Algoritmi e complessità

Indice

Parte I — Teoria della complessità

1. Notazione

Useremo i principali **insiemi numerici** come $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ e, ogni tanto, le loro versioni con soli elementi positivi $\mathbb{N}^+, \mathbb{Z}^+, \mathbb{Q}^+, \mathbb{R}^+$.

Un **magma** è una struttura algebrica $(A, \oplus)$ formata da un insieme e un'operazione. Se essa è:

- dotata di $\oplus$ **associativa** allora è detta **semigruppo**;
- dotata di un elemento $\overline{e} \in A$ tale che

$$\forall x \in A \quad x \oplus e = e \oplus x = x$$

  allora è detta **monoide**; l'elemento $\overline{e}$ è chiamato **elemento neutro** e in un monoide esso è unico; alcuni monoidi importanti sono $(\mathbb{N}, +)$ oppure $(\mathbb{N}, \cdot)$;
- dotata di $\oplus$ **commutativa** allora si aggiunge **abeliano** alla sua definizione.

Un **monoide libero** è un monoide i cui elementi sono generati da una base. Vediamo un importante monoide libero che useremo spesso durante il corso.

Partiamo da un **alfabeto** Σ, ovvero un insieme finito non vuoto di **lettere/simboli**. Definiamo Σ^* come l'insieme di tutte le sequenze di lettere dell'alfabeto Σ; queste sequenze sono dette **parole/stringhe** e una generica parola è $w \in \Sigma^*$ nella forma

$$w = w_0 ... w_{n-1} \mid n \geq 0 \land w_i \in \Sigma.$$

Usiamo $n \geq 0$ perché esiste anche la **parola vuota** ε. L'insieme Σ^* è **numerabile**.

Data una parola $w \in \Sigma^*$ indichiamo con $|w|$ il **numero di simboli** di w. La parola vuota è tale che $|\varepsilon| = 0$, ed è l'unica con questa proprietà.

Un'operazione che possiamo definire sulle parole è la **concatenazione**: l'operazione è

$$\cdot : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$$

ed è tale che, date

$$x = x_0 ... x_{n-1} \quad y = y_0 ... y_{m-1} \mid x, y \in \Sigma^*,$$

posso calcolare $z = x \cdot y$ come

$$z = x_0 ... x_{n-1} y_0 ... y_{m-1}.$$

Dato il magma $(\Sigma^*, \cdot)$, esso è:

- **semigruppo** perché $\cdot$ associativa;
- **non abeliano** perché $\cdot$ non commutativa (*lo sarebbe se* $\Sigma = \{x\}$);
- dotato di **neutro** $e = \varepsilon$.

Ma allora $(\Sigma^* \cdot)$ è un monoide. Esso è anche un **monoide libero** su Σ.

Cambiamo argomento. Chiamiamo

$$B^A = \{f \mid f : A \longrightarrow B\}$$

l'insieme di tutte funzioni da A in B; usiamo questa notazione perché la **cardinalità** di questo insieme, se A e B sono finiti, è esattamente $|B|^{|A|}$.

Spesso useremo un numero K come *«insieme»*: questo va inteso come l'insieme formato da K termini, ovvero l'insieme $\{0, 1, ..., k-1\}$. Ad esempio,

$$0 = \varnothing \quad | \quad 1 = \{0\} \quad | \quad 2 = \{0, 1\} \quad | \quad ...$$

Date queste due definizioni, vediamo qualche insieme particolare.

Esempio 1.1: Indichiamo con 2^A l'insieme

$$\{f \mid f : A \longrightarrow \{0,1\}\},$$

ovvero l'insieme delle funzioni che classificano gli elementi di A in un dato sottoinsieme di A, cioè ogni funzione determina un certo sottoinsieme. Possiamo quindi dire che

$$2^A \simeq \{X \mid X \text{ sottoinsieme di } A\}.$$

Questo insieme si chiama anche **insieme delle parti**, si indica con $\mathcal{P}(A)$ e ha cardinalità $2^{|A|}$, se A è finito.

Esempio 1.2: Indichiamo con A^2 l'insieme

$$\{f \mid f : \{0,1\} \longrightarrow A\}$$

l'insieme che rappresenta il **prodotto cartesiano**: infatti,

$$A^2 \simeq A \times A.$$

Esempio 1.3: Indichiamo con 2^* l'insieme delle stringhe binarie, ma allora l'insieme 2^{2^*} è la famiglia di tutti i linguaggi binari, ad esempio $\varnothing$, 2^*, $\{\varepsilon, 0, 00, 000, \ldots\}$, eccetera.

2. Algoritmi 101

2.1. Definizioni

In questo corso vedremo una serie di algoritmi che useremo per risolvere dei problemi, ma cos'è un problema?

Un **problema** Π è formato da:

- un insieme di input possibili $I_\Pi \subseteq 2^*$;
- un insieme di output possibili $O_\Pi \subseteq 2^*$;
- una funzione

$$\mathrm{Sol}_\Pi : I_\Pi \longrightarrow 2^{O_\Pi}/\{\oslash\}.$$

  Usiamo l'insieme delle parti come codominio perché potrei avere più risposte corrette per lo stesso problema.

Un **algoritmo** lo possiamo identificare in una **Macchina di Turing**. Sappiamo già come è fatta, l'abbiamo vista tante volte, ovvero contiene:

- un **nastro** bidirezionale infinito con input e blank:
- una **testina** di lettura/scrittura two-way;
- un **controllo a stati finiti**;
- un **programma/tabella** che permette l'evoluzione della computazione.

Perché usiamo una MdT quando abbiamo a disposizione una macchina a registri, ad esempio una macchina RAM, WHILE o lambda-calcolo?

La **tesi di Church-Turing** afferma un risultato molto importante che possiamo dare in più *«salse»*:

- tutte le macchine create e che saranno create sono equivalenti, ovvero quello che fai con una macchina lo fai anche con l'altra;
- nessuna definizione di algoritmo può essere diversa da una macchina di Turing;
- la famiglia dei problemi di decisione che si possono risolvere è uguale per tutte le macchine;
- i linguaggi di programmazione sono Turing-completi, ovvero se ipotizziamo una memoria infinita allora è come avere una MdT.

Anche un computer quantistico è una MdT, come calcolo almeno, perché in tempo si ha la quantum supremacy.

Un **algoritmo** A per Π è una MdT tale che

$$x \in I_\Pi \rightsquigarrow \boxed{\text{A}} \rightsquigarrow y \in O_\Pi$$

tale che $y \in \mathrm{Sol}_\Pi(x)$, ovvero quello che mi restituisce l'algoritmo è sempre la risposta corretta.

Ma tutti i problemi sono risolvibili? La risposta è no, e lo possiamo vedere con le cardinalità:

- i problemi di decisione sono i problemi dell'insieme 2^{2^*}, ovvero data una stringa binaria devo dire se essa sta o meno nell'insieme delle istanze positive del problema di decisione; questo insieme è tale che

$$\left|2^{2^*}\right| \approxeq \left|2^{\mathbb{N}}\right| \approxeq |\mathbb{R}|;$$

- i programmi non sono così tanti: visto che i programmi sono stringhe, e visto che Σ^* è numerabile, le stringhe su un linguaggio sono tali che $2^* \sim \mathbb{N}$.

Si dimostra che $\mathbb{N} \nsim \mathbb{R}$, quindi sicuramente esistono dei problemi che non sono risolvibili.

2.2. Teoria della complessità

Una volta ristretto lo studio ai solo problemi risolvibili possiamo chiederci quanto efficientemente lo riusciamo a fare: questa branca di studio è detta **teoria della complessità**.

In questo ambito vogliamo vedere quante risorse spendiamo durante l'esecuzione dell'algoritmo o del programma. Abbiamo in realtà due diverse teorie della complessità: algoritmica e strutturale.

La **teoria della complessità algoritmica** ci chiede di:

- stabilire se un problema Π è risolubile;
- se sì, con che costo rispetto a qualche risorsa.

Le risorse che possiamo studiare sono:

- **tempo**, come numero di passi o tempo cronometrato;
- **spazio**;
- numero di CPU nel punto di carico massimo;
- somma dei tempi delle CPU;
- energia dissipata.

Noi useremo quasi sempre il **tempo**. Definiamo

$$T_A : I_\Pi \longrightarrow \mathbb{N}$$

la funzione che ci dice, per ogni input, quanto ci mette l'algoritmo A a terminare su quell'input.

Questo approccio però non è molto comodo. Andiamo a raccogliere per lunghezza e definiamo

$$t_A : \mathbb{N} \longrightarrow \mathbb{N}$$

la funzione

$$t_A(n) = \max\{T_A(x) \mid x \in I_\Pi \wedge |x| = n\}$$

che va ad applicare quella che è la filosofia **worst case**. In poche parole, andiamo a raccogliere gli input con la stessa lunghezza e prendiamo, per ciascuna categoria, il numero di passi massimo che è stato rilevato. Anche questa soluzione però non è bellissima: è una soluzione del tipo *«STA ANDANDO TUTTO MALEEEEE»*.

Abbiamo altre soluzioni? Sì, ma non fanno meglio di quanto abbiamo già:

- la soluzione **best case** è troppo sbilanciata verso il *«sta andando tutto bene»*;
- la soluzione **average case** è complicata perché serve una distribuzione di probabilità.

A questo punto teniamo l'approccio worst case perché rispetto agli altri due non va a rendere complicati i conti. Inoltre, prendere il massimo ci dà la certezza di non fare peggio di quel valore.

La **teoria della complessità strutturale** invece, fissato Π un problema, si chiede quale sia la complessità di Π. La differenza sembra minima con la complessità algoritmica, ma è abissale: mi interessa la complessità del problema, non del singolo algoritmo che lo risolve. Sto chiedendo quale sia la complessità del migliore algoritmo che lo risolve.

Per entrambe le complessità ci serviremo della **complessità asintotica**, ovvero per n molto grandi vogliamo vedere il comportamento dei vari algoritmi, perché *«con i dati piccoli sono bravi tutti»*.

Il simbolo per eccellenza che useremo è l'O-grande: se un algoritmo ha complessità $O(f(n))$ vuol dire che $f(n)$ domina il tempo t_A del nostro algoritmo in esame. Piccolo appunto: dobbiamo stare comunque attenti alle costanti dentro O e Ω, quindi prendiamo tutte le complessità con le pinze.

Torniamo alle due teorie. Durante lo studio di queste ultime abbiamo due squadre di operai che fanno due lavori:

- **upper bound**: cerchiamo una soluzione per l'algoritmo, e cerchiamo poi di migliorarla continuamente abbassandone la complessità; in poche parole, questa squadra cerca di abbassare sempre di più la soglia indicata con $O(f(n))$ per avere una soluzione sempre migliore;
- **lower bound**: cerchiamo di dimostrare che il problema non si può risolvere in meno di $f(n)$ risorse; indichiamo questo *«non faccio meglio»* con $\Omega(f(n))$ e, al contrario dell'altra squadra, questo valore cerchiamo di alzarlo il più possibile; non dobbiamo esibire un algoritmo, bensì una prova.

Quando le due complessità coincidono abbiamo chiuso la questione:

- non faccio meglio di $f(n)$,
- non faccio peggio di $f(n)$,

ma allora ci metto esattamente $f(n)$, a meno di costanti, e questa situazione si indica con $\Theta(f(n))$.

È molto raro arrivare ad avere una complessità con Θ: l'ordinamento di array è $\Theta(n \log(n))$, ma è uno dei pochi casi, di solito si ha gap abbastanza grande.

Il problema sorge quando l'upper bound è esponenziale e il lower bound è polinomiale: ci troviamo in una **zona grigia** che potrebbe portarci ad algoritmi molto efficienti o ad algoritmi totalmente inefficienti. I problemi interessanti sono spesso e volentieri nella zona grigia.

3. Problemi di decisione

Se in un problema mi viene chiesto di *«decidere qualcosa»*, siamo davanti ad un **problema di decisione**: questi sono particolari perché hanno $O_\Pi = \{0, 1\}$ e hanno **una sola risposta possibile**, vero o falso. Non posso ciò avere un sottoinsieme di risposte possibili, ma una *«risposta secca»*.

3.1. Classi di complessità

Non possiamo studiare ogni problema singolarmente, quindi creiamo delle **classi di problemi** e andiamo a studiarli tutti assieme.

Le due classi più famose sono P e NP:

- P è la **classe dei problemi di decisione risolvibili in tempo polinomiale**; ci chiederemo sempre se un problema Π sta in P, questo perché ci permetterà di scrivere degli algoritmi efficienti per tale problema;
- NP è la **classe dei problemi di decisione risolvibili in tempo polinomiale su macchine non deterministiche**.

Cosa sono le **macchine non deterministiche**? Supponiamo di avere un linguaggio speciale, chiamato N-python, dotato di una istruzione pazza

$$x =?$$

che, quando viene eseguita, sdoppia l'esecuzione del programma, assegnando $x = 0$ nella prima istanza e $x = 1$ nella seconda istanza. Queste due istanze vengono eseguite in parallelo. Questa istruzione può essere però eseguita un numero arbitrario di volte su un numero arbitrario di variabili: questo genera un **albero di computazioni**, nel quale abbiamo delle foglie che contengono uno dei tanti assegnamenti di 0 e 1 delle variabili *«sdoppiate»*.

Tutte queste istanze y_i che abbiamo nelle foglie le controlliamo:

- rispondiamo *SI* se **ESISTE** un *SI* tra tutte le y_i;
- rispondiamo *NO* se **TUTTE** le y_i sono *NO*.

Questa macchina è però impossibile da costruire: posso continuare a forkare il mio programma, ma prima o poi le CPU le finisco per la computazione parallela.

Molti problemi che non sappiamo se stanno in P sappiamo però che sono in NP. Il problema più famoso è **CNF-SAT**: l'input è un'espressione logica in forma normale congiunta del tipo

$$\varphi = (x_1 \vee x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_3 \vee x_1),$$

formata da una serie clausole unite da *AND*. Ogni clausola è combinazione di *letterali* (normali o negati) legati da *OR*. Data φ formula in CNF, ci chiediamo se sia **soddisfacibile**, ovvero se esiste un assegnamento che rende φ vera. Un **assegnamento** è una lista di valori di verità che diamo alle variabili x_i per cercare di rendere vera φ.

Questo problema è facilmente *«scrivibile»* in una macchina non deterministica: per ogni variabile $x_i \mid i = 1, \dots, n$ eseguo l'istruzione magica $x_i =?$ che genera così tutti i possibili assegnamenti alle variabili, che sono 2^n, e poi controllo ogni assegnamento alla fine delle generazioni. Se almeno uno rende vera φ rispondo *SI*. Il tempo è polinomiale: ho rami esponenziali ma ogni ramo deve solo controllare n variabili.

Come siamo messi con CNF-SAT? Non sappiamo se sta in P, ma sicuramente sappiamo che sta in NP. Tantissimi problemi hanno questa caratteristica. Ma esiste una relazione tra le classi P e NP?

La relazione più ovvia è $P \subseteq NP$: se un problema lo so risolvere senza l'istruzione magica in tempo polinomiale allora creo un programma in N-python identico che però non usa l'istruzione magica che viene eseguito in tempo polinomiale.

Quello che non sappiamo è l'implicazione inversa, ovvero se $NP \subseteq P$ e quindi se $P = NP$. Questo problema è stato definito da **Cook**, che affermava di *«avere a portata di mano il problema»*, e invece...

Abbiamo quindi due situazioni possibili:
- se $P = NP$ è una situazione rassicurante perché so che tutto quello che ho davanti è polinomiale;
- se $P \neq NP$ è una situazione meno rassicurante perché so che esiste qualcosa di non risolvibile ma non so se il problema che ho sotto mano ha o meno questa proprietà.

Per studiare questa funzione possiamo utilizzare la **riduzione in tempo polinomiale**, una relazione tra problemi di decisione. Diciamo che Π_1 è **riducibile in tempo polinomiale** a Π_2, e si indica con

$$\Pi_1 \underset{p}{\leq} \Pi_2,$$

se e solo se $\exists f : I_{\Pi_1} \longrightarrow I_{\Pi_2}$ tale che:
- f è calcolabile in tempo polinomiale;
- $\text{Sol}_{\Pi_1}(x) = \text{SI} \iff \text{Sol}_{\Pi_2}(f(x)) = \text{SI}$.

Grazie a questa funzione riesco a cambiare, in tempo polinomiale, da Π_1 a Π_2 e, se riesco a risolvere una delle due, allora riesco a risolvere anche l'altra. Il $\leq$ indica che il primo problema «non è più difficile» del secondo.

Teorema 3.1.1: Se $\Pi_1 \underset{p}{\leq} \Pi_2$ e $P_2 \in P$ allora $P_1 \in P$.

Teorema 3.1.2 (*Teorema di Cook*): Il problema CNF-SAT è in NP e

$$\forall \Pi \in NP \quad \Pi \underset{p}{\leq} \textit{CNF-SAT}\ .$$

Questo teorema è un risultato enorme: afferma che CNF-SAT è un problema al quale tutti gli altri si possono ridurre in tempo polinomiale. In realtà CNF-SAT non è l'unico problema: l'insieme di problemi che hanno questa proprietà è detto insieme dei problemi **NP-completi**, ed è definito come

$$NP\text{-}C = \left\{ \Pi \in NP \mid \forall \Pi' \in NP \quad \Pi' \underset{p}{\leq} \Pi \right\}.$$

Per dimostrare che un problema è NP-completo basta far vedere che CNF-SAT si riduce a quel problema, vista la proprietà transitiva della riduzione polinomiale. Se un problema è in NP-C lo possiamo definire come *«roba probabilmente difficile»*.

Corollario 3.1.2.1: Se $\Pi \in NP$-C e $\Pi \in P$ allora $P = NP$.

Questo corollario ci permette di ridurre la ricerca della risposta di $P = NP$ ai soli problemi in NP-C.

4. Problemi di ottimizzazione

Durante questo corso non vedremo quasi mai problemi di decisione, ma ci occuperemo quasi interamente di **problemi di ottimizzazione**. Questi problemi sono un caso particolare dei problemi.

Dato Π un problema di ottimizzazione, allora questo è definito da:

- **input**: $I_\Pi \subseteq 2^*$;
- **soluzioni ammissibili**: esiste una funzione

  $$\mathrm{Amm}_\Pi : I_\Pi \longrightarrow 2^{2^*}/\{\oslash\}$$

  che mappa ogni input in un insieme di soluzioni ammissibili:
- **obiettivo**: esiste una funzione

  $$c_\Pi : 2^* \times 2^* \longrightarrow \mathbb{N}$$

  tale che $\forall x \in I_\Pi$ e $\forall y \in \mathrm{Amm}_\Pi(x)$ la funzione $c_\Pi(x, y)$ mi dà il costo di quella soluzione; questa funzione è detta **funzione obiettivo**;
- **tipo**: identificatore di c_Π, che può essere una funzione di massimizzazione o minimizzazione, ovvero $T_\Pi \in \{\min, \max\}$.

Esempio 4.1: Vediamo **MAX CNF-SAT**, una versione alternativa di CNF-SAT. Questo problema è definito da:

- **input**: formula logica in forma normale congiunta;
- **soluzioni ammissibili**: data φ formula CNF, in questo insieme A ho tutti i possibili assegnamenti a_i delle variabili di φ;
- **obiettivo**: $c_\Pi(\varphi, a_i \in A)$ deve contare il numero di *clausole* rese vere dall'assegnamento a_i;
- **tipo**: $T_\Pi = \max$.

Sicuramente questo problema è non polinomiale: se ce l'avessimo, questo mi darebbe l'assegnamento massimo, che poi in tempo polinomiale posso buttare dentro CNF-SAT per vedere se φ con tale assegnamento è soddisfacibile, ma questo non è possibile perché CNF-SAT non è risolvibile in tempo polinomiale (*o almeno, abbiamo assunto tale nozione*).

Ad ogni problema di ottimizzazione Π possiamo associare un problema di decisione $\hat{\Pi}$ con:

- **input**: $I_{\hat{\Pi}} = \{(x, k) \mid x \in I_\Pi \wedge k \in \mathbb{N}\}$;
- **domanda**: la risposta sull'input (x, k) è
  - *SI* se e solo se $\exists y \in \mathrm{Amm}_\Pi(x)$ tale che:
    - $c_\Pi(x, y) \leq k$ se $T_\Pi = \min$;
    - $c_\Pi(x, y) \geq k$ se $T_\Pi = \max$;
  - *NO* altrimenti.

Il valore k fa da bound al valore minimo o massimo che vogliamo accettare.

Esempio 4.2: Vediamo il problema MAX-$\widehat{\text{CNF}}$-SAT:

- **input**: $I = \{(\varphi, k) \mid \varphi \text{ formula CNF} \wedge k \in \mathbb{N}\}$;
- **domanda**: la risposta a (φ, k) è *SI* se e solo se esiste un assegnamento che rende vere almeno k clausole di φ.

La classe di complessità che contiene i problemi di ottimizzazione Π risolvibili in tempo polinomiale è la classe PO.

Teorema 4.1: Se $\Pi \in PO$ allora il suo problema di decisione associato $\hat{\Pi} \in P$.

Corollario 4.1.1: Se $\hat{\Pi} \in NP\text{-}C$ allora $\Pi \notin PO$.

Noi useremo spesso problemi che hanno problemi di decisione associati $NP\text{-}C$. Ci sono dei problemi in PO? Certo: i problemi di programmazione lineare sono tutti problemi in PO.

Cosa si fa se, dato un problema di ottimizzazione, vediamo che il suo associato è NPC?

Una possibile soluzione sono le **euristiche**, però non sappiamo se funzionano bene o funzionano male, perché dipendono molto dall'input.

Una soluzione migliore sono le **funzioni approssimate**: sono funzioni polinomiali che mi danno soluzioni non ottime ma molto vicine all'ottimo rispetto ad un errore che scegliamo arbitrariamente.

Dato Π problema di ottimizzazione, chiamiamo $\text{opt}_\Pi(x)$ il valore ottimo della funzione obiettivo su input x. Dato un algoritmo approssimato per Π, ovvero un algoritmo tale che

$$x \in I_\Pi \rightsquigarrow A \rightsquigarrow y \in \text{Amm}_\Pi(x)$$

mi ritorna una soluzione ammissibile, non per forza ottima, definisco **rapporto di prestazioni** il valore

$$R_\Pi(x) = \max\left\{\frac{c_\Pi(x,y)}{\text{opt}_\Pi(x)}, \frac{\text{opt}_\Pi(x)}{c_\Pi(x,y)}\right\}.$$

Si dice che A è una α-approssimazione per Π se e solo se

$$\forall x \in I_\Pi \mid R_\Pi \leq \alpha.$$

In poche parole, su ogni input possibile vado male al massimo quanto α.

È una definizione un po' contorta ma funziona: se la funzione obiettivo è di massimizzazione allora la prima frazione ha

$$\text{num} \leq \text{den},$$

mentre la seconda frazione mi dà sempre un valore ≥ 1, che quindi sarà il valore scelto per R_Π. Nel caso di funzione obiettivo di minimizzazione la situazione è capovolta.

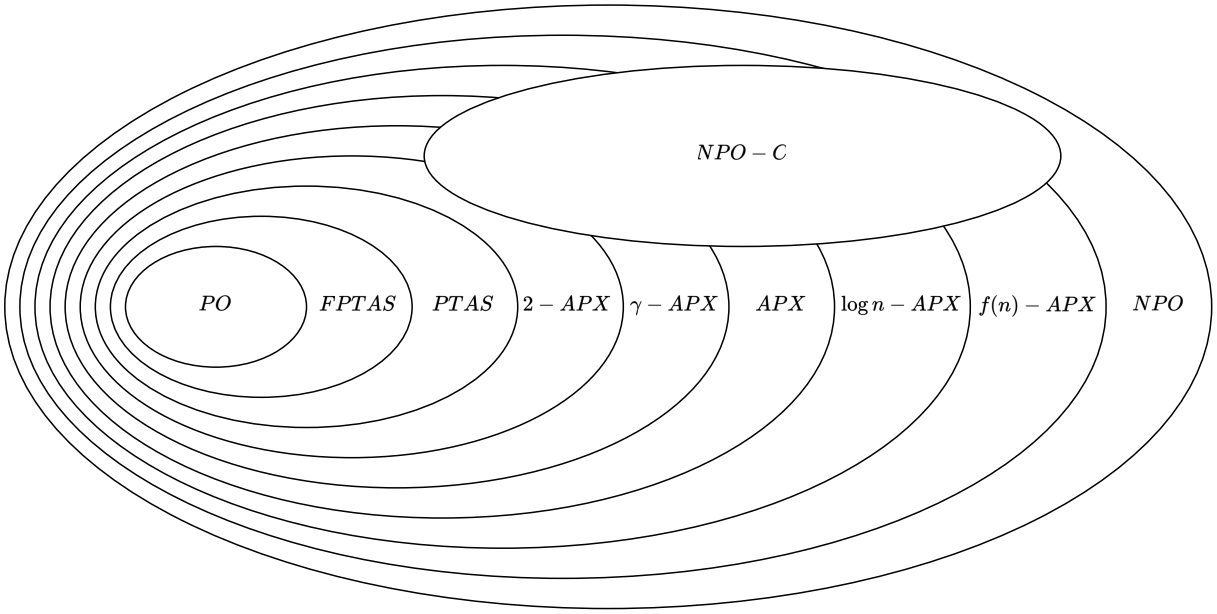
Esempio 4.3: Se $R_\Pi = 1$ allora l'algoritmo su quell'input è ottimo e mi dà la soluzione migliore. Se, ad esempio, $R_\Pi = 2$, allora ho ottenuto

- un costo doppio dell'ottimo (*minimizzazione*);
- un costo dimezzato dell'ottimo (*massimizzazione*).

Vorremmo sempre avere un algoritmo 1-approssimante perché siamo all'ottimo, ma se non riusciamo ad avere ciò vorremmo almeno esserci molto vicino.

4.1. Classi di complessità

Quali altri classi esistono in questo zoo della classi di ottimizzazione?



In ordine, PO è la più piccola, poi ci sono le classi γ-APX, con $\gamma \in \mathbb{R}^{\geq 1}$, che rappresentano tutte le classi che contengono i problemi con algoritmi γ-approssimati. La classe che li tiene tutti è

$$APX = \bigcup_{\alpha \geq 1} \alpha\text{-}APX \ .$$

Una classe ancora più grande è $\log(n)$-APX: questa classe non usa una costante per definirsi, ma più l'input diventa grande più l'approssimazione diventa peggiore. La classe generale è la classe $f(n)$-APX, i cui algoritmi dipendono dalla funzione $f(n)$.

Il container più grande di tutti è NPO, che contiene anche gli NPO-completi, classe trasversale a tutti gli APX. La definizione di questa classe è abbastanza complicata.

Rompiamo l'ordine e vediamo infine altre due classi:

- $PTAS$ (*Polynomial Time Approximation Scheme*): classe che contiene gli algoritmi approssimabili a piacere, ma che non sono totalmente polinomiali;
- $FPTAS$ (*Fully Polynomial Time Approximation Scheme*): classe che contiene gli algoritmi approssimabili a piacere, ma che mantengono un tempo polinomiale al ridursi dell'errore.

Parte II — Algoritmi di approssimazione

1. Max Matching

Il primo (*e unico*) problema di ottimizzazione della classe PO che vedremo sarà il problema di **Max Matching**. Esso è definito da:

- **input**: grafo $G = (V, E)$ non orientato e bipartito, ma quest'ultima condizione può anche essere tolta. Un **grafo bipartito** è un grafo nel quale i vertici sono divisi in due blocchi (*colori*) e i lati vanno da un vertice del primo blocco ad un vertice del secondo blocco (*e viceversa, ma tanto è non orientato*). In poche parole, non ho lati che collegano vertici nello stesso blocco;
- **soluzioni ammissibili**: un **matching** M, una scelta di lati tale che i vertici del grafo risultano incisi al più da un lato. Il matching lo vediamo come un matrimonio, quindi dobbiamo evitare i matrimoni poligami, ovvero far incidere ogni vertice da più di un arco; non dobbiamo far sposare tutti, accettiamo anche i single (*come me* 😢);
- **obiettivo**: numero di match $|M|$;
- **tipo**: max.

Una soluzione in tempo polinomiale sfrutta l'**algoritmo del cammino aumentante**.

Un cammino aumentante si applica ad un grafo con un matching M parziale. Per trovare i cammini aumentanti ci serviranno i **vertici esposti**, ovvero vertici sui quali non incidono i lati presi nel matching.

Un **cammino aumentante** (*augmenting path*) è un cammino che parte e arriva su un vertice esposto e alterna *«lati liberi»* e lati di M. Se so che esiste un cammino aumentante, scambio i lati presi e quelli non presi facendo un'operazione di **switch**: il matching cambia ma soprattutto aumenta di 1 il numero di lati presi.

Questa è un'informazione pazza: se so che esiste un cammino aumentante il matching non è massimo e lo posso quindi migliorare.

Lemma 1.1: Se esiste un cammino aumentante per il matching M allora M non è massimo.

Dimostrazione 1.1: Banale, se esiste un cammino aumentante allora tramite l'operazione di switch ottengo un matching che contiene un lato in più di M, quindi M non è massimo. ■

Lemma 1.2: Se il matching M non è massimo allora esiste un cammino aumentante per M.

Dimostrazione 1.2: Sia M' un matching tale che $|M'| > |M|$, che esiste per ipotesi. I matching sono un insieme di lati, quindi potremmo avere:

- lati solo in M (M/M');
- lati solo in M' (M'/M);
- lati sia in M sia in M' ($M \cap M'$).

Prendiamo i lati che sono in

$$M \Delta M' = (M/M') \cup (M'/M) = (M \cup M')/(M \cap M')$$

differenza simmetrica dei due match.

Osserviamo che nessun vertice può avere più di due lati incidenti in $M \Delta M'$. Possiamo dire di più: se un vertice ha esattamente due lati incidenti allora questi arrivano da due match diversi per definizione di match.

Se disegniamo il grafo con i soli vertici che sono incisi dai lati di $M \Delta M'$, abbiamo solo vertici di grado 1 e vertici di grado 2. Un grafo di questo tipo ha solo cammini e cicli, non esiste altro.

Se consideriamo i cicli, essi sono formati da vertici di grado 2, che hanno due lati incidenti ma arrivano da due matching diversi. Questo implica che ogni ciclo copre lo stesso numero di lati di M/M' e lati di M'/M e hanno lunghezza pari. Ma visto che $|M'| > |M|$ deve esistere qualcosa nel grafo che abbia più lati di M' al suo interno.

Se non è un ciclo, allora è un cammino: infatti, quello detto poco fa sui cicli implica che esiste un cammino nel grafo che è formato da più lati di M'/M (*esattamente uno in più*), ovvero un cammino che inizia a finisce con lati di M'/M. Questo cammino, dal punto di vista di M, è aumentante: alterna lati di M con lati che non sono in M (*per definizione di differenza*) e ai bordi ci sono due vertici che non sono incisi da lati di M. ■

Per trovare un cammino aumentante dobbiamo fare una **visita di grafo**. Una visita è un modo sistematico che si usa per scoprire un grafo. Abbiamo tre tipi di nodi:

- nodi sconosciuti (*bianchi*);
- nodi conosciuti ma non ancora visitati, che sono in una zona detta **frontiera** (*grigi*);
- nodi visitati (*neri*).

Vediamo come funziona l'algoritmo di visita, usando la funzione $c(x) \leftarrow t$ che assegna al vertice x il colore t.

Algoritmo di visita

```
1:  for x in V:
2:      c(x) ← W
3:  F ← {x_seed}
4:  c(x_seed) ← G
5:  while F ≠ ⦰:
6:      x ← pick(F)
7:      visit(x)
8:      c(x) ← B
9:      for y in neighbor(x):
10:         if c(y) == W:
11:             F ← F ∪ {y}
12:             c(y) ← G
```

Se il grafo è **connesso** lo riesco a visitare tutto e ogni vertice lo visito una volta sola. Se non è connesso visito solo la componente connessa del seme e, per continuare la visita, devo mettere un nuovo

seme nella frontiera per andare avanti. Questo algoritmo è quindi ottimo per trovare le **componenti connesse** del grafo.

La funzione `pick` determina il comportamento di questa visita: se utilizziamo uno **stack** stiamo facendo una **DFS** (*visita in profondità*), se utilizziamo invece una **pila** stiamo facendo una **BFS** (*visita in ampiezza*). Infatti, in base a come si comporta la funzione `pick` abbiamo un ordine di scelta dei nodi diverso.

La BFS è interessante perché, a partire dal seme, nella frontiera metto i nodi vicini al seme, poi i vicini dei vicini del seme, poi eccetera. In poche parole, visito nell'ordine i nodi alla stessa distanza dal seme. Questo è uno dei modi standard per calcolare le distanze in un grafo non orientato e non pesato. Andremo quindi ad usare una BFS per trovare i cammini aumentanti.

Find Augmenting

1: $X \leftarrow$ vertici esposti in M
2: for x in X:
3: | BFS(x) con alternanza di lati in M e lati fuori da M
4: | Se durante la ricerca trovo un altro vertice di X
5: | | ritorno il cammino trovato

La BFS ha tempo proporzionale al numero di lati, quindi Find Augmenting impiega tempo $O(nm)$. Quanti aumenti posso fare? Al massimo $\frac{n}{2}$, quindi ho $O\left(\frac{n^2}{2}m\right)$, che è al massimo $O(n^4)$.

Teorema 1.1: Bipartite Max Matching è in PO.

Corollario 1.1.1: Il problema di decisione Perfect Matching è in P.

Se il grafo non fosse bipartito avremmo l'**algoritmo di fioritura**, che non sfrutta la BFS.

2. Load Balancing

Usciamo fuori dai problemi in *PO* e vediamo il problema del **Load Balancing**. Esso è definito da:

- **input**:
  - $m > 0$ numero di macchine;
  - $n > 0$ numero di task;
  - $(t_i)_{i \in n} > 0$ durate dei task;
- **soluzione ammissibile**: funzione

$$\alpha : n \longrightarrow m$$

  che assegna ogni task ad una macchina. Il **carico** di una macchina j è la quantità

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

  Il **carico generale** è

$$L = \max_j L_j;$$

- **funzione obiettivo**: L;
- **tipo**: min.

Teorema 2.1: Load Balancing è *NPO-C*.

Che problemi stanno in *NPO*? È difficile dare una definizione di non determinismo nei problemi di ottimizzazione, però possiamo definire *NPO-C*: un problema di ottimizzazione Π è *NPO*-completo se e solo se:

- $\Pi \in NPO$;
- $\hat{\Pi} \in NP\text{-}C$.

Se un problema in *NPO-C* fosse polinomiale allora il suo problema di decisione associato sarebbe polinomiale, e questo non può succedere.

Vediamo un **algoritmo greedy**, una tecnica di soluzione che cerca di ottimizzare *«in modo miope»*, ovvero costruisce passo dopo passo la soluzione prendendo ogni volta la direzione che sembra ottima in quel momento.

Greedy Load Balancing

input
- $m > 0$ numero di macchine
- $n > 0$ numero di task
- $(t_i)_{i \in n}$ durata di ogni task

1: for i in m:
2: $\quad A_i \leftarrow \varnothing$ (task assegnate alla macchina)
3: $\quad L_i \leftarrow 0$ (carico della macchina)
4: for j in n:
5: $\quad i \leftarrow \arg\min_{t \in m} L_t$ (indice della macchina con meno carico)
6: $\quad A_i \leftarrow A_i \cup \{j\}$
7: $\quad L_i \leftarrow L_i + t_j$

Greedy Load Balancing

8: **output** α assegna ogni elemento j di A_i alla macchina i

Il tempo d'esecuzione di questo algoritmo è $O(nm)$, ed è molto comodo perché è possibile utilizzarlo anche **online**, ovvero quando i task non sono tutti conosciuti ma possono arrivare anche durante l'esecuzione dell'algoritmo.

Teorema 2.2: Greedy Load Balancing è una 2-approssimazione per Load Balancing.

Dimostrazione 2.1: Chiamiamo L^* il valore della funzione obiettivo nella soluzione ottima.

Osserviamo che:

1. vale
$$L^* \geq \frac{1}{m} \sum_j t_j,$$
ovvero il carico migliore ci mette almeno un tempo uguale allo *«spezzamento perfetto»*, cioè quello che assegna ad ogni macchina lo stesso carico (*caso ideale, che segue la media*);
2. vale
$$L^* \geq \max_j t_j,$$
ovvero una macchina deve impiegare almeno il tempo più grande tra quelli disponibili.

Guardiamo la macchina che dà il massimo carico, ovvero sia $\hat{i}$ tale che $L_{\hat{i}} = L$ e sia $\hat{j}$ l'ultimo compito che le è stato assegnato. Se assegno $\hat{j}$ vuol dire che poco prima questa macchina era la più scarica, quindi

$$L_{\hat{i}} - t_{\hat{j}} = \underbrace{L_{\hat{i}}'}_{\text{carico in quel momento}} \leq L_i \quad \forall i \in m.$$

Sommiamo rispetto al numero di macchine, quindi

$$\sum_{i \in m} L_{\hat{i}} - t_{\hat{j}} = m\left(L_{\hat{i}} - t_{\hat{j}}\right) \leq \sum_{i \in m} L_i = \sum_{j \in n} t_j.$$

Dividiamo tutto per m e otteniamo

$$L_{\hat{i}} - t_{\hat{j}} \leq \frac{1}{m} \sum_j t_j \overset{(1)}{\leq} L^*.$$

Sappiamo che

$$L = L_{\hat{i}} = \underbrace{L_{\hat{i}} - t_{\hat{j}}}_{\leq L^*} + \underbrace{t_{\hat{j}}}_{\leq L^* \text{ per } (2)} \leq 2L^*$$

quindi

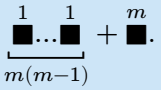
$$\frac{L}{L^*} \leq 2. \qquad \blacksquare$$

L'algoritmo Greedy Load Balancing non è **tight**, ovvero posso costruire un input che si avvicini a piacere all'approssimazione del problema.

Teorema 2.3: $\forall \varepsilon > 0$ esiste un input per Load Balancing tale che Greedy Load Balancing produce un output

$$2 - \varepsilon \leq \frac{L}{L^*} \leq 2.$$

Dimostrazione 2.2: Scelgo un numero di macchine $m > \frac{1}{\varepsilon}$ e un numero di task $n = m(m-1)+1$. Come sono fatti questi task? Li dividiamo come



Ragionando con un approccio greedy, assegno tutti gli $m(m-1)$ task da 1 alle m macchine, quindi ognuna ha $m-1$ task, poi manca solo quella che dura m, che assegno a caso visto che hanno tutte lo stesso carico. Il carico della macchina scelta sarà $L = m - 1 + m = 2m - 1$.

Ragionando invece su come dovrebbe essere la configurazione ottima, questa dovrebbe assegnare la task grande m alla prima macchina e le altre $m(m-1)$ alle restanti $m-1$ macchine, quindi ognuna, compresa la prima, ha ora un carico di m, quindi $L^* = m$.

Confrontiamo i due valori:

$$\frac{L}{L^*} = \frac{2m-1}{m} = 2 - \frac{1}{m} \geq 2 - \varepsilon. \qquad \blacksquare$$

Vediamo ora un algoritmo migliore per il Load Balancing.

Sorted Greedy Load Balancing

input

- $m > 0$ numero di macchine
- $n > 0$ numero di task
- $(t_i)_{i \in n}$ durata di ogni task

1: Ordina in maniera decrescente l'insieme $(t_i)_{i \in n}$
2: Usa l'algoritmo Greedy Load Balancing

Teorema 2.4: Sorted Greedy Load Balancing è una $\frac{3}{2}$-approssimazione per Load Balancing.

Dimostrazione 2.3: Osserviamo che se $n \leq m$ la soluzione prodotta è ottima. Consideriamo quindi il caso $n > m$.

Osserviamo inoltre che $L^* \geq 2t_m$, ovvero che $\frac{1}{2}L^* \geq t_m$, con

$$\underbrace{t_0 \geq \ldots \geq t_m}_{m+1 \text{ task}} \geq t_{m+1} \geq \ldots \geq t_{n-1}.$$

Infatti, la macchina che riceve 2 task ha carico

$$\geq t_i + t_j \geq 2t_m.$$

Sia $\hat{i}$ l'indice della macchina con carico massimo, ovvero $L_{\hat{i}} = L$. Se $\hat{i}$ ha un compito solo, la soluzione è ottima. Consideriamo allora $\hat{i}$ con più di un compito: sia $\hat{j}$ l'ultimo compito assegnato a quella macchina. So che $\hat{j} \geq m$, perché le prime m task le do ad ogni macchina i distinta, allora

$$L = L_{\hat{i}} = \underbrace{L_{\hat{i}} - t_{\hat{j}}}_{\leq L^*} + \underbrace{t_{\hat{j}}}_{\leq t_m \leq \frac{1}{2}L^*} \leq \frac{3}{2}L^*.$$

Ma allora

$$\frac{L}{L^*} \leq \frac{3}{2}.$$

■

Graham nel 1969 ha poi dimostrato che questo algoritmo in realtà è

$$\frac{4}{3}\text{-}APX\ .$$

Hochbaum e Shmoys hanno dimostrato, nel 1988, che:

- Load Balancing ammette un *PTAS*;
- Load Balancing non ammette un *FPTAS*.

3. Center Selection

Vediamo un esempio: abbiamo un insieme di magazzini di Amazon e vogliamo scegliere tra questi dei «*super magazzini*», ovvero magazzini ai quali gli altri magazzini si riferiscono. L'insieme di super magazzino più magazzini che si riferiscono al super magazzino è detto **cella di Voronoi**.

Uno **spazio metrico** è una coppia (Ω, d) con Ω insieme e $d : \Omega \times \Omega \longrightarrow \mathbb{R}^{\geq 0}$ **metrica** che rispetta le seguenti proprietà:

- **simmetria**: $d(x, y) = d(y, x)$;
- **identità degli indiscernibili**: $d(x, y) = 0$ se e solo se $x = y$;
- **disuguaglianza triangolare**: $d(x, y) \leq d(x, z) + d(z, y)$, ovvero aggiungere una tappa intermedia al mio percorso non mi può mai far guadagnare della distanza.

Il nostro mondo, ad esempio, non è uno spazio metrico perché non rispetta la simmetria.

Su $\Omega = \mathbb{R}^n$ si usa quasi sempre la **metrica euclidea**, ovvero

$$d(\overline{x}, \overline{y}) = \sqrt{\sum_i (x_i - y_i)^2}.$$

Un'altra metrica molto famosa è quella **Manhattan**, la *«metrica dei taxi»*, ovvero mi è permesso spostarmi come a Manhattan, cioè in orizzontale e verticale, a gradino. La metrica è tale che

$$d(\overline{x}, \overline{y}) = \sum_i |x_i - y_i|$$

e, oltre ad essere uno spazio metrico, è una **ultra metrica**, ovvero $d(x, y) \leq \max\{d(x, z), d(y, z)\}$.

Fissato (Ω, d) spazio metrico, definiamo:

- **input**:
  - insieme $S \subseteq \Omega$ di n punti in uno spazio metrico;
  - budget $k > 0$ che indica quanti magazzini voglio costruire;
- **soluzione accettabile**: insieme $C \subseteq S$ di centri, con $|C| \leq k$. Definisco:
  - $\text{dist}(x, A) = \min_{y \in A} d(x, y)$ minima distanza tra x e gli elementi di A;
  - $\forall s \in S \quad \delta_C(s) = \text{dist}(s, C)$ raggio di copertura del nodo s;
  - $\rho_C = \max_{s \in S} \delta_C(s)$ il massimo raggio di copertura;
- **obiettivo**: ρ_C;
- **tipo**: min.

Vogliamo minimizzare il massimo raggio di copertura. Questo problema è un problema *NPO-C*.

Costruiamo una prima soluzione facendo finta di sapere un dato che però dopo non sapremo.

Center Selection Plus v1

input:
- $S \subseteq \Omega \mid |S| = n > 0$
- $k > 0$
- $r \in \mathbb{R}^{>0}$

1: $C \leftarrow \varnothing$ insieme dei centri scelti
2: while $S \neq \varnothing$:
3: $\quad \overline{s} \leftarrow$ take-any-from(S)
4: $\quad C \leftarrow C \cup \{\overline{s}\}$
5: $\quad$ for s in S tali che $d(s, \overline{s}) \leq 2r$:

Center Selection Plus v1

```
    ⌊ ⌊ S ← S/{s}
  if |C| ≤ k
    ⌊ output C
  else
    ⌊ output impossibile
```

Teorema 3.1: Se Center Selection Plus emette un output allora esso è una

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

approssimazione.

Dimostrazione 3.1: Se arriviamo alla fine dell'algoritmo vuol dire che S è stato svuotato, perché ogni $s \in S$ è stato cancellato da un certo $\overline{s}$ per la sua distanza. Sappiamo che

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

ma allora

$$\frac{\rho_C}{\rho^*} \leq \frac{2r}{\rho^*}.$$

■

Teorema 3.2: Se $r \geq \rho^*$ l'algoritmo emette un output.

Dimostrazione 3.2: Sia C^* una soluzione ottima, cioè una serie di centri che ha come raggio di copertura ρ^*. Sia $\overline{s} \in C$, chiamiamo $\overline{c}^* \in C^*$ il centro al quale si riferisce $\overline{s}$ nella soluzione ottima.

Sia X l'insieme dei punti che nella soluzione ottima C^* si rivolgono a $\overline{c}^*$, allora $\forall s \in X$ posso dire che

$$d(s, \overline{s}) \leq d(s, \overline{c}^*) + d(\overline{c}^*, \overline{s})$$

ma le due distanze sono più piccole del raggio di copertura ottimo perché si riferiscono a lui, quindi

$$d(s, \overline{s}) \leq 2\rho^* \leq 2r$$

ma allora verrebbero tutti cancellati da X quando seleziono s, come abbiamo scritto nell'algoritmo sopra.

Dopo $\leq k$ passi ho cancellato tutto, quindi ho un output. ■

Questi teoremi ci dicono che l'output è approssimato in base a r, che poi mi dà anche una cosa in più per capire se ho soluzioni. Ma che valori possiamo scegliere per r? Andiamo a vedere:

- se $r = \rho^*$ allora ho una 2-approssimazione;
- se $r > \rho^*$ allora l'approssimazione diventa peggiore, ma abbiamo comunque un output;
- se $r < \frac{1}{2}\rho^*$ sto approssimando con $\frac{2r}{\rho^*} < \frac{\rho^*}{\rho^*} = 1$ che però è impossibile:
- se $\frac{1}{2}\rho^* \leq r < \rho^*$ è random.

Vorremmo sempre avere ρ^* ma non ce l'abbiamo.

Vediamo una versione di Greedy Center Selection che sarà utile per dimostrare alcune proprietà.

Center Selection Plus v2

input:
- $S \subseteq \Omega \mid |S| = n > 0$
- $k > 0$
- $r \in \mathbb{R}^{>0}$

```
C ← {s̄} per un certo s̄ ∈ S
while true:
    if ∃s̄ ∈ S | d(s̄, C) > 2r:
        C ← C ∪ {s̄}
    else
        break
if |C| ≤ k
    output C
else
    output impossibile
```

Vediamo finalmente l'algoritmo che andremo poi a studiare, ovvero quello senza le indicazioni sul valore di r che tanto vorremmo.

Greedy Center Selection

input:
- $S \subseteq \Omega \mid |S| = n > 0$
- $k > 0$

```
if |S| ≤ k:
    output S
C ← {s̄} per un certo s̄ ∈ S
while |C| < k:
    seleziona s̄ ∈ S che massimizzi d(s̄, C)
    C ← C ∪ {s̄}
output C
```

Con questo algoritmo scelgo esattamente k centri, il primo scelto a caso, gli altri $k-1$ il più lontano possibile da quelli che ho già scelto.

Teorema 3.3: Greedy Center Selection è una 2-approssimazione per Center Selection.

Dimostrazione 3.3: Ci servirà Center Selection Plus v2 per dimostrare questo teorema.

Per assurdo supponiamo che l'algoritmo Greedy Center Selection emetta una soluzione con $\rho > 2\rho^*$. Questo vuol dire che esiste un certo elemento $\hat{s} \in S$ tale che $d(\hat{s}, C) > 2\rho^*$.

Sia $\overline{s}_i$ l'i-esimo centro aggiunto a C e sia $\overline{C}_i$ l'insieme dei centri in quel momento; possiamo dire che

$$\underbrace{d\left(\overline{s}_i, \overline{C}_i\right) \geq d\left(\hat{s}, \overline{C}_i\right)}_{\text{prendo massima distanza}} \geq d(\hat{s}, C) > 2\rho^*.$$

Questa esecuzione è una delle esecuzioni possibili di Center Selection Plus v2 quando $r = \rho^*$. Noi sappiamo che questo algoritmo produce un output corretto, quindi termina entro k iterazioni quando non ci sono più elementi a distanza maggiore di $2\rho^*$, quindi tutti gli $s \in S$ sono tali che $d(s, C) \leq 2\rho^*$ ma questo non è vero, perché esiste $\hat{s}$ tale che $d(\hat{s}, C) > 2\rho^*$.

Questo è un assurdo, quindi la soluzione è tale che $\rho \leq 2\rho^*$ e quindi

$$\frac{\rho}{\rho^*} \leq 2.$$

■

Questo algoritmo è un esempio di **inapprossimabilità** o di **algoritmo tight**, ovvero non esistono algoritmi che approssimano Center Selection in un modo migliore di una 2-approssimazione.

Teorema 3.4: Se $P \neq NP$ non esiste un algoritmo polinomiale che α-approssimi Center Selection per qualche $\alpha < 2$.

Dimostrazione 3.4: Usiamo il problema NP-C Dominating Set:

- **input**:
  - grafo $G = (V, E)$;
  - $k > 0$;
- **output**: ci chiediamo se $\exists D \subseteq V \mid |D| \leq k$ tale che

$$\forall x \in V/\{D\} \quad \exists d \in D \mid xd \in E.$$

Questo problema deve scegliere dove mettere delle *guardie* in un grafo, ovvero dei nodi che coprono i nodi a loro adiacenti, in modo che tutti i nodi siamo coperti.

Dati G e k input di Dominating Set dobbiamo costruire una istanza di Center Selection.

Partiamo con il definire lo spazio metrico (Ω, d) con:

- insieme $\Omega = S = V$;
- funzione distanza

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

Dimostriamo che è uno spazio metrico:

- simmetria: banale;
- identità degli indiscernibili: banale;
- disuguaglianza triangolare: notiamo che in questa disuguaglianza

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

  abbiamo il membro di sinistra che può assumere valori in $\{1,2\}$, stessa cosa per i due addendi del membro di destra, ma allora il membro di destra ha come valori possibili $\{2,3,4\}$, quindi vale la disuguaglianza triangolare.

Come budget prendiamo esattamente k numero di guardie.

Ho creato il mio input per Center Selection. Chiediamoci quanto vale $\rho^*(S,k)$, ma questa assume solo valori in $\{1,2\}$ per come ho definito la distanza. Vediamo i due casi distinti:

- distanza 1: tutte le distanze sono 1, essendo questa il massimo tra tutte le distanze (*e non posso averla a zero*), ma allora vuol dire che tutti i vertici non scelti sono a distanza 1 dai centri scelti, e quindi sono *«coperti»* dalle guardie, quindi abbiamo un Dominating Set;
- distanza 2: non abbiamo un Dominating Set, avendo distanza 2 un vertice non viene coperto dai vertici che abbiamo selezionato.

Noi abbiamo un algoritmo α-approssimante per Center Selection con $\alpha < 2$, che fa

$$(S,k) \rightsquigarrow \text{ALGORITMO} \rightsquigarrow \rho^*(S,k) \leq \underbrace{\rho(S,k)}_{\text{risultato}} \leq \alpha\rho^*(S,k).$$

Sappiamo che la distanza migliore è 1 o 2, ma allora:

- se $\rho^* = 1$ ottengo $1 \leq \rho(S,k) \leq \alpha$;
- se $\rho^* = 2$ ottengo $2 \leq \rho(S,k) \leq 2\alpha$.

Nel primo caso devo rispondere *SI* al problema di Dominating Set, nel secondo caso devo rispondere *NO*. I due insiemi sono ovviamente disgiunti.

Ma questo è un assurdo: avrei un algoritmo polinomiale per Dominating Set, quindi deve valere per forza $\alpha \geq 2$ ■

4. Set Cover

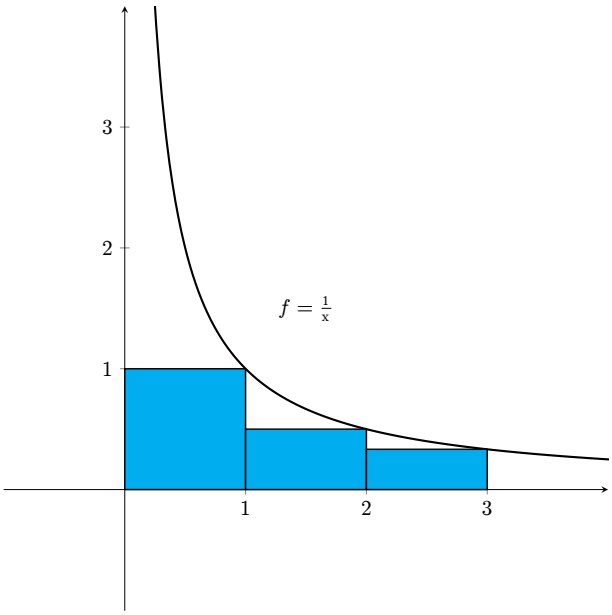
4.1. Funzione armonica

Diamo prima una definizione di **funzione armonica**: essa è la funzione

$$H : \mathbb{N}^{>0} \longrightarrow \mathbb{R}$$

tale che

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



Vediamo due proprietà importanti di questa funzione.

La prima proprietà ci dà un **upper bound** alla funzione armonica, ovvero

$$H(n) \leq 1 + \int_1^n \frac{1}{x}\,\mathrm{d}x = 1 + [\ln(x)]_1^n = 1 + \ln(n) - \cancel{\ln(1)} = 1 + \ln(n).$$

La seconda proprietà afferma che

$$\int_t^{t+1} \frac{1}{x}\,\mathrm{d}x \leq 1 \cdot \frac{1}{t} = \frac{1}{t}.$$

Stiamo dando un altro upper bound: visto che $\frac{1}{x}$ è decrescente possiamo stimare la sua area tra t e $t+1$ usando un rettangolo con altezza uguale all'altezza della funzione calcolata in t.

Questa proprietà ci dà un **lower bound** alla funzione armonica, ovvero

$$H(n) = \frac{1}{1} + ... + \frac{1}{n} \geq \int_1^2 \frac{1}{x}\,\mathrm{d}x + ... + \int_n^{n+1} \frac{1}{x}\,\mathrm{d}x = \int_1^{n+1} \frac{1}{x}\,\mathrm{d}x = \ln(n+1).$$

Grazie a queste proprietà abbiamo trovato dei **bound** per la funzione armonica:

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

4.2. Algoritmi

Il problema di (*Min*) **Set Cover** ci chiede di coprire tutte le zone di una città con i mezzi di trasporto spendendo il meno possibile scegliendo tra una serie di offerte che coprono un certo numero di aree.

Vediamo la definizione di questo problema:

- **input**:
  - m insiemi $S_0, ..., S_{m-1}$ non per forza disgiunti tali che $U = \bigcup_{i \in m} S_i$; questo insieme viene detto **insieme universo** (*tutte le zone della città*);
  - m costi $w_0, ..., w_{m-1}$ associati ad ogni insieme S_i (*costo delle offerte*);
- **soluzioni ammissibili**: $I \subseteq m$ tale che $\bigcup_{i \in I} S_i = U$ (*scelgo offerte che mi coprono tutta la città*);
- **obiettivo**: $\sum_{i \in I} w_i$;
- **tipo**: min.

Questo problema è *NPO-C*. La motivazione principale è la presenza di due tendenze contrastanti:

- scelgo insiemi grandi per coprire il più possibile subito, ma vado a spendere troppo;
- scelgo insiemi piccoli che mi vanno a costare poco, ma potrei prenderne così tanto da superare la soluzione precedente.

Costruiremo la nostra soluzione aggiungendo mano a mano ad un insieme le offerte scelte. La scelta di un'offerta la andiamo a fare guardando il numero di elementi nuovi che andrebbe a coprire: per avere una buona metrica guarderemo il rapporto tra quanto paghiamo e quanti elementi nuovi inseriamo, e lo andremo a minimizzare.

Greedy Set Cover

1: $R \leftarrow U$
2: $I \leftarrow \varnothing$
3: while $R \neq \varnothing$:
4: Scegli S_i che minimizzi $\frac{w_i}{|S_i \cap R|}$
5: $I \leftarrow I \cup \{i\}$
6: $R \leftarrow R / S_i$
7: **output** I

Il valore

$$\frac{w_i}{|S_i \cap R|}$$

è un **costo** che associamo ad ogni insieme: infatti, questo ci dà una indicazione di quanto paghiamo nello scegliere quell'insieme considerando sia il suo costo sia quanti nuovi elementi va a coprire. Indichiamo con $c(s)$ questa quantità $\forall s \in S_i \cap R$.

Lemma 4.2.1: Alla fine di Greedy Set Cover abbiamo

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

Dimostrazione 4.2.1: Il costo totale delle offerte scelte è $w = \sum_{i \in I} w_i$, ma w_i è la somma di tutti i costi $c(s)$ associati ad ogni elemento di $s \in S_i \cap R$, quindi

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

■

Lemma 4.2.2: Per ogni k vale

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

Dimostrazione 4.2.2: Assumo $S_k = \{s_1, ..., s_d\}$ enumerato in ordine di copertura.

Consideriamo l'iterazione che copre s_j usando un certo S_h, cosa possiamo dire di R?

Sicuramente $\{s_j, s_{j+1}, ..., s_d\} \subseteq R$ ma allora $|S_k \cap R| \geq d - j + 1$ e quindi

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

Andiamo a valutare

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

$$= w_k \left(\frac{1}{1} + ... + \frac{1}{d} \right) = w_k H(|S_k|).$$

■

Teorema 4.2.1: Sia $M = \max_i |S_i|$, allora Greedy Set Cover è una $H(M)$-approssimazione per Set Cover.

Dimostrazione 4.2.3: Sia I^* una soluzione ottima, ovvero un insieme di indici tali che

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

Per il lemma 2 vale

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

perché $H(M) \geq H(|S_i|)$ vista la monotonia di H.

Possiamo scrivere anche che

$$\underbrace{\sum_{i \in I^*} \sum_{s \in S_i} c(s)}_{\text{più di una volta}} \geq \sum_{s \in U} c(s) = w.$$

Uniamo i due risultati e otteniamo

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

Ma allora

$$\frac{w}{w^*} \leq H(M).$$ ■

L'approssimazione che ho non è esatta, ma dipende dall'input.

Notiamo che $|M| \leq |U| = n$, ma $H(M) \leq H(n)$ per monotonia di H e quindi

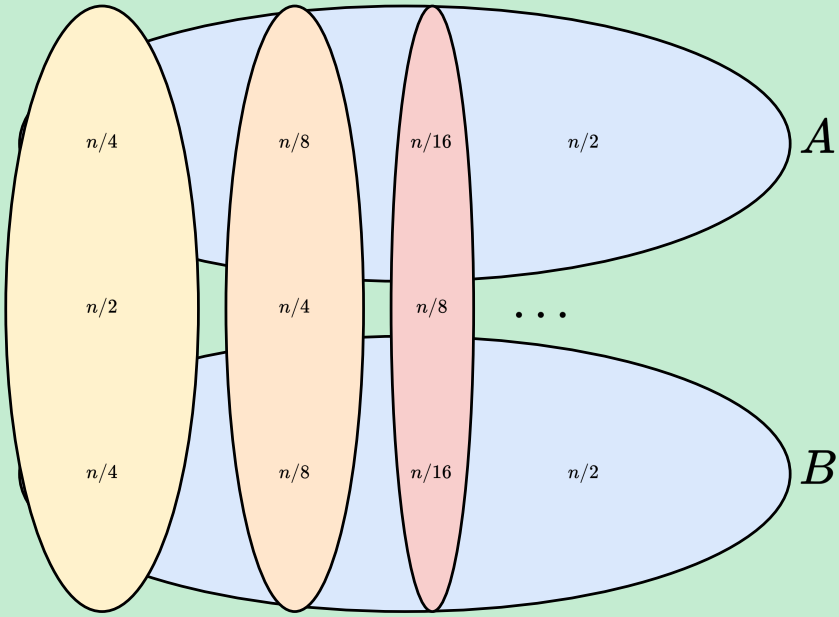
$$H(M) \leq H(n) = O(\log(n)).$$

Corollario 4.2.1.1: Greedy Set Cover è una $O(\log(n))$-approssimazione per Set Cover.

Questo qui non è un **algoritmo tight**: infatti, riusciremo a creare un input ad hoc che si avvicina al tasso di approssimazione a meno di un errore molto piccolo.

Esempio 4.2.1: Il nostro universo è formato da:

- 2 insiemi A, B di costo $1 + \varepsilon$ con $\frac{n}{2}$ punti;
- 1 insieme *giallo* di costo 1 con $\frac{n}{2}$ punti, $\frac{n}{4}$ punti da A e $\frac{n}{4}$ punti da B;
- 1 insieme *arancio* di costo 1 con $\frac{n}{4}$ punti, $\frac{n}{8}$ punti da A e $\frac{n}{8}$ punti da B;
- 1 insieme *rosso* di costo 1 con $\frac{n}{8}$ punti, $\frac{n}{16}$ punti da A e $\frac{n}{16}$ punti da B;
- eccetera.



Simuliamo l'algoritmo Greedy Set Cover su questo particolare input.

Prima iterazione:

- gli insiemi A, B hanno costo $\frac{2+2\varepsilon}{n}$;
- gli insiemi che pescano da A, B hanno costo $\frac{2}{n}$, $\frac{4}{n}$, $\frac{8}{n}$, eccetera.

L'algoritmo seleziona quindi l'insieme giallo con costo $\frac{2}{n}$. Rimangono da coprire $\frac{n}{2}$ elementi.

Seconda iterazione:

- gli insiemi A, B hanno costo $\frac{4+4\varepsilon}{n}$;
- gli insiemi che pescano da A, B hanno costo $\frac{4}{n}$, $\frac{8}{n}$, eccetera.

L'algoritmo seleziona quindi l'insieme arancio con costo $\frac{4}{n}$. Rimangono da coprire $\frac{n}{4}$ elementi.

E così via fino a quando non copro totalmente $A \cup B$ selezionando solo gli insiemi «trasversali». Ogni insieme costa 1 e gli insiemi sono $\log(n)$, quindi

$$w = \log(n).$$

Il costo ottimo si ottiene scegliendo gli insiemi A e B, quindi

$$w^* = 2 + 2\varepsilon.$$

L'approssimazione ottenuta è

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

Teorema 4.2.2: Se $P \neq NP$ non esiste un algoritmo che approssimi Set Cover meglio di

$$(1 - o(1)) \log(n).$$

5. Vertex Cover

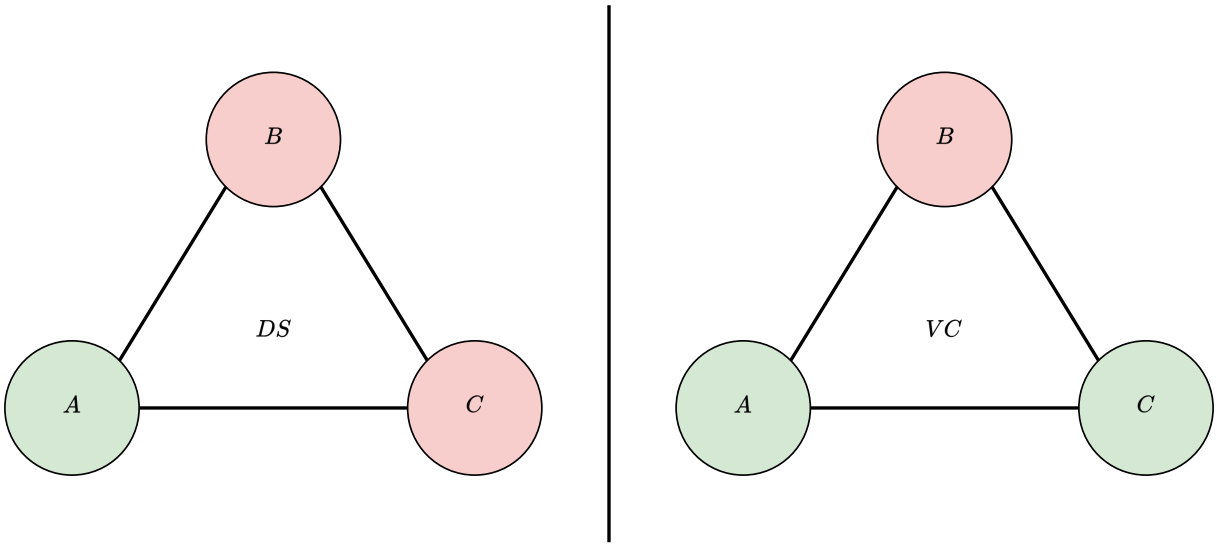
Il problema di **Vertex Cover** assomiglia spaventosamente a Dominating Set, più problemi per noi durante l'esame.

Vediamo come è definito:

- **input**:
  - grafo non orientato $G = (V, E)$;
  - pesi $w_i \in \mathbb{R}^{>0} \quad \forall i \in V$;
- **soluzione ammissibile**: insieme di vertici $X \subseteq V$ tale che $\forall e \in E$ allora $e \cap X \neq \emptyset$, ovvero ogni lato deve avere almeno un vertice a lui incidente nell'insieme X;
- **funzione obiettivo**: $\sum_{i \in X} w_i$;
- **tipo**: min.

Sembra il problema di Dominating Set ma ci sono differenze:

- in Dominating Set andiamo a scegliere una serie di vertici X, dette guardie, e tutti i *vertici* non scelti hanno almeno un arco che va in una guardia
- in Vertex Cover andiamo a scegliere una serie di vertici X e tutti i *lati* hanno almeno un vertice nell'insieme scelto.



Se abbiamo un Vertex Cover allora abbiamo anche un Dominating Set.

Questo problema è ovviamente *NPO-C*. Noi vedremo un algoritmo basato sul pricing.

I **problemi basati sul pricing** sono problemi economici dove ho:

- agenti che pagano una certa quota per avere un certo servizio;
- agenti che decidono se entrare o meno nel gioco accettando o meno le offerte degli altri agenti.

I primi agenti sono i *lati*, che pagano una certa quota p_e per farsi coprire da un certo vertice, mentre i secondi sono i vertici, che vedono se entrare o meno nel gioco in base a quanto vengono pagati.

Associamo ad ogni lato un prezzo che pagherebbero ai vertici incidenti per farsi coprire da loro.

Il **pricing** è un insieme di prezzi

$$(p_e)_{e \in E}.$$

Un pricing è **equo** se e solo se

$$\forall i \in V \quad \sum_{e \in E \wedge i \in e} p_e \leq w_i.$$

In poche parole, i lati che incidono sul vertice i devono offrire al massimo quello che chiede il vertice. È una mafia, perché dovrei dare di più? Noi useremo solo pricing equi. Il pricing equo più ovvio è quello che vale 0 su tutti i lati.

Lemma 5.1: Se $(p_e)_{e \in E}$ è un pricing equo allora

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

Dimostrazione 5.1: Sappiamo che $X^* \subseteq V$ è una soluzione ottima e

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

Per definizione di equità vale

$$\sum_{e \in E \wedge i \in e} p_e \leq w_i.$$

Sommiamo su tutti gli $i \in X^*$ quindi

$$\sum_{e \in E} p_e \leq \sum_{i \in X^*} \sum_{e \in E \wedge i \in e} p_e \leq \sum_{i \in X^*} w_i = w^*.$$

■

Data un pricing, esso è **stretto** sul vertice i se e solo se

$$\sum_{e \in E \wedge i \in e} p_e < w_i.$$

È una proprietà che vale su un vertice, non so tutto il grafo. Questa proprietà afferma che il vertice i non è contento di quello che riceve, non riceve quello che vuole.

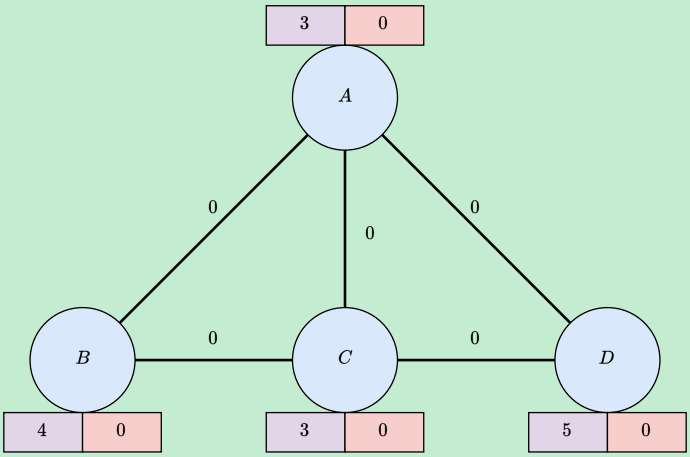
Pricing Vertex Cover

input
- grafo $G = (V, E)$ non orientato
- costi $w_i \in \mathbb{R}^{>0} \quad \forall i \in V$

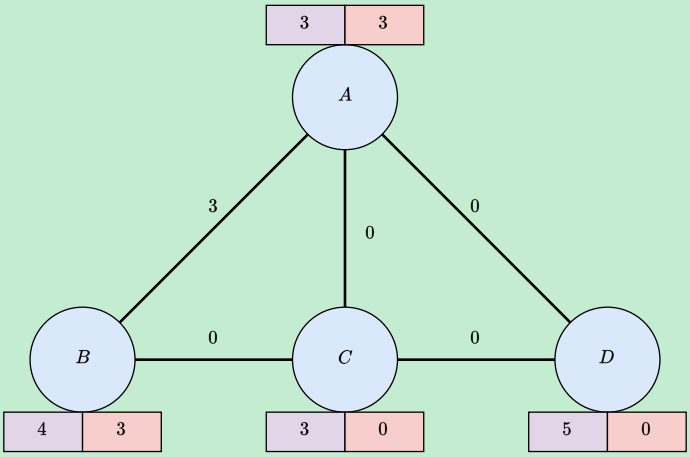
1: $p_e \leftarrow 0 \quad \forall e \in E$
2: while $\exists e = (i, j)$ tale che $(p_e)_{e \in E}$ è stretto su i e su j:
3: sia $\overline{e}$ il lato che minimizza $\Delta = \min\left(\left(w_i - \sum_{e \in E \wedge i \in e} p_e\right), \left(w_j - \sum_{e \in E \wedge j \in e} p_e\right)\right)$
4: $p_{\overline{e}} \leftarrow p_{\overline{e}} + \Delta$, ovvero alzo l'offerta del minimo indispensabile
5: **output** $\{i \in V \mid p_e$ non è stretto su $i\}$ insieme dei vertici contenti

Esempio 5.1: Nelle seguenti figure abbiamo il peso di un vertice w_i indicato in violetto e quanto il vertice $i \in V$ viene pagato dai lati p_i indicato in rosso.

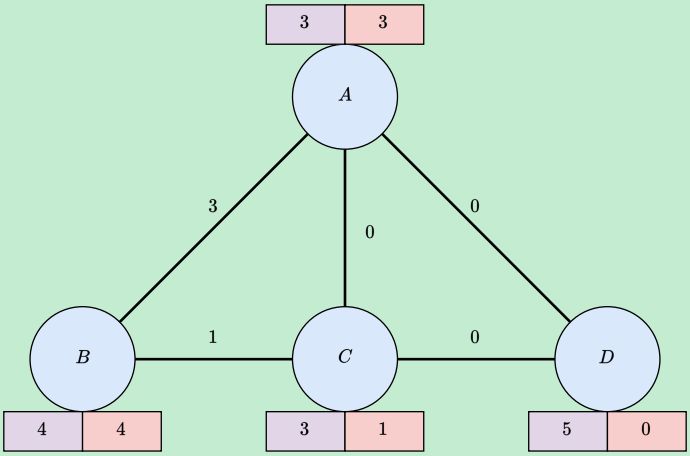
All'inizio dell'algoritmo siamo nella seguente configurazione, con il pricing banale.



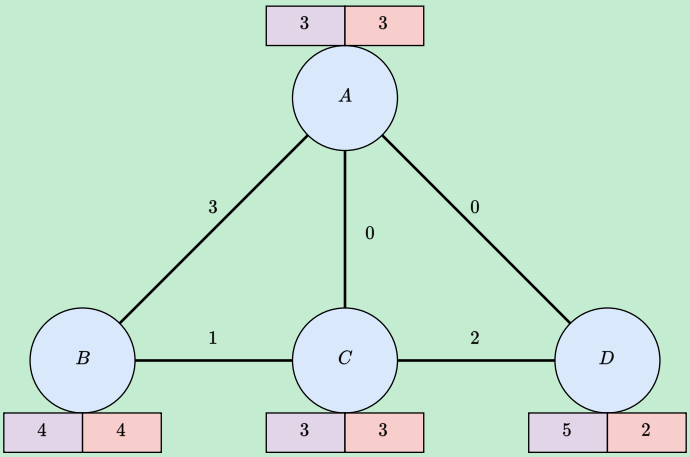
Alla prima iterazione del while tutti i lati hanno entrambi i vertici con pricing stretto e tutti i lati verrebbero aggiornati con il valore 3. Scegliamo di modificare il lato AB.



Alla seconda iterazione del while i lati BC e CD hanno entrambi i vertici con pricing stretto, il lato BC verrebbe aggiornato con il valore 1. Scegliamo di modificare il lato BC.



Alla terza iterazione del while il lato CD ha entrambi i vertici con pricing stretto e verrebbe aggiornato con il valore 2. Scegliamo di modificare il lato CD.



Non eseguiamo più nessuna iterazione del while perché ogni lato ha almeno un vertice con pricing non stretto.

Selezioniamo quindi i vertici A,B e C che sono gli unici che hanno il pricing non stretto.

Perché siamo sicuri che questo algoritmo sia uno di approssimazione? Perché i lati non parlano tra di loro, non si mettono d'accordo su cosa fare.

L'algoritmo che abbiamo visto non vuole avere lati con entrambi i vertici incidenti stretti: infatti, se così fosse quel lato non verrebbe coperto perché i vertici non sono invogliati ad entrare nell'affare.

Lemma 5.2: Alla fine di Pricing Vertex Cover abbiamo

$$w \leq 2 \sum_{e \in E} p_e.$$

Dimostrazione 5.2: Noi paghiamo

$$w = \sum_{i \in V_{\text{out}}} w_i$$

ma il costo w_i è la somma di tutte le offerte dei lati incidenti, quindi

$$w = \sum_{i \in V_{\text{out}}} \sum_{e \in E \wedge i \in e} p_e.$$

Stiamo sommando per ogni vertice nell'output i lati a che incidono su di esso, ma questi compaiono al massimo due volte, ovvero se entrambi i vertici di un lato stanno nell'output, quindi

$$w \leq 2 \sum_{e \in E} p_e.$$

■

Teorema 5.1: Pricing Vertex Cover è una 2-approssimazione per Vertex Cover.

Dimostrazione 5.3: Vediamo

$$\frac{w}{w^*} \underset{l2}{\leq} \frac{2 \sum_{e \in E} p_e}{w^*} \underset{l1}{\leq} \frac{2 \sum_{e \in E} p_e}{\sum_{e \in E} p_e} = 2.$$

■

Non sappiamo molto di più: non sappiamo se possiamo andare meglio di 2, ma sappiamo che per questo problema non possiamo scrivere un PTAS.

6. Disjoint Paths

Idea del problema: abbiamo grafo orientato con k sorgenti $s_0, ..., s_{k-1}$ e altrettante destinazioni $t_0, ..., t_{k-1}$. Cerchiamo di creare il maggior numero di cammini $s_i \to t_i$ passando per i lati al massimo una volta. In realtà noi useremo un parametro di congestione c che ci permette di passare al massimo c volte per un lato. Questo problema è anche detto problema dei **cammini disgiunti**.

Diamo una definizione più formale di questo problema:

- **input**:
  - grafo $G = (N, A)$ orientato;
  - lista $s_0, ..., s_{k-1} \in N$ di sorgenti;
  - lista $t_0, ..., t_{k-1} \in N$ di destinazioni;
  - parametro di congestione $c \in \mathbb{N}^+$;
- **soluzione ammissibile**: insieme $I \subseteq k$ tale che i cammini $\pi_i : s_i \to t_i$ non usano archi di G più di c volte; in altre parole, ogni arco è usato al massimo da c cammini;
- **obiettivo**: $|I|$;
- **tipo**: max.

Usiamo anche una funzione di lunghezza

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

che verrà modificata nel tempo e ci permetterà di implementare il prossimo algoritmo. Con questa funzione definiamo la **lunghezza di un cammino** come la quantità

$$l(\pi = \langle x_1, ..., x_i \rangle) = l(x_1, x_2) + ... + l(x_{i-1}, x_i).$$

Greedy Paths

---

input:

grafo $G = (N, A)$ orientato
lista $s_0, ..., s_{k-1} \in N$ di sorgenti
lista $t_0, ..., t_{k-1} \in N$ di destinazioni
parametro di congestione $c \in \mathbb{N}^+$
moltiplicatore $\beta > 1$

$I \leftarrow \varnothing$ (*coppie già collegate*)
$P \leftarrow \varnothing$ (*cammini già fatti*)
for $a \in A$:
  $l(a) \leftarrow 1$
forever and ever:
  trova il cammino più corto $\pi_i : s_i \to t_i$ con $i \notin I$ (*coppia ancora non collegata*)
  if $\nexists \pi_i$:
    break
  $I \leftarrow I \cup \{i\}$
  $P \leftarrow P \cup \{\pi_i\}$
  $\forall a \in \pi_i$:
    $l(a) \leftarrow \beta \cdot l(a)$
    if $l(a) = \beta^c$:
      rimuovi a dal grafo

Greedy Paths

15: **output** I e P

Definizione 6.1 (*Cammino corto*): In un certo istante di esecuzione, un cammino π è **corto** se e solo se

$$l(\pi) < \beta^c.$$

I cammini si possono solo allungare, quindi il passaggio di *«stato»* è da *corto* a *lungo*.

Definizione 6.2 (*Cammino utile*): In un certo istante di esecuzione, un cammino π è **utile** se collega un coppia nuova $i \notin I$.

L'algoritmo considera solo cammini utili e, inoltre, ogni volta il più corto. All'inizio consideriamo cammini **corti e utili**, poi cammini **lunghi e utili**, poi ci fermiamo.

Noi studieremo l'algoritmo quando finiamo i cammini corti. Sia $\bar{l}$ la funzione lunghezza in quel momento e siano $\overline{I}$ e $\overline{P}$ le coppie già collegate con il loro percorso associato sempre in quel momento. Nella fase di output abbiamo l_o, I_o e P_o.

Lemma 6.1: Sia $i \in I^*/I_o$, allora

$$\bar{l}(\pi_i^*) \geq \beta^c.$$

Dimostrazione 6.1: Per assurdo sia

$$\bar{l}(\pi_i^*) < \beta^c.$$

Questo non è possibile: nel grafo avremmo un cammino corto che potremmo selezionare, ma noi abbiamo appena esaurito i cammini corti. ■

Lemma 6.2: Sia $m = |A|$, allora

$$\sum_{a \in A} \bar{l}(a) \leq \beta^{c+1} \left|\overline{I}\right| + m.$$

Dimostrazione 6.2: Dimostriamolo per induzione.

All'inizio dell'algoritmo abbiamo

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

Supponiamo ora che la lunghezza l_1 passi alla lunghezza l_2 scegliendo la coppia i-esima con il cammino π_i. Supponiamo inoltre di non essere ancora arrivati a $\overline{l}$.

Possiamo dire che

$$l_2(a) = \begin{cases} l_1(a) & \text{se } a \notin \pi_i \\ \beta l_1(a) & \text{se } a \in \pi_i \end{cases}.$$

Valutiamo la differenza

$$\begin{aligned}\sum_{a\in A} l_2(a) - \sum_{a\in A} l_1(a) &= \sum_{a\notin\pi_i} (l_2(a) - l_1(a)) + \sum_{a\in\pi_i} (l_2(a) - l_1(a)) = \\ &= \sum_{a\in\pi_i} (l_2(a) - l_1(a)) = \sum_{a\in\pi_i} (\beta l_1(a) - l_1(a)) = \\ &= (\beta - 1)\sum_{a\in\pi_i} l_1(a) = (\beta-1) l_1(\pi_i) < (\beta-1)\beta^c \\ &\leq \beta^{c+1}\end{aligned}$$

Ad ogni passo aggiungiamo un cammino, quindi la funzione lunghezza aumenta ogni volta di β^{c+1}. Noi facciamo $\left|\overline{I}\right|$ aggiunte prima di arrivare in $\overline{I}$, quindi aggiungiamo $\beta^{c+1}\left|\overline{I}\right|$, al quale aggiungiamo il valore iniziale m della lunghezza *«spostandolo a destra»*. ■

Facciamo un paio di osservazioni importanti per calcolare il γ dell'approssimazione.

Osserviamo che

$$\sum_{i\in I^*/I_o} \bar{l}(\pi_i^*) \geq \beta^c |I^*/I_o|$$

sommando su (I/I_o) in entrambi i membri della disequazione. Osserviamo inoltre che nella soluzione ottima nessun arco è usato più di c volte. Quindi possiamo maggiorare il costo con il costo di tutti gli archi moltiplicati per c, ovvero

$$\sum_{i\in I^*/I_o} \bar{l}(\pi_i^*) \leq c \sum_{a\in A} \bar{l}(a) \underset{l2}{\leq} c\left(\beta^{c+1}\left|\overline{I}\right| + m\right).$$

Grazie a queste due osservazioni possiamo trovare il γ che approssima questo algoritmo.

Teorema 6.1: Greedy Path fornisce una $\left(2cm^{\frac{1}{c+1}} + 1\right)$-approssimazione per Disjoint Paths.

Dimostrazione 6.3: Osserviamo che

$$\begin{aligned}\beta^c|I^*| &\leq \beta^c|I^*/I_o| + \beta^c|I^* \cap I_o| \\ &\underset{o1}{\leq} \sum_{i \in I^*/I_o} \bar{l}(\pi_i^*) + \beta^c|I_o| \\ &\underset{o2}{\leq} c\Big(\beta^{c+1}|\bar{I}| + m\Big) + \beta^c|I_o| \\ &\leq c(\beta^{c+1}|I_o| + m) + \beta^c|I_o|.\end{aligned}$$

Dividiamo tutto per β^c e otteniamo

$$|I^*| \leq c\beta|I_o| + \frac{cm}{\beta^c} + |I_o| \leq c\beta|I_o| + |I_o|\frac{cm}{\beta^c} + |I_o| = |I_o|\left(c\beta + \frac{cm}{\beta^c} + 1\right).$$

Ma allora

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

Vorremmo scegliere un β che minimizzi questa funzione: la soluzione gentilmente calcolata è

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

Con questo valore otteniamo

$$\frac{|I^*|}{|I_o|} \leq c\Big(m^{\frac{1}{c+1}} + m^{1-\frac{c}{c+1}}\Big) = c\Big(m^{\frac{1}{c+1}} + m^{\frac{1}{c+1}}\Big) + 1 = 2cm^{\frac{1}{c+1}} + 1. \qquad \blacksquare$$

Questa funzione è particolare: ha un fattore lineare moltiplicativo in c che peggiora l'approssimazione, ma al tempo stesso si alza anche l'indice della radice. Infatti:

- se $c = 1$ approssimo con $2\sqrt{m} + 1$;
- se $c = 2$ approssimo con $4\sqrt[3]{m} + 1$;
- se $c = 3$ approssimo con $6\sqrt[4]{m} + 1$;
- eccetera.

7. Vertex Cover, il ritorno

7.1. Programmazione lineare e lineare intera

Introduciamo la **programmazione lineare** (LP). Essa è un problema di ottimizzazione definito da:

- **input**:
  - $A \in \mathbb{Q}^{m \times n}$ matrice di coefficienti;
  - $b \in \mathbb{Q}^m$ vettore di termini noti;
  - $c \in \mathbb{Q}^n$ vettore di pesi;
- **soluzioni ammissibili**: $x \in \mathbb{Q}^n$ tali che $Ax \geq b$;
- **funzione obiettivo**: $c^T x$;
- **tipo**: min (*in realtà non cambia niente*).

Ogni riga della matrice Ax rappresenta un vincolo di disuguaglianza con b. Sia i vincoli sia la funzione obiettivo sono delle **funzioni lineari**.

Un po' di storia: negli anni '50 escono le prime tecniche di risoluzione, la più popolare è l'**algoritmo del simplesso**, molto efficiente nella realtà ma esponenziale nella teoria. Per 30 anni circa la questione LP $\in$ *PO* è rimasta aperta, quando nel 1984 si è avuta la conferma di quella relazione grazie all'**algoritmo di Karmarkan**.

Vediamo una versione leggermente diversa di LP nelle premesse, ma profondamente diversa nei risultati. Introduciamo la **programmazione lineare intera** (ILP), un problema definito da:

- **input**:
  - $A \in \mathbb{Q}^{m \times n}$ matrice di coefficienti;
  - $b \in \mathbb{Q}^m$ vettore di termini noti;
  - $c \in \mathbb{Q}^n$ vettore di pesi;
- **soluzioni ammissibili**: $x \in \mathbb{Z}^n$ tali che $Ax \geq b$;
- **funzione obiettivo**: $c^T x$;
- **tipo**: min (*in realtà non cambia niente*).

La differenza sembra minima, ma è in realtà enorme: infatti, ILP diventa un problema *NPO-C* solo per l'imposizione di soluzioni intere.

7.2. Algoritmo di arrotondamento

Vediamo una soluzione di **Vertex Cover con arrotondamento** (*rounding*) che utilizza la LP.

Il problema Vertex Cover ha in input un grafo $G = (V, E)$ non orientato con una serie di pesi $w_i \in \mathbb{Q}^{>0} \quad \forall i \in V$. Sia π un'istanza di Vertex Cover. Creo il problema ILP(π) con:

- **variabili**: $x \in \mathbb{Z}^n$;
- **vincoli**:
  - $x_i + x_j \geq 1 \quad \forall \{i, j\} \in E$, ovvero almeno uno dei vertici di ogni lato deve essere nella soluzione;
  - $x_i \geq 0 \quad \forall i \in V$, ovvero impongo la non negatività;
  - $x_i \leq 1 \quad \forall i \in V$, ovvero impongo valori booleani alle variabili;
- **obiettivo**: $\min_x \left(\sum_{i \in V} w_i x_i \right)$.

Consideriamo ora il rilassamento di questo problema, chiamato ILP(π), identico al problema precedente tranne che la variabile ora è $x \in \mathbb{Q}^n$. Il risultato è un vettore che contiene anche dei valori razionali in $\mathbb{Q}$. Come ci comportiamo? Useremo questa soluzione intermedia come input al **rounding**. Come?

Sia $\pi : (G = (V, E), w_i)$, allora

$$\pi \rightsquigarrow \text{ILP}(\pi) \rightsquigarrow \text{LP}(\pi) = x^* \rightsquigarrow \text{rounding}(x^*) = \overline{x} \mid \overline{x}_i = \begin{cases} 0 & \text{se } x_i^* < 0.5 \\ 1 & \text{se } x_i^* \geq 0.5 \end{cases} \quad \forall i \in V.$$

Sia w^*_{ILP} l'ottimo ottenuto nel problema ILP, e sia w^*_{LP} l'ottimo ottenuto nel problema LP rilassato.

Lemma 7.2.1:

$$w^*_{\text{LP}} \leq w^*_{\text{ILP}}.$$

Dimostrazione 7.2.1: Il problema rilassato LP ha un super-insieme di soluzioni ammissibili rispetto al problema originale, quindi l'ottimo può solo essere migliore, o al massimo uguale. ■

Lemma 7.2.2: $\overline{x}$ è una soluzione ammissibile di $\text{ILP}(\pi)$.

Dimostrazione 7.2.2: I vincoli da verificare sono

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

con $x \in \mathbb{Z}^n$. Partiamo dal secondo vincolo: come abbiamo ottenuti i vari x_i? Sappiamo che

$$\overline{x}_i = \begin{cases} 0 & \text{se } x_i^* < 0.5 \\ 1 & \text{se } x_i^* \geq 0.5 \end{cases} \quad \forall i \in V,$$

ma allora il secondo vincolo è verificato, avendo ogni x_i valore nell'insieme $\{0, 1\}$.

Vediamo ora il primo vincolo: l'unico caso nel quale non è rispettato è quando $\overline{x}_i + \overline{x}_j = 0$, ovvero $\overline{x}_i = \overline{x}_j = 0$. Se ai due vertici è assegnato 0 vuol dire che $x_i^* < 0.5$ e $x_j^* < 0.5$, ma questo è assurdo: infatti, nella soluzione ottima in LP vale il vincolo $x_i^* + x_j^* \geq 1$, che non è però soddisfatto dalla somma di due quantità minori di 0.5. ■

Lemma 7.2.3:

$$\forall i \quad \overline{x}_i \leq 2x_i^*.$$

Dimostrazione 7.2.3: Dobbiamo controllare i due possibili valori di $\overline{x}_i$.

Se $\overline{x}_i = 0$ allora

$$0 \leq x_i^* < 0.5 \overset{*2}{\implies} \overline{x}_i = 0 \leq 2x_i^* < 1 \implies \overline{x}_i \leq 2x_i^*.$$

Se $\overline{x}_i = 1$ allora

$$x_i^* \geq 0.5 \overset{*2}{\Longrightarrow} 2x_i^* \geq 1 = \overline{x}_i \Longrightarrow \overline{x}_i \leq 2x_i^*.$$

■

Lemma 7.2.4:

$$w \leq 2w_{\text{LP}}^*.$$

Dimostrazione 7.2.4: Molto facile:

$$w = \sum_i w_i \overline{x}_i \underset{l3}{\leq} 2 \sum_i w_i x_i^* = 2w_{\text{LP}}^*.$$

■

Teorema 7.2.1: L'algoritmo basato su rounding è una 2-approssimazione per Vertex Cover.

Dimostrazione 7.2.5: Dopo quattro estenuanti lemmi possiamo affermare che

$$\frac{w}{w_{\text{ILP}}^*} \underset{1}{\leq} \frac{w}{w_{\text{LP}}^*} \underset{4}{\leq} \frac{2w_{\text{LP}}^*}{w_{\text{LP}}^*} = 2.$$

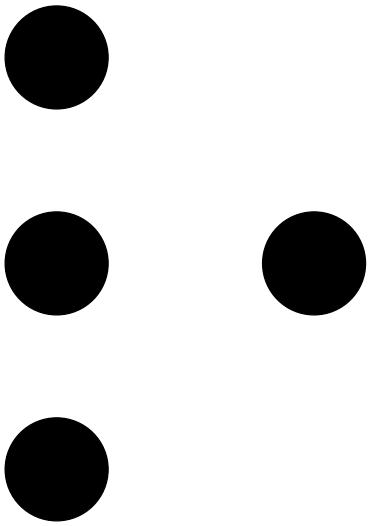
■

Vertex Cover è **tight**, si può dimostrare ma non lo facciamo.

8. Commesso Viaggiatore (TSP)

8.1. Teoria dei grafi

La **teoria dei grafi** nasce con Eulero a fine ′700 con il **problema dei ponti di Konigsberg** (*oggi Kaliningrad*): abbiamo un fiume con due isole, e i collegamenti argine-isole e isole-isole sono definiti da 7 ponti.



Possiamo scegliere un punto di partenza della città che mi permetta di passare per tutti i punti una e una sola volta per poi tornare al punto di partenza? Eulero, un fratello, astrae, quindi il problema si riduce alla ricerca di un **circuito euleriano** in un grafo non orientato.

In realtà, quello che stiamo utilizzando è un **multigrafo**: questo perché esistono due vertici che hanno almeno due lati incidenti.

Teorema 8.1.1 (*di Eulero*): Esiste un circuito euleriano se e solo se il grafo è connesso e tutti i vertici hanno grado pari.

Dimostrazione 8.1.1:

{$\Longleftarrow$ solo questo}

Partiamo da un vertice x_0 e ci spostiamo su un lato che incide su x_0 fino al vertice x_1. Ora, x_1 ha grado pari quindi da x_1 ho almeno un altro lato, che va a x_2. Ora, eccetera.

Cosa può succedere in questa costruzione? Abbiamo due alternative:

- incappiamo in un ciclo con un nodo che abbiamo già visitato, ma non ci sono problemi; infatti, se per esempio da x_3 vado a x_1 devo avere un altro nodo per avere il grado di x_1 pari;
- arriviamo a x_0: abbiamo ancora due casi:
  - abbiamo esaurito i lati;
  - abbiamo cancellato un numero pari di lati da ogni nodo e ripartiamo con il percorso.

Finito, odio il quadratino. ■

Vediamo ora il lemma delle strette di mano: dato un gruppo di persone che si stringono la mano, il numero di persone che stringono la mano ad un numero dispari di persone è pari.

Lemma 8.1.1 (*Handshaking lemma*): In un grafo, il numero di vertici di grado dispari è pari.

Dimostrazione 8.1.2: Calcoliamo

$$S = \sum_{x \in V} d(x).$$

Sicuramente S è un numero pari, questo perché stiamo contando ogni lato due volte, visto che ogni lato incide su due vertici. Visto che la somma di numeri pari è ancora un numero pari, togliamo da S tutti gli addendi pari, rimanendo solo con i dispari.

Ma allora sto sommando tutti gli $x \in V$ tali che $d(x)$ è dispari. ■

8.2. Algoritmi

Il problema del **commesso viaggiatore** (*travelling salesman problem*) è il seguente: abbiamo un insieme di città collegate da strade di una certa lunghezza e il commesso vuole partire da un punto (*casa sua*), passare per tutte le città una sola volta, tornare al punto di partenza e camminare il meno possibile.

Vediamo ora la definizione rigorosa:

- **input**:
  - grafo non orientato $G = (V, E)$;
  - lunghezze $\langle \delta_e \rangle_{e \in E}$;
- **soluzioni ammissibili**: circuiti che passano per ogni vertice esattamente una volta; i circuiti che hanno questa proprietà sono detti **circuiti hamiltoniani**;
- **funzione obiettivo**: lunghezza del circuito, ovvero la somma delle lunghezze dei lati scelti per il circuito hamiltoniano;
- **tipo**: min.

Questo problema è molto difficile, anche nella sua versione approssimata, per due motivi:

- non è detto che un circuito hamiltoniano esista;
- riusciamo a fare bene solo in alcuni casi speciali, ovvero con particolari grafi o lunghezze.

Noi faremo due modifiche al problema TSP per renderlo *«studiabile»*.

Lemma 8.2.1: TSP $\equiv$ TSP su cricche.

Dimostrazione 8.2.1: Se abbiamo a disposizione un algoritmo che risolve il TSP su cricche e vogliamo risolvere il TSP generale, basta aggiungere dei lati di peso spropositato fino a creare una cricca, e passare la nuova istanza al nostro algoritmo.

Se abbiamo a disposizione un algoritmo che risolve il TSP generale e vogliamo risolvere il TSP su cricche, non dobbiamo modificare niente: il TSP generale lavora su qualsiasi grafo, quindi anche sulle cricche. ■

La prima modifica che facciamo a TSP è questa: assumiamo che i nostri grafi siano completi, ovvero siano delle **cricche**. Chiamiamo $G = K_m$ la cricca di m nodi.

La seconda modifica che introduciamo è rendere la distanza uno **spazio metrico**, ovvero le distanze rispettano la disuguaglianza triangolare

$$\forall x, y, z \quad \delta_{xy} \leq \delta_{xz} + \delta_{zy}.$$

Queste due modifiche rendono trasformando TSP nel problema **TSP metrico**.

Ci mancano infine *due ingredienti, due algoritmi*:

- **minimum spanning tree**: dato un grafo connesso pesato (*sui lati*), trovare un **albero di copertura** (*ovvero che tocca tutti i vertici*) di peso totale minimo. Questo problema si risolve in tempo polinomiale con vari algoritmi, il più famoso è l'**algoritmo di Kruskal**;
- **minimum-weight perfect matching**: data una cricca pesata con un numero pari di vertici, trovare un matching perfetto di peso minimo. In modo informale: i nodi si amano tutti (*cricca*), c'è attrito tra le coppie (*peso*), ma devo far sposare tutti (*matching perfetto*). Questo problema si risolve in tempo polinomiale con l'**algoritmo di Blossom**.

Possiamo finalmente vedere l'algoritmo per il TSP metrico.

Algoritmo di Christofides

input
- cricca $G = (V, E)$
- pesi $\langle \delta_e \rangle_{e \in E}$ che sono una metrica

1: $T =$ minimum spanning tree di G
2: $D =$ insieme dei vertici di grado dispari in T
- Per l'handshaking lemma, $|D|$ ha cardinalità pari

3: $M =$ minimum-weight perfect matching su D
- Può succedere che $T \bigcap M \neq \emptyset$, ovvero dei lati sono usati per entrambi gli insiemi

4: $H = T \bigcup M$ unione disgiunta che contiene i doppioni, quindi ho un multigrafo
- D ora ha tutti vertici di grado pari, visto che ho aggiunto i lati del match M

5: $\pi =$ cammino euleriano, la cui esistenza è garantita dal teorema di Eulero
6: $\widetilde{\pi}$ = trasformazione di π in un circuito hamiltoniano tramite strozzatura dei cappi
- se ho un arco (x, y), con y già toccato una volta, vado nel nodo dopo y
- sotto ho una cricca, quindi posso andare dove voglio con gli archi
- prendo tutti i vertici ma saltando i doppioni

7: **output** $\widetilde{\pi}$

Vediamo due lemmi che saranno utili per calcolare l'approssimazione che fornisce questo algoritmo.

Lemma 8.2.2:

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

Dimostrazione 8.2.2: Sia π^* un TSP ottimo. Se da π^* togliamo un qualunque lato e otteniamo un albero di copertura. Siccome T è albero di copertura minimo, allora

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

Lemma 8.2.3:

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

Dimostrazione 8.2.3: Sia π^* il TSP ottimo. Dentro questo TSP ho anche i vertici di D, nodi che nell'albero che avevano grado dispari. Costruiamo un circuito sui vertici di D nell'ordine in cui questi vertici compaiono in π^*. Chiamiamo questo circuito $\overline{\pi}^*$.

Sappiamo che

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

perché sto prendendo delle scorciatoie e quindi vale la disuguaglianza triangolare.

Dividiamo i lati di $\overline{\pi}^*$ in due insiemi M_1 e M_2 alternando l'inserimento dei lati. Ognuno dei due insiemi è un perfect matching su D. Ma M è il minimum perfect matching, quindi

$$\delta(M) \leq \delta(M_1) \quad e \quad \delta(M) \leq \delta(M_2),$$

quindi

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

Ma allora

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

Teorema 8.2.1: L'algoritmo di Christofides è una $\frac{3}{2}$-approssimazione per il TSP metrico.

Dimostrazione 8.2.4: Osserviamo che $\delta(\pi) = \delta(T) + \delta(M)$, ma per i due lemmi precedenti sappiamo che

$$\delta(\pi) \underset{l1+l2}{\leq} \delta^* + \frac{1}{2}\delta^* = \frac{3}{2}\delta^*.$$

Noi però diamo in output $\widetilde{\pi}$. Per la disuguaglianza triangolare, visto che noi strozziamo quando passiamo al circuito hamiltoniano, abbiamo che

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

quindi

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

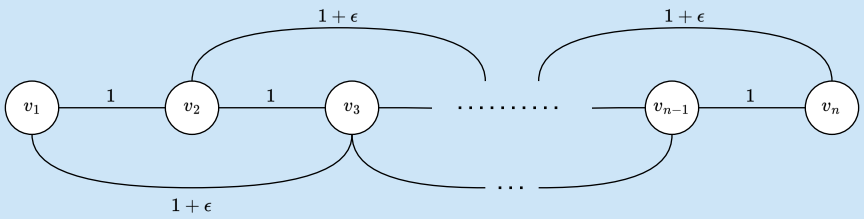
■

Teorema 8.2.2: L'analisi fatta è stretta.

Dimostrazione 8.2.5: Dato n un numero intero pari e $\varepsilon \in (0, 1)$ l'errore, costruisco il grafo $G_{(n,\varepsilon)}$ formato dai vertici $v_1, v_2, ..., v_n$ collegati da:

- lati (v_i, v_{i+1}) di lunghezza 1;
- lati (v_i, v_{i+2}) di lunghezza $1 + \varepsilon$.

Devo avere quel bound sulla ε perché senza la distanza considerata non sarebbe una metrica.



Trasformo il grafo in una cricca ma i lati aggiunti devono mantenere la metrica. Costruiamo quindi $K_{(n,\varepsilon)}$ a partire da $G_{(n,\varepsilon)}$ con i lati nuovi di lunghezza uguale allo shortest path tra i due vertici incidenti. In questo modo rispetto naturalmente la disuguaglianza triangolare, essendo uguale alla somma dei due cammini minimi.

L'algoritmo di Christofides per prima cosa trova l'albero di copertura minimo T, che è quello formato dai soli lati di lunghezza 1. Non conviene scegliere i lati lunghi $1 + \varepsilon$ perché prendendoli siamo obbligati a prendere un altro lato (*quello per collegare il vertice saltato*) e avere quindi $2 + \varepsilon > 1$. Abbiamo quindi $\delta(T) = n - 1$.

Tutti i vertici interni a T hanno grado pari, mentre i due vertici sugli estremi hanno grado dispari, quindi il minimum-weight perfect matching è quello che collega v_1 con v_n. Per fare ciò prendo i

lati di lunghezza $1+\varepsilon$ fino all'ultimo vertice di indice dispari v_{n-1}, e infine prendo l'ultimo lato di lunghezza 1. Abbiamo quindi $\delta(M) = (1+\varepsilon)\frac{n}{2} + 1$.

Ma allora

$$\delta = \delta(T) + \delta(M) = (n-1) + \left(\frac{n}{2} + \varepsilon\frac{n}{2} + 1\right) = \frac{3}{2}n + \varepsilon\frac{n}{2}.$$

Il circuito hamiltoniano minimo si trova partendo da v_2. Esso si trova facendo:

- $v_2 \longrightarrow v_1$ di lunghezza 1;
- $v_1 \overset{*}{\longrightarrow} v_{n-1}$ di lunghezza $(1+\varepsilon)\frac{n}{2}$;
- $v_{n-1} \longrightarrow v_n$ di lunghezza 1;
- $v_n \overset{*}{\longrightarrow} v_2$ di lunghezza $(1+\varepsilon)\frac{n}{2}$.

Ma allora

$$\delta^* = 1 + (1+\varepsilon)\frac{n}{2} + (1+\varepsilon)\frac{n}{2} + 1 = (1+\varepsilon)n + 2.$$

Calcoliamo infine

$$\frac{\delta}{\delta^*} = \frac{\frac{3}{2}n + \varepsilon\frac{n}{2}}{(1+\varepsilon)n + 2} \underset{n\to\infty}{\longrightarrow} \frac{3}{2}.$$

■

L'algoritmo di Christofides è stato ideato verso la fine degli anni '70, ed è il migliore attualmente a nostra disposizione. Possiamo fare di meglio modificando il grafo, imponendo le distanze tutte uguali a 1 o 2. Questo algoritmo è stato scoperto in parallelo da un personaggio russo durante la guerra fredda, ma visto che non si parlavano nessuno sapeva dell'altro.

Lemma 8.2.4: Il problema di decidere se un grafo ammetta un circuito hamiltoniano è *NP-C*.

Teorema 8.2.3: Non esiste $\alpha > 1$ tale che il TSP generale sia α-approssimabile.

Dimostrazione 8.2.6: Supponiamo per assurdo di avere un algoritmo α-approssimante per TSP, con $\alpha > 1$.

Dato un grafo $G = (V, E)$ vogliamo sapere se esso ha un CH. Trasformiamo G in una istanza accettabile per TSP: per fare ciò dobbiamo rendere il grafo una cricca e dare dei pesi ad ogni lato. Creiamo quindi la cricca

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

con

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

Se G aveva un CH allora G' ne ha uno di lunghezza $\leq n$. Se G non ha un CH allora qualunque CH di G' ha lunghezza $\geq \lceil \alpha n \rceil + 1$.

Valuto il mio algoritmo α-approssimante per TSP sull'istanza G'. Se questo fosse un algoritmo esatto allora troverebbe il migliore CH, e guardando l'output (*la lunghezza del CH*) potrei dire se lo stesso CH era in G, quindi otterrei un assurdo.

Questo purtroppo è un algoritmo approssimato, quindi abbiamo due casi:

- se G ha un CH allora in G' ho un CH di lunghezza n, ma essendo un algoritmo approssimato mi viene restituito un CH un pelo più lungo ma comunque $\leq \alpha n$;
- se G non ha un CH allora in G' ho un CH di lunghezza $\geq \lceil \alpha n \rceil + 1$.

I due intervalli non si sovrappongono: infatti, se per assurdo $\alpha n \geq \lceil \alpha n \rceil + 1$, allora

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

che è un assurdo. I due insiemi quindi non si sovrappongono.

Ma questo è impossibile: riesco a decidere se G ha un CH in tempo polinomiale, ma per il lemma precedente questo questo non è possibile in tempo polinomiale. ■

9. PTAS per 2-LoadBalancing

Ricordiamo cosa sono i *PTAS*: sono problemi che sono **approssimabili a meno di una qualunque costante**. In poche parole, dato un problema, posso scegliere una qualunque costante (*tasso di approssimazione*) tale che esiste un algoritmo che mi risolve il dato problema con il tasso di approssimazione scelto. Purtroppo, più il tasso di errore scelto è basso, più è esponenziale il tempo.

Il problema per il quale vediamo un PTAS è quello che abbiamo già visto, nel quale però viene fissato il numero di macchine a 2. In questo caso, il carico generale sarà peggiore di Load Balancing puro, ma almeno il problema è un PTAS.

Diamo la definizione rigorosa del problema:

- **input**: insieme di task $t_0, ..., t_{n-1} > 0$;
- **soluzione ammissibile**: funzione di assegnamento delle n task a 2 macchine, ovvero $\alpha : n \longrightarrow 2$;
- **funzione obiettivo**: $\max\left(\sum_{i \mid \alpha(i)=0} t_i, \sum_{i \mid \alpha(i)=1} t_i\right)$;
- **tipo**: min.

Vediamo l'algoritmo magico per questo problema.

Algoritmo magico

input
- insieme di task $t_0, ..., t_{n-1}$
- macchine M_0 e M_1
- tasso di approssimazione $\varepsilon > 0$ per ottenere una $(1+\varepsilon)$-approssimazione

1: if $\varepsilon \geq 1$
2: - Assegna tutti i task t_i alla macchina M_0
3: - Salta al punto 10
4: Ordina i task t_i in ordine decrescente
5: **FASE 1**
6: - $k \leftarrow \lceil \frac{1}{\varepsilon} - 1 \rceil$
7: - Si cerca esaustivamente (*tempo* 2^k) l'assegnamento ottimo dei primi k task
8: **FASE 2**
9: - I restanti $n - k$ task sono assegnati in modo greedy
10: **output** massimo carico

Tutti i *PTAS* sono così: la parte più importante del problema (*in questo caso, l'assegnamento dei task più pesanti*) la vogliamo risolvere in modo esatto, ma questo è impattante sul tempo ed è difficile. Il resto invece lo approssimo.

Teorema 9.1: L'algoritmo magico è una $(1+\varepsilon)$-approssimazione per 2-LoadBalancing.

Dimostrazione 9.1: Sia $T = \sum_{i=0}^{n-1} t_i$. Vediamo due casi distinti per il valore di ε.

Se $\varepsilon \geq 1$ ci stiamo accontentando di una 2-approssimazione. Sappiamo infatti che

$$L^* \geq \frac{\sum_{i=0}^{n-1} t_i}{2} \quad (*)$$

se riusciamo a fare una divisione «perfetta». Visto che assegniamo tutto ad una macchina, allora $L = \sum_{i=0}^{n-1} t_i = T$ e quindi

$$\frac{L}{L^*} \overset{*}{\leq} \frac{T}{\frac{T}{2}} = 2.$$

Se $\varepsilon \in (0,1)$ alla fine della prima fase ho carichi y_0 e y_1. Dopo la seconda fase ho invece carichi L_0 e L_1. Assumiamo che $L_0 \geq L_1$. Ci sono anche qui due casi da analizzare.

Il primo caso avviene quando $L_0 = y_0$: in questo caso tutti i task della seconda fase sono finiti a M_1. In poche parole, in ogni step della seconda fase avevo M_1 più scarica di M_0. Ma questo che abbiamo ottenuto è l'assegnamento globale ottimo: infatti, partendo da una soluzione già ottima, se avessi sbagliato qualcosa nella fase greedy avrei assegnato almeno un task a M_0, ma questo avrebbe peggiorato lo sbilanciamento ottenuto nella fase ottima.

Il secondo caso è quello *«normale»*: assegno un po' di task alle due macchine ma non in modo esclusivo. Sia t_h l'ultimo task assegnato a M_0, il più piccolo presente in M_0. Sappiamo che

$$L_0 - t_h \underset{\text{assegnamento}}{\leq} L_1' \underset{\text{altro}}{\leq} L_1.$$

Sommiamo L_0 ad entrambi i membri e dividiamo per 2, quindi

$$2L_0 - t_h \leq \underbrace{L_0 + L_1}_{=T} \implies L_0 - \frac{t_h}{2} \leq \frac{T}{2} \implies L_0 \leq \frac{T}{2} + \frac{t_h}{2}. \quad (**)$$

Sappiamo che

$$T = t_0 + \ldots + t_k + t_{k+1} + \ldots + t_{n-1}$$

e che i t_i sono ordinati in senso decrescente. Ma allora, *stando molto larghi*, vale

$$T = \underbrace{t_0 + \ldots + t_k}_{\forall i=0,\ldots,k \quad t_i \geq t_h} + \underbrace{t_{k+1} + \ldots + t_{n-1}}_{>0} \geq (k+1)t_h. \quad (***)$$

Valutiamo finalmente

$$\frac{L}{L^*} \overset{\text{HP}}{=} \frac{L_0}{L^*} \overset{*}{\leq} \frac{L_0}{\frac{T}{2}} \overset{**}{\leq} \frac{\frac{T}{2} + \frac{t_h}{2}}{\frac{T}{2}} = 1 + \frac{t_h}{T} \overset{***}{\leq} 1 + \frac{t_h}{(k+1)t_h} = 1 + \frac{1}{k+1}.$$

Ricordando che $k = \left\lceil \frac{1}{\varepsilon} - 1 \right\rceil$, infine vale

$$\frac{L}{L^*} \leq 1 + \frac{1}{\frac{1}{\varepsilon} - 1 + 1} = 1 + \frac{1}{\frac{1}{\varepsilon}} = 1 + \varepsilon.$$

■

Se ε molto basso potrei richiedere nella fase 1 di assegnare tutti i task, non eseguendo di fatto la seconda parte. Possiamo imporre ad esempio il vincolo $\left\lceil \frac{1}{\varepsilon} - 1 \right\rceil \leq n$ o assegnare in modo ottimo le prime $\max\left(\left\lceil \frac{1}{\varepsilon} - 1 \right\rceil, n\right)$ task.

Teorema 9.2: L'algoritmo magico richiede tempo $O\left(\underbrace{n\log(n)}_{\text{ordinamento}} + \underbrace{2^{\min((\frac{1}{\varepsilon}),n)}}_{\text{ass.es}} + \underbrace{\left(n-\frac{1}{\varepsilon}\right)}_{\text{ass.appr}}\right)$.

Dimostrazione 9.2: La dimostrazione è scritta negli underbracket. Nella fase 1 di assegnamento esatto questo lo posso fare in $2^k \approx 2^{\frac{1}{\varepsilon}}$ modi. La fase 2 ha tempo $O(n-k)$. L'ordinamento avviene con qualunque algoritmo non banale. ■

10. Knapsack

Dei pirati arrivano in un grotta e trovano degli oggetti, ognuno con un valore $v_0, ..., v_{n-1}$. I pirati vorrebbero portare a casa gli oggetti con più valore, ma ogni oggetto ha un peso $w_0, ..., w_{n-1}$ e i bro hanno uno zaino con una certa capacità massima W. Vogliamo portare a casa il massimo valore senza rompere lo zaino. Il problema di decisione associato, ovvero chiedersi se sia possibile portare a casa più di k oggetti, è un problema *NP-C*.

Vediamo la definizione formale:

- **input**:
  - $n > 0$ oggetti;
  - $w_i > 0$ pesi con $i \in n$;
  - $W > 0$ capacità dello zaino;
- **soluzione ammissibile**: $X \subseteq n$ tale che $\sum_{i \in X} w_i \leq W$;
- **funzione obiettivo**: $v = \sum_{i \in X} w_i$;
- **tipo**: max.

Il problema è *NPO-C* visto che il suo problema di decisione associato è *NP-C*.

Useremo una tecnica chiamata **programmazione dinamica**: quest'ultima risolve un certo problema individuando k parametri che andranno a formare una tabella k-dimensionale. Al posto di risolvere il problema noi andremo a riempire la tabella, è più facile rispetto alla risoluzione del problema originale. Abbiamo una serie di vincoli:

- le prime *«righe»* e le prime *«colonne»* (*facciamo finta di essere in 2d*) devono essere facili da riempire;
- dobbiamo essere in grado di riempire ogni cella partendo da celle che abbiamo già riempito precedentemente, ovvero dobbiamo avere una **strategia di riempimento**.

Ogni cella è un problema, che prende in input i parametri che indicizzano quella cella.

Questa tecnica è molto comoda quando i problemi sono di natura esponenziale. Il numero di problemi dipende dai valori dei parametri. Se i parametri sono nell'input del problema, la dimensione della tabella è esponenziale.

Prima di vedere la PD applicata a Knapsack, introduciamo il concetto di **pseudo-polinomialità**.

Esempio 10.1: Voglio un programma che decide se un numero naturale x è primo (*BTW è stato trovato finalmente un algoritmo polinomiale per questo problema*). Il programma è molto tremendamente difficile da scrivere, vediamolo in GO.

```
input(x)
for i := 2; i < x; i++ {
  if x % i == 0 {
    return false
  }
}
return true
```

Il ciclo `for` viene eseguito $O(x)$ volte. Quindi l'algoritmo è polinomiale? **NO**. Il concetto di polinomialità deve essere inteso nella lunghezza dell'input, non nel valore dell'input. Questo perché avendo un algoritmo polinomiale nel valore dell'input, se quest'ultimo viene raddoppiato allora l'algoritmo si comporta ancora bene, ma se raddoppia la lunghezza dell'input il valore cresce spropositatamente e l'algoritmo inizia a fare fatica.

Infatti, possiamo scrivere

$$x = 2^{\log_2(x)},$$

quindi la lunghezza di x è esponenziale nel mio input.

Lemma 10.1: Se un algoritmo è polinomiale nella lunghezza dell'input allora lo è anche nel valore dell'input, ma non vale il viceversa.

Questo algoritmo è chiamato pseudo-polinomiale, ovvero un algoritmo che è lineare nel tempo ma esponenziale nella lunghezza dell'input (????).

10.1. PD: prima versione

Prima versione di PD (focus pesi)

input

- $n > 0$ numero di oggetti
- $v_0, ..., v_{n-1}$ valore degli oggetti
- $w_0, ..., w_{n-1}$ peso degli oggetti
- $W > 0$ capacità dello zaino
- questo input ha lunghezza $O(n)$

Parametri della tabella

- Righe: capacità dello zaino w, lavora sui valori $0, ..., W$
- Colonne: quanta parte dell'input considerare i, lavora sui valori $0, ..., n$
  - Considero gli oggetti fino a $(i-1)$ compreso
- In una cella inserisco il valore massimo che riesco a portare a casa

Strategia di riempimento

- La prima riga è 0 perché la capienza w è 0
- La prima colonna è 0 perché sto considerando 0 oggetti

  L'algoritmo riempie per righe tutte quelle successive alla prima
- Data la cella $v[w, i]$:
  - Considero la cella $v[w, i-1]$, ovvero a parità di peso prendo i primi $i-1$ oggetti
  - Se non prendo l'oggetto i-esimo allora $v[w, i] = v[w, i-1]$
  - Se prendo l'oggetto i-esimo allora $v[w, i] = \max(v[w, i-1], v_{i-1} + v[w - w_{i-1}, i-1])$
    - In poche parole, vedo cosa prendevo con w_{i-1} peso in meno e ci sommo il nuovo peso

    In forma contratta:

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

output: $v[W, n]$

La matrice v è esponenziale in W, visto che è un parametro di input del problema originale. Per far si che questo algoritmo funzioni, è essenziale che i pesi w_i siano tutti interi e anche la capacità W. Se la tabella avesse valori continui allora avremmo degli algoritmi polinomiali per risolverla.

10.2. PD: seconda versione

Seconda versione di PD (focus valori)

input

- $n > 0$ numero di oggetti
- $v_0, ..., v_{n-1}$ valore degli oggetti
- $w_0, ..., w_{n-1}$ peso degli oggetti
- $W > 0$ capacità dello zaino
- questo input ha lunghezza $O(n)$

Parametri della tabella

- Righe: massimo valore che portiamo a casa v, lavora sui valori $0, ..., \sum_i v_i$
- Colonne: quanta parte dell'input considerare i, lavora sui valori $0, ..., n$
  - Considero gli oggetti fino a $(i-1)$ compreso
- In una cella inserisco la capacità minima dello zaino che contiene quegli elementi
  - A volte questa capacità sarà impossibile (*infinita*)

1: **Strategia di riempimento**

2: La prima riga è 0 perché non voglio portare a casa niente

3: La prima colonna, dalla seconda riga, è ∞ perché voglio portare a casa qualcosa ma non ho oggetti disponibili

L'algoritmo riempie per righe tutte quelle successive alla prima

4: Data la cella $w[v, i]$:

5: Considero la cella $w[v, i-1]$, ovvero a parità di valore richiesto considero un oggetto in meno

6: Se non prendo l'oggetto i-esimo allora $w[v, i] = \min(w[v, i-1], w_{i-1})$

7: Se prendo l'oggetto i-esimo allora $w[v, i] = \min(w[v, i-1], w_{i-1} + w[v - v_{i-1}, i-1])$

In forma contratta:

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

L'ultima colonna contiene il peso minimo degli zaini considerando tutti gli oggetti

8: Filtro le entry $w[v, n]$ tali che $w[v, n] > W$ perché non sono ammissibili

9: **output**: l'indice della riga che contiene l'ultima entry accettabile

Anche qui, la matrice w è esponenziale ma nella somma dei valori v.

10.3. FPTAS

I due algoritmi di PD visti trovano la soluzione esatta, ma vista la loro pseudo-polinomialità, essi sono esponenziali. Vediamo quindi un algoritmo polinomiale per la risoluzione, ma approssimato.

Parliamo di **Turchia**: prima degli anni '90 la *lira turca* non valeva assolutamente niente. Idea geniale del governo: negli anni '90 viene introdotta la *lira pesante*, che valeva 10.000 lire turche originali. In poche parole, ogni prezzo è stato diviso per 10.000.

In cosa può esserci utile la Turchia, **oltre al trapianto di capelli**?

Dobbiamo abbattere la dimensione della tabella: possiamo esprimere i valori degli oggetti in una **moneta dei pirati pesante**, in modo che i valori diventino piccoli e che la matrice si comprima.

Abbiamo però un problema: se comprimiamo tanto i valori questi potrebbero confondersi, visto che gli indici della tabella sono **interi**. Noi diciamo chissene frega, almeno stiamo rendendo la soluzione polinomiale.

Una cosa che non abbiamo mai detto ma che è banale: gli algoritmi che utilizziamo per approssimare sono comunque ammissibili, ovvero non andiamo mai fuori dai bound del problema. In poche parole:

- accettiamo una cosa **sub-ottima**, che è peggio dell'ottimo;
- non accettiamo **mai** una soluzione **sub-ammissibile**, che va fuori dai bound.

Andiamo a comprimere la seconda tabella perché la prima rischia di rompere i vincoli di ammissibilità. Questa tecnica di compressione si chiama **scaling**. Non posso comprimere la prima tabella perché le righe mi danno l'**ammissibilità**: se comprimo potrei ottenere soluzioni che nella compressa vanno bene ma non vanno bene in quella normale.

FPTAS per Knapsack

input

problema $\Pi = (v_i, w_i, W)$

errore $0 < \varepsilon \leq 1$ per avere una $1 + \varepsilon$ approssimazione di Π

Definiamo

1: $$\vartheta = \frac{\varepsilon v_{\max}}{2n}$$

Definiamo

2: $$\overline{\Pi} = \left(\overline{v}_i = \left\lceil \frac{v_i}{\vartheta} \right\rceil \vartheta, w_i, W\right)$$

I valori degli oggetti di $\overline{\Pi}$ sono molto simili a quelli di Π

Definiamo

3: $$\hat{\Pi} = \left(\widetilde{v} = \left\lceil \frac{v_i}{\vartheta} \right\rceil, w_i, W\right)$$

4: Risolviamo $\hat{\Pi}$ in modo esatto con la seconda versione dell'algoritmo PD

Osserviamo che l'**ammissibilità** è la stessa in tutti i problemi: stiamo considerando gli stessi pesi e la stessa capacità dello zaino, quindi le soluzioni ammissibili sono uguali. Le soluzioni ottime però cambiano, visto che dipendono dai valori, e questi sono diversi.

Osserviamo inoltre che $\overline{X}^* = \hat{X}^*$ perché i valori differiscono per una costante.

Lemma 10.3.1: Sia X una soluzione ammissibile. Allora

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

Dimostrazione 10.3.1: I valori di $\overline{\Pi}$ sono un pelo più grandi di quelli di Π, quindi

$$\sum_{i \in X} v_i \leq \sum_{i \in X} \overline{v}_i \underset{\text{OPT}}{\leq} \sum_{i \in \overline{X}^*} \overline{v}_i.$$

Visto che i $\overline{v}_i$ sono ottenuti con una approssimazione per eccesso, ogni $\overline{v}_i$ è al massimo un ϑ in più, quindi

$$\sum_{i \in X} v_i \leq \sum_{i \in \overline{X}^*} (v_i + \vartheta) = \sum_{i \in \overline{X}^*} v_i + \left|\overline{X}^*\right| \vartheta \leq \sum_{i \in \overline{X}^*} v_i + n\vartheta = \sum_{i \in \overline{X}^*} v_i + n \frac{\varepsilon v_{\max}}{2n}.$$

Abbiamo ottenuto quindi

$$\sum_{i \in X} v_i \leq \sum_{i \in \overline{X}^*} v_i + \frac{\varepsilon v_{\max}}{2}. \quad (*)$$

Questa disuguaglianza è vera per ogni soluzione ammissibile. Una di queste è $X_{\max} = \{i_{\max}\}$, soluzione che contiene l'indice dell'elemento di valore massimo (*dopo aver tolto i valori che hanno peso maggiore della capacità dello zaino*).

Ma allora

$$\sum_{i \in X_{\max}} v_i = v_{\max} \leq \sum_{i \in \overline{X}^*} v_i + \frac{\varepsilon v_{\max}}{2} \underset{(\varepsilon \leq 1)}{\leq} \sum_{i \in \overline{X}^*} v_i + \frac{v_{\max}}{2}.$$

Portando a sinistra il termine $\frac{v_{\max}}{2}$ otteniamo

$$\frac{v_{\max}}{2} \leq \sum_{i \in \overline{X}^*} v_i. \quad (**)$$

Uniamo $(*)$ e $(**)$ e otteniamo

$$\sum_{i \in X} v_i \leq \sum_{i \in \overline{X}^*} v_i + \frac{\varepsilon v_{\max}}{2} \overset{(**)}{\leq} \sum_{i \in \overline{X}^*} v_i + \varepsilon \left(\sum_{i \in \overline{X}^*} v_i \right) = (1 + \varepsilon) \sum_{i \in \overline{X}^*} v_i.$$

Abbiamo prima osservato che $\hat{X}^* = \overline{X}^*$, ma allora

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

■

Teorema 10.3.1: Vale inoltre

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

Dimostrazione 10.3.2: Per il lemma precedente vale

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

Considero $X = X^*$. Allora

$$(1+\varepsilon)\sum_{i\in\hat{X}^*} v_i \geq \sum_{i\in X^*} v_i = v^*.$$

■

Corollario 10.3.1.1: L'algoritmo FPTAS è una $(1+\varepsilon)$-approssimazione per Knapsack.

Manca da analizzare la **complessità** di questo FPTAS: come è fatta la matrice? Quante sono le celle?

La matrice è formata da:

- **righe**: vanno da 0 a $\sum_i \hat{v}_i$, ma ogni indice di riga è sicuramente $\leq \hat{v}_{\max}$, quindi $\sum_i \hat{v}_i \leq n\hat{v}_{\max}$;
- **colonne**: n.

La dimensione è quindi

$$\text{rows}\cdot\text{cols} \leq n^2\hat{v}_{\max} = n^2\left\lceil\frac{v_{\max}}{\vartheta}\right\rceil = n^2\left\lceil\frac{2nv_{\max}}{\varepsilon v_{\max}}\right\rceil = O\left(\frac{n^3}{\varepsilon}\right).$$

La dimensione della tabella risulta quindi polinomiale in n ma anche in ε. Questa è una enorme differenza rispetto al PTAS della lezione precedente: infatti, il PTAS per 2-LoadBalancing aveva ε in un esponenziale.

Parte III — Algoritmi probabilistici

1. Introduzione

Introduciamo gli **algoritmi probabilistici**: quello che cambia, rispetto a quanto fatto fino a questo momento, è il **paradigma** che utilizziamo.

Gli **algoritmi deterministici** che abbiamo usato prendevano in input $x \in I_\Pi$ e restituivano un output $y \in O_\Pi$ in maniera totalmente deterministica.

Gli **algoritmi probabilistici** hanno la possibilità di pescare da una **sorgente** casuale, che estrae bit con probabilità uniforme. In termini di MdT, la sorgente casuale è un nastro casuale che contiene 0 e 1. L'algoritmo non è più deterministico: se conoscessimo a prescindere l'input e la porzione di *«nastro random»* visitata potremmo simulare deterministicamente il comportamento, ma visto che non lo conosciamo il processo non è deterministico.

Anche l'output viene modificato: infatti, non abbiamo più un output specifico ma abbiamo una distribuzione di probabilità $P(y \mid x)$ che mi descrive la probabilità di avere l'output y sapendo che è stato inserito in input x.

L'essere probabilistico influisce su due fattori:

- **output**: definito già come distribuzione di probabilità;
- **tempo di esecuzione**: in base ad alcune *«scelte»* dell'algoritmo potremmo metterci più o meno tempo.

A volte i due fattori sono influenzati contemporaneamente dal probabilismo.

Prima di vedere il nostro primo algoritmo probabilistico, vediamo un **piccolo problema** di questo paradigma. Infatti, nessuna macchina deterministica è in grado di simulare una vera MdT probabilistica, ovvero non esistono macchine in grado di produrre dei bit random.

Per ovviare a questo problema, intrinseco dell'architettura che stiamo utilizzando, ci accontentiamo dei **generatori di numeri pseudo-randomici** (*PRNG Pseudo-Random Number Generator*). Queste strutture sono deterministiche ma la loro esecuzione fa sembrare la generazione effettivamente casuale. In poche parole, sono funzioni deterministiche che generano sequenze deterministiche che però sembrano essere sequenze casuali.

2. Taglio minimo globale (mincut)

Il primo problema che vediamo risolto con algoritmi probabilistici è il problema del **taglio minimo globale** (*o **mincut***). Esso è definito da:

- **input**: grafo non orientato $G = (V, E)$;
- **soluzione ammissibile**: insieme di vertici $X \subseteq V$ tale che:
  - $X \neq \varnothing$;
  - $X^C \neq \varnothing$;
- **funzione obiettivo**: vogliamo valutare la quantità $k = \left|\{e \in E \mid e \cap X \neq \varnothing \wedge e \cap X^C \neq \varnothing\}\right|$, ovvero il numero di lati che hanno un vertice in X e l'altro nel suo complemento; in poche parole, se disegno G, voglio contare il numero di lati che fanno da ponte dalla zona X alla zona X^C;
- **tipo**: min.

Anche se non sembra, anche questo problema è *NPO-C*.

Lemma 2.1: Il taglio minimo è $\leq$ del grado minimo dei vertici.

Questo è un bound molto molto ampio: ad esempio, unendo due cricche K_n e K_m con un lato, il taglio minimo vale $k = 1$, che è molto più piccolo del grado minimo $n - 1$ o $m - 1$.

Prima di vedere l'algoritmo risolutivo per mincut diamo la definizione di **contrazione di grafi**: dato un grafo G, la contrazione $G \downarrow e$ sul lato e si calcola con i seguenti passi:

- eliminare il lato e dal grafo;
- unire i due vertici sui quali e incideva in un unico vertice;
- unire ogni vicino dei vertici fusi al nuovo vertice;
  - questa operazione può causare la creazione di due lati paralleli e quindi di un multigrafo.

Visto l'ultimo punto, di solito la definizione di contrazione è data sui multigrafi.

Questa operazione va a ridurre ad ogni iterazione il numero di lati e di vertici del multigrafo.

Algoritmo di Karger

input
  multigrafo $G = (V, E)$
1: if G non connesso
2: **output** una componente connessa qualunque
    il taglio vale 0
3: else
4: while $|V| > 2$
5: Sia e un lato a caso
6: Calcola $G \downarrow e$
      Questa operazione contrae tutti i lati paralleli ad e
7: **output** la classe di equivalenza di uno dei due vertici rimanenti
  I vertici finali sono insiemi di vertici, ma anche classi di equivalenza

Mostreremo che esiste una probabilità non nulla di ottenere la soluzione ottima. Negli altri casi, il risultato ottenuto sarà arbitrariamente brutto.

Durante l'esecuzione dell'algoritmo abbiamo una serie di multigrafi $G_1 \longrightarrow G_2 \longrightarrow \dots$ che parte da $G_1 = G$. Sia G_i il multigrafo che abbiamo prima dell'i-esima iterazione. Chiamiamo X^* il taglio minimo e k^* la sua **dimensione**, ovvero il numero di archi che tagliano il taglio minimo.

Vista l'operazione di contrazione, abbiamo che:

- G_i ha $n - i + 1$ vertici (*in ogni iterazione perdiamo un vertice*);
- G_i ha $\leq m - i + 1$ lati (*in ogni iterazione cancelliamo tutti i lati paralleli*).

Inoltre, ogni taglio di G_i corrisponde ad un taglio di G con la stessa dimensione.

In particolare, il grado minimo di G_i è $\geq k^*$: infatti, se avessi un taglio più piccolo anche G ce l'avrebbe ma questo è impossibile perché X^* con k^* è il taglio minimo.

Sia ora m_i il numero di lati di G_i, allora

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

ma il numero di vertici in G_i è $n - i + 1$ e sappiamo che il grado minimo è $\geq$ della dimensione ottima, quindi

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

Sia E_i l'evento che ci dice se all'i-esima iterazione **NON** contraiamo uno dei lati tagliati dal taglio minimo. Dio ci ha detto qual è il taglio minimo, sappiamo quali sono i lati del taglio minimo.

Lemma 2.2: Vale

$$P(E_i \mid E_1, \dots, E_{i-1}) \geq \frac{n - i - 1}{n - i + 1}.$$

Dimostrazione 2.1: Valutiamo

$$\begin{aligned} P(E_i \mid E_1, \dots, E_{i-1}) &= 1 - P\big(\overline{E}_i \mid E_1, \dots, E_{i-1}\big) = 1 - \frac{k^*}{m_i} \\ &\geq 1 - \frac{2k^*}{k^*(n - i + 1)} = \frac{n - i + 1 - 2}{n - i + 1} = \frac{n - i - 1}{n - i + 1}. \end{aligned}$$

Fanculo il quadrato. ■

Questo lemma vuole vedere la probabilità di non contrarre un lato prezioso sapendo le probabilità dello stesso evento nei multigrafi precedenti.

Teorema 2.1: L'algoritmo di Karger emette il taglio minimo con probabilità

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

Dimostrazione 2.2: Il taglio minimo si ottiene quando non contraggo mai i lati preziosi, quindi

$$P = P(E_1 \wedge E_2 \wedge ... \wedge E_{n-2}) = \\ \underset{\text{CR}}{=} P(E_1) \cdot P(E_2 \mid E_1) \cdot P(E_3 \mid E_2, E_1) \cdot ... \cdot P(E_{n-2} \mid E_{n-3}, ..., E_1)$$

per la Chain Rule della probabilità. Noi queste quantità le conosciamo, ovvero

$$P \underset{\text{lemma}}{\geq} \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot ... \cdot \frac{1}{3} = \\ = \frac{(n-2)!}{n \cdot (n-1) \cdot ... \cdot 3} \cdot \frac{2}{2} = \frac{2(n-2)!}{n(n-1)(n-2)!} = \frac{2}{n(n-1)}.$$

Ricordando che

$$\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)(n-2)!}{(n-2)!2} = \frac{n(n-1)}{2},$$

allora

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

■

Questa quantità decresce quadraticamente con n, ovvero più è grande il multigrafo e più ho probabilità bassa di trovare il taglio minimo.

Corollario 2.1.1: Eseguendo l'algoritmo di Karger

$$\binom{n}{2} \ln(n)$$

volte si ottiene il taglio minimo con probabilità $P \geq 1 - \frac{1}{n}$.

Dimostrazione 2.3: Dall'analisi matematica si sa che (*non lo sapevo*)

$$\forall x > 1 \quad \frac{1}{4} \leq \left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}. \quad (*)$$

Valutiamo la probabilità di **NON** ottenere il taglio minimo, ovvero nessuna delle prove che abbiamo fatto ha dato il taglio minimo, ma allora, per il teorema precedente, vale

$$P_{\text{singolo}} \leq 1 - \frac{1}{\binom{n}{2}} \underset{\text{ind}}{\longrightarrow} P \leq \left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}\ln(n)} \overset{*}{\leq} \left(\frac{1}{e}\right)^{\ln(n)} = \frac{1}{n}.$$

Noi vogliamo la probabilità di ottenere il taglio minimo, quindi

$$P \geq 1 - \frac{1}{n}.$$

■

I bro statistici dicono che questo evento è **quasi certo**, ovvero se n va a infinito allora la probabilità di trovare il taglio minimo va a 1.

3. Set Cover, il ritorno

3.1. Concetti preliminari

Vediamo **cose**: introduciamo qualche risultato matematico preliminare che però non dimostriamo.

Definizione 3.1.1 (*Chain rule*): Legge comodissima nella probabilità, essa afferma che

$$P(E_1 \wedge ... \wedge E_n) = P(E_1) \cdot P(E_2 \mid E_1) \cdot ... \cdot P(E_n \mid E_1, ..., E_{n-1}).$$

Definizione 3.1.2 (*Analisi 1*): Vale

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

Definizione 3.1.3 (*Analisi 2*): Vale

$$\forall x \in [0, 1] \quad 1 - x \leq e^{-x}.$$

Definizione 3.1.4 (*Union bound*): Legge comoda in molte dimostrazioni, essa afferma che

$$P\left(\bigcup_i E_i\right) \leq \sum_i P(E_i).$$

Questa legge afferma che la probabilità dell'unione non supera la somma delle singole probabilità, questo perché gli eventi potrebbero sovrapporsi e quindi essere contati più volte.

Definizione 3.1.5 (*Disuguaglianza di Markov*): Date

- $\mathbb{X} \geq 0$ variabile aleatoria non negativa con media finita e
- $\alpha > 0$ valore reale positivo

vale

$$P(\mathbb{X} \geq \alpha) \leq \frac{\mathbb{E}[\mathbb{X}]}{\alpha}.$$

Le disuguaglianze che hanno questa forma sono dette **leggi di concentrazione**, e sono molto comode perché ci indicano la probabilità che una variabile aleatoria si discosti da un valore considerando la media di quella variabile. Notiamo inoltre come questa relazione sia **indipendente** da come è distribuita la variabile $\mathbb{X}$.

3.2. Algoritmo probabilistico

Avevamo già visto Set Cover, e come **metafora** avevamo usato quella degli spazi pubblicitari o degli abbonamenti di Milano: vogliamo scegliere dei pacchetti per coprire tutto un insieme universo spendendo il meno possibile.

Definiamo nuovamente il problema in termini formali:

- **input**:
  - $m > 0$ insiemi $S_0, ..., S_{m-1}$ tali che $\bigcup_{i \in m} S_i = U$;
  - $m > 0$ pesi $w_0, ..., w_{m-1} \in \mathbb{Q}^+$;
- **soluzione ammissibile**: sottoinsieme di abbonamenti $I \subseteq m$ tale che $\bigcup_{i \in I} S_i = U$;
- **funzione obiettivo**: $w = \sum_{i \in I} w_i$;
- **tipo**: min.

Avevamo già visto un algoritmo di approssimazione per questo problema, oggi vediamo invece un **algoritmo probabilistico**. Prima di tutto, sia m il numero di insiemi e $n = |U|$.

Questo problema lo possiamo scrivere nei termini della **programmazione lineare intera**:

- **variabili**: $x_0, ..., x_{m-1}$ booleane che ci dicono se prendiamo o meno l'insieme $S_i \mid i \in m$;
- **funzione obiettivo**: min $w_0 x_0 + ... + w_{m-1} x_{m-1}$;
- **vincoli**:
  1. $\forall i \in m \quad 0 \leq x_i \leq 1$, ovvero ogni variabile deve essere 0 oppure 1;
  2. $\forall p \in U \quad \sum_{i \mid p \in S_i} x_i \geq 1$, ovvero per ogni elemento dell'universo da coprire prendo gli insiemi che lo contengono e richiedo l'esistenza di una variabile che lo copre.

Chiamiamo questo problema Π. Se rilassiamo Π sui numeri reali otteniamo un problema $\hat{\Pi}$ con risultato approssimato ma sicuramente migliore perché abbiamo ampliato lo spazio delle soluzioni ammissibili.

Arrotondamento aleatorio

input
insiemi $S_0, ..., S_{m-1}$
pesi $w_0, ..., w_{m-1} \in \mathbb{Q}^+$;
$k \in \mathbb{Z}$

Costruiamo il problema Π di PLI
Risolviamo la sua versione rilassata $\hat{\Pi}$ ottenendo $\hat{x}$
$I \leftarrow \varnothing$
PRIMA VERSIONE DEL FOR
for $i \in m$
    for $t = 1, ..., \lceil k + \ln(n) \rceil$
        Con probabilità $\hat{x}_i$ aggiungiamo i a I
SECONDA VERSIONE DEL FOR
for $i \in m$
    Aggiungiamo i a I con probabilità $1 - (1 - \hat{x})^{k + \ln(n)}$
output I

Questo algoritmo è molto semplice: lancio *«tante volte»* la moneta con probabilità $\hat{x}_i$ per decidere se inserire l'insieme S_i nella copertura. Purtroppo, questo algoritmo non dà garanzie di dare una soluzione ammissibile, è improbabile ma potrebbe succedere. Se invece la soluzione data è ammissibile, non è detto che sia ottima. Uno sfigato praticamente.

L'algoritmo di Karger era **GG** perché dava sempre una soluzione ammissibile (*visto che i tagli non sono soggetti a vincoli*) ma spesso dava anche l'ottimo. Qua abbiamo una $P > 0$ di restituire soluzioni non ammissibili, e, se per puro caso abbiamo una soluzione ammissibile, non sempre sarà ottima, ma menomale non lo sarà con bassa probabilità.

Teorema 3.2.1: L'algoritmo di arrotondamento aleatorio ha le seguenti proprietà:

1. produce una soluzione ammissibile con probabilità
$$P \geq 1 - e^{-k};$$
2. per ogni $\alpha \geq 1$ vale
$$P(\text{TA} \geq \alpha(k + \ln(n))) \leq \frac{1}{\alpha}$$
con TA tasso di approssimazione della soluzione.

Il primo punto riguarda l'ammissibilità, mentre il secondo punto ci dice che la probabilità che l'algoritmo vada male è bassa. Nel primo punto notiamo che se k assume valori grandi allora facciamo molti più conti, però abbiamo una buona probabilità di ottenere una soluzione ammissibile.

Dimostrazione 3.2.1: Diamo prima un po' di notazione. Per questa dimostrazione abbiamo bisogno di:

- insiemi $S_0, ..., S_{m-1}$;
- pesi $w_0, ..., w_{m-1}$;
- insieme universo $U = \bigcup_{i=0}^{m} S_i$ con $n = |U|$;
- problemi Π (*normale*) e $\hat{\Pi}$ (*rilassato*);
- soluzione $\hat{x}_0, ..., \hat{x}_{m-1} \in [0, 1]$ del problema $\hat{\Pi}$;
- valore della funzione obiettivo $\hat{w} = w_0\hat{x}_0 + ... + w_{m-1}\hat{x}_{m-1}$;
- conosciamo a culo anche w^*.

Ovviamente vale $\hat{w} \leq w^*$ perché il rilassamento di un problema di PLI è migliore del problema originale (*problema di minimo*).

[**PRIMO PUNTO**]

Sia E_p l'evento «il punto $p \in U$ non è coperto». Sia amm l'evento «la soluzione che abbiamo davanti è ammissibile». Valutiamo

$$
\begin{aligned}
P(\text{amm}) &= 1 - P(\overline{\text{amm}}) = \\
&= 1 - P(\text{almeno un punto di } U \text{ non è coperto}) = \\
&= 1 - P\left(\bigcup_{p\in U} E_p\right) \\
&\underset{\text{UB}}{\geq} 1 - \sum_{p\in U} P(E_p) = \\
&= 1 - \sum_{p\in U} P(p \text{ non è coperto}) = \\
&= \text{nessun insieme che contiene } p \text{ è stato scelto, visto che non lo copriamo} = \\
&= 1 - \sum_{p\in U}\left(\prod_{i\in m \wedge p\in S_i} P(i \text{ non è stato scelto})\right) = \\
&= 1 - \sum_{p\in U}\left(\prod_{i\in m \wedge p\in S_i} (1-\hat{x}_i)^{\lceil k+\ln(n)\rceil}\right) \\
&\geq 1 - \sum_{p\in U}\left(\prod_{i\in m \wedge p\in S_i} (1-\hat{x}_i)^{k+\ln(n)}\right) \\
&\underset{a2}{\geq} 1 - \sum_{p\in U}\left(\prod_{i\in m \wedge p\in S_i} e^{-\hat{x}_i(k+\ln(n))}\right) = \\
&= \text{faccio il prodotto di esponenziali, portando la sommatoria all'esponente} = \\
&= 1 - \sum_{p\in U} e^{-(k+\ln(n))\sum_{i\in m\wedge p\in S_i}\hat{x}_i} \\
&= \text{la sommatoria all'esponente somma valori} \geq 1 \text{ per il vincolo PLI} = \\
&\geq 1 - \sum_{p\in U} e^{-(k+\ln(n))} = \\
&= 1 - \sum_{p\in U} \frac{1}{n} e^{-k} = \\
&= 1 - e^{-k} \underbrace{\sum_{p\in U}}_{|U|=n} \frac{1}{n} = 1 - e^{-k}.
\end{aligned}
$$

[SECONDO PUNTO]

Valutiamo la quantità

$$
P(S_i \text{ scelto}) = P\left(\bigcup_{j=1}^{k\ln(n)} \hat{x}_i\right) \underset{\text{UB}}{\leq} \sum_{j=1}^{k\ln(n)} \hat{x}_i = (k+\ln(n))\hat{x}_i.
$$

Questo vale perché mi basta che l'insieme venga scelto in una di quelle volte.

Valutiamo ora

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

Usiamo la disuguaglianza di Markov

$$P(\mathbb{X} \geq \beta) \leq \frac{\mathbb{E}[\mathbb{X}]}{\beta} \quad (*)$$

scegliendo $\beta = \alpha(k+\ln(n))w^*$. Valutiamo infine

$$\begin{aligned}P\left(\frac{w}{w^*} \geq \alpha(k+\ln(n))\right) &= P(w \geq \alpha(k+\ln(n))w^*) = \\ &= P(w \geq \beta) \\ &\underset{(*)}{\leq} \frac{\mathbb{E}[w]}{\beta} \leq \frac{(k+\ln(n))w^*}{\alpha(k+\ln(n))w^*} = \frac{1}{\alpha}.\end{aligned}$$

Madonna abbiamo finito. ■

Questo bound è molto largo: possiamo dare un bound più preciso scegliendo un preciso valore di k.

Corollario 3.2.1.1: Per $k = 3$ abbiamo il 45% di probabilità di ottenere una soluzione ammissibile con rapporto di approssimazione $\leq 6 + 2\ln(n)$.

Dimostrazione 3.2.2: Dato il nostro insieme degli eventi possibili, lo possiamo dividere in tre zone:

1. zona degli eventi con soluzione non ammissibile, che indichiamo con E_{na};
2. zona degli eventi con soluzione ammissibile ma con tasso di approssimazione cattivo, ovvero $> 6 + 2\ln(n)$, che indichiamo con E_{bad};
3. zona degli eventi con soluzione ammissibile, che indichiamo con P_{ok}.

La probabilità è uno strumento matematico, definito in teoria delle misure, usato per misurare delle quantità. Noi andremo a misurare queste aree.

Valutiamo in ordine le zone, partendo dalla prima, ovvero

$$P(E_{\text{na}}) \underset{\text{th}}{\leq} e^{-k} = e^{-3}.$$

Valutiamo poi la seconda, ovvero

$$P(E_{\text{bad}}) = P(\text{TA} > 6 + 2\ln(n)) \underset{\alpha=2}{\leq} \frac{1}{2}.$$

Valutiamo infine la terza zona, quella che ci interessa, ovvero

$$P_{\text{ok}} = 1 - P(E_{\text{na}} \cup E_{\text{bad}}) \underset{\text{UB}}{\geq} 1 - (P(E_{\text{na}}) + P(E_{\text{bad}})) = 1 - \left(e^{-3} + \frac{1}{2}\right) \approx 45\%. \qquad \blacksquare$$

Come in tutti gli algoritmi probabilistici, più volte eseguo l'algoritmo e più pompo la probabilità di ottenere un risultato prima ammissibile e poi ottimo.

4. MAX E_k-SAT [$k \geq 3$]

Forse uno dei problemi più famosi del mondo nella sua versione di decisione, esso è definito da:

- **input**: formula CNF in cui ogni clausola contiene esattamente k letterali;
- **soluzioni ammissibili**: assegnamento di valori di verità alle variabili che compaiono nell'input;
- **funzione obiettivo**: numero di clausole soddisfatte dall'assegnamento;
- **tipo**: max.

Le **formule CNF** sono congiunzioni di *«pezzi»*

$$P_1 \wedge ... \wedge P_t$$

e ogni *«pezzo»* P_i è una disgiunzione di k letterali

$$l_1 \vee ... \vee l_k,$$

che possiamo trovare positivi o negativi. I (*droga armi*) pezzi (*che cadono*) sono le **clausole**, che contengono appunto i **letterali**.

Una proprietà interessante delle CNF è che ogni formula logica che coinvolge variabili booleane può essere trasformata in una CNF normale e poi in una E_k-CNF.

Teorema 4.1: MAX E_k-SAT è un problema *NPO-C*.

Dimostrazione 4.1: Per assurdo, supponiamo di saper risolvere MAX E_k-SAT in tempo polinomiale. Sia T il numero di clausole che vengono soddisfatte dall'assegnamento generato dall'algoritmo esatto polinomiale. Ma allora riesco a risolvere il problema di decisione E_k-SAT in tempo polinomiale, semplicemente osservando se T è uguale al numero di clausole. Ma E_k-SAT è un problema *NP-C*, quindi questo è un assurdo. ■

Vediamo quindi un algoritmo probabilistico per questo problema.

Algoritmo mega difficile

input
  Formula CNF in cui ogni clausola contiene k letterali;

1: Assegna ad ogni variabile un valore a caso in {true, false}
2: **output** assegnamento fatto

Teorema 4.2: L'algoritmo mega difficile, data una formula CNF formata da t clausole, ne rende vere almeno

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

cioè

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

Dimostrazione 4.2: Diamo prima un po' di notazione. Siano:

- φ la formula CNF di input;
- k il numero di letterali che compaiono in ogni clausola;
- n il numero di variabili diverse che compaiono in φ;
- $x_1, ..., x_n$ le variabili che compaiono in φ;
- T il numero di clausole che vengono soddisfatte dall'assegnamento.

Introduciamo n variabili aleatorie

$$X_i \sim U(\{0, 1\})$$

distribuite come un modello uniforme sull'insieme $\{0, 1\}$. Ogni variabile X_i indica se la variabile i-esima è vera o falsa.

Introduciamo infine t variabili aleatorie C_j tali che

$$C_j = \begin{cases} 1 & \text{se la clausola } K_j \text{ è soddisfatta} \\ 0 & \text{altrimenti} \end{cases}.$$

Aggiungiamo un teorema alle **COSE**, un bel teorema, il **teorema delle probabilità totali**.

Calcoliamo il valore atteso del numero di clausole coperte come

$$\mathbb{E}[T] = \sum_{b_1 \in \{0,1\}} ... \sum_{b_n \in \{0,1\}} \mathbb{E}[T \mid X_1 = b_1, ..., X_n = b_n] P(X_1 = b_1, ..., X_n = b_n).$$

Con il teorema delle probabilità totali stiamo condizionando ai valori delle variabili X, che coprono tutto il mio insieme di casi possibili. Sistemiamo questa quantità spezzando la probabilità in singole probabilità:

$$\mathbb{E}[T] = \sum_{b_1} ... \sum_{b_n} \mathbb{E}[C_1 + ... + C_t \mid X_1 = b_1, ..., X_n = b_n] P(X_1 = b_1) ... P(X_n = b_n) =$$

$$= \frac{1}{2^n} \sum_{b_1} ... \sum_{b_n} \mathbb{E}[C_1 + ... + C_n \mid X_1 = b_1, ..., X_n = b_n].$$

Uso la linearità del valore atteso, quindi

$$\mathbb{E}[T] = \frac{1}{2^n} \sum_{b_1} ... \sum_{b_n} \sum_{j=1}^{t} \mathbb{E}\left[C_j \mid X_1 = b_1, ..., X_n = b_n\right].$$

Guardiamo il valore atteso. Sto assegnando un valore di verità a tutte le variabili, ma così facendo so esattamente il valore di C_j: dipende infatti dai valori che ho appena assegnato.

Supponiamo di guardare la j-esima clausola, formata da k variabili. Per avere questa clausola falsa, devo assegnare *VERO* a tutte le variabili negative e *FALSO* a tutte le variabili positive, visto

che abbiamo una disgiunzione. Esiste quindi un solo modo per assegnare queste k variabili per avere la clausola falsa. Le altre $n-k$ variabili le assegno a caso.

Facendo ciò abbiamo 2^{n-k} modi per rendere C_j falsa, visto che $n-k$ sono messe a caso con 2 valori possibili. Ma allora $2^n - 2^{n-k}$ sono il numero di modi che rendono vera la clausola.

Sistemiamo il valore atteso, considerando solo i casi che rendono vera la clausola, ovvero:

$$\begin{aligned}\mathbb{E}[T] &= \frac{1}{2^n}\sum_{j=1}^{t}\left(2^n - 2^{n-k}\right) = \\ &= \frac{2^n - 2^{n-k}}{2^n}t.\end{aligned}$$

Moltiplico tutto per 2^{k-n} ottenendo infine

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

Lemma 4.1: Vale

$$\forall j = 0, ..., n \quad \exists b_1, ..., b_j \in \{0,1\} \mid \mathbb{E}\left[T \mid X_1 = b_1, ..., X_j = b_j\right] \geq \frac{2^k - 1}{2^k}t,$$

dove t è il numero di clausole.

Questo lemma è più generale rispetto al teorema precedente: ci sta dicendo che possiamo fissare un numero qualsiasi j di variabili e mantenere comunque il numero di minimo di clausole soddisfatte.

Dimostrazione 4.3: Dimostriamo per induzione sul numero di variabili fissate j.

Passo base: se $j = 0$ sto dimostrando il teorema precedente, dove non fissavo niente.

Passo induttivo: per HP di induzione sappiamo che

$$\mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}\right] \geq \frac{2^k - 1}{2^k}t.$$

Usiamo il teorema delle probabilità totali, condizionando il valore atteso alla variabile X_j, ovvero:

$$\begin{aligned}\mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}\right] = \mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}, X_j = 0\right]P\left(X_j = 0\right) + \\ + \mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}, X_j = 1\right]P\left(X_j = 1\right).\end{aligned}$$

Le due probabilità valgono entrambe $\frac{1}{2}$ essendo le X_i uniformi, quindi

$$\begin{aligned}\mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}\right] = \frac{1}{2}\big(\mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}, X_j = 0\right] + \\ + \mathbb{E}\left[T \mid X_1 = b_1, ..., X_{j-1} = b_{j-1}, X_j = 1\right]\big).\end{aligned}$$

Chiamo A e B i due valori attesi. Per assurdo siano

$$A, B < \frac{2^k - 1}{2^k} t.$$

La loro somma è

$$A + B < \frac{2^k - 1}{2^k} 2t.$$

Ma allora

$$\mathbb{E} = \frac{1}{2}(A + B) < \frac{2^k - 1}{2^k}.$$

Ma questo non è possibile: all'inizio avevamo, per ipotesi di induzione, che

$$\mathbb{E} \geq \frac{2^k - 1}{2^k} t,$$

ma unendo l'ultima disuguaglianza otteniamo un assurdo.

Visto l'assurdo, almeno uno dei due valori attesi deve essere maggiore o uguale a quella quantità, il che dimostra il nostro lemma. ■

Questo lemma ci dà l'idea di quella che è la **de-randomizzazione**: possiamo fare un assegnamento deterministico continuando a garantire il numero di clausole soddisfatte.

Corollario 4.2.1: Esiste un assegnamento deterministico che soddisfa almeno

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

clausole.

Non siamo più nel probabilistico. Ma come determiniamo questo assegnamento di variabili?

Assegnamento deterministico

input

formula φ E_k-CNF
la formula contiene t clausole $K_1 \wedge ... \wedge K_t$
la formula utilizza n variabili $x_1, ..., x_n$

1: $D \leftarrow \varnothing$
2: for $i = 1, ..., n$
3: $\quad \Delta_0 = 0$
4: $\quad \Delta_1 = 0$
5: $\quad \Delta_{D_0} = \varnothing$
6: $\quad \Delta_{D_1} = \varnothing$
7: $\quad$ for $j = 1, ..., t$

Assegnamento deterministico

```
        if j ∈ D
            continue
        if x_i non compare in K_j
            continue
        Sia h il numero di variabili in K_j con indice ≥ i
        if x_i compare positiva in K_j
            Δ_0 = Δ_0 − 1/2^h
            Δ_1 = Δ_1 + 1/2^h
            Δ_{D_1} = Δ_{D_1} ∪ {j}
        else
            Δ_0 = Δ_0 + 1/2^h
            Δ_1 = Δ_1 − 1/2^h
            Δ_{D_0} = Δ_{D_0} ∪ {j}
    if Δ_0 ≥ Δ_1
        x[i] = false
        D ← D ∪ Δ_{D_0}
    else
        x[i] = true
        D ← D ∪ Δ_{D_1}
output x[1], ..., x[n]
```

Cerchiamo di spiegare questo algoritmo:

- il primo ciclo for decide quale variabile i devo assegnare ora;
- il secondo ciclo for scorre le clausole j:
  - l'insieme D contiene le clausole già soddisfatte;
  - se la clausola $j \in D$ o se la variabile $x_i \notin K_j$ allora possiamo andare avanti, visto che la clausola è già soddisfatta o la variabile non è presente.
  - per ogni clausola da controllare, modifico i valori Δ_0 e Δ_1 in base a quante variabili di indice $\geq i$ ho nella clausola corrente; questi valori Δ ci dicono quanto variano i valori attesi delle clausole soddisfatte ponendo vera o falsa la variabile in esame.

Come mai proprio $\frac{1}{2^h}$? Supponiamo di avere h variabili da assegnare per una tale clausola, allora ho

$$\frac{2^h - 1}{2^h}$$

modi di renderla vera. Supponiamo di assegnare *VERO* alla i-esima variabile, che è positiva, ma facendo ciò ho reso vera la clausola, portando a 1 il suo valore atteso. Se invece supponiamo di assegnare *FALSO* alla i-esima variabile, che è negativa, ho $h - 1$ variabili ancora da controllare per renderla vera, ovvero otteniamo

$$\frac{2^{h-1} - 1}{2^{h-1}}.$$

Se calcoliamo Δ_1 (*o l'altra quantità*) come differenza tra il valore dopo e il valore prima del valore atteso, otteniamo

$$1 - \frac{2^h - 1}{2^h} = \frac{1}{2^h}.$$

Gli insiemi Δ_D contano quante clausole andremo a soddisfare con la scelta fatta.

IN SINTESI:

- siamo partiti da un algoritmo probabilistico con una buona *«mira»* nel rendere vere le clausole;
- abbiamo deciso di fissare qualche variabile garantendo comunque la buona *«mira»*;
- abbiamo fatto vedere che possiamo trovare deterministicamente questo assegnamento.

Parliamo brevemente del fattore di approssimazione dell'algoritmo probabilistico: questo sarebbe

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

ma vista la natura probabilistica dell'algoritmo, il fattore di approssimazione andrebbe calcolato utilizzando le probabilità, e questo è abbastanza sbatti.

Ricordandoci però che abbiamo ottenuto un algoritmo deterministico usando la de-randomizzazione, possiamo affermare che il fattore di approssimazione è quello calcolato poco fa.

Parte IV — Teorema PCP e inapprossimabilità di problemi

1. Teorema PCP

Iniziamo con un argomento **molto** (*enfasi sul molto*) divertente, il **teorema PCP**.

Negli anni '70 nasce la **teoria della complessità strutturale**: essa vuole risolvere problemi che non erano ancora stati risolti dividendoli in classi, così da usare soluzioni più astratte e generali dei singoli problemi. La domanda cruciale e molto ambiziosa alla quale cercava (*e cerca ancora oggi*) risposta è la famosissima

$$P = NP\,.$$

Inoltre, purtroppo per noi che dobbiamo studiarle, esiste uno zoo di classi di complessità, e tra queste conosciamo anche qualche relazione, ma le relazioni più importanti e interessanti non sono ancora state risolte (*vedi P e NP*).

Dal **Teorema di Cook** del 1972 (*credo SAT problema NPC*) l'unico gioiello della teoria della complessità è il **teorema PCP** del 1998, ideato da Arora e Safra.

Ma andiamo per ordine, prima del teorema ne dobbiamo mangiare di pasta con il tonno.

Torniamo nel mondo dei problemi di decisione. Una cosa molto bella che possiamo fare è trasformare il problema di decisione in un linguaggio $L \subseteq 2^*$. Ora, dato un input $x \in 2^*$, ci chiediamo se esso appartenga o meno a L. Abbiamo cambiato il problema, è diventato un problema di appartenenza ad un linguaggio, ma che non cambia niente sull'esito del problema.

L'algoritmo che decide questa appartenenza è un **decisore**, che noi rappresentiamo con una MdT, l'oggetto più formale e astratto che rappresenta un algoritmo deterministico.

Le MdT hanno forme alternative: una di queste molto particolare è la **MdT con oracolo**.

L'input di queste macchine è sempre $x \in 2^*$ e l'output è sempre una risposta *SI/NO*. Questa macchina, però, può accedere ad un **oracolo** $w \in 2^*$ durante la sua esecuzione, una stringa binaria. La risposta quindi dipende non solo dall'input, ma anche dall'oracolo, ovvero la MdT diventa $M(x, w)$. L'accesso all'oracolo non è *diretto*, ma la MdT usa un nastro di supporto, detto **nastro delle query**. Ogni volta che la MdT vuole un elemento della stringa dell'oracolo scrive sul nastro delle query la posizione (*in binario*) della posizione da interrogare e la macchina, andando nello **stato di query** $q_?$, estrae l'elemento richiesto.

Pensiamo all'oracolo w come ad una dimostrazione: noi abbiamo in input x e vogliamo verificare che x abbiamo una certa proprietà usando delle informazioni aggiuntive dell'oracolo w.

Vista questa natura, queste macchine sono dette **verificatori**.

Esempio 1.1: Vogliamo costruire una MdT che ci dice se $x \in 2^*$ è primo. Nell'oracolo potrei andare a scrivere un divisore proprio di x. e la MdT deve controllare se questo oracolo ha avuto ragione oppure no. Ovviamente, questo controllo lo fa in tempo polinomiale.

In poche parole, la MdT verifica se l'oracolo sta dando una dimostrazione credibile oppure no.

Come vedremo nel seguente teorema, queste MdT con oracolo non sono altro che delle MdT non deterministiche: infatti, quando le MdT non deterministiche sdoppiavano il loro comportamento, noi possiamo vederlo in una MdT con oracolo come la richiesta di query dall'oracolo.

Teorema 1.1: Un linguaggio $L \subseteq 2^*$ sta in NP se esiste una MdT con oracolo V tale che:
1. $V(x, w)$ lavora in tempo polinomiale in $|x|$;
2. $\forall x \in 2^*$ vale

$$\exists w \in 2^* \mid V(x, w) = \text{SI} \iff x \in L.$$

L'oracolo fa il ruolo del **non determinismo**. Il secondo punto rappresenta *lo sbatti* del non determinismo: comodo che per vedere l'appartenenza devo avere almeno un ramo vero, scomodo che per vedere la non appartenenza devo avere tutti i rami falsi.

Espandiamo ancora la nostra macchina: introduciamo i **verificatori probabilistici**. Essi aggiungono un ingrediente, dei **bit random** (*ma dai*).

Come prima, abbiamo l'input $x \in 2^*$ e l'oracolo $w \in 2^*$ al quale la MdT ha accesso. In questa versione estesa la MdT può accedere anche ad una **sorgente di bit random**. Sulla base di queste tre variabili la MdT deve rispondere *SI/NO*.

A noi interesseranno verificatori probabilistici V per un linguaggio L tali che:
- V lavora in tempo polinomiale in $|x|$;
- se
  - $x \in L$ allora $\exists w \in 2^*$ tale che $V(x, w)$ accetta con probabilità 1. In poche parole, qualsiasi sia la sequenza di bit random la MdT accetta x;
  - $x \notin L$ allora $\forall w \in 2^*$ abbiamo che $V(x, w)$ rifiuta con probabilità $\geq \frac{1}{2}$. In poche parole, in almeno metà delle sequenze di bit random la MdT rifiuta x.

Vista la proprietà 1, la dimensione dei bit random ai quali accediamo è polinomiale.

Abbiamo quindi due ingredienti, uno **oracolico** (*non deterministico*) e uno **randomico**.

Questi due ingredienti serviranno per definire la classe di complessità PCP e il teorema PCP. Infatti, **PCP** è l'acronimo di **Probabilistically Checkable Proof**, ovvero una dimostrazione (*oracolo*) che possiamo controllare in modo probabilistico (*random*).

Date due funzioni $r, q : \mathbb{N} \longrightarrow \mathbb{N}$ chiamiamo

$$\text{PCP}[r, q]$$

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 alla sorgente di bit random.

Vediamo qualche classe importante che possiamo definire a partire da questo *template*.

La prima classe che creiamo è

$$\text{PCP}[0, 0] = P.$$

Infatti, non potendo fare query e non potendo usare dei bit random, la decisione dipende solamente dall'input x che abbiamo dato alla MdT.

La seconda classe che creiamo è

$$\text{PCP}[0, \text{poly}] = NP\,.$$

Infatti, non possiamo usare dei bit random ma abbiamo accessi polinomiali all'oracolo, che è esattamente quello che fa una MdT non deterministica. Inoltre, non avendo delle probabilità, questa macchina accetta con probabilità 1 e rifiuta con probabilità 1, nel senso che non esistono probabilità intermedie (*prima avevamo* $\frac{1}{2}$).

Teorema 1.2 (*Teorema PCP*): Vale

$$NP = \mathrm{PCP}[O(\log(n)), O(1)].$$

Ho un trade-off rispetto alla definizione di NP che abbiamo dato prima del teorema:

- voglio avere accesso ai bit random ($0 \rightsquigarrow O(\log(n))$);
- per far ciò devo rinunciare ad un buon numero di query (poly $\rightsquigarrow O(1)$).

Vediamo come questo passaggio renda *«più potente»* la componente randomica: passo da polinomiale a costante, ma per compensare ciò mi basta solo un fattore logaritmico. Abbiamo appena affermato la superiorità della componente randomica sulla componente non deterministica.

Quello che è sorprendente è quel $O(1)$: stiamo dicendo che, dato un linguaggio, esiste un numero (*e anche una funzione logaritmica, ma quella chissene*) perfetto per quel linguaggio e tale che ogni stringa di input viene accettata (*almeno un ramo*) o rifiutata (*ogni ramo*) entro quel numero di query.

Per verificare che è un numero va bene per un dato problema bisogna costruire un verificatore che:

- $\forall x \in 2^*$ allora V usa al massimo le risorse definite dalle funzioni r e $q1$;
- se $x \in L$ allora V accetta con probabilità 1 qualsiasi sia la stringa di bit randomici;
- se $x \notin L$ allora V rifiuta con probabilità $\geq \frac{1}{2}$ qualsiasi sia l'oracolo.

Tutto bello, ma a cosa serve il teorema PCP? Noi lo useremo per fornire delle **dimostrazioni di inapprossimabilità** (*a meno di certe costanti*) usando delle riduzioni a PCP.

Facciamo un'ultima restrizione a queste macchine. Stiamo creando un verificatore $V \in \mathrm{PCP}[r, q]$ che, su input $x \in 2^*$, faccia esattamente:

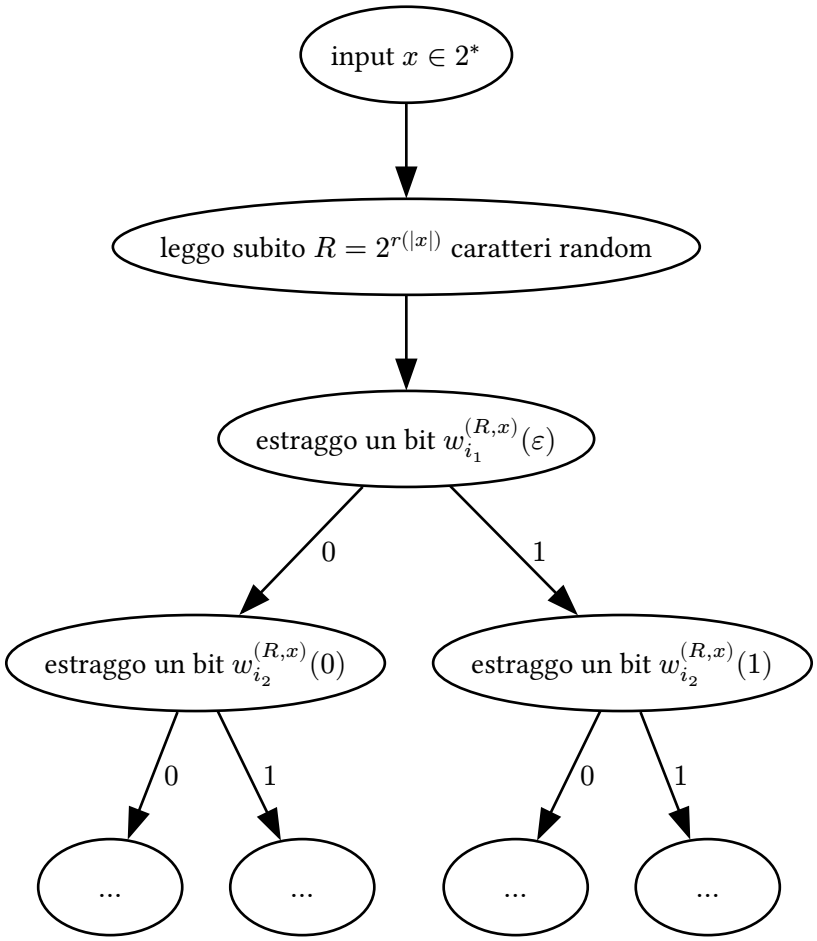
- $q(|x|)$ query;
- $r(|x|)$ estrazioni di bit random.

Se il verificatore V fa di meno query/estrazioni facciamo eseguire a V delle query/estrazioni a vuoto. Questo comportamento sicuramente non modifica il comportamento del verificatore.

Immaginiamo un verificatore

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

per NP. Questa macchina funziona nel seguente modo:



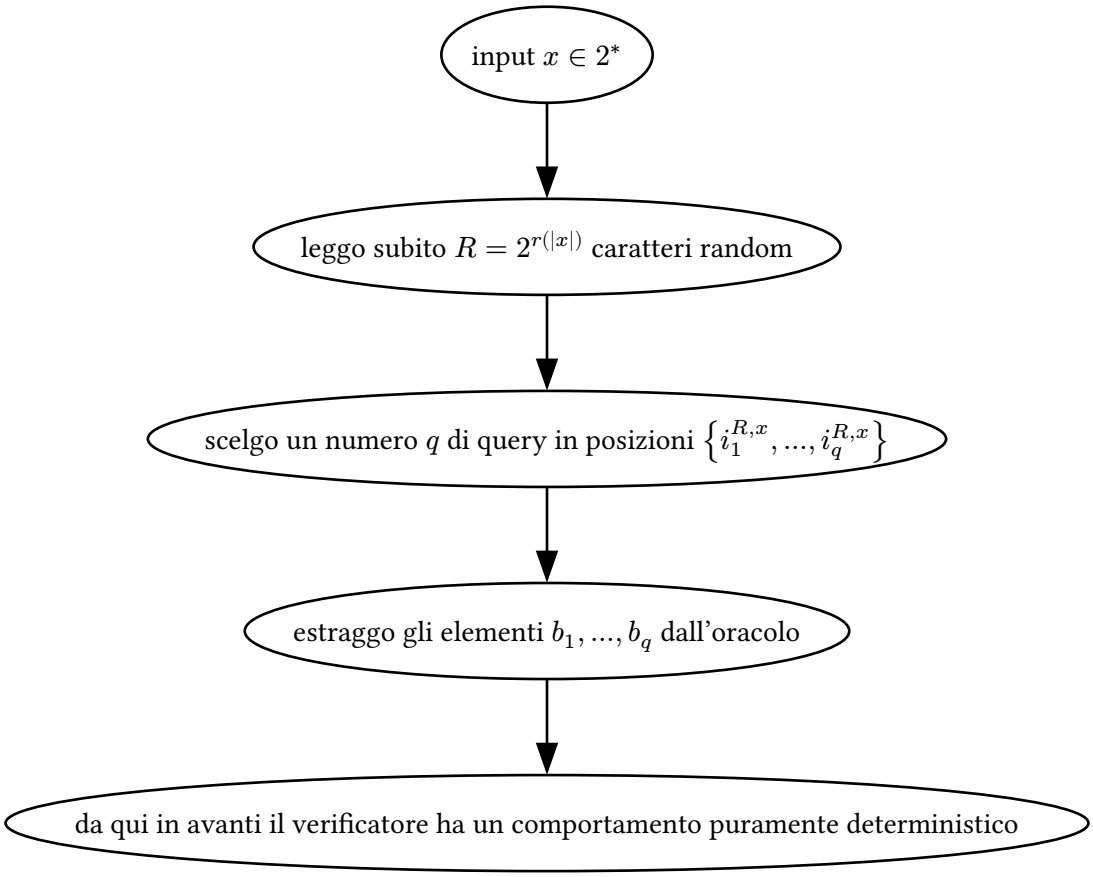
La quantità $w_{i_t}^{(R,x)}\left(w_{i_1}, ..., w_{i_{t-1}}\right)$ indica il bit estratto dall'oracolo sapendo R, x e tutti i bit estratti precedentemente. Quello che si va a creare è un mega albero di computazione.

Questa proprietà si chiama **adattività**, che caratterizza un **verificatore adattivo**. Quest'ultimo è un verificatore le cui risposte dell'oracolo dipendono sì dai bit random e dall'input ma anche da quello che abbiamo ottenuto prima come estrazioni. Ma a noi non piace un verificatore adattivo.

L'altezza dell'albero mostrato sopra è q, ottenuto facendo una serie di richieste all'oracolo. Il numero totale di richieste è

$$\overline{q} = 2^{q-1} + 2^{q-2} + ... + 2^1 + 2^0 = \sum_{i=0}^{q-1} 2^i.$$

Per rimuovere l'adattività faccio subito all'oracolo tutte le richieste $\overline{q}$ e, da quel momento, finisco la computazione in maniera deterministica usando le query fatte. Questa nuova versione del verificatore si chiama **verificatore non adattivo**.



Sto chiedendo tutte le query subito, poi faccio partire la computazione senza interrogare di nuovo l'oracolo. Ho sicuramente un costo, ovvero un aumento del numero di query, ma questo è comunque un numero che è costante.

Esempio 1.2: Siano:

- $z = 10110110$;
- $r(|z|) = 2$;
- $R = 01$;
- $q = 3$;
- $I = \{i_3, i_{15}, i_{27}\}$.

Estraggo i bit w_3, w_{15}, w_{27}. Ho quindi 2^3 casi possibili:

w_3	w_{15}	w_{27}	
0	0	0	N
0	0	1	S
0	1	0	S
0	1	1	N
1	0	0	S
1	0	1	S
1	1	0	S
1	1	1	N

Questa tabella esprime il comportamento che seguirà il verificatore dopo l'estrazione. Il comportamento espresso è un comportamento puramente deterministico.

2. Inapprossimabilità di MAX E_k-SAT

Vediamo come il teorema PCP può aiutarci a dimostrare l'inapprossimabilità di alcuni problemi. Partiamo con MAX E_k-SAT, che avevamo introdotto parlando di algoritmi probabilistici.

Lemma 2.1: Ogni k-CNF ($k \geq 3$) con t clausole si può trasformare in una 3-CNF con $(k-2)t$ clausole, preservando la soddisfacibilità.

Questo lemma in realtà è un se e solo se: se rendo vera una k-CNF allora lo stesso assegnamento rende vera la 3-CNF (*assegnando correttamente le variabili ausiliarie, se serve*). Il contrario vale se limitiamo la soddisfacibilità alle sole variabili non ausiliarie: le variabili ausiliarie vengono inserite positive e negative, quindi una clausole che contiene una variabile positiva avrà una clausola associata con la variabile negativa che, per essere soddisfatta, si deve affidare alle variabili che avevamo già all'inizio.

Teorema 2.1: Vale

$$\forall k \geq 3 \quad \exists \varepsilon > 0$$

tale che MAX E_k-SAT non è $(1+\varepsilon)$-approssimabile.

Questo teorema ci dice che non possiamo avvicinarci a 1 quanto vogliamo.

Dimostrazione 2.1 *(Per $k = 3$)*: Scegliamo un linguaggio $L \in NP\text{-}C$. Viste le relazioni tra classi che abbiamo visto a inizio corso, vale $L \in NP$ e quindi anche

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

Per assurdo, immaginiamo di avere un algoritmo di approssimazione per MAX E_3-SAT che approssima con tasso di approssimazione $\alpha < 1 + \varepsilon$, con

$$\varepsilon = \frac{1}{2q2^q}.$$

Sia V il verificatore per il linguaggio L. Il verificatore è non adattivo, ovvero:

- prende in input $z \in 2^*$;
- estrae una stringa di bit random $R \in 2^{r(|z|)}$;
- sceglie delle posizioni $i_1^{z,R}, ..., i_q^{z,R}$;
- ottiene le risposte $b_1, ..., b_q$.

Ci convinciamo del fatto che si può scrivere una q-CNF, che chiamiamo $\Psi_{z,R}$ soddisfacibile se e solo se V accetta z. Questa CNF dipende dalle variabili logiche $w_0, w_1, ...$ dove w_i è vera se e solo se l'i-esimo carattere dell'oracolo è 1.

Per ottenere una CNF prendo la tabella dipendente dalle variabili w_i e, selezionando le righe dei *NO*, creo una CNF formata dalle variabili naturali se in quella riga sono false e dalle variabili negate se in quella riga sono vere.

Esempio 2.1: Nell'esempio della lezione scorsa, dove viene spiegato il PCP, la CNF risultante è

$$\Phi = (w_3 \vee w_{15} \vee w_{27}) \wedge (w_3 \vee \neg w_{15} \vee \neg w_{27}) \wedge (\neg w_3 \vee \neg w_{15} \vee \neg w_{27}).$$

La formula $\Psi_{z,R}$ è una q-CNF con al massimo 2^q clausole, ovvero il numero combinazioni di q letterali. Trasformo $\Psi_{z,R}$ in una 3-CNF $\varphi_{z,R}$ con al massimo $q2^q$ clausole, per il lemma precedente. Consideriamo, per semplicità, che abbia esattamente $q2^q$.

Definiamo la formula

$$\Phi_z = \bigwedge_{R \in 2^{|z|}} \varphi_{z,R}.$$

Questa è un'enorme 3-CNF con al massimo $q2^q 2^{r(|z|)}$ clausole. Come prima, per semplicità, consideriamo la formula con esattamente quel numero di clausole.

Il numero di clausole è enorme, ma è comunque polinomiale:

- $q2^q$ è un numero, molto grande, ma che dipende solo dal numero q che definisce il linguaggio;
- $2^{r(|z|)}$ è un esponenziale di un logaritmo, quindi è un polinomio.

Do in pasto questa formula al mio algoritmo per MAX E_3-SAT.

Abbiamo due casi:

- se $z \in L$ allora $\exists w$ che fa accettare V con probabilità 1, e sono soddisfatte $q2^q 2^{r(|z|)}$ clausole;
- se $z \notin L$ allora $\forall w$ il nostro V rifiuta con probabilità $\geq \frac{1}{2}$. Andiamo a quantificare il massimo numero di clausole soddisfatte. Dividiamo l'insieme $2^{r(|z|)}$ in due insiemi: il primo di questi accetta la stringa z, il secondo invece non accetta la stringa z, ma per non accettare una stringa basta che una clausola non sia soddisfatta, quindi al massimo ho

  $$\underbrace{\frac{2^{r(|z|)}}{2} q2^q}_{\text{metà accetto}} + \underbrace{\frac{2^{r(|z|)}}{2}(q2^q - 1)}_{\text{metà non accetto}} = \frac{2^{r(|z|)}}{2}(q2^q + q2^q - 1) = q2^q 2^{r(|z|)} - \frac{2^{r(|z|)}}{2}$$

  clausole soddisfatte.

Il nostro algoritmo è approssimato, quindi:

- se $z \in L$ ci verrà dato un risultato

  $$\geq \frac{q2^q 2^{r(|z|)}}{1+\varepsilon};$$

- se $z \notin L$ ci verrà dato un risultato

  $$\leq q2^q 2^{r(|z|)} - \frac{2^{r(|z|)}}{2}.$$

Vediamo se questi due disequazioni danno degli insiemi che non si sovrappongono.

Siano $A = q2^q$ e $B = 2^{r(|z|)}$. Per assurdo sia

$$AB - \frac{B}{2} > \frac{AB}{1+\varepsilon},$$

ovvero per assurdo i due insiemi si sovrappongono. Ma allora

$$AB + AB\varepsilon - \frac{B}{2} - \frac{B}{2}\varepsilon - AB > 0$$
$$\varepsilon\left(AB - \frac{B}{2}\right) - \frac{B}{2} > 0$$
$$\frac{1}{2A}\left(AB - \frac{B}{2}\right) - \frac{B}{2} > 0$$
$$\frac{B}{2} - \frac{B}{4A} - \frac{B}{2} > 0$$
$$-\frac{B}{4A} > 0$$

che è impossibile visto che stiamo considerando solo quantità positive.

Ma allora i due insiemi non si sovrappongono, quindi riesco a decidere se z appartiene o meno a L in tempo polinomiale, ma questo non è possibile perché $P \neq NP$. ■

3. Inapprossimabilità di MAX Independent Set

Non abbiamo visto il problema **MAX Independent Set** durante il corso, quindi vediamo rapidamente la sua definizione formale:

- **input**: grafo $G = (V, E)$ non orientato;
- **soluzione ammissibile**: un **insieme indipendente**, ovvero una **anti-cricca**, un sottoinsieme $X \subseteq V$ tale che

$$\binom{X}{2} \cap E = \varnothing;$$

- **funzione obiettivo**: $|X|$;
- **tipo**: max.

Ovviamente, questo è un problema *NPO-C*.

Teorema 3.1: Per ogni $\varepsilon > 0$ MAX Independent Set non è $(2 - \varepsilon)$-approssimabile.

Non riusciamo quindi ad approssimare MAX Independent Set meglio di 2.

Dimostrazione 3.1: Sia L un linguaggio *NP-C* a nostra scelta, ovvero

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

Fissato $z \in 2^*$, il nostro verificatore:

- genera una sequenza di bit random $R \in 2^{r(|z|)}$;
- genera delle posizioni $i_1^{z,R}, ..., i_q^{z,R}$;
- fa le query all'oracolo all'oracolo, ottenendo $b_1, ..., b_q$.

Tutti questi dati formano una z-**configurazione**: essa è una coppia

$$\left(R, \left\{i_1^{z,R} : b_1, ..., i_q^{z,R} : b_q\right\}\right).$$

Il numero di queste configurazioni dipende dal numero di scelte di R e, fissata una di queste, anche dal numero di scelte dell'oracolo.

Costruiamo G_z un grafo non orientato, dove:

- i vertici sono le z-configurazioni **accettanti**, ovvero le configurazioni che mi portano ad accettare z nel linguaggio L;
- i lati tra due vertici v e v' esistono se e solo se:
  - $R = R'$ oppure
  - $\exists k, k' \in \{1, ..., q\}$ tali che $i_k^{z,R} = i_{k'}^{z,R'} \wedge b_k \neq b_{k'}$. In poche parole, esiste una posizione in comune che ha avuto due risposte diverse dall'oracolo. Questo lato in particolare evidenzia una **relazione di inconsistenza**.

Il grafo ha un numero di vertici **polinomiale**: i bit random sono $2^{r(|z|)}$ ma la funzione $r(|z|)$ è logaritmica quindi ho un fattore polinomiale in $|z|$. Il numero di possibili risposte è 2^q che però è un numero. Quindi il numero di vertici è polinomiale.

Supponiamo per assurdo di avere un algoritmo approssimato per MAX Independent Set con grado di approssimazione $\alpha < 2 + \varepsilon$.

Abbiamo due casi:

- se $z \in L$ il MAX Independent Set ha dimensione

$$\geq 2^{r(|z|)}$$

per il Lemma 3.1, ma noi abbiamo in mano un algoritmo $(2-\varepsilon)$-approssimante, quindi quello che ci viene restituito è un valore

$$\geq \frac{2^{r(|z|)}}{2-\varepsilon};$$

- se $z \notin L$ il MAX Independent Set ha dimensione

$$\leq 2^{r(|z|)-1}$$

per il Lemma 3.2.

Questi insiemi sono disgiunti: infatti,

$$2^{r(|z|)-1} = \frac{2^{|z|}}{2} < \frac{2^{|z|}}{2-\varepsilon}$$

quindi sto decidendo in tempo polinomiale l'appartenenza di una stringa in un linguaggio *NP-C*, ma questo è impossibile se $P \neq NP$, quindi non esistono algoritmi α-approssimanti per MAX Independent Set. ■

Vediamo la dimostrazione dei due lemmi che abbiamo usato nella dimostrazione.

Lemma 3.1: Se $z \in L$ allora G_z ha un Independent Set di dimensione $\geq 2^{r(|z|)}$.

Dimostrazione 3.2: Se $z \in L$ allora $\exists \overline{w}$ che fa accettare con probabilità 1. Prendiamo tutte le configurazioni accettanti $\left(R, \left\{i_1^{z,R} : b_1, ..., i_q^{z,R} : b_q\right\}\right)$ consistenti con $\overline{w}$. Devo scegliere l'unico nodo che ha le risposte uguali ai bit di $\overline{w}$. Visto che abbiamo $2^{r(|z|)}$ possibili R, l'insieme di tutti questi vertici ha cardinalità $2^{r(|z|)}$. Per semplicità, noi consideriamo che questa cardinalità sia almeno quel valore. Inoltre, tutti questi elementi sono non collegati da archi, visto che sono consistenti, quindi è anche un Independent Set. ■

Lemma 3.2: Se $z \notin L$ ogni Independent Set di G_z ha cardinalità $\leq 2^{r(|z|)-1}$.

Dimostrazione 3.3: Per assurdo, supponiamo che S sia un Independent Set di G_z con cardinalità $> 2^{r(|z|)-1}$.

S contiene tutti nodi che hanno R diversi tra loro e, considerando le risposte dell'oracolo, a parità di indice ho sicuramente la stessa risposta, altrimenti avremmo un arco tra i due vertici e quindi non sarebbe un Independent Set.

A fronte di S si può costruire un $\overline{w}$ compatibile con tutti i vertici di S. Stiamo quindi accettando con probabilità

$$> 2^{r(|z|)-1} = \frac{2^{r(|z|)}}{2}$$

ovvero stiamo accettando con più della metà delle possibili stringhe random. Noi però non stiamo accettando z, quindi almeno metà delle configurazioni non devono accettare per definizione, quindi questo è un assurdo. ■

Parte V — Strutture succinte

1. Introduzione alle strutture succinte

Le **strutture succinte** sono oggetti estremamente piccoli, talmente piccoli che andremo a contarne i bit per ridurli il più possibile. Come vedremo però, ci sarà un limite a tutto ciò.

Una **struttura dati** è un **Abstract Data Type** (*ADT*): esso rappresenta un insieme di primitive che devono essere implementate. In base a come implementiamo i metodi, avremo una certa occupazione in spazio e un certo dispendio temporale. Le implementazioni, ovviamente, non differiscono per funzioni, ma differiscono solo per complessità in spazio e tempo.

Le strutture succinte sono SD che occupano poco spazio. Per il tempo andremo nella soluzione *«a babbo»*, ovvero *«basta che vadano»*. In poche parole vogliamo tutto: SD piccola ma molto veloce. Di solito, questo trade-off è difficile da avere nella realtà.

Teorema 1.1 (*Information-theoretical lower bound*): Data una struttura dati di taglia n, che contiene V_n valori possibili $v_1, ..., v_{V_n}$, ognuno dei quali utilizza $x_1, ..., x_{V_n}$ bit, il **teorema di Shannon** afferma che

$$\frac{x_1 + ... + x_{V_n}}{V_n} \geq \log_2(V_n).$$

Quel fattore $\log_2(V_n)$ si chiama **information-theoretical lower bound**, e ci dà una dimensione media minima in bit che serve per codificare una struttura dati di V_n valori di taglia n.

Chiamiamo Z_n questo bound sul numero di bit. Sia D_n il numero di bit utilizzati per rappresentare una struttura. Allora la struttura è:

- **implicita** se $D_n = Z_n + O(1)$;
- **succinta** se $D_n = Z_n + o(Z_n)$;
- **compatta** se $D_n = O(Z_n)$.

2. Rank e Select

Vogliamo implementare l'ADT che definisce il comportamento **Rank e Select**.

Dato un array $b \in 2^n$ di n bit, abbiamo due operazioni:

- la primitiva **rank** conta il numero di 1 prima di una data posizione, ovvero
$$\text{rank}_b(p) = |\{i \mid i < p \wedge b_i = 1\}|;$$
- la primitiva **select** dice in che posizione è presente il k-esimo 1 (*a partire da* 0), ovvero
$$\text{select}_b(k) = \max\{p \mid \text{rank}_b(p) \leq k\}.$$

Valgono due **proprietà**:

$$\text{rank}_b(\text{select}_b(i)) = i$$
$$\text{select}_b(\text{rank}_b(i)) \geq i \wedge \text{select}_b(\text{rank}_b(i)) = i \iff b_i = 1$$

Le strutture che considereremo sono dette **statiche**: una volta costruito l'oggetto, esso non viene più modificato dalle primitive, quindi è buona cosa costruire in maniera intelligente queste strutture.

Le strutture di controllo (*tabelle*) che usiamo sono dette **indici**. Il tempo di costruzione di questi indici non lo considereremo mai.

Vediamo due implementazioni naive di questa ADT:

- se non costruisco niente lo spazio utilizzato dalla struttura è 0 ma il tempo è lineare perché ogni volta devo calcolarmi i valori di rank e select scorrendo l'array;
- se costruisco le tabelle di rank e select per intero occupiamo $2n \log_2(n)$ bit di spazio ma abbiamo tempo di accesso $O(1)$ per avere le risposte.

Per salvare un array di n bit ci servono

$$\geq \log_2(D_n) = \log_2(2^n) = n \text{ bit}.$$

2.1. Struttura di Jacobson per Rank

La struttura di Jacobson è stata pensata negli anni '80 ed è una **struttura multi-livello**.

Dato l'array b lo divido in **super-blocchi** di lunghezza $(\log(n))^2$. Ogni super-blocco poi lo dividiamo in **blocchi** di lunghezza $\frac{1}{2}\log(n)$.

Ogni super-blocco S_i memorizza quanti 1 sono contenuti dall'inizio del vettore fino a S_i (*escluso*). Invece, ogni blocco B_{ij} memorizza quanti 1 ci sono dall'inizio del super blocco S_i fino a se stesso (*escluso*). Vediamo l'occupazione in memoria di queste strutture.

La **tabella dei super-blocchi** ha $\frac{n}{(\log(n))^2}$ righe e contiene dei valori che sono al massimo lunghi (secondo me sono $\left(\log(n)^2\right)$, e invece sono) n, che in bit sono $\log(n)$. La grandezza di questa tabella è quindi

$$\frac{n}{(\log(n))^2}\log(n) = \frac{n}{\log(n)} = o(n).$$

La **tabella dei blocchi** ha $\frac{n}{\frac{1}{2}\log(n)}$ righe e contiene dei valori che sono al massimo lunghi $(\log(n))^2$, che in bit sono $\log\left(\log(n)^2\right)$. La grandezza di questa tabella è quindi

$$\frac{n}{\frac{1}{2}\log(n)}\log(\log(n)^2) = \frac{4n\log(\log(n))}{\log(n)} = o(n).$$

Manca una cosa: cosa succede se ci viene chiesta una posizione interna al blocco?

Usiamo il **four-russians trick**, il *trucco dei quattro russi.*

Per ogni possibile configurazione dei blocchi, che sono $2^{\frac{1}{2}\log(n)}$, memorizziamo la tabella di rank di quella configurazione. Abbiamo quindi $\frac{1}{2}\log(n)$ righe, che contengono valore massimo $\frac{1}{2}\log(n)$, che in bit sono $\log\left(\frac{1}{2}\log(n)\right)$. L'occupazione totale di tutte queste strutture è

$$2^{\frac{1}{2}\log(n)}\frac{1}{2}\log(n)\log\left(\frac{1}{2}\log(n)\right) = \sqrt{n}\log\left(\sqrt{n}\right)\log\left(\log\left(\sqrt{n}\right)\right) = o(n).$$

La struttura di Jacobson è quindi **succinta**.

2.2. Struttura di Clarke per Select

La **struttura di Clarke per Select** è stata ideata molto dopo rispetto alla struttura di Jacobson.

Come funziona questa struttura? È ancora una **struttura multi-livello**, che salverà le entry della select *«ogni tanto»*, ogni tot multipli.

2.2.1. Primo livello

Il **primo livello** della struttura memorizza le select dei valori multipli di $\log(n)\log(\log(n))$. Il numero di righe di questa tabella è

$$\frac{n}{\log(n)\log(\log(n))}.$$

Per ogni riga indico la posizione dell'1 richiesto, quindi al massimo n, che in bit è $\log(n)$. La dimensione di questa tabella è quindi

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

2.2.2. Secondo livello

Stiamo salvando delle posizioni p_i, ognuna delle quali indica la posizione dell'$[i\log(n)\log(\log(n))]$-esimo bit 1 del vettore. Ragioniamo su p_{i+1} e p_i. La successione è necessariamente crescente, e inoltre la differenza non può essere più bassa del multiplo, perché di mezzo ho esattamente quel valore moltiplicativo. Quindi

$$p_{i+1} - p_i \geq \log(n)\log(\log(n)).$$

Questa differenza ci dice quanto sono sparsi i nostri 1 nel blocco. Se ho l'uguale allora ho solo 1, altrimenti ne ho di meno. In poche parole, questo valore indica la **densità** degli 1.

Il **secondo livello** dipende dalla densità $r_i = p_{i+1} - p_i$. Ho due casi da considerare:

- se $r_i \geq (\log(n)\log(\log(n)))^2$ ho una densità bassa di 1, si dice che il vettore in quello span è **sparso**. La tabella della select viene memorizzata esplicitamente. Quanto occupiamo in memoria? Gli 1 da memorizzare sono quelli tra un pezzo e l'altro, quindi $\log(n)\log(\log(n))$ righe. La posizione che scrivo dentro è al massimo r_i (*l'indice dell'*1 *parte dall'inizio del blocco*), quindi $\log(r_i)$ in bit. Come spazio totale otteniamo

$$\log(n)\log(\log(n))\log(r_i) = \frac{(\log(n)\log(\log(n)))^2}{\log(n)\log(\log(n))}\log(r_i) \leq \frac{r_i\log(r_i)}{\log(n)\log(\log(n))} \underset{r_i \leq n}{\leq} \frac{r_i}{\log(\log(n))};$$

- se $r_i < (\log(n)\log(\log(n)))^2$ ho una densità alta, si dice che il vettore in quello span è **denso**, ho tantissimi 1. In questo caso memorizzo le posizioni multiple di $\log(r_i)\log(\log(n))$. Quanta memoria serve? Sto usando

$$\frac{\log(n)\log(\log(n))}{\log(r_i)\log(\log(n))}\log(r_i) = \log(n) \leq \frac{r_i}{\log(\log(n))}$$

bit di memoria.

In entrambi i casi analizzati usiamo al massimo, nel secondo livello, un numero di bit uguale a

$$\frac{r_i}{\log(\log(n))}$$

per ogni blocco, quindi

$$\begin{aligned}\frac{r_0}{\log(\log(n))} + \frac{r_1}{\log(\log(n))} + ... = \frac{p_1 - p_0}{\log(\log(n))} + \frac{p_2 - p_1}{\log(\log(n))} + ... = \\ = \frac{p_n - p_0}{\log(\log(n))} \leq \frac{n}{\log(\log(n))} = o(n).\end{aligned}$$

2.2.3. Terzo livello

Nel caso denso mi mancano comunque degli 1 da indicizzare, che sono quelli nelle posizioni non multiple. Serve un **terzo livello**. Nel secondo livello ho memorizzato le posizioni s_i^j del blocco che inizia in i. Come prima, vediamo la differenza $t_i^j = s_i^{j+1} - s_i^j$. Ricordiamoci che siamo nel caso denso, quindi vale $r_i < (\log(n)\log(\log(n)))^2$. Vale inoltre

$$t_i^j \geq \log(r_i)\log(\log(n)).$$

Anche qui abbiamo due casistiche:

- se $t_i^j \geq \log\left(t_i^j\right)\log(r_i)(\log(\log(n)))^2$ siamo ancora nel caso **sparso**. Come prima, memorizzo tutte le tabelle esplicitamente. Che occupazione abbiamo? Il numero di righe è $\log(r_i)\log(\log(n))$, ognuna di queste tiene una quantità che è al massimo t_i^j (*contiamo sempre da inizio blocco*), quindi in bit sono $\log\left(t_i^j\right)$. L'occupazione totale è quindi

$$\log(r_i)\log(\log(n))\log\left(t_i^j\right) = \frac{\log\left(t_i^j\right)\log(r_i)(\log(\log(n)))^2}{\log(\log(n))} \leq \frac{t_i^j}{\log(\log(n))};$$

- se $t_i^j < \log\left(t_i^j\right)\log(r_i)(\log(\log(n)))^2$ usiamo il **four-russians trick**: per questa struttura prendiamo tutte le possibili tabelle di select. Osserviamo che

$$\begin{aligned}\log\left(t_i^j\right) \underset{\text{DEF}}{\leq} \log(r_i) \underset{\text{DEF}}{\leq} \log\left((\log(n)\log(\log(n)))^2\right) = \\ = 2\log(\log(n)\log(\log(n))) = 2\log(\log(n)) + 2\log(\log(\log(n))) \leq 4\log(\log(n)).\end{aligned}$$

  Quindi

$$\begin{aligned}t_i^j < \log\left(t_i^j\right)\log(r_i)(\log(\log(n)))^2 \\ \leq 4\log(\log(n))4\log(\log(n))(\log(\log(n)))^2 \leq 16(\log(\log(n)))^4.\end{aligned}$$

Le tabelle del four-russians trick hanno t_i^j righe, ognuna che contiene valori che sono al massimo t_i^j (*come solito, partiamo dal blocco corrente a contare*), che in bit sono $\log\left(t_i^j\right)$. Le tabelle sono $2^{t_i^j}$ quindi lo spazio occupato è

$$2^{t_i^j} t_i^j \log\left(t_i^j\right) \leq 2^{16(\log(\log(n)))^4} 16(\log(\log(n)))^4 \log(16(\log(\log(n)))^4) = o(n).$$

Come prima, se sommiamo per ogni blocco j abbiamo una somma **telescopica** che ci porta a $o(n)$.

3. Alberi binari

3.1. Introduzione

La parola **albero** è molto *overloaded*, ovvero viene usata per esprimere tanti concetti diversi che però hanno delle differenze.

In **teoria dei grafi**, un albero è *un grafo non orientato aciclico connesso*. Se cade la richiesta di connessione siamo di fronte ad una **foresta**.

Nell'**informatica classica**, un albero è uguale a quello della teoria dei grafi ma, una volta scelto un nodo come radice, l'intero albero viene *«appeso per gravità»* a questa radice. In teoria dei grafi questi si chiamerebbero **alberi radicati**.

Possiamo distinguere gli alberi in base ad alcuni fattori:

- **numero di figli**:
  - un albero si dice **generico** se ogni nodo ha un numero arbitrario di figli;
  - un albero si dice n-ario se ogni nodo ha n oppure 0 nodi;
- **ordinamento**:
  - un albero si dice **ordinato** se, per ogni nodo v dell'albero, i nodi nel sotto-albero sinistro sono $<$ di v e i nodi del sotto-albero destro sono $>$ di v;
  - un albero si dice **non ordinato** se è tutto a caso.

Un albero generico di solito è anche ordinato, perché è molto più facile da salvare e rappresentare.

Un **albero binario** è un albero 2-ario. I nodi dell'albero si chiamano **nodi** e sono di due tipi:

- i nodi che hanno figli sono detti **nodi interni**;
- i nodi che non hanno figli sono detti **foglie** o **nodi esterni**.

Se vogliamo considerare anche alberi che hanno nodi con un figlio andiamo ad aggiungere delle *foglie fittizie* per completare l'albero.

Sia T un albero. Indichiamo con:

- $E(T)$ il numero di nodi esterni dell'albero;
- $I(T)$ il numero di nodi interni dell'albero.

Teorema 3.1.1: Per ogni albero binario T vale

$$E(T) = I(T) + 1.$$

Dimostrazione 3.1.1: Dimostriamo questo teorema per induzione strutturale.

Caso base: albero binario banale T.

L'albero T è formato dalla sola radice. Esso ha

$$E(T) = 1 \quad | \quad I(T) = 0,$$

ma allora $1 = 0 + 1$, quindi la relazione è verificata.

Passo ricorsivo: albero binario T formato dalla radice R e due sotto-alberi T_1 (*sx*) e T_2 (*dx*).

Guardiamo l'albero T: esso è formato dai due **alberoni** T_1 e T_2, quindi

$$E(T) = E(T_1) + E(T_2) \quad | \quad I(T) = I(T_1) + I(T_2) + R = I(T_1) + I(T_2) + 1.$$

Per ipotesi induttiva valgono

$$E(T_1) = I(T_1) + 1 \wedge E(T_2) = I(T_2) + 1.$$

Ma allora

$$E(T) = E(T_1) + E(T_2) = I(T_1) + I(T_2) + 1 + 1 = I(T) + 1.$$ ■

Il **numero totale di nodi** è $2n + 1$.

Teorema 3.1.2: Ci sono

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

alberi binari con n nodi interni.

Il numero C_n è detto n-esimo **numero di Catalano**. Quanto vale C_n? Lo possiamo approssimare?

Usiamo l'**approssimazione di Stirling**: essa afferma che

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Grazie a questa approssimiamo C_n con

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

Calcoliamo Z_n come

$$\begin{aligned}
Z_n &= \log_2(C_n) = \log_2(4^n) - \log_2\left(\sqrt{\pi n^3}\right) = \\
&= 2n - \frac{1}{2}\log_2(\pi n^3) = 2n - \left(\frac{1}{2}\log_2(\pi) + \frac{3}{2}\log_2(n)\right) = 2n - O(\log(n)).
\end{aligned}$$

3.2. Struttura succinta per alberi binari

Per ora ignoriamo i dati contenuti nei nodi, memorizziamo solo la struttura.

Numeriamo i nodi a partire dall'alto andando verso il basso, partendo da sinistra e andando verso destra. In poche parole, numera facendo una ricerca in ampiezza. I numeri partono da 0.

Sia n il numero di nodi interni. Usiamo un array binario b di $2n+1$ elementi, che nella cella i avrà 1 se il nodo è interno, altrimenti conterrà 0.

Le **operazioni** che vogliamo saper fare su questo albero sono:
- dato un nodo, voglio sapere i suoi figli;
- dato un nodo, voglio sapere se è una foglia.

Immaginiamo di avere un albero T e consideriamo un sotto-albero T' di T, scelto con la stessa radice di T. Dato un nodo p di T', voglio sapere quali sono i figli di questo nodo. Per come abbiamo numerato l'albero, basta contare i nodi alla sua destra fino al figlio sinistro (*figlio di sinistra*) e poi basta fare $+1$ (*figlio di destra*). Mi serve quindi il numero di nodi di T', ma questo lo sappiamo, è due volte i nodi interni più uno, ovvero $2n'+1$.

Quanti sono questi nodi? Noi sappiamo il numero di interni, ovvero il rank, quindi $2\text{rank}_b(p)+1$, e questo è il figlio sinistro di p, l'altro è uguale ma ha $+2$.

Per sapere se è una foglia basta vedere se il vettore in posizione p è 0.

Per sapere il mio genitore, devo chiedermi chi è $p' \mid p'$ genitore di p, qui mi serve la select. Sappiamo che questo p' è tale che $2\text{rank}_b(p')+1=p$ oppure $2\text{rank}_b(p')+2=p$, quindi

$$\text{rank}_b(p') = \left\lfloor \frac{p}{2} - \frac{1}{2} \right\rfloor.$$

Applichiamo la select ad entrambi i membri e otteniamo

$$p' = \text{select}_b\left(\left\lfloor \frac{p}{2} - \frac{1}{2} \right\rfloor \right).$$

Per i **dati ancillari**, ovvero i dati dei nodi, usiamo un array parallelo al vettore dei nodi che tiene i dati, ma questo approccio non è molto succinto. Se i dati sono contenuti sono nei nodi interni, uso un vettore indicizzato di lunghezza uguale ai nodi interni al quale accedo tramite rank sul vettore che già possediamo.

Quanta memoria stiamo occupando? Stiamo utilizzando:
- un array di $2n+1$ bit;
- una struttura di rank e select che occupa $o(2n+1) = o(n)$.

Sappiamo che $Z_n = 2n - O(\log(n))$, quindi abbiamo una **struttura succinta** perché occupiamo $2n$ a meno di un o-piccolo.

4. Codice di Elias-Fano per sequenze monotone

Nei web crawler, quando salvo il grafo del web, per ogni nodo ho una lista dei successori, che dice quali a pagine punta la pagina corrente. Vogliamo ordinare queste sequenze in maniera crescente, e useremo il **codice di Elias-Fano** per fare tutto ciò.

Data una sequenza di valori $0 \leq x_0 \leq \ldots \leq x_{n-1} < u$ vogliamo memorizzare questa lista.

Calcoliamo

$$l = \max\left\{0, \left\lfloor \log_2\left(\frac{u}{n}\right)\right\rfloor\right\}.$$

Memorizziamo esplicitamente gli l bit inferiori (*meno significativi*)

$$l_0 = x_0 \bmod 2^l \quad | \quad \ldots \quad | \quad l_{n-1} = x_{n-1} \bmod 2^l$$

di ogni numero. I bit superiori, che sono

$$\left\lfloor \frac{x_0}{2^l} \right\rfloor \quad | \quad \ldots \quad | \quad \left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor$$

li memorizziamo come differenze, ovvero memorizziamo

$$u_i = \left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor$$

assumendo che $x_{-1} = 0$.

La sequenza iniziale è non decrescente, quindi tutte queste differenze sono maggiori o uguali a 0.

Questi valori vengono memorizzati in **unario** in un unico vettore u. Scrivere in unario vuol dire tanti zeri quanto vale u_i e poi un uno. Quanto occupiamo in memoria?

Per i bit inferiori ci servono ln bit, ovvero n valori diversi ognuno di l bit. Per i bit superiori ci serve

$$\begin{aligned}
\sum_{i=0}^{n-1} u_i + 1 &= \sum_{i=0}^{n-1} \left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor + 1 = \\
&= n + \sum_{i=0}^{n-1} \left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor = \\
&= \text{la serie è telescopica} = \\
&= n + \left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor - \left\lfloor \frac{x_{-1}}{2^l} \right\rfloor = \\
&= n + \frac{x_{n-1}}{2^l} \leq n + \frac{u}{2^l} = n + \frac{u}{2^{\lfloor \log_2(\frac{u}{n}) \rfloor}} \leq n + \frac{u}{2^{\log(\frac{u}{n})-1}} = n + \frac{2u}{\frac{u}{n}} = 3n.
\end{aligned}$$

In totale abbiamo quindi

$$D_n = ln + 3n = (l+3)n = \left(\left\lfloor \log_2\left(\frac{u}{n}\right)\right\rfloor + 3\right)n = 2n + n\left\lceil \log_2\left(\frac{u}{n}\right)\right\rceil.$$

Cosa ci rappresenta l'operazione $\text{select}_u(i)$? Ci dice dove è l'i-esimo 1, ma il vettore u contiene una serie di zeri unari e poi un uno che fa da separatore. Quindi

$$\begin{aligned}\text{select}_b(i) &= i + u_0 + \ldots + u_i = \\ &= i + \left\lfloor \frac{x_0}{2^l} \right\rfloor + \left(\left\lfloor \frac{x_1}{2^l} \right\rfloor - \left\lfloor \frac{x_0}{2^l} \right\rfloor \right) + \ldots = \\ &= \text{serie telescopica} = \\ &= i + \left\lfloor \frac{x_i}{2^l} \right\rfloor.\end{aligned}$$

Vogliamo calcolare l'information theoretical lower bound. Fissati n e u, quante sono le sequenze monotone $0 \leq x_0 \leq \ldots \leq x_{n-1} < u$ possibili?

Vogliamo sapere quanti insiemi di cardinalità n ci sono nell'insieme $\{0, \ldots, u-1\}$. Questo è abbastanza complicato, quindi usiamo un trick. Un altro modo per vedere questa domanda è chiedersi quante soluzioni ha, in $\mathbb{N}^n$, l'equazione

$$c_0 + c_1 + \ldots + c_{n-1} = n.$$

Le c_i sono delle variabili che ci dicono quante volte nella sequenza delle x compare il numero x_i.

Per contare si usa la **tecnica delle stelle e delle barre** (*stars & stripes counting*).

Esempio 4.1: La sequenza

$$**||*|||*|$$

indica che ci sono 8 numeri da selezionare e che viene scelto il primo numero 2 volte e il terzo e il settimo numero 1 volta.

In generale, le barre vengono usate per dividere i numeri, e le stelle vengono usate per indicare il numero di x_i che sono presenti nell'insieme. Devo avere n stelle, purché le metta in mezzo alle barre. Le barre sono $u-1$ e le stelle sono n. Quante sequenze di n stelle e $u-1$ barre esistono?

Ho $u+n-1$ caratteri nello stringone, devo scegliere dove sono le barre, quindi ho

$$\binom{u+n-1}{u-1} = \frac{(u+n-1)!}{(u-1)!(u+n-1-u+1)!} = \frac{(u+n-1)!}{n!(u-1)!}$$

possibili scelte. Quindi

$$Z_n = \log_2\left(\binom{u+n-1}{u-1}\right) = \log_2\left(\binom{u+n-1}{n}\right) = \log_2\left(\frac{(u+n-1)!}{n!(u-1)!}\right).$$

Sappiamo che

$$\log\left(\binom{A}{B}\right) \sim B\log\left(\frac{A}{B}\right) + (A-B)\log\left(\frac{A}{A-B}\right).$$

Quindi

$$Z_n = \log\left(\binom{u+n-1}{n}\right) \sim n\log\left(\frac{u+n-1}{n}\right) + (u-1)\underbrace{\log\left(\frac{u+n-1}{u-1}\right)}_{=0 \text{ trascurabile se } n \ll u} =$$

$$= n\log\left(\left(\frac{u}{n}\right)\left(1+\frac{n}{u}-\frac{1}{u}\right)\right) =$$

$$= n\log\left(\frac{u}{n}\right) + n\log\left(1+\frac{n}{u}-\frac{1}{u}\right) =$$

$$= \text{sappiamo che } x \approx \log(1+x) =$$

$$\approx n\log\left(\frac{u}{n}\right) + n\left(\frac{n}{u}-\frac{1}{u}\right) =$$

$$\leq n\log\left(\frac{u}{n}\right) + \frac{n^2}{u}.$$

Assumiamo, per semplicità, che $n \leq \sqrt{u}$, ma allora

$$Z_n = n\log\left(\frac{u}{n}\right) + 1.$$

Ricordiamoci che

$$D_n = 2n + n\left\lceil \log\left(\frac{u}{n}\right)\right\rceil + o(n).$$

Ma allora $D_n = O(Z_n)$. Non abbiamo una struttura succinta ma almeno abbiamo lo stesso ordine di grandezza, quindi è una **struttura compatta**.

5. Hash minimali perfetti

Le funzioni hash sono funzioni molto rudimentali: partono da qualsiasi universo U e ci ritornano, a prescindere da U, un valore nell'insieme $\{0, ..., m-1\}$. In poche parole sono funzioni nella forma

$$h : U \longrightarrow m.$$

Il valore m indica il numero di **bucket**, ovvero il numero di contenitori che usa la funzione hash per distribuire i valori di U. Perché usiamo i bucket? Perché di solito $U \gg m$ e quindi esistono sicuramente delle **collisioni**, ovvero dei valori che vengono mappati nello stesso bucket.

Imponiamo dei requisiti alle funzioni hash:

1. h si calcoli in tempo costante;
2. h sia *molto iniettiva*, ovvero non possiamo avere una funzione h iniettiva ($U \gg m$) ma almeno chiediamo che h spalmi bene i valori di U nei vari bucket, mettendone circa $\frac{U}{m}$ in ognuno.

Sia $X \subseteq U$ con $|X| = n$. Vogliamo avere una funzione hash h che non generi collisioni su X, ovvero

$$\forall x, y \in U \quad x \neq y \Longrightarrow h(x) \neq h(y).$$

Questa cosa è molto difficile, ecco perché chiediamo la *«molto iniettiva»*.

Costruiamo una struttura dati che utilizza una funzione hash h scelta uniformemente a caso. Possiamo farlo? Sì, le funzioni hash sono finite, quindi posso farlo, ma questo in realtà è impossibile:

1. l'insieme di tutte le funzioni hash è enorme, dove lo mettiamo?
2. se scelgo a caso, potrei non avere una funzione che lavora in tempo o in spazio costante.

Una cosa che si fa spesso è associare ad ogni carattere della stringa un **peso**, e poi calcolare la somma pesata modulo la grandezza massima. Questo ci permette di avere infinite funzioni hash, modificando solamente il vettore dei pesi.

Un'altra soluzione è la **full randomness assumption**: assumiamo che h sia scelta veramente a caso, e che non ci siano questi problemi.

Per permettere tutto ciò andremo a restringere la famiglia di funzioni hash ad un sottoinsieme pià maneggevole e che abbia delle buone proprietà.

Sia

$$H_{U,m} \subseteq m^U$$

insieme dal quale scelgo uniformemente a caso. Gli elementi di H sono *belli* perché

$$\forall t \in m \quad \forall x \in U \quad P[h(x) = t] = \frac{1}{m}.$$

In poche parole, la probabilità per un elemento di U di finire in un bucket è uguale per ogni bucket e questa proprietà vale per ogni elemento di U.

Una funzione hash $h : U \longrightarrow m$ è **perfetta** per l'insieme $X \subseteq U$ se e solo se h è iniettiva su X, ovvero deve valere $|X| \leq m$.

Una funzione hash $h : U \longrightarrow m$ è **minimale** se $|X| = m$.

La probabilità di avere un hash perfetto, scegliendo a caso, è molto bassa, e questa scende ancora di più se consideriamo un hash minimale.

5.1. Tecnica MWHC su grafi

La **tecnica MWHC** (*Majewski, Worwald, Havaś e Czech*) è stato presentato in un articolo del 1980 con l'obiettivo di risolvere un problema legato agli **hash minimali perfetti ordinati** (*OPPMH, Ordered Preserving Perfect Minimal Hash*), ovvero hash che mappano in un ordine deciso.

Sia U l'insieme universo e sia $X \subseteq U$ con $|X| = n$. Vogliamo una funzione che associ ad ogni $x_i \in X$ un valore di r bit. Fissiamo anche $m \geq n$. Scegliamo a caso due funzioni

$$h_1, h_2 : U \longrightarrow m$$

usando il trucchetto dei pesi.

Costruiamo un grafo. Esso ha:

- **vertici** indicizzati da $\{1, ..., m\}$;
- **archi** tra i vertici i e j se e solo se

$$\exists x \in X \mid \{h_1(x), h_2(x)\} = \{i, j\}.$$

In poche parole, inseriamo un arco tra due vertici se otteniamo i da h_1 e j da h_2 a parità di input.

Cosa non vogliamo che succeda:

1. $h_1(x) = h_2(x)$, ovvero ho un **cappio**;
2. considerando x, y tali che $x \neq y$ otteniamo lo stesso arco, ovvero abbiamo un multigrafo;
3. il grafo è ciclico.

Se succede anche solo una di queste cose buttiamo via le due funzioni hash e ne scegliamo altre due (*ovvero cambiamo i pesi*). Ma le troveremo prima o poi due funzioni h_1 e h_2 che ci vanno bene?

Teorema 5.1.1: Se $m > 2.09n$ le funzioni hash h_1 e h_2 hanno quasi sempre le proprietà desiderate e il numero di tentativi attesi è ≈ 2.

Vogliamo assegnare ad ogni $x \in X$ un valore $f(x) \in 2^r$ di r bit. Etichettiamo l'arco tra due nodi con il valore $f(x)$, dove x è il valore di X che ha generato quell'arco.

Andiamo a scrivere G come un sistema di equazioni del tipo

$$(A[h_1(x)] + A[h_2(x)]) \bmod 2^r = f(x) \quad \forall x \in X$$

dove $A[1], ..., A[m]$ sono variabili. Se il grafo è aciclico, allora il sistema ammette una soluzione.

La soluzione di questo sistema lo andiamo a salvare in un vettore chiamato A.

Grazie a questo posso costruire la funzione hash richiesta: infatti, non so come ma funziona, se richiediamo $f(x)$ con $x \in X$, basta calcolare la somma delle posizioni di A trovate usando h_1 e h_2.

Che occupazione in spazio abbiamo?

Il valore di m è scelto, quindi non lo salvo, come non salvo le funzioni h_1 e h_2. L'unica quantità da salvare è il vettore delle soluzioni A, formato da m entry che contengono valori di r bit, quindi abbiamo una occupazione di

$$mr = 2.09nr$$

bit in totale.

Questa funzione hash che abbiamo creato è perfetta per quello che ci serve, ma cosa succede se calcoliamo la funzione ottenuta in un valore che non abbiamo previsto?

Vista la natura **key-less** di questo *«dizionario»*, una query con valori non previsti ci restituisce comunque un valore, che dipende dal vettore delle soluzioni, ma è completamente senza senso.

Nonostante questo piccolo svantaggio, un vantaggio di questa soluzione è che l'occupazione di memoria è veramente minima.

5.2. Tecnica MWHC su iper-grafi

La tecnica MWHC funziona bene anche sugli iper-grafi, ovvero grafi che sono formati da iper-archi (*archi formati da più archi, Eulero perché mi fai questo*).

La scelta di m per un I-grafo dipende sempre da n e da un fattore numerico γ. Per:

- $I = 1$ vale il γ precedente;
- $I = 2$ abbiamo un γ minore di $I = 1$;
- $I = 3$ abbiamo il γ migliore, ovvero $\gamma = 1.23$;
- $I \geq 4$ il valore di γ torna a salire.

Negli iper-grafi dobbiamo modificare la nozione di aciclicità con la nozione di **peelability**.

Un iper-grafo (V, E) ammette una **peeling sequence** se e solo se esiste una sequenza di coppie arco-vertice $(e_1, x_1), ..., (e_n, x_n)$ tale che:

- $x_i \in e_i$;
- $\forall j < i \quad x_i \notin e_j$.

I valori x_i sono detti **hinge**, ovvero i perni. Per trovare una peeling sequence basta risolvere un sistema di equazioni diofantee abbastanza semplice.

Anche in questo caso, abbiamo

$$\gamma n r$$

bit di memoria occupati da A, ma in questo caso possiamo fare meglio.

Le equazioni diofantee da risolvere sono n, una per ogni lato, e gli hinge, che sono al massimo n, sono gli unici valori non zero che dobbiamo salvare dentro il vettore A. Al posto di salvare quest'ultimo per intero, salvo solo le entry non nulle e uso un vettore di bit sul quale uso rank e select per ricavare la posizione di queste entry.

Le entry non nulle sono al massimo n e contengono un valore che è di r bit, quindi mi servono nr bit. Il vettore di bit è lungo quanto il numero di vertici, quindi mi servono γn bit. Infine, la struttura di rank e select occupa $o(n)$ bit. L'occupazione totale, in bit, è quindi

$$nr + \gamma n + o(n).$$

Quando ci conviene questa soluzione? Chiediamoci per quali valori di r vale

$$\begin{aligned} \gamma n r &> nr + \gamma n \\ \gamma n r - nr &> \gamma n \\ nr(\gamma - 1) &> \gamma n \\ r &> \frac{\gamma}{\gamma - 1}. \end{aligned}$$

Considerando $\gamma = 1.23$, otteniamo

$$r > 5.$$