agli oggetti, $i < n$, capacità dello zaino $W > 0$
- **Soluzioni ammissibili**: $X \subseteq n$ tale che

$$\sum_{i \in X} w_i \leq W$$

- **Funzione obiettivo**: la somma dei valori

$$v = \sum_{i \in X} v_i$$

- **Tipo**: massimizzazione, max

Side quest: Programmazione Dinamica: Vogliamo risolvere un certo problema π, nel quale sono presenti dei parametri $\pi[\overline{a}, \overline{b}]$, ma il problema si può presentare in varianti con valori diversi da $\overline{a}$ e $\overline{b}$.

Voglio poter **risolvere** una certa **istanza di un problema a partire da istanze precedenti**/più semplici/con valori più semplici risolte in precedenza.
Devo quindi risolvere soluzioni più semplici per arrivare a quelle più grosse, le soluzioni dipendono da quelle prima.

Il numero di problemi da risolvere prima della soluzione che cerco è pari a tutte le possibili combinazioni di valori di parametri fino a $\overline{a}$ e $\overline{b}$. Si possono pensare le diverse istanze da risolvere come se fossero in una tabella con da un lato tutti i valori possibili del parametro $\overline{a}$ e dell'altro i possibili valori di $\overline{b}$.

	0	1	...	$\overline{a}$
0				
1				
...				
$\overline{b}$				

In ognuna delle celle la soluzione dell'istanza di π con i relativi valori dei parametri.

Fondamentale i parametri siano valori interi, altrimenti si può risolvere tutto in tempo polinomiale.

Side-side-quest: pseudopolinomialità: Un algoritmo è pseudopolinomiale quando il suo **tempo di esecuzione dipende dal valore numerico degli input** considerati, al posto che dalla grandezza della rappresentazione. Un valore n in binario richiede $b = \lceil \log_2 n \rceil$ bit, ma il valore considerato è $n \leq 2^b$, quindi il tempo d'esecuzione è esponenziale rispetto alla dimensione dell'input.

8.2.1 Soluzione di Programmazione Dinamica: Versione A

Partendo dalle n coppie di valori-pesi.

I parametri secondo cui iterare con la programmazione dinamica (righe e colonne della tabella) sono:

- W, **capienza dello zaino**, risolvo il problema per tutti i valori da 0 a W
- **Numero di oggetti**, considero solo i primi i oggetti, variando il valore di i tra 0 e n

Posso costruire una tabella con in **ogni riga tutte le soluzione per un possibile valore per la capienza dello zaino** W e la colonna i contiene le soluzioni considerando solo i primi i oggetti. Quindi la cella $3, 2$ contiene il valore massimo ottenibile con i primi 3 oggetti e capienza 2.

Ogni cella della tabella contiene il valore massimo che i pirati si portano a casa, i.e., il risultato del problema, i.e., il **valore della funzione obiettivo**.

La prima riga, non importa quanti oggetti considero, se lo zaino ha capienza 0 non posso metterci nulla.
La prima colonna anch'essa è tutti zero, in quanto non ho oggetti, anche con un camion al posto dello zaino nella grotta non trovo nulla.

Quindi come posso trovare il **valore** $v(w, i)$ **per una qualsiasi cella?** Presupponendo di riempire la tabella da sinistra verso destra, dall'alto verso il basso. La w rappresenta la capienza considerata, la funzione v rappresenta il valore portato a casa e la i indica che abbiamo considerato i primi i oggetti, quelli che hanno da indice 0 a $i - 1$.

Guardo **i valori precedenti**, partendo da $v(w, i - 1)$:

- Se decido di non aggiungere l'i-esimo elemento il risultato non cambia, in quanto ho già trovato il risultato migliore per quello stesso spazio
- Altrimenti, se voglio aggiungere l'i-esimo elemento con valore v_i, aggiungo il suo valore, ma lo zaino deve essere abbastanza capiente

Formalmente:

$$v(w,i) = \begin{cases} v(w, i-1) & \text{se } w < w_{i-1} \\ \max\left(v(w,i-1), v_{i-1} + v(w - w_{i-1}, i-1)\right) & \text{altrimenti} \end{cases}$$

Quindi, se non c'è abbastanza spazio ($w < w_{i-1}$) per aggiungere l'i-esimo elemento il valore rimane lo stesso.

Altrimenti prendo il massimo tra il valore della cella subito a sinistra e il valore della migliore cella la cui capienza associata mi permette di aggiungere l'i-esimo elemento.

Il numero di colonne è n, ma tutti i possibili valori di W sono pseudo-polinomiali $\implies$ complessità esponenziale.

8.2.2 Soluzione di Programmazione Dinamica: Versione B

Tengo la nostra tabella per la programmazione dinamica, in riga sempre il numero di oggetti considerati, ma **in colonna metto il valore che i pirati vogliono portarsi a casa**, di conseguenza **nelle celle** calcolo **la minima capacità di uno zaino per ottenere quel valore**. Stesso problema, stiamo solo cambiando la rappresentazione.

Se il valore obiettivo è 0, la dimensione necessaria è sempre 0, si tratta di un pirata molto umile.
Se voglio portare a casa un valore > 0, ma sto considerando 0 elementi risulta difficile $\implies$ la prima colonna è tutta $+\infty$ (tranne la prima cella a zero).

Come ottengo **una cella qualunque** $w(v, i)$?

Se decido di non portare a casa anche l'i-esimo oggetto rimane tutto come $w(v, i-1)$, altrimenti diventa $w_{i-1} + w(v - v_{i-1}, i-1)$, quindi

$$w(v,i) = \begin{cases} \min\left(w(v,i-1), w_{i-1}\right) & \text{se } v < v_{i-1} \\ \min\left(w(v,i-1), w_{i-1} + w(v - v_{i-1}, i-1)\right) & \text{altrimenti} \end{cases}$$

Se il valore cercato è minore del valore dell'oggetto che ho appena introdotto posso scegliere di prendere solo quest'ultimo.

Altrimenti prendo il minimo tra il valore subito a sinistra e la somma tra il peso dell'i-esimo oggetto e il peso della cella contenente il valore minimo necessario per arrivare al nostro obiettivo una volta sommato il valore dell'oggetto appena inserito.

Quindi, compilo la tabella, ma **avrò delle celle in cui il peso minimo è too much**, $> W$, l'**output** per il nostro problema lo **ottengo risalendo** dalla cella in basso a destra verso l'alto **finché non trovo un valore di capienza minima ammissibile** $w < W$, i.e., prendo il valore ammissibile più alto possibile.

8.2.3 Algoritmo DP Approssimato

Problema: gli algoritmi precedenti sono esatti, ma sono entrambi esponenziali nel numero di righe.

Per **approssimare il problema** possiamo "*comprimere le righe*", ovvero **dividiamo tutti i valori per un certo numero**, **diminuendo** così il **numero di righe** necessarie, fino a far diventare l'algoritmo polinomiale.

Si può fare **solo con la seconda versione**, in cui le righe rappresentano la funzione obiettivo (che possiamo approssimare, andando a ottenere un subottimo), mentre con la prima versione non possiamo comprimere la capacità dello zaino, in quanto parte dei constraint del problema, rischieremmo di andare verso soluzioni non ammissibili.

Quindi, applichiamo il secondo algoritmo, ma con **scaling**.

Al problema $\pi(v_i, w_i, W)$ aggiungiamo $\epsilon \in (0, 1]$ e vogliamo una $(1+\epsilon)$-**approssimazione** di π

$$\theta = \frac{\epsilon \cdot v_{max}}{2n}$$

Introduciamo **due ulteriori versioni del problema**:

$$\overline{\pi}\left(\overline{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, w_i, W\right)$$

Quindi arrotondo un po' i valori per eccesso, mentre in

$$\hat{\pi} = \left(\hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil, w_i, W\right)$$

approssimo ma lascio compresso, quindi noi vogliamo risolvere $\hat{\pi}$ in modo esatto tramite il secondo algoritmo di programmazione dinamica visto precedentemente.

Dobbiamo capire **come sono correlate le tre soluzioni**:

$$\hat{X}^* \subseteq n, \quad \overline{X}^* \subseteq n \quad X^* \subseteq n$$

Osservazione: l'**ammissibilità è la stessa** per tutti e 3 i problemi, tutti i problemi hanno le stesse soluzioni ammissibili.

Osservazione: Le soluzioni

$$\overline{X}^* = \hat{X}^*$$

in quanto sono problemi uguali, i valori sono solo moltiplicati per una costante. **Queste soluzioni sono uguali**.

Lemma: Sia X una soluzione ammissibile,

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

La soluzione ottima dei nostri problemi modificati, moltiplicata per la costante di approssimazione, è sempre $\geq$ di qualsiasi soluzione ammissibile.

Proof. Dato che $\overline{\pi}$ ha solo degli arrotondamenti per eccesso, tutti i valori saranno un po' più grandi, quindi

$$\sum_{i\in X} v_i \leq \sum_{i\in X} \overline{v}_i \leq \sum_{i\in\overline{X}^*} \overline{v}_i$$

E ovviamente, la soluzione ottima per $\hat{\pi}$ ha un valore maggiore di una qualsiasi soluzione ammissibile.

Per come sono definiti

$$\overline{v}_i - v_i \leq \theta$$

Quindi

$$\sum_{i\in\overline{X}^*} \overline{v}_i \leq \sum_{i\in\overline{X}^*} (v_i+\theta) = \sum_{i\in\overline{X}^*} v_i + |\overline{X}^*|\theta \leq$$

$$\leq \sum_{i \in \overline{X}^*} v_i + n\theta = \sum_{i \in \overline{X}^*} v_i + n\frac{\epsilon \cdot v_{max}}{2n}$$

Quindi

$$\sum_{i \in X} v_i \leq \sum_{i \in \overline{X}^*} v_i + \frac{\epsilon \cdot v_{max}}{2}$$

è vera per ogni soluzione ammissibile X.

Prendo solo l'indice dell'elemento del valore massimo:

$$X = \{i_{max}\}$$

(ovviamente escludendo gli elementi con $w_i > W$).

Questa è una soluzione ammissibile, quindi

$$v_{max} \leq \sum_{i \in \overline{X}^*} v_i + \frac{\epsilon \cdot v_{max}}{2} \leq$$

Essendo $\epsilon \leq 1$

$$\leq \sum_{i \in \overline{X}^*} v_i + \frac{v_{max}}{2}$$

$$\implies \frac{v_{max}}{2} \leq \sum_{i \in \overline{X}^*} v_i$$

Quindi la disuguaglianza precedente

$$\sum_{i \in X} v_i \leq \sum_{i \in \overline{X}^*} v_i + \frac{\epsilon \cdot v_{max}}{2}$$

diventa

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

Ma ricordando che $\hat{X}^* = \overline{X}^*$

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

□

Teorema: Il lemma vale per tutte le soluzioni, anche quella originale

$$(1+\epsilon)\hat{v}^*\theta \geq v^*$$

Proof. Il Lemma dice che

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

Ma la prima parte è $\leq \hat{v}^*\theta$ e la seconda è una qualsiasi soluzione ammissibile, quindi anche X^* e di conseguenza v^*.

□

Comprimo i valori, calcolo tutto, ma alla fine devo **decomprimerli** per sapere il mio risultato, per questo ho il θ aggiunto, la discrepanza è data dall'approssimazione per eccesso effettuata.

Corollario: L'algoritmo è una $(1+\epsilon)$**-approssimazione**.

Complessità: Devo guardare le celle della tabella, ovvero il numero di colonne per il numero di righe, quindi

$$n \cdot \sum \hat{v}_i \leq n^2 \hat{v}_{max} = n^2 \left\lceil \frac{v_{max}}{\theta} \right\rceil =$$

$$= n^2 \left\lceil \frac{2v_{max}n}{\epsilon \cdot v_{max}} \right\rceil = O\left(\frac{n^3}{\epsilon}\right)$$

la disuguaglianza è data dal fatto che ognuno degli n elementi è per forza minore del massimo v_{max}

In totale quindi diventa

$$O\left(\frac{n^3}{\epsilon}\right)$$

Ovvero, polinomiale in n e in ϵ, quindi **è un FPTAS**.

9 Algoritmi probabilistici

Cambia la **nozione di algoritmo**, oltre all'input del problema l'algoritmo è in grado di leggere da una **sorgente di bit casuali** (0 o 1 con pari probabilità). Sostanzialmente, una DTM con accesso anche a un nastro casuale utilizzabile quando necessario.

L'**output** quindi non è più deterministico, ma una **distribuzione di probabilità che dipende dall'input**. Anche il **tempo di esecuzione** diventa una **distribuzione** in base all'input.

Aggiungiamo un **componente probabilistico** agli algoritmi. L'output è diventa completamente deterministico se potessimo fissare i bit generati dalla componente casuale.

Nessun computer è in grado di simulare veramente una sorgente random, in quanto macchine deterministiche. Senza congegni strani ci si accontenta di **generatori pseudo-casuali PRNG**, ovvero una sequenza che sembra casuale ma che parte da un seed, e quindi riproducibile. Creano sequenze deterministiche che *sembrano* casuali.

9.1 Problema del Taglio Minimo Globale (MinCut)

Definizione:

- **Input**: un grafo non orientato $G = (V, E)$
- **Soluzione ammissibile**: insieme di vertici $X \subseteq V$, con $X \neq \emptyset$ e che non deve avere complemento vuoto $X^C \neq \emptyset$ (dividere i vertici in due gruppi)
- **Funzione obiettivo**: l'insieme dei lati che hanno un'estremità in X ed una estremità nel complemento di X

  $$\{e \in E | e \cap X \neq \emptyset, e \cap X^C \neq \emptyset\}$$

  i.e., lati facente parti del taglio
- **Tipo**: minimizzazione min

Si tratta di un problema $\in \mathcal{NPO}c$.

Un **taglio** è una partizione dei vertici di un grafo V in due sottoinsiemi disgiunti S, T. Gli archi che collegano un vertice di S a uno di T formano il taglio. Un taglio è **minimo** se il suo peso è minimo rispetto a tutti gli altri possibili tagli (insieme di lati più piccolo possibile).

Lemma: Il taglio minimo è $\leq$ grado minimo.

Il taglio può essere al minimo 1, nel caso di un ponte tra due gruppi di nodi. Al massimo può essere il grado minimo in quanto basta tagliare quel numero di lati per dividere un nodo da tutti gli altri.

9.1.1 Algoritmo di Karger

Contrazione di un grafo su un lato e: Partendo da un grafo sul quale è presente un lato e legato ai vertici u e v, una contrazione si ottiene rimuovendo e e facendo coincidere u e v.

Da notare che se un terzo vertice x è collegato sia a u che a v, allora bisogna collegare con due archi il vertice risultato della contrazione $\{u, v\}$ a x. Contrarre un lato che ha dei "paralleli" vuol dire contrarre anche quest'ultimi.

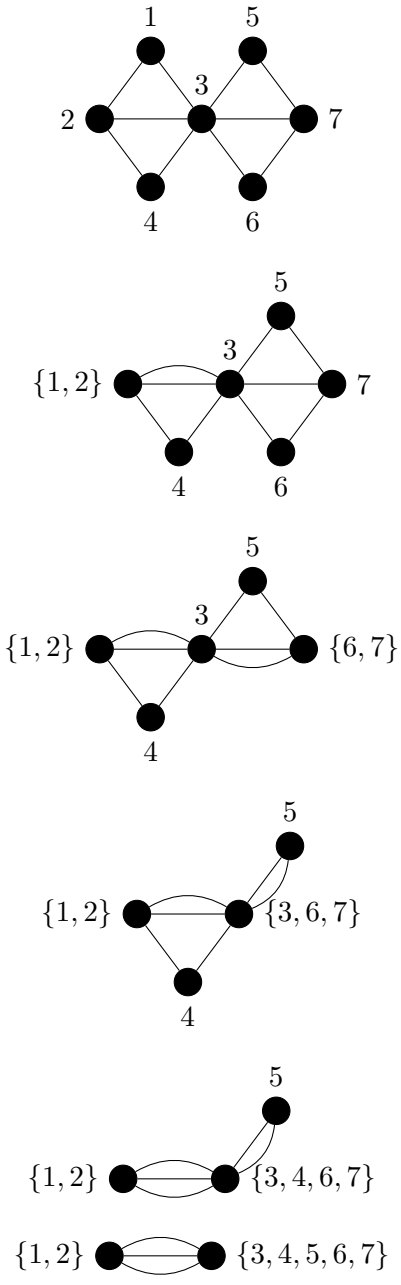
Indichiamo la contrazione di e su G con $G \downarrow e$.

Passaggi dell'algoritmo:

- Se G non è connesso, emetti una componente connessa qualunque
- Altrimenti, finché ci sono più di 2 vertici, scegli un **lato casuale** e **contrai**
- Emetti la **classe di equivalenza** di uno dei due vertici

I due gruppi risultanti dal taglio corrisponderanno ai due gruppi di vertici "compressi" e i lati da tagliare saranno i vertici presenti tra quei gruppi di nodi.

Esempio: Grafo G



Quindi possiamo dividere i vertici in $X = \{1, 2\}$ e $X^C = \{3, 4, 5, 6, 7\}$ tagliando, nel grafo originale i lati $(1, 3), (2, 3), (2, 4)$.

Questo algoritmo **può ottenere la soluzione ottima**, ma anche **soluzioni arbitrariamente brutte**, non è un algoritmo di approssimazione, è **probabilistico**, potrebbe trovare qualcosa di molto lontano dall'ottimo, l'importante è che ci sia anche una probabilità di trovare quest'ultimo. Dobbiamo dimostrare che è presente una probabilità positiva di ottenere l'ottimo.

Proof. Chiamiamo il grafo iniziale G_1 e G_i il grafo prima dell'i-esima iterazione

$$G = G_1 \xrightarrow{G_1 \downarrow e_1} G_2 \xrightarrow{G_2 \downarrow e_2} \dots$$

Chiamiamo X^* il taglio minimo e k^* la dimensione di quest'ultimo.

Osservazioni:

1. G_i ha $n - i + 1$ vertici (contrarre riduce i vertici sempre di 1)
2. G_i ha $\leq m - i + 1$ lati dato che ne cancelliamo almeno 1 (potrebbero esserci paralleli) ogni volta
3. Ogni taglio di G_i corrisponde a un taglio di G con la stessa dimensione
4. Quindi il grado minimo di G_i è $\geq k^*$, altrimenti (come visto nel Lemma) il taglio minimo non sarebbe davvero minimo
5. Se m_i sono i lati di G_i

   $$2m_i = \sum_{v \in V_{G_i}} d_{G_i}(v)$$

   sarà pari alla metà della somma dei gradi di tutti i vertici, che a sua volta (per l'osservazione precedente)

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

Chiamando l'evento E_i = "all'i-esima contrazione non contraiamo uno dei lati tagliati dal taglio minimo" (quindi dalla soluzione ottima).

Lemma:

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

Dove la prima parte rappresenta la probabilità di togliere all'i-esimo passaggio uno dei lati necessari per il taglio minimo, dopo aver avuto la fortuna che ciò non sia accaduto per i primi $i-1$ passi.

Di conseguenza:

$$\begin{aligned} P[E_i|E_1, \ldots, E_{i-1}] &= 1 - P[\overline{E}_i|E_1, \ldots, E_{i-1}] \\ &= 1 - \frac{k^*}{m_i} \\ &\geq 1 - \frac{k^* \cdot 2}{k^*(n-i+1)} \\ &= \frac{n-i-1}{n-i+1} \end{aligned}$$

Quindi questa è la probabilità che all'i-esimo passo io non ho mai escluso nessuno dei lati necessari per il taglio minimo.

□

Teorema: L'algoritmo di Karger emette il **taglio minimo** con **probabilità**

$$\geq \frac{1}{\binom{n}{2}}$$

Proof. Sapendo che

$$P[E_1 \wedge E_2 \wedge \ldots \wedge E_{n-2}] = P[E_1] \cdot P[E_2|E_1] \cdot \ldots \cdot P[E_{n-2}|E_1, \ldots, E_{n-3}]$$

Per la proprietà della catena di eventi. Ma ognuna di quelle probabilità è nota, per il Lemma:

$$\begin{aligned} \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \ldots \cdot \frac{1}{3} &= \frac{(n-2)!2}{n!} \\ &= \frac{2}{n(n-1)} \\ &= \frac{1}{\binom{n}{2}} \end{aligned}$$

□

Quindi una probabilità relativamente piccola, che scala molto in fretta con n, ma **positiva**.

Corollario: Eseguendo l'algoritmo di Karger $\binom{n}{2} \ln n$ volte si ottiene il taglio minimo con probabilità $\geq 1 - \frac{1}{n}$ (possiamo dire "quasi certo", dato che tende ad 1 per $n \to +\infty$).

Proof. Dimostrazione: Sapendo che $\forall x > 1$

$$\frac{1}{4} \leq 1 - \left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}$$

Qual è la probabilità di NON trovare mai l'ottimo? Sarà

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

□

Quindi, eseguendo un algoritmo probabilistico che ha tempo polinomiale un numero polinomiale di volte ho sempre tempo polinomiale come risultato, ma sono *quasi* certo di ottenere l'ottimo.

Se eseguo tante volte un algoritmo che può darmi soluzioni molto brutte, ma potenzialmente anche l'ottimo, prima o poi qualcosa di buono lo trovo.

Cose®

Una serie di *Cose*® che potrebbero essere utili:

1. **Chain rule (regola della catena):** la probabilità della congiunzione di eventi si può sempre esprimere come probabilità del primo per probabilità del secondo dato il primo, fino all'ultimo

$$P[E_1 \wedge E_2 \wedge \ldots \wedge E_n] = P[E_1] \cdot P[E_2|E_1] \cdot \ldots \cdot P[E_n|E_1 \ldots E_{n-1}]$$

2. $\forall x > 1$

$$\frac{1}{4} \leq \left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}$$

3. **Union bound:** la probabilità dell'unione di eventi è minore della somma di probabilità degli eventi (uguale solo se sono tutti disgiunti):

$$P\left[\bigcup_i E_i\right] \leq \sum_i P[E_i]$$

4. **Disuguaglianza di Markov:** avendo una variabile aleatoria x non negativa e con media finita e un $\alpha > 0$, allora

$$P[x \geq \alpha] \leq \frac{E[x]}{\alpha}$$

   la probabilità che x sia maggiore di α è minore uguale della media di x diviso α.

5. $\forall x \in [0, 1]$

$$1 - x \leq e^{-x}$$

6. **Teorema delle probabilità totali:** la probabilità di un evento può essere calcolata sommando le probabilità condizionate di tale evento rispetto a un insieme completo di eventi mutuamente esclusivi.

9.2 Problema SetCover

Già visto, per più informazioni guarda il `capitolo a riguardo`.

Definizione:

- **Input**: $S_0, \ldots, S_{m-1}$, con $\bigcup_{i \in m} = U$ con pesi $w_o, \ldots, w_{m-1} \in Q^+$
- **Soluzioni ammissibili**: $I \subseteq m$ tale che

$$\bigcup_{i \in I} S_i = U$$

- **Funzione obiettivo**:

$$w = \sum_{i \in I} w_i$$

- **Tipo**: minimizzazione, min

Chiamo $n := |U|$ la **cardinalità dell'universo**.

Si può formulare come un **problema di programmazione lineare intera ILP**. Prendo $x_0, \ldots, x_{m-1}$ variabili, ognuna di queste dice se includere o meno il relativo insieme. La funzione da minimizzare è

$$w_0 x_0 + w_1 x_1 + \ldots + w_n x_n$$

Quindi i vincoli sono:

$$\begin{cases} 0 \leq x_j \leq 1 & \forall j \in n \\ \sum_{i:p \in S_i} x_i \geq 1 & \forall p \in U \end{cases}$$

La prima riga vuol dire che **ogni variabile può essere tra 0 e 1** (ma abbiamo solo valori interi), di conseguenza prendere o meno il set.
La seconda riga vuol dire che **almeno un insieme deve coprire ogni punto dell'universo**.

Risolvendo questo problema sugli interi si ottiene la soluzione ottima, ma si può rilassare su numeri reali per risolverlo (non ottimamente) in tempo polinomiale. Consideriamo $\hat{\pi}$ la **versione rilassata** di programmazione lineare non intera di questo problema.

9.2.1 Arrotondamento Aleatorio

L'idea dietro l'algoritmo di arrotondamento aleatorio per SetCover è la seguente.

Algorithm 10 ArrotondamentoAleatorioSetCover()

Costruiamo π
Risolviamo la versione rilassata $\hat{\pi}$, ottenendo soluzione $\hat{x}$ (vettore di valori soluzione)
$I \leftarrow \emptyset$
for $i \in m$ **do**
  for $t = 1, \ldots, \lceil k + \ln n \rceil$ **do**
    aggiungiamo i a I con probabilità $\hat{x}_i$
  end for
end for
Output I

Il `for` per l'inserimento può essere visto anche come: "Inserisci i in I con probabilità $1 - (1 - \hat{x}_i)^{\lceil k + \ln n \rceil}$".

Usiamo la **soluzione $\hat{x}_i$ come probabilità**, più il valore è alto più è probabile che verrà inserito nella soluzione finale. k è un parametro, dentro l'algoritmo, mentre α è esterno, scelto da noi.

Sebbene improbabile, **potrebbe non portare a una soluzione ammissibile**, che non copre tutto l'universo. Quando anche fornisce una soluzione ammissibile **non è detto che sia ottima**.

Teorema: L'algoritmo ha queste proprietà:

1. Produce una **soluzione ammissibile con probabilità** $\geq 1 - e^{-k}$.

2. Per ogni $\alpha > 0$, la **probabilità** che il **fattore di approssimazione** (quanto lontano dall'ottimo è la soluzione) sia $\geq \alpha(k + \ln n)$ è $\leq 1/\alpha$

$$P[\text{fatt. appr. } \geq \alpha(k + \ln n)] \leq \frac{1}{\alpha}$$

   Insomma, la probabilità che l'algoritmo vada male è bassa.

Proof. **Notazione**: ricordando come è fatto l'input (vedi sopra), risolvendo la versione $\hat{\pi}$ ottengo $\hat{x}_0, \ldots, \hat{x}_{m-1} \in [0, 1]$ e la soluzione del problema rilassato è

$$\hat{w} = w_0\hat{x}_0 + \ldots + w_{m-1}\hat{x}_{m-1}$$

E w^* è la soluzione ottima, ovviamente $\hat{w} \leq w^*$.

Punto 1: Probabilità che la soluzione sia ammissibile: chiamando

$$E_p = \text{il punto } p \in U \text{ non è coperto}$$

$$\begin{aligned} P[\text{sol. amm.}] &= 1 - P[\text{sol. non amm.}] \\ &= 1 - P[\text{almeno un punto di } U \text{ non coperto}] \\ &= 1 - P\left[\bigcup_{p \in U} E_p\right] \geq \end{aligned}$$

Usando l'union bound

$$\begin{aligned} &\geq 1 - \sum_{p \in U} P[E_p] \\ &= 1 - \sum_{p \in U} P[p \text{ non è coperto}] \\ &= 1 - \sum_{p \in U} \left(\prod_{i \in m, p \in S_i} P[i \text{ non è stato scelto}] \right) \end{aligned}$$

Quindi la probabilità che un punto non sia coperto è il prodotto delle probabilità di non aver scelto nessun insieme che lo contengo, e posso dire meglio quando questa cosa succede, per definizione dell'algoritmo:

$$
\begin{aligned}
&= 1 - \sum_{p \in U} \prod_{i \in m, p \in S_i} (1 - \hat{x}_i)^{\lceil k + \ln n \rceil} \\
&\geq 1 - \sum_{p \in U} \prod_{i \in m, p \in S_i} (1 - \hat{x}_i)^{k + \ln n} \geq
\end{aligned}
$$

Uso il fatto che $1 - x \leq e^{-x}$:

$$
\begin{aligned}
&\geq 1 - \sum_{p \in U} \prod_{i \in m, p \in S_i} e^{-\hat{x}_i (k + \ln n)} \\
&= 1 - \sum_{p \in U} e^{-(k + \ln n) \sum_{i \in m, p \in S_i} \hat{x}_i} \geq
\end{aligned}
$$

Ma uno dei vincoli delle LP per la seconda sommatoria è che questa sia ≥ 1, quindi

$$
\begin{aligned}
&\geq 1 - \sum_{p \in U} e^{-(k + \ln n)} \\
&= 1 - \sum_{p \in U} \frac{1}{n} e^{-k} \\
&= 1 - e^{-k} \sum_{p \in U} \frac{1}{n} \\
&= 1 - e^{-k}
\end{aligned}
$$

Ricordando che $|U| = n$.

Punto 2: La probabilità che S_i venga scelto è $\leq (k + \ln n)\hat{x}_i$ (per union bound). Il valore atteso di w è

$$\begin{aligned} E[w] &= \sum_{i \in m} w_i P[S_i \text{ sia scelto}] \\ &\leq \sum_{i \in m} w_i (k + \ln n)\hat{x}_i \\ &= (k + \ln n) \sum_{i \in m} w_i \hat{x}_i = \end{aligned}$$

Ma è la definizione di $\hat{w}$ quindi

$$\begin{aligned} &= (k + \ln n)\hat{w} \\ &\leq (k + \ln n)w^* \end{aligned}$$

Vogliamo usare la disuguaglianza di Markov, scegliendo come parametro $\beta. = \alpha(k + \ln n)w^*$. Mi importa che

$$P\left[x \geq \beta\right] \leq \frac{E[x]}{\beta}$$

Quindi

$$\begin{aligned} P\left[\frac{w}{w^*} \geq \alpha(k + \ln n)\right] &= P\left[w \geq \alpha(k + \ln n)w^*\right] \\ &= P[w \geq \beta] \leq \end{aligned}$$

Per la disuguaglianza di Markov

$$\begin{aligned} &\leq \frac{E[w]}{\beta} \\ &\leq \frac{(k + \ln n)w^*}{\alpha(k + \ln n)w^*} \\ &= \frac{1}{\alpha} \end{aligned}$$

□

Corollario: Per $k = 3$, c'è il 45% di **probabilità di ottenere una soluzione ammissibile con rapporto di approssimazione** $\leq 6 + 2 \ln n$.

Proof. La probabilità che una soluzione sia non ammissibile è piccola, il resto delle volte la soluzione è ammissibile ma il rapporto di approssimazione non ci piace abbastanza.

Quindi abbiamo uno spazio di soluzioni non ammissibili $E_{\text{non-amm}}$, delle soluzioni che non vogliamo E_{bad} e le soluzioni che effettivamente ci vanno bene E_{ok}.

La probabilità delle soluzioni non ammissibili è:

$$P[E_{\text{non-amm}}] \leq e^{-k} = e^{-3}$$

Dal rapporto di approssimazione possiamo capire che $\alpha = 2$, quindi la probabilità per le soluzioni che non vogliamo è

$$P[E_{\text{bad}}] = P[\text{fatt. appr. } > 6 + 2 \ln n] \leq \frac{1}{2}$$

Per le soluzioni buone invece:

$$\begin{aligned} P[E_{\text{ok}}] &= 1 - P[E_{\text{non-amm}} \cup E_{\text{bad}}] \\ &\geq 1 - (P[E_{\text{non-amm}}] + P[E_{\text{bad}}]) \\ &\geq 1 - \left(e^{-3} + \frac{1}{2} \right) \approx 45\% \end{aligned}$$

□

9.3 Problema MaxEkSat

Dove k è un parametro ≥ 3.

Definizione:

- **Input**: una formula CNF in cui ogni clausola contiene esattamente k letterali
- **Soluzioni Ammissibili**: assegnamenti di valori di verità (per le variabili che compaiono nell'input)
- **Funzione obiettivo**: numero di clausole soddisfatte
- **Tipo**: massimizzazione, max

Ovviamente è $\mathcal{NPO}$, in quanto facilmente riconducibile a un SAT generico.

9.3.1 Algoritmo Probabilistico

Passaggi:

- Assegna a ogni variabile un valore a caso $\in \{true, false\}$

Teorema: Questo algoritmo **rende vere** almeno

$$\frac{2^k - 1}{2^k} = 1 - \frac{1}{2^k}$$

delle clausole totali.

Alternativamente, questo algoritmo, data una formula con t clausole, ne **rende vere almeno**

$$\frac{2^k - 1}{2^k} t$$

Cioè

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

Un po' di **notazione:** sia φ l'**input**, chiamiamo

- n il numero di variabili che compaiono
- k il numero di letterali per clausola
- $x_1, \ldots, x_n$ le variabili che compaiono nelle clausole
- $K_1, \ldots, K_t$ le clausole

Variabili aleatorie:

- x_i: variabile uniforme in $(0, 1)$, variabile che uso per decidere se rendere vera o falsa la variabile i-esima. Assegniamo 0 o 1 a caso per tutte
- C_i: variabile aleatoria che ha valore:

$$C_i = \begin{cases} 1 & \text{se } K_i \text{ soddisfatta} \\ 0 & \text{altrimenti} \end{cases}$$

- T: numero di clausole soddisfatte

Proof. Qual è il **numero atteso di clausole soddisfatte** $E[T]$?

$$\begin{aligned} E[T] &= \sum_{b_1 \in \{0,1\}} \dots \sum_{b_n \in \{0,1\}} E[T | x_1 = b_1, \dots, x_n = b_n] P[x_1 = b_1, \dots, x_n = b_n] \\ &= \sum_{b_1 \in \{0,1\}} \dots \sum_{b_n \in \{0,1\}} E[C_1 + \dots + C_t | x_1 = b_1, \dots, x_n = b_n] P[x_1 = b_1] \dots P[x_n = b_n] \end{aligned}$$

Ma ognuna delle n probabilità è $1/2$ (dato che l'assegnamento è casuale), quindi

$$= \frac{1}{2^n} \sum_{b_1 \in \{0,1\}} \dots \sum_{b_n \in \{0,1\}} E[C_1 + \dots + C_t | x_1 = b_1, \dots, x_n = b_n]$$

Sostituisco alla probabilità il **valore atteso**:

$$= \frac{1}{2^n} \sum_{b_1 \in \{0,1\}} \dots \sum_{b_n \in \{0,1\}} \sum_{j=1}^{t} E[C_j | x_1 = b_1, \dots, x_n = b_n]$$

Dove, ogni clausola K_j può essere falsa solo nel caso in cui ognuna delle variabili x_i che la compone si presenta nella forma:

- $false$ se $x_i \in K_j$
- $true$ se $\neg x_i \in K_j$

Quindi c'è un singolo assegnamento che rende K_j falsa, mentre ci sono $2^n - 2^{n-k}$ modi per renderla vera.

Quindi la sommatoria sopra diventa

$$= \frac{1}{2^n} \sum_{j=1}^{t} (2^n - 2^{n-k}) = \frac{2^n - 2^{n-k}}{2^n} t = \frac{2^{k-n}2^n - 2^{k-n}2^{n-k}}{2^{k-n}2^n} t = \frac{2^k - 1}{2^k} t$$

□

Lemma: Per ogni $j = 0, \ldots, n$ esistono $b_1, \ldots, b_j \in \{0, 1\}$ tali che

$$E[T|x_1 = b_1, \ldots, x_j = b_j] \geq \frac{2^k - 1}{2^k} t$$

dove t è il numero di clausole.

Proof. Per induzione su j:

- **Base:** Per $j = 0$ è il teorema.
- **Passo induttivo:** per ipotesi di induzione

  $$\frac{2^k - 1}{2^k} t \leq E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}] =$$

  Per il teorema delle probabilità totali:

  $$\begin{aligned} &= E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 0]P[x_j = 0] \\ &\qquad + E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 1]P[x_j = 1] \\ &= \frac{1}{2}(E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 0] \\ &\qquad + E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 1]) \end{aligned}$$

  Ipotizzando che per assurdo i due valori siano:

  $$E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 0] < \frac{2^k - 1}{2^k} t$$

  $$E[T|x_1 = b_1, \ldots, x_{j-1} = b_{j-1}, x_j = 1] < \frac{2^k - 1}{2^k} t$$

  Ma questo vorrebbe dire che la loro somma è

  $$< \frac{2^k - 1}{2^k} 2t$$

  Ma questo va contro l'ipotesi induttiva, quindi almeno uno dei due deve essere $\geq$.

□

Corollario: Esiste un **assegnamento deterministico che soddisfa almeno**

$$\frac{2^k - 1}{2^k} t$$

clausole.
Input: Formula k-CNF fatta da t clausole $K_1 \wedge \ldots \wedge K_t$ e n variabili $x_1, \ldots, x_n$.
Per capire come assegnarle.

$D \leftarrow \emptyset$
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 $j \in D$ **or** x_i non compare in K_j **then**
      continue
    end if
    $h \leftarrow$ numero di variabili in K_j con indice $\geq i$
    if x_i compare positiva in K_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
      $\Delta_0 \leftarrow \Delta_0 + \frac{1}{2^h}$
      $\Delta_1 \leftarrow \Delta_1 - \frac{1}{2^h}$
      $\Delta_{D_0} \leftarrow \Delta_{D_0} \cup \{j\}$
    end if
  end for
  if $\Delta_0 \geq \Delta_1$ **then**
    $x[i] = false$
    $D \leftarrow D \cup \Delta_{D_0}$
  else
    $x[i] = true$
    $D \leftarrow D \cup \Delta_{D_1}$
  end if
end for
Output $x[i], \ldots x[n]$

Questo algoritmo tenta di "mimare" quello visto nel Lemma, conoscendo un assegnamento "buono" delle prime j variabili posso conoscere un assegnamento "buono" delle prime $j+1$.

Notazione:

- Δ_1/Δ_0: valore atteso se viene posta ad 1 e a 0 la variabile, rispettivamente.

- $\Delta_{D_1}/\Delta_{D_0}$: l'insieme di clausole chiuse da un determinato assegnamento ($true$ o $false$ rispettivamente).

Per ogni variabile (`for` esterno), per ogni clausola (`for` interno) determina cosa succederebbe al valore atteso se assegnassi un certo valore di verità alla variabile (`if/else`, rispettivamente *true/false*).

(Non ho capito, ma farò finta)

10 Teoria della Complessità di Approssimazione

Torniamo nel mondo magico dei **problemi di decisione**.

Ricordiamo che si possono sempre esprimere come: conoscendo un **linguaggio** $L \subseteq 2^*$ (insieme di stringhe binarie), dobbiamo **stabilire se l'input** $x \in 2^*$ (stringa binaria) **appartiene a** L (linguaggio), quindi stabilire se $x \in L$, tramite un **algoritmo di decisione**.

10.1 MdT con Oracolo o Verificatore

Con **MdT con Oracolo o verificatore** si intende una macchina di Turing nella quale l'input è sempre una stringa binaria $x \in 2^*$, ma è presente anche un "**oracolo**", il quale contiene un'**ulteriore stringa** $w \in 2^*$, a cui la MdT può accedere quando necessario. L'**output** sarà quindi una **decisione** (sì/no) e **dipende da input e stringa dell'oracolo**.

Per accedere all'oracolo la MdT scrive sul **nastro delle query**: scrivendo una posizione su questo nastro l'oracolo restituirà il bit presente in quella posizione sulla stringa dell'oracolo.

Possiamo pensare alle MdT con oracolo come se quest'ultimo fosse una "*dimostrazione*" per l'input, per questo vengono anche chiamate **verificatori**.

Teorema: un linguaggio $L \subseteq 2^*$ sta in $\mathcal{NP}$ se e solo se esiste una MdT V con oracolo tale che:

1. $V(x, w)$ lavora in **tempo polinomiale** in $|x|$
2. $\forall x \in 2^*$, $\exists w \in 2^*$ tale che $V(x, w) =$ sì, se e solo se $x \in L$. Se x deve avere risposta "sì", esiste una w che fa ottenere "sì"

Equivalente ad una NDTM, al posto di dipendere da input e tutte le uscite, dipende da input e stringa di oracolo, il non determinismo viene introdotto in maniera differente ma sono equivalenti.

Quindi $\mathcal{NP}$ possiamo definirla come la **classi di problemi risolvibili da verificatori in tempo polinomiale**.

10.1.1 Verificatori Probabilistici

Input e output come prima, ma il **verificatore probabilistico** ha **accesso** anche a una **sorgente di bit random** e risponde "sì" o "no" in base anche a questi. Si aggiunge una parte di randomness.

La macchina in sé è deterministica, ma il **risultato dipende da input, oracolo e bit casuali**.

Il verificatore probabilistico V per il linguaggio L:

1. Lavora in **tempo polinomiale** per $|x|$.

2. Se $x \in L$, allora $\exists w \in 2^*$ tale che $V(x, w)$ **accetta con probabilità 1**. Se una stringa è accettabile, deve esserci una dimostrazione/stringa w dell'oracolo che la accetta indipendentemente dalla sorgente casuale.

3. Se $x \notin L$, $\forall w \in 2^*$, $V(x, w)$ **rifiuta con probabilità** $\geq 1/2$. Se una stringa è da non accettare, nonostante i bit casuali ho una buona probabilità di non accettarla.

10.2 Probabilistically Checkable Proof PCP

Dai verificatori probabilistici viene il nome PCP: Probabilistically Checkable Proof.

Date due funzioni $r, q : \mathbb{N} \to \mathbb{N}$, **chiamo**

$$PCP[r, q]$$

La **classe dei linguaggi accettati da un verificatore probabilistico** che su input x faccia $\leq q(|x|)$ query all'oracolo e $\leq r(|x|)$ letture di bit random.

Esempi di classi:

- Se non posso fare query o prendere bit random:

  $$PCP[0, 0] = \mathcal{P}$$

  La decisione dipende solo dall'input, quindi questa equivale a $\mathcal{P}$.

- Se posso fare query all'oracolo ma non posso accedere a i bit random

  $$PCP[0, poly] = \mathcal{NP}$$

  Senza probabilità le "probabilità" diventano certezze, ovvero 0 o 1. Questa corrisponde a introdurre il non determinismo, ovvero coincide con $\mathcal{NP}$.

Teorema PCP [Arora, Safra, 1998]: La classe $\mathcal{NP}$ equivale a:

$$\mathcal{NP} = PCP[O(\log n), O(1)]$$

Estraggo un **numero logaritmico di bit** e faccio un **numero costante di query all'oracolo**, introdurre un po' di randomness fa fare meno query all'oracolo. Inoltre la randomness aumenta logaritmicamente, al posto di avere una quantità polinomiale di letture all'oracolo, quindi la randomness è esponenzialmente "più potente" del non determinismo.

Ad **esempio**:

$$SAT \in PCP[5\log n + 7\log\log n + 12, 157]$$

D'ora in poi facciamo riferimento a uno specifico verificatore V che usa r **bit random** e q **query**

$$V \in PCP[r, q]$$

Su **input** $x \in 2^*$:

- Fa esattamente $q(|x|)$ **query**
- Estrae esattamente $r(|x|)$ **bit random**

Si può assumere questo **senza perdita di generalità**, al massimo potrebbe usarne di meno, ma non è una restrizione (posso fare quelle che mancano "a caso" e non usare, basta sia un upper bound).

Ci interessa il caso in cui le **query siano una costante** $O(1)$.

Inoltre possiamo assumere che i **bit random li estragga tutti assieme** come prima cosa, per poi tenerli da parte e usarli quando servono.

Considerando un **verificatore**

$$V \in PCP[r(n), q]$$

Dove q è una costante mentre $r(n)$ è un verificatore $\in O(\log n)$.
Questo verificatore

- riceve input $w \in 2^*$
- Estrae $R \in 2^{r(|x|)}bit$
- Richiede all'oracolo la posizione $w_{i_1^{R,x}}$
- Può rispondere 0 o 1 e per ogni possibilità richiederà un numero q di posizioni su w e le **richieste effettuate** potrebbero essere diverse, **in base alle risposte precedenti potrebbero cambiare le richieste effettuate**

Si chiama **verificatore adattivo**, le **query** fatte all'oracolo **dipendono anche dalle risposte delle query precedenti**. Sarebbe carino avere un verificatore non adattivo.

Quindi al posto di avere un albero di esecuzioni, guardiamo quali sono **tutte le possibili richieste**, le quali saranno una quantità finita

$$\overline{q} = 2^{q-1}, 2^{q-2}, \ldots, 1$$

Dopo aver estratto tutti i bit random che ci servono, **estraiamo** anche **tutte le possibili $\overline{q}$ query all'oracolo**. Dopo di questo l'esecuzione diventa quella di una MdT deterministica.

Questo ci permette di **trasformare qualunque verificatore adattivo** in un verificatore **non adattivo**.

Daremo per scontato che il verificatore:

1. **Legga** una **stringa** $R \in 2^{r(|x|)}$ di bit random
2. **Effettui** un numero q di **query** $\{i_1^{R,x}, \ldots, i_q^{R,x}\}$
3. Da qui il **comportamento** è **puramente deterministico**, dipendente da input x, stringa random R e risposte ottenute dall'oracolo

Se la stringa è da accettare, qualunque sia la stringa di bit random scelti, c'è una stringa dell'oracolo che mi fa accettare l'input. Se non è da accettare invece, per almeno metà delle stringhe di bit random viene rifiutato (probabilità $\geq 1/2$).

Esempio: Prendendo

$$L \in PCP[r(n), q] = \mathcal{NP}$$

Come **funziona il verificatore**?

- Prende in ingresso l'input
$$z \in 2^*$$
- Estrae una stringa random
$$R \in 2^{r(|z|)}$$
- Scelgo le posizioni, che dipenderanno dall'input e da R, le quali saranno esattamente q
$$i_1^{z,R}, \ldots, i_q^{z,R} \in \mathbb{N}$$
che portano ad altrettante richieste all'oracolo e altrettanti bit:
$$b_1, \ldots, b_q$$
- Da qui in poi ci sarà da fare computazione deterministica, che dipende da:
$$z, R, b_1, \ldots, b_q$$

10.2.1 Inapprossimabilità di MaxEkSat

Ogni k-CNF (con $k \geq 3$) con t clausole si può **trasformare** in una 3-CNF con $(k-2)t$ clausole **preservando la soddisfacibilità**.

Esempio:

$$x_7 \vee \neg x_4 \vee x_9 \vee \neg x_{15}$$
$$\implies (x_7 \vee \neg x_4 \vee y_1) \wedge (x_9 \vee \neg x_{15} \vee \neg y_1)$$

Se quella sopra è soddisfacibile, almeno uno dei letterali sopra è vero, l'assegnamento si può estendere a uno che rende vera anche quella sotto, mentre se tutti quelli sopra sono falsi, non c'è modo di avere la variabile ausiliaria $y_1, \neg y_1$ vera in entrambi i casi.

Teorema: Per ogni $k \geq 3$, esiste un $\epsilon > 0$ tale che MaxEkSat non è $(1+\epsilon)$ approssimabile (supponendo che $\mathcal{P} \neq \mathcal{NP}$).

Proof. (Per $k = 3$). Scegliamo un $L \in \mathcal{NP}c$, quindi

$$L \in \mathcal{NP} = PCP[r(n), q]$$

per qualche $r(n) \in O(\log n)$, $q \in \mathbb{N}$.

Sia V il verificatore

$$z \in 2^*, \;\; R \in 2^{r(|z|)}, \;\; i_1^{z,R}, \ldots, i_q^{z,R} \to b_1, \ldots, b_q$$

Si può scrivere una q-CNF soddisfacibile se e solo se V accetta

$$\psi_{z,R}$$

che dipende dalle variabili logiche $w_0, w_1, \ldots$ dove $w_i = true$ se e solo se l'i-esimo carattere dell'oracolo è un 1 (corrispondono ai caratteri sulla stringa dell'oracolo).

Quindi $\psi_{z,R}$ è una q-CNF con $\leq 2^q$ clausole.
Può diventare $\varphi_{z,R}$, ovvero una 3-CNF con $\leq q2^q$ clausole.

Posso creare, in tempo polinomiale:

$$\Phi_z = \bigwedge_{R \in 2^{r(|z|)}} \varphi_{z,R}$$

3-CNF con $\leq q2^q 2^{r(|z|)}$ clausole (un numero grosso, ma tutto polinomiale).

Quindi dando in pasto questa formula Φ_z a un algoritmo (ipotetico, cerchiamo un assurdo) di approssimazione per MaxE3Sat con tasso di approssimazione $\alpha < 1 + \epsilon$, definendo

$$\epsilon = \frac{1}{2q2^q}$$

Se $z \in L$, allora $\exists w$ che fa accettare V con probabilità 1.
Significa che Φ_z è soddisfacibile, quindi $q2^q 2^{r(|z|)}$ clausole soddisfacibili.

Se $z \notin L$, $\forall w$ allora V rifiuta con probabilità $\geq 1/2$.
Significa che il massimo numero di clausole soddisfacibili in Φ_z è

$$\frac{q2^q 2^{r(|z|)}}{2} + \frac{2^{r(|z|)}}{2}(q2^q - 1) = q2^q 2^{r(|z|)} - \frac{2^{r(|z|)}}{2}$$

Fornendo Φ_z a un algoritmo approssimato che permette di approssimare MaxE3Sat a meno di $1 + \epsilon$ il numero di clausole che verranno soddisfatte sarà:

- se $z \in L$
$$\geq \frac{q2^q 2^{r(|z|)}}{1 + \epsilon}$$
- se $z \notin L$
$$\leq q2^q 2^{r(|z|)} - \frac{2^{r(|z|)}}{2}$$

Chiamando:

$$A = q2^q, \quad B = 2^{r(|z|)}, \quad \epsilon = \frac{1}{2A}$$

Dando Φ_z in pasto all'algoritmo MaxE3Sat approssimato, quindi fornirà un numero di clausole soddisfacibili:

- se $z \in L$

$$\geq \frac{AB}{1+\epsilon}$$

- se $z \notin L$

$$\leq AB - \frac{B}{2}$$

Si possono distinguere questi due casi? Ipotizzando che per assurdo:

$$\begin{aligned}
AB - \frac{B}{2} > \frac{AB}{1+\epsilon} \implies & \\
\implies & AB + AB\epsilon - \frac{B}{2} - \frac{B}{2}\epsilon - AB > 0 \\
\implies & \epsilon\left(AB - \frac{B}{2}\right) - \frac{B}{2} > 0 \\
\implies & \frac{1}{2A}\left(AB - \frac{B}{2}\right) - \frac{B}{2} > 0 \\
\implies & \frac{B}{2} - \frac{B}{4A} - \frac{B}{2} > 0 \\
\implies & -\frac{B}{4A} > 0
\end{aligned}$$

Ed è impossibile, quindi i due intervalli non si sovrappongono.
In tempo polinomiale ho determinato l'appartenenza ad L, quindi il linguaggio originale, il che è impossibile (a meno che $\mathcal{P} = \mathcal{NP}$).

□

Questo è il modo in cui si usa PCP, tipicamente la costruzione è, partendo da un problema $\mathcal{NP}c$, far vedere che si riesce a trasformare il comportamento del verificatore in una formula (fatto in tempo polinomiale), che data in pasto a un algoritmo che non esiste permette di confermare l'appartenenza al linguaggio.
Al variare del problema varia q, quindi varia anche ϵ, dando una valutazione precisa di q si potrebbe dare un numero ad ϵ.

10.3 Inapprossimabilità di MaxIndependentSet

Il problema MaxIndependentSet consiste nel trovare l'insieme massimo di vertici indipendenti, ovvero senza lati tra di loro.

Definizione:

- **Input:** grafo non orientato $G = (V, E)$
- **Soluzioni ammissibili:** insiemi indipendenti (insiemi di vertici tali che non ci sono lati tra loro, contrario di clique)

$$X \subseteq V \text{ t.c. } \binom{X}{2} \cap E = \emptyset$$

- **Funzione obiettivo:** cardinalità dell'insieme di vertici, $|X|$
- **Tipo:** massimizzazione max

Teorema: Per ogni $\epsilon > 0$, MaxIndependentSet non è $(2-\epsilon)$ **approssimabile**.

Proof. Consideriamo un linguaggio $L \in \mathcal{NP}c$, quindi L è verificabile da

$$L \in PCP[r(n), q]$$

con $r(n) \in O(\log n)$, $q \in \mathbb{N}$.

Fissiamo

$$z \in 2^*$$

Generiamo una sequenza di bit random e q query all'oracolo

$$R \in 2^{r(|z|)} \qquad i_1^{z,R}, \ldots, i_q^{z,R} \to b_1, \ldots, b_q$$

Tutto questo genera una z-configurazione, ovvero una coppia:

$$(R, \{i_1^{z,R} : b_1, \ldots, i_q^{z,R} : b_q\})$$

Per uno z fissato, è composta da una serie di $2^{r(|z|)}$ bit random e le q coppie bit interrogato-risposta. Ci sono tante z-configurazioni, uno per ogni possibile R e per ogni possibile interrogazione.

Costruisco un grafo non orientato G_z dove:

- i vertici sono le z-configurazioni accettanti.
- c'è un lato tra due configurazioni della forma

  $$(R, \{i_1^{z,R} : b_1, \ldots, i_q^{z,R} : b_q\}), \quad (R', \{i_1^{z,R'} : b'_1, \ldots, i_q^{z,R'} : b'_q\})$$

  se e solo se $R = R' \vee \exists k, k' \in \{1, \ldots, q\}$ tali che $i_k^{z,R} = i_{k'}^{z,R'}$ ma $b_k \neq b_{k'}$. Ovvero, se le due stringhe random sono uguali oppure se ci sono due interrogazioni alla stessa posizione dell'oracolo che forniscono risposte diverse.

La relazione descritta dai lati si chiama "relazione di incompatibilità", due soluzioni collegate sono incompatibili.

Questo è un grafone, i bit random sono in quantità $2^{r(|z|)}$ e le interrogazioni sono 2^q, quindi $2^{r(|z|)} \cdot 2^q$ in totale, grosso ma comunque polinomiale.

Fact 1: Se $z \in L$, allora G_z ha un insieme indipendente di dimensione $\geq 2^{r(|z|)}$.

Proof. Se $z \in L$, allora $\exists \overline{w}$ che fa accettare con probabilità 1. Prendiamo tutte le configurazioni accettanti

$$(R, \{i_1^{z,R} : b_1, \ldots, i_q^{z,R} : b_q\})$$

compatibili con $\overline{w}$.

Proprietà:

1. questo insieme di vertici ha dimensione $\geq 2^{r(|z|)}$, in quanto qualunque sia R devo accettare
2. sono tutti non collegati da archi, ovvero indipendenti, in quanto se fossero collegate sarebbero incompatibili

□

Fact 2: Se $z \notin L$, allora ogni insieme indipendente di G_z ha cardinalità $\leq 2^{r(|z|)-1}$.

Proof. Supponendo che S sia un insieme indipendente di G_z, con cardinalità $> 2^{r(|z|)-1}$. Questo S è un insieme di configurazioni al cui interno non ci possono essere configurazioni incompatibili tra loro, quindi per ogni posizione richiesta da una qualsiasi configurazione con una qualsiasi R risulta sempre lo stesso valore.

Da S si può costruire un $\overline{w}$ compatibile con tutti i vertici di S.
$\implies$ $\overline{w}$ viene accettato in tutte le configurazioni, quindi accetta con probabilità $> 1/2$, ma per definizione è impossibile.

□

Ora diamo G_z in pasto a un algoritmo approssimato per MaxIndependentSet con grado di approssimazione $< 2 - \epsilon$.
I casi sono due

- $z \in L$: l'ottimo è $\geq 2^{r(|z|)}$
- $z \notin L$: l'ottimo $< 2^{r(|z|)}$

Il nostro algoritmo è approssimato quindi ottiene

$$\frac{2^{r(|z|)}}{2-\epsilon} > 2^{r(|z|)-1} = \frac{2^{r(|z|)}}{2}$$

Comunque sempre maggiore di $2^{r(|z|)-1}$, ed è impossibile ottenerlo in tempo polinomiale (a meno che $\mathcal{P} = \mathcal{NP}$), quindi assurdo.

□

11 Strutture succinte, quasi-succinte e compatte

Abbiamo lavorato con dei razzi, ora si parla di oggetti estremamente piccoli, strutture dati molto semplici.

Intanto, cos'è una struttura dati? Intendiamo un **Abstract Data Type ADT**, ovvero un **insieme di primitive**, ad esempio su uno stack le primitive potrebbero essere `push`, `pop` e `is_empty`. Esistono modi formali, assiomatici, per definire le ADT.

Ogni ADT si può implementare e possono esserci molte **implementazioni** diverse, che funzionalmente corrispondono alla stessa ADT, ma differiscono per **complessità in spazio e tempo**.

11.1 Information-Theoretical Lower Bound

Supponendo di avere una struttura dati specifica che per **taglia** n abbia V_n **valori possibili**, che chiameremo:

$$v_1, v_2, \ldots, v_{V_n}$$

E ognuno di questi possibili **valori** utilizza un **certo numero di bit**

$$x_1, x_2, \ldots, x_{V_n}$$

Allora il **teorema di Shannon** dice che

$$\frac{x_1 + x_2 + \ldots + x_{V_n}}{V_n} \geq \log_2 V_n$$

Sostanzialmente, la **media delle dimensioni di tutti i valori** non può essere meglio del $\log_2$ del numero di valori possibili, quindi una "compressione", una rappresentazione dei dati non può essere meglio della media, ci saranno altre rappresentazioni per cui la rappresentazione occuperà più spazio.

Questo $\log_2 V_n$ si chiama **information theoretical lower bound** e fornisce la **dimensione media in bit per una struttura dati**.

Definiamo quindi

$$Z_n := \log_2 V_n$$

come il **numero di bit minimi per rappresentare una ADT**.

Per una struttura dati che implementa l'ADT e occupa D_n bit, allora

$$D_n \geq Z_n$$

Ci possono essere più casi: le strutture possono essere

- **Implicite:** $D_n = Z_n + O(1)$, una costante in più del lower bound, e.g., $Z_n + 3$ bit.
- **Succinte:** $D_n = Z_n + o(Z_n)$, un o in più del lower bound, e.g., $Z_n + \log Z_n$ bit.
- **Compatte:** $D_n = O(Z_n)$, un O del lower bound, e.g., $Z_n + \sqrt{Z_n}$ bit.
- Tutte le altre, peggiori.

11.2 Struttura Succinta per Rank e Select

Rank e Select ADT. Supponendo di avere un array di n bit, $\underline{b} \in 2^n$.

Il **rank** è quanti 1 ci sono prima di una determinata posizione:

$$rank_{\underline{b}}(p) = |\{i | i < p \text{ e } b_i = 1\}|$$

Select è il duale, quindi dove si trova l'i-esimo 1

$$select_{\underline{b}}(k) = \max\{p | rank_{\underline{b}}(p) \leq k\}$$

Quindi, **rank** quanti 1 fino a una posizione, mentre **select** dice dove solo gli 1.

Proprietà:

- $rank(select(i)) = i$
- $select(rank(p)) \geq p$, viene $=$ se e solo se in posizione p c'è un 1

Calcoliamo l'**information theoretical lower bound**. I valori possono essere

$$V_n = 2^n$$

in quanto questi sono i possibili valori di un vettore di n bit. Di conseguenza

$$Z_n = \log_2 V_n = n$$

Per rappresentare un vettore di n bit servono n bit (e grazie al).

Per **fare rank e select** sul vettore possiamo:

- a ogni richiesta calcolare la risposta, quindi occupiamo spazio n (solo il vettore) ma le richieste richiedono tempo $O(n)$
- costruisco una tabella per tutti i possibili valori di rank e select, utilizzando spazio $2n \log n$ le richieste richiedono tempo $O(1)$ (solo il lookup)

11.2.1 Struttura succinta di Jacobson per Rank

Si tratta di una **struttura multilivello**.

Il **vettore di bit** viene diviso in "**superblocchi**", i quali sono blocchi di lunghezza $(\log n)^2$ bit (si presuppone che n sia potenza di due per evitare ceil vari).

Ogni superblocco viene **diviso in blocchi** di lunghezza $\frac{1}{2}\log n$.

Cosa memorizza Jacobson:

1. Per ogni **superblocco** memorizza quanti 1 ci sono **prima del superblocco**.

2. Per ogni **blocco** B_{ij} memorizza quanti 1 ci sono **dall'inizio del superblocco** S_i fino al blocco B_{ij} escluso.

3. **Four russians trick**: una tabella rank per tutti i possibili valori dei blocchi

Spazio: Quanto **spazio** occupiamo per i **superblocchi**? Serve una **tabella per ogni superblocco** che indica gli 1:

- ci sono $\frac{n}{(\log n)^2}$ **righe**
- per ogni riga ci possono essere n valori, quindi servono $\log n$ bit per rappresentarli

In **totale**

$$\frac{n}{(\log n)^2}\log n = \frac{n}{\log n} = o(n)$$

Per i superblocchi uso $o(n)$ bit aggiuntivi.

Quanto **spazio per i blocchi**? Mi serve una tabella che indica gli 1 dall'inizio del superblocco:

- ci sono $\frac{n}{1/2 \log n}$ blocchi, di conseguenza altrettante **righe**
- per ogni riga devo scrivere quanti 1 dall'inizio del blocco, ovvero un numero che occuperà al massimo $\log((\log n)^2)$ bit

In **totale**

$$\frac{n}{\frac{1}{2}\log n} 2 \log\log n = \frac{4n \log\log n}{\log n} = o(n)$$

Quindi per blocchi e superblocchi uso $o(n)$ bit aggiuntivi.

Four russians trick: I blocchi sono lunghi $1/2 \log n$, quindi son corti, per rispondere a query su dati molto piccoli memorizzo tutte le possibili tabelle di rank e vado a cercare quella che mi serve (quella che corrisponde al blocco in questione).

Quanto **spazio** occupa? Ci sono

- $2^{1/2 \log n}$ possibili **blocchi**
- per ognuno dei quali ci sono $1/2 \log n$ **righe**
- per ogni riga servono $\log(1/2 \log n)$ bit

In **totale**

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

Quindi **tutti i dati aggiuntivi** in totale occupano **spazio** $o(n)$.
Non possiamo tornare al vettore originale da questa struttura dati in quanto ci serve per fare il four russians trick.

Quindi abbiamo una **struttura succinta** che risponde in **tempo costante**, per avere un determinato rank prendiamo il rank prima del superblocco, lo sommiamo al rank prima del blocco e poi interroghiamo la tabella corrispondente al blocco all'interno del quale ci troviamo.

11.2.2 Struttura di Clarke per la Select

Ricordiamo che la `select` serve a memorizzare le posizioni degli 1. L'idea è quella di memorizzare gli 1 non sempre ma solo *ogni tanto*. Anche questa è una struttura **multilivello**

Primo livello: memorizza solo i **valori di select** per valori **multipli di** $\log n \log\log n$.

Quanto **spazio** occupa?

- Ci sono $\frac{n}{\log n \log\log n}$ posizioni da memorizzare
- Ogni numero da memorizzare chiede $\log n$ bit

In **totale**:

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

Chiamiamo p_i è la **posizione** del $i \cdot \log n \log\log n$-esimo 1.
Quindi sappiamo che:

$$p_{i+1} - p_i \geq \log n \log\log n$$

In quanto tra una posizione e l'altra devono per forza esserci almeno $\log n \log\log n$ valori, e questo succede solo nel caso siano tutti 1 di fila.

Secondo livello: Dipende da $r_i = p_{i+1} - p_i$, che sappiamo essere $r_i \geq \log n \log\log n$.
Abbiamo due casi:

1. **Caso sparso:** $r_i \geq (\log n \log\log n)^2$, gli 1 sono lontani tra loro, ci sono molti 0. In questo caso la **tabella della select viene memorizzata esplicitamente** (come differenza da p_i ovviamente).

   Quanto **spazio** occupa?
   - La tabella deve memorizzare $\log n \log\log n$ 1, quindi altrettante righe
   - vengono memorizzate come differenza, quindi servono $\log r_i$ bit per ogni riga

   In **totale**:

   $$\begin{aligned}(\log n \log\log n) \log r_i &= \frac{(\log n \log\log n)^2}{\log n \log\log n} \log r_i \\ &\leq \frac{r_i \log r_i}{\log n \log\log n} \\ &\leq \frac{r_i}{\log\log n}\end{aligned}$$

   Il primo $\leq$ viene dall'ipotesi del caso sparso, mentre il secondo viene semplificando $\log r_i$ con $\log n$, ricordando che $r_i \leq n$.

2. **Caso denso:** dove r_i è compreso tra

   $$\log n \log\log n \leq r_i < (\log n \log\log n)^2$$

   Memorizzo le posizioni multiple di $\log r_i \log\log n$.

   Quanto **spazio** occupo?
   - Memorizzo un numero di righe pari a "un 1 ogni tanto", definito come $(\log n \log\log n)/(\log r_i \log\log n)$
   - Ognuno di questi richiede $\log r_i$ bit

   In **totale**:

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

   l'ultimo passaggio viene dalla nostra ipotesi per r_i (bound sinistro).

Complessivamente per il **secondo livello**

$$\leq \frac{r_0}{\log\log n} + \frac{r_1}{\log\log n} + \cdots =$$
$$= \frac{p_1 - p_0}{\log\log n} + \frac{p_2 - p_1}{\log\log n} + \cdots =$$

Che è una somma telescopica, quindi rimane

$$= \frac{p_{\frac{n}{\log n \log\log n}} - p_0}{\log\log n} \leq \frac{n}{\log\log n} = o(n)$$

Quindi in **totale** un $o(n)$, ma **manca un pezzo** per il caso denso del secondo livello, in cui non abbiamo memorizzato tutti gli 1.

Terzo livello: Chiamando $s_i^0, \ldots s_i^q$ le **posizioni memorizzate al secondo livello** nel **caso denso**, tra ognuna di queste posizioni saranno presenti $\log r_i \log\log n$ bit pari a 1, quindi

$$t_i^j = s_i^{j+1} - s_i^j \implies t_i^j \geq \log r_i \log\log n$$

Ci sono anche qui **due casi**:

1. **Caso sparso:**
$$t_i^j \geq \log t_i^j \log r_i (\log\log n)^2$$
In questo caso **memorizza esplicitamente la tabella**.

   Quanto **spazio** occupo?
   - servono $\log r_i \log\log n$ posizioni e altrettante righe della tabella
   - ogni riga contiene un numero che al massimo è t_i^j, quindi usa $\log t_i^j$ bit

   In **totale**
$$(\log r_i \log\log n) \log t_i^j = \frac{\log t_i^j \log r_i (\log\log n)^2}{\log\log n}$$
$$= \frac{t_i^j}{\log\log n}$$

Quindi, **complessivamente**, per tutti i vari t_i^j arriva a occupare:

$$\sum_j \frac{t_i^j}{\log\log n} = \sum_j \frac{s_i^{j+1} - s_i^j}{\log\log n} = \frac{s_i^k - s_i^0}{\log\log n}$$

Dove s_i^k è la posizione dell'ultimo 1 all'interno del blocco considerato. Possiamo **maggiorare** questa somma con

$$\frac{p_{i+1} - p_i}{\log\log n}$$

in quanto s_i^k è la posizione prima di p_{i+1} e s_i^0 è la posizione subito dopo s_i. Per lo spazio complessivo tra tutti i livelli:

$$\sum_i \frac{p_{i+1} - p_i}{\log\log n} = \frac{p_{\frac{n}{\log n \log\log n}} - p_0}{\log\log n} \leq \frac{n}{\log\log n} = o(n)$$

In totale:

$$\sum_i \sum_j \frac{t_i^j}{\log\log n} \leq \sum_i \frac{r_i}{\log\log n} \leq \frac{n}{\log\log n} = o(n)$$

Similmente al livello precedente, anche questo caso **occupa** $o(n)$.

2. **Caso denso:**

$$t_i^j < \log t_i^j \log r_i (\log\log n)^2$$

Si usa il four russians trick, **memorizziamo esplicitamente tutte le possibili tabelle di select**.

Considerando che

$$\begin{aligned}\log t_i^j &\leq \log r_i \\ &\leq \log\left((\log n \log\log n)^2\right) \\ &= 2\log(\log n \log\log n) \\ &= 2\log\log n + 2\log\log\log n \\ &\leq 4\log\log n\end{aligned}$$

Di conseguenza

$$t_i^j \leq \log t_i^j \log r_i (\log\log n)^2$$

Ma considerando che

$$\begin{cases}\log t_i^j \leq 4\log\log n \\ \log r_i \leq 4\log\log n\end{cases}$$

complessivamente

$$t_i^j \leq 16(\log\log n)^4$$

Quanto **spazio** occupa?

- Ci sono $2^{t_i^j}$ tabelle
- ognuna da t_i^j righe
- ogni riga occupa $\log t_i^j$ bit

Quindi in **totale**:

$$\begin{aligned}2^{t_i^j} \cdot t_i^j \cdot \log t_i^j &\leq 2^{16(\log\log n)^4} 16(\log\log n)^4 \log\left(16(\log\log n)^4\right) \\ &= (16(\log\log n)^4)^2 \\ &= o(n)\end{aligned}$$

Quindi, nuovamente. il four russians trick **occupa** $o(n)$.

Per entrambe le strutture, si possono implementare solo i primi n livelli e lasciar stare gli ultimi (tendenzialmente non implementare il four russians trick), l'operazione non è più tempo costante ma diventa logaritmica, si risparmia lo spazio dei livelli non implementati.

11.3 Alberi Binari

Per la teoria dei grafi: un albero è un **grafo non orientato**, **aciclico** e **connesso**. Facendo cadere l'ipotesi di connessione è una foresta.
Di solito in informatica si sceglie un nodo come radice e si usano **alberi radicati**.

Inoltre l'albero può essere **ordinato** o **non ordinato**. Nel caso ordinato ogni nodo ha una lista di figli, quindi c'è un ordine relativo dei figli. Di solito si intende ordinato, in quanto più facile.

Un **albero binario** ha la restrizione di **avere 2 o 0 figli** (non può esserci 1 figlio solo).

I nodi **con figli** si chiamano "**nodi interni**", mentre i nodi **senza** si chiamano "**nodi esterni**" o "**foglie**".

Quindi da qui in poi con "**albero binario**" si intende un **albero radicato ordinato in cui tutti i nodi hanno 2 o 0 figli**.

Teorema: Il numero di nodi esterni in un albero binario è pari al numero di nodi interni + 1

$$\#\text{ nodi esterni} = \#\text{ nodi interni } + 1$$

Proof. Fatta per induzione strutturale:

- **Base**: considerando l'albero più semplice possibile, ovvero solo la radice $\square$: il numero di nodi esterni è 1 (la radice) ed il numero di nodi interno è 0, quindi dato che $1 = 0 + 1$

  $$E(\square) = I(\square) + 1 = 0 + 1 = 1$$

  la proprietà vale per il caso base.

- **Caso ricorsivo**: considerando i due possibili sotto-alberi di un nodo T_1, T_2, per ipotesi induttiva $E(T_1) = I(T_1) + 1$ e $E(T_2) = I(T_2) + 1$, quindi per l'intero albero T:

  $$\begin{aligned} E(T) &= E(T_1) + E(T_2) \\ &= I(T_1) + 1 + I(T_2) + 1 \\ &= I(T_1) + I(T_2) + 1 + 1 \\ &= I(T) + 1 \end{aligned}$$

$\square$

Teorema: Ci sono

$$C_n = \frac{1}{n+1} \left(\begin{array}{c} 2n \\ n \end{array} \right)$$

alberi binari con n nodi interni (numero di Catalano).

Proof. Trust me bro.

$\square$

Approssimazione di Stirling:

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

Quanto vale C_n asintoticamente?

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

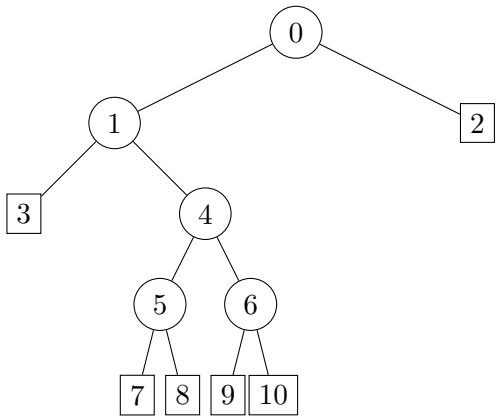
Information Theoretical Lower Bound: Questo vuol dire che

$$\begin{aligned} Z_n = \log C_n &\approx \log 2^n - \log\sqrt{\pi n^3} \\ &= 2n - o(\log n) \end{aligned}$$

Quindi il l'**information theoretical lower bound** per memorizzare un albero è $2n - o(\log n)$.

11.3.1 Rappresentazione Succinta

Volendo rappresentare un albero con una certa struttura, **numeriamo i nodi** dall'alto verso il basso e da sinistra verso destra. **Esempio**:



Quindi $n = 5$ nodi interni e 6 nodi esterni.

Usiamo un **array di** $2n + 1$ **bit** (11 in questo caso) e se il **nodo** è **interno** inserisco **1**, **altrimenti 0**. Per l'esempio precedente:

0	1	2	3	4	5	6	7	8	9	10
1	1	0	0	1	1	1	0	0	0	0

Trovare i figli: Voglio sapere dove sono i **figli di ogni nodo**. Per sapere tale indice per un nodo p mi basta sapere quanti sono i nodi a destra del nodo considerato e quanti sono i nodi a sinistra del figlio di sinistra del nodo considerato, li sommo a p e trovo l'indice del figlio.

I nodi totali dell'albero sono $2n + 1$, i nodi interni n. I nodi interni prima di p sono $rank_b(p)$, quindi l'indice del figlio sinistro è

$$f_{sx}(p) = 2rank_b(p) + 1$$

$$f_{dx}(p) = 2rank_b(p) + 2$$

Trovare il genitore: Ma chi è p' tale che p' è **genitore di** p? Non so se p è il figlio di sinistra o destra, quindi varrà una di queste due proprietà

$$\begin{array}{ll} 2rank(p')+1 & = p \\ 2rank(p')+2 & = p \end{array} \implies rank(p') = \left\lfloor \frac{p}{2} - \frac{1}{2} \right\rfloor$$

Ma select è circa l'inverso di rank, quindi:

$$p' = select \left\lfloor \frac{p}{2} - \frac{1}{2} \right\rfloor$$

Di conseguenza definiamo:

$$parent(p) := select_b \left(\left\lfloor \frac{p}{2} - \frac{1}{2} \right\rfloor \right)$$

In questo modo tramite rank e select possiamo trovare sia padri che figli di un certo nodo all'interno dell'albero.

Dati satellite: Ma se devo memorizzare **dati ausiliari** per ogni nodo oltre che la struttura?

- Se i dati sono su **tutti i nodi** allora mi serve un vettore ausiliario di $2n+1$ **celle**.
- Se i dati sono **solo su i nodi interni** posso usare un vettore ausiliario con dimensione pari ai nodi interni e faccio una **rank per sapere la cella a cui accedere**.

Occupazione di memoria: Dimensione dell'array:

$$D_n = 2n + 1 + o(n)$$

L'**information theoretical lower bound** è

$$Z_n = 2n - o(\log n)$$

Quindi la struttura è **succinta** (considerando le precedenti strutture succinte per rank e select).

11.4 Codifica di Elias-Fano per sequenze monotone

Elias-Fano è un modo **succinto** (compatto nel caso più generale, ma succinto in realtà) per **rappresentare delle sequenze crescenti**.

Partendo da una **sequenza di valori non decrescenti**

$$x_0 \leq x_1 \leq x_2 \leq \ \ldots \ \leq x_{n-1} < u$$

Si calcola un numero ℓ tale che

$$\ell = \max\left\{0, \left\lfloor \log \frac{u}{n} \right\rfloor\right\}$$

Dividiamo in due parti la rappresentazione:

1. Gli ℓ **bit inferiori** (meno significativi)

$$\begin{array}{rcl} \ell_0 & = & x_0 \mod 2^\ell \\ \ell_1 & = & x_1 \mod 2^\ell \\ \vdots & & \\ \ell_{n-1} & = & x_{n-1} \mod 2^\ell \end{array}$$

   Vengono **memorizzati esplicitamente**.

2. I **bit superiori** (più significativi)

$$\left\lfloor \frac{x_0}{2^\ell} \right\rfloor, \left\lfloor \frac{x_1}{2^\ell} \right\rfloor, \ldots, \left\lfloor \frac{x_{n-1}}{2^\ell} \right\rfloor$$

   Vengono memorizzati come **differenza tra l'uno e il successivo** (Δ-compressione)

$$u_0, u_1, u_2, \ldots$$

   Dove:

$$u_i = \left\lfloor \frac{x_i}{2^\ell} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^\ell} \right\rfloor$$

   Con il presupposto che $x_{-1} = 0$.
   Questi valori vengono **memorizzati in unario** in un unico vettore dove scrivo **tanti 0 quanti il valore**, **terminati da un 1**, esempio:

$$3\ 7 \rightarrow 0001\ 00000001$$

Occupazione di spazio:

- La **prima parte** richiede $\ell \cdot n$ **bit**, in quanto memorizzati esplicitamente.

- Per la **seconda parte** servono

$$\begin{aligned}\sum_{i=0}^{n-1} u_i + 1 &= \sum_{i=0}^{n-1} \left(\left\lfloor \frac{x_i}{2^\ell} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^\ell} \right\rfloor + 1 \right) \\ &= n + \sum_{i=0}^{n-1} \left(\left\lfloor \frac{x_i}{2^\ell} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^\ell} \right\rfloor \right) =\end{aligned}$$

  E questa è una somma telescopica, quindi rimane solo

$$\begin{aligned}= n + \frac{x_{n-1}}{2^\ell} - \frac{x_{-1}}{2^\ell} &= n + \frac{x_{n-1}}{2^\ell} \\ &\leq n + \frac{u}{2^\ell} \\ &= n + \frac{u}{2^{\lfloor \log \frac{u}{n} \rfloor}} \\ &\leq n + \frac{u}{2^{\log \frac{u}{n} - 1}} \\ &= n + \frac{2u}{u/n} \\ &= 3n\end{aligned}$$

  Quindi per la seconda parte **servono** $3n$ **bit**.

In **totale** servono:

$$\begin{aligned}D_n &= \ell \cdot n + 3n \\ &= (\ell + 3)n \\ &= \left(\left\lfloor \log \frac{u}{n} \right\rfloor + 3 \right) n \\ &= 2n + n \left\lceil \log \frac{u}{n} \right\rceil + o(n)\end{aligned}$$

Quindi un $2n + n \left\lceil \log \frac{u}{n} \right\rceil + o(n)$ bit.

Trovare un valore: Quindi per trovare il valore x_i cosa faccio?

- Per la **parte bassa** dei bit è facile in quanto è memorizzata esplicitamente.
- Per la **parte alta** faccio una select sul vettore che contiene tutte le parti alte in unario per trovare l'i-esimo 1:

$$\begin{aligned} select_{\underline{u}}(i) &= i + u_0 + u_1 + \ldots + u_i \\ &= i + \left\lfloor \frac{x_0}{2^\ell} \right\rfloor + \left\lfloor \frac{x_1}{2^\ell} \right\rfloor - \left\lfloor \frac{x_0}{2^\ell} \right\rfloor + \ldots \\ &= i + \left\lfloor \frac{x_i}{2^\ell} \right\rfloor \end{aligned}$$

Trovate entrambe basta concatenare il valore.

11.4.1 Information theoretical lower bound

Fissati n e u, **quante sono le possibili sequenze**

$$0 \leq x_0 \leq x_1 \leq \ldots \leq x_{n-1} < u$$

Equivalente a dire: quanti possibili insiemi di cardinalità n con ripetizioni ci sono?

Ancora equivalente al numero di soluzioni $\in \mathbb{N}^u$ di

$$c_0 + c_1 + \ldots + c_{u-1} = n$$

Dove c_i rappresenta quanti valori i sono presenti nella sequenza iniziale.

Le soluzioni possiamo contarle con lo stars & stripes counting:

- Scriviamo $u - 1$ barre
- Tra ogni barra e quella dopo poniamo c_i stelle, quindi ci saranno n stelle

Quante sequenze di n stelle e $u-1$ barre esistono?

$$\begin{aligned}\left(\begin{array}{c} u+n-1 \\ u-1 \end{array} \right) &= \frac{(u+n-1)!}{(u-1)!(u+n-1-u+1)!} \\ &= \frac{(u+n-1)!}{n!(u-1)!}\end{aligned}$$

Di conseguenza l'**information theoretical lower bound** è

$$Z_n = \log \left(\begin{array}{c} u+n-1 \\ u-1 \end{array} \right) = \log \left(\begin{array}{c} u+n-1 \\ n \end{array} \right)$$

Ma sapendo che:

$$\log \left(\begin{array}{c} A \\ B \end{array} \right) \sim B \log \frac{A}{B} + (A-B) \log \frac{A}{A-B}$$

E applicandola ponendo $A = u+n-1$ e $B = n$, quindi:

$$Z_n \sim n \log \frac{u+n-1}{n} + (u-1) \log \frac{u+n-1}{u-1}$$

Ma considerando $u \gg n$ (l'universo molto più grande dei valori che effettivamente usiamo), quindi il secondo termine (da $u-1$) è tutto ≈ 0, quindi:

$$\approx n \log \frac{u}{n} \left(1 + \frac{n}{u} - \frac{1}{u} \right) = n \log \frac{u}{n} + n \log \left(1 + \frac{n}{u} - \frac{1}{u} \right)$$

Sapendo che $x \approx \log(1+x)$:

$$Z_n \approx n \log \frac{u}{n} + n \left(\frac{n}{u} - \frac{1}{u} \right) \leq n \log \frac{u}{n} + \frac{n^2}{u}$$

sempre considerando $n \ll u$.

Riprendendo il D_n di prima:

$$D_n = 2n + n \left\lceil \log \frac{u}{n} \right\rceil + o(n)$$

Che è dominato dal $2n$, quindi

$$D_n = O(Z_n)$$

La struttura quindi, in questo caso, non è succinta ma **compatta**.

Risulta compatta con assunzioni generiche, **supponendo che** $n \leq \sqrt{u}$, abbastanza sensata considerando che abbiamo presupposto che $u \gg n$, la struttura anziché compatta **diventa succinta**.

11.5 Funzioni di Hash

Su un universo U, una **funzione di hash** è

$$h : U \to m$$

dove $|U| > |m|$. Questa viene chiamata "funzione di hash per U con m **bucket**".

Requisiti di una funzione di hash h:

1. h si deve calcolare in **tempo costante** (facile da calcolare).

2. h deve essere "molto iniettiva", non devono **esserci collisioni**, ma ci saranno per forza, quindi h deve spargere bene gli elementi su m bucket. Collision resistance, difficile trovare $h(x) = h(y)$ per $x \neq y$.

Solitamente per le strutture dati si usa la "**full randomness assumption**", ovvero che la funzione di hash sia scelta in modo completamente casuale tra tutti i possibili, cosa che non regge nel mondo reale.

Quindi, di solito, si **restringe la quantità di funzioni di hash possibili**, in quanto, probabilmente, non serviranno tutte.

Considerando una funzione di hash

$$h : U \to 100$$

E stringhe richieste alla funzione della forma:

$$x = x_1 \ \ldots \ x_n \in U$$

Quindi **scegliamo dei pesi**, con valore uniformemente a caso (qua possiamo farlo, questo è lo step fully random):

$$w_1, \ \ldots \ , w_{100}$$

E **definiamo** la nostra **funzione di hash** come:

$$h(x) := \left(\sum_{i=1}^{n} x_i \cdot w_i \right) \mod 100$$

Ovvero il prodotto di ogni componente della stringa per il corrispondente peso.

In questo modo stiamo definendo una particolare **famiglia di funzioni di hash**, all'interno della quale posso scegliere uniformemente a caso

$$\mathcal{H}_{U,m} \subseteq m^U$$

All'interno della quale vale la **proprietà**:

$$\forall x \in U \quad P[h(x) = t] = \frac{1}{m} \quad \forall t \in m$$

Ovvero, all'interno della funzione i valori sono suddivisi equamente tra tutti i bucket possibili.

Quindi, non possiamo scegliere uniformemente a caso, quindi restringo l'universo a qualcosa che posso scegliere davvero uniformemente a caso.

Funzione di Hash Perfetta: Una funzione di **hash è perfetta** per l'insieme X iff è **iniettiva** su X (è necessario che $m \geq |X|$, altrimenti è, ovviamente, impossibile).

Funzione di Hash minimale: Si dice "**minimale**" la funzione di hash perfetta con il **numero di bucket minori possibili** per un dato universo, quindi con $|X| = m$.

11.5.1 Tecnica MWHC

Per costruire una funzione di hash minimale. Dai nomi degli inventori: Majewski, Worwald, Havaś & Czech.

L'obiettivo della tecnica è quello di ottenere una **funzione di hash minimale perfetta ordinata**, ovvero che permette di mappare i valori in un ordine preciso (order-preserving minimal perfect hash, opmpf per gli amici). Questa tecnica può essere utilizzata per rappresentare funzioni statiche (generiche) ad r bit, quindi un qualsiasi dizionario di valori statico.

Partendo da un **universo** U e un **insieme** $X \subseteq U$, **con** $|X| = n$; la **funzione** f associa a ogni **valore** $x_i \in X$ un **vettore** di r bit

$$f : X \to 2^r$$

I Bro ci hanno spiegato come memorizzare una funzione di questo tipo.

Primi **step**:

1. **Fissano un** $m \geq n$
2. Scelgono **uniformemente a caso due funzioni di hash**

$$h_1, h_2 : U \to m$$

3. Costruiscono un **grafo** dove
   - i **vertici** sono i valori da 0 a m, quindi m vertici
   - c'è un **arco** tra due nodi i, j ogni volta che è presente una stringa tale che
   $$x \in X, \quad \{h_1(x), h_2(x)\} = \{i, j\}$$
   quindi tendenzialmente ci sono n lati

Situazioni spiacevoli:

- $h_1(x) = h_2(x)$ per qualche x, non vogliamo loop-de-loops
- $x, y \in X$, $x \neq y$ tali che $\{h_1(x), h_2(x)\} = \{h_1(y), h_2(y)\}$, in quanto vogliamo una corrispondenza 1 a 1 tra archi ed elementi di X
- il grafo è ciclico (vogliamo non lo sia), quindi per cercare di evitarlo scegliamo un m "abbastanza grande"

Se ci si ritrova in una di queste situazioni si **cambia funzioni**. Questi eventi, con m **abbastanza grande**, sono **poco probabili**.

Teorema: Con $m > 2.09n$, le funzioni h_1, h_2 hanno **quasi sempre** (asintoticamente) le **proprietà desiderate**, e il numero di **tentativi attesi** è circa 2.

Ogni arco del grafo corrisponde a un valore $x \in X$, di conseguenza ha associato un unico $f(x) \in 2^r$.

Possiamo creare un **sistema di equazioni** con m variabili

$$A[0], A[1], \ldots, A[m-1] \in 2^r$$

Dove ci sono $x \in X$ **equazioni** della forma:

$$\forall x \in X \quad (A[h_1(x)] + A[h_2(x)]) \mod 2^r = f(x)$$

Questo sistema **ammette soluzione** (grazie al fatto che il grafo è aciclico, ma non spiegheremo il perché, trust me bro) e memorizzo tali valori.

La **struttura dati contiene** le m **variabili** e i **valori** h_1, h_2 e m.

Ma quando devo usare la struttura? Calcolo

$$f(v) = A[h_1(v)] + A[h_2(v)] \mod 2^r$$

Ed è tempo costante, rispetto alla dimensione dell'input.

Quanto spazio occupa?

- la parte delle **funzioni di hash e** m occupa uno spazio che non dipende dai valori che vogliamo memorizzare quindi spazio **costante** $O(1)$ (che può essere o meno rilevante)
- le **variabili da memorizzare** occupano spazio in relazione alla dimensione di r, sono m soluzioni, quindi in **totale** $m \cdot r$ bit.

Da notare che non stiamo memorizzando le chiavi, stiamo rappresentando una funzione a partire da dei valori definiti, se provo a usare la struttura con una "chiave" non parte dell'insieme definito inizialmente funzionerà lo stesso, anche se non è detto che il risultato abbia senso.

11.5.2 Aciclicità → Peelability

Questa costruzione si può avere anche su **ipergrafi**, usando **più funzioni di hash**, minimizzando il bound del valore m a $1.23n$ con un 3-ipergrafo (3 funzioni di hash, ogni arco può collegare fino a 3 nodi); a salire di dimensioni aumenta il valore.

Con un **3-ipergrafo il teorema vale per** $m > \gamma n$, $\gamma = 1.23$.

Per un ipergrafo la nozione di aciclicità diventa "**peelability**".

Un ipergrafo (V, E) **ammette una peeling sequence** iff esiste un modo per **ordinare gli iperarchi** $e_1, \ldots, e_n \in E$ e una **sequenza di vertici** $x_1, \ldots, x_n \in V$ in modo da avere una **sequenza**

$$\begin{array}{ll} x_1, & e_1 \\ x_2, & e_2 \\ \vdots & \\ x_n, & e_n \end{array}$$

tale che:

1. $x_i \in e_i$, il lato si attacca a quel vertice (*hinge*)
2. $x_i \notin e_j \ \forall j < i$, ovvero andando avanti sono **sempre vertici nuovi**, non viene riusato lo stesso vertice

In un grafo normale è facile vedere come una peeling sequence esista solo nel caso aciclico. Per il caso in più dimensioni te lo lascio immaginare, ma per un 3-ipergrafo è come "sbucciare" uno dei "piani" che uniscono i vertici (ovvero i lati) per volta, senza toccare nodi vecchi.

Spazio occupato: Per la costruzione con MWHC su un 3-ipergrafo

$$\gamma nr \text{ bit}$$

Ma molte entry nel vettore delle soluzioni potrebbero non essere necessarie (se non presenti nel sistema di equazioni possono essere poste a 0), quindi posso **memorizzare solo i valori utili** (entry non zero), più un vettore di bit per memorizzare la posizione, su cui bisogna fare rank e select per trovare gli indici giusti.

Quindi la **versione compressa** occupa:

$$nr + \gamma n + o(n)$$

Che **conviene quando**

$$nr + \gamma n < \gamma nr \implies r > 5$$

per $\gamma = 1.23$.