Algoritmi e complessità

Indice

1. Lezione 01 [26/09] 4
1.1. Notazione 4
1.1.1. Insiemi numerici 4
1.1.2. Monoide libero 4
1.1.3. Funzioni 4
1.2. Algoritmi 101 5
2. Lezione 02 [03/10] 8
2.1. Upper e lower bound 8
2.2. Classi di complessità [1] 8
2.3. Problemi di ottimizzazione 10
2.4. Classi di complessità [2] 12
3. Lezione 03 14
3.1. Max Matching 14
3.2. Load Balancing 16
4. Lezione 04 [10/10] 19
4.1. Ancora Load Balancing 19
4.2. Center Selection 20
5. Lezione 05 [11/10] 24
5.1. Ancora Center Selection 24
5.2. Set Cover 26
5.2.1. Funzione armonica 26
5.2.2. Definizione del problema 27
6. Lezione 06 [17/10] 29
6.1. Ancora Set Cover 29
6.2. Vertex Cover 31
7. Lezione 07 [18/10] 36
7.1. Ancora Vertex Cover 36
7.2. Problema dei cammini disgiunti (disjoint paths) 37
8. Lezione 08 [24/10] 41
8.1. Vertex Cover, il ritorno 41
8.1.1. Programmazione lineare, intera e non 41
8.1.2. Vertex Cover con arrotondamento 41
9. Lezione 09 [25/10] 44
9.1. Implementazione di Vertex Cover 44
9.2. La nascita della teoria dei grafi 44
10. Lezione 10 [07/11] 46
10.1. Problema del commesso viaggiatore [TSP] 46
10.2. Algoritmo di Christofides 47
11. Lezione 11 [14/11] 51
11.1. Inapprossimabilità di TSP 51
11.2. Un PTAS per 2-LoadBalancing 52
11.3. Introduzione a Knapsack 54
12. Lezione 12 [14/11] 55
12.1. Knapsack 55

12.1.1. Prima versione 56
12.1.2. Seconda versione 56
12.1.3. FPTAS per Knapsack 57
13. Lezione 13 [21/11] 61
14. Lezione 14 [22/11] 62
14.1. Algoritmi probabilistici 62
14.2. Taglio minimo globale [mincut] 62

1. Lezione 01 [26/09]

1.1. Notazione

1.1.1. Insiemi numerici

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

1.1.2. Monoide libero

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

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

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

  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 $(\mathbb{N}, *)$
- dotata di $\cdot$ **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 $|\varepsilon| = 0$.

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.

1.1.3. Funzioni

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, ha cardinalità $|B|^{|A|}$.

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, $0 = \varnothing$, $1 = \{0\}$, $2 = \{0, 1\}$, eccetera.

Date queste due definizioni, vediamo qualche insieme particolare.

Indichiamo con 2^A l'insieme

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

ovvero l'insieme delle funzioni che classificano gli elementi di un 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.

Indichiamo con A^2 l'insieme

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

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

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

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^*, $\{\varepsilon, 0, 00, 000, ...\}$, eccetera.

1.2. Algoritmi 101

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 $\text{Sol}_\Pi : I_\Pi \longrightarrow 2^{O_\Pi}/\{\varnothing\}$; usiamo l'insieme delle parti come codominio perché potrei avere più risposte corrette per lo stesso problema.

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

Un algoritmo per Boldi è una **Macchina di Turing**. Sappiamo già come è fatta, ovvero:

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

Perché usiamo una MdT quando abbiamo a disposizione una macchina a registri (RAM, WHILE, lambda-calcolo)?

La **tesi di Church-Turing** afferma un risultato molto importante che però 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 $y \in \text{Sol}_\Pi(x)$, ovvero quello che mi restituisce l'algoritmo è sempre la risposta corretta.

Ma tutti i problemi sono risolvibili? No, grazie Mereghetti.

Questo lo vediamo con le cardinalità:

- i problemi di decisione sono i problemi dell'insieme 2^{2^*}, ovvero data una stringa binaria (il nostro input) devo dire se essa sta o meno nell'insieme; 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.

Una volta che abbiamo ristretto il nostro studio ai solo problemi risolvibili (noi considereremo solo quelli) 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

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

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

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

tale che

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

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» (grande cit.).

Abbiamo altre soluzioni? Sì, ma non sono il massimo:

- 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.

Useremo inoltre la **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 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.

2. Lezione 02 [03/10]

2.1. Upper e lower bound

Fissato Π un problema, qual è la complessità di Π? Mi interessa la complessità del problema, non del singolo algoritmo che lo risolve: in poche parole, sto chiedendo quale sia la complessità del migliore algoritmo che lo risolve.

Questa è quella che chiamiamo **complessità strutturale**: non guardo i singoli algoritmi ma i problemi nel loro complesso.

Durante questo studio 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; in matematichese, 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, una prova.

Piccolo appunto: dobbiamo stare comunque attenti alle costanti dentro O e Ω, quindi prendiamo tutte le complessità un po' con le pinze.

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 nella zona grigia, menomale molti sono solo nella «zona polinomiale», purtroppo molti sono solo nella «zona esponenziale».

2.2. Classi di complessità [1]

Viene più comodo creare delle **classi di problemi** e 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, che ha l'istruzione esotica

$$x = ?$$

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 assegnazione 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 $x_i \quad i = 1, ..., n$ 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, quindi se $NP \subseteq P$ e quindi se $P = NP$. Questo problema è stato definito da **Cook**, che affermava di «avere 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 2.2.1: Se $\Pi_1 \underset{p}{\leq} \Pi_2$ e $P_2 \in P$ allora $P_1 \in P$.

Teorema 2.2.2 (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

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

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 2.2.2.1: Se $\Pi \in \textit{NP-C}$ e $\Pi \in P$ allora $P = NP$.

Questo corollario ci permette di ridurre la ricerca ai soli problemi in NP-C.

2.3. 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

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

  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 $T_\Pi \in \{\min, \max\}$.

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

- **input**: formula logiche in forma normale congiunta;
- **soluzioni ammissibili**: data φ formula, 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 tirerebbe fuori 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 $\widetilde{\Pi}$ con:
- **input**: $I_{\widetilde{\Pi}} = \{(x,k) \mid x \in I_\Pi \wedge k \in \mathbb{N}\}$;
- **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.

Vediamo il problema $\widetilde{\text{MAX-SAT}}$:
- **inputs**: $I = \{(\varphi, k) \mid \varphi \text{ formula CNF} \wedge k \in \mathbb{N}\}$;
- **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 2.3.1: Se $\Pi \in PO$ allora il suo problema di decisione associato $\widetilde{\Pi} \in P$.

Corollario 2.3.1.1: Se $\widetilde{\Pi} \in NP\text{-}C$ allora $\Pi \notin PO$.

Noi useremo spesso problemi che hanno problemi di decisione associati *NP-C*. Ci sono dei problemi in *PO*? Certo: i problemi di programmazione lineare sono tutti problemi in *PO*, ma noi ne vedremo almeno un altro di questa classe.

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

Una possibile soluzione sono le **euristiche**, però non sappiamo se funzionano bene o funzionano male, perché magari 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 $\mathrm{opt}_\Pi(x)$ il valore ottimo della funzione obiettivo su input x. Dato un algoritmo approssimato per Π, ovvero un algoritmo tale che

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

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

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

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' esotica 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.

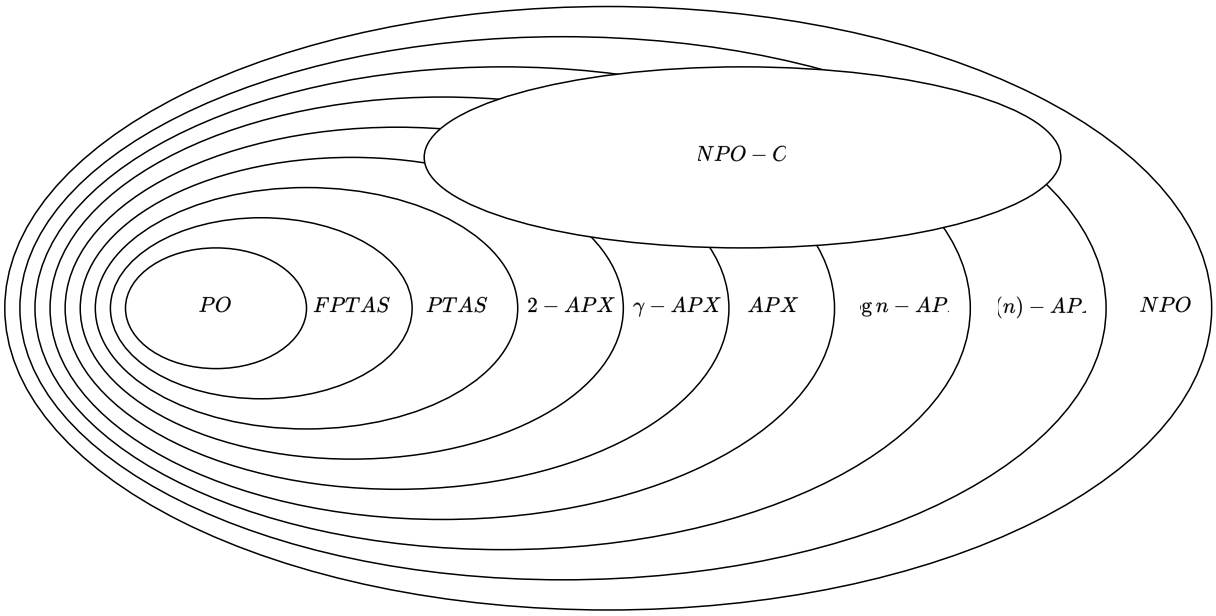
Se $R_\Pi = 1$ allora l'algoritmo su quell'input è ottimo e mi dà la soluzione migliore. Se, ad esempio, $R_\Pi = 2$, allora ho ottenuto

- 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.

2.4. Classi di complessità [2]

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.

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

Rompiamo l'ordine e vediamo infine altre due classi:

- *PTAS* (*Polynomial Time Approximation Scheme*): classe che contiene gli algoritmi approssimabili a piacere;
- *FPTAS* (*Fully Polynomial Time Approximation Scheme*): classe che mantiene un tempo polinomiale al ridursi dell'errore, visto che più l'errore è vicino a 1 e più tempo ci metto.

3. Lezione 03

3.1. Max Matching

Il primo (*e unico*) problema di ottimizzazione in PO che vedremo sarà il problema di **Max Matching**.

Questo problema 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 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, quindi siamo

POCO LGBTQ+ FRIENDLY (godo)

- **soluzioni ammissibili**: una soluzione ammissibile è un **matching** M, una scelta di lati tale che i vertici del grafo risultano incisi al più da un lato. In poche parole, **viva i matrimoni non poligami**, dobbiamo far sposare persone che si piacciono e solo una volta, ma accettiamo anche i single (*come me*);
- **obiettivo**: numero di match $|M|$;
- **tipo**: massimizzare il numero di match, quindi 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 3.1.1: Se esiste un cammino aumentante per il matching M allora M non è massimo.

Lemma 3.1.2: Se il matching M non è massimo allora esiste un cammino aumentante per M.

Dimostrazione 3.1.1: 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

```
for x in V:
    c(x) ← W
F ← {x_seed}
c(x_seed) ← G
while F ≠ ∅:
    x ← pick(F)
    visit(x)
    c(x) ← B
    for y in neighbor(x):
        if c(y) == W:
            F ← F ∪ {y}
            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

```
X ← vertici esposti in M
for x in X:
    BFS(x) con alternanza di lati in M e lati fuori da M
    se durante la ricerca trovo un altro vertice di X
        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 3.1.1: Bipartite Max Matching è in PO.

Corollario 3.1.1.1: 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.

3.2. Load Balancing

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

- **input**: abbiamo tre dati:
  - $m > 0$ 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**: max.

Teorema 3.2.1: Load Balancing è NPO-completo.

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:

- $\Pi \in NPO$;
- $\widetilde{\Pi} \in NP$-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

```
for i in m:
    A_i ← ⦸ (task assegnate alla macchina)
    L_i ← 0 (carico della macchina)
for j in n:
    i ← arg min_{i∈m} L_i (indice macchina con meno carico)
    A_i ← A_i ∪ {j}
    L_i ← L_i + t_j
α assegna ogni elemento 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 3.2.2: Greedy Load Balancing è una 2-approssimazione per Load Balancing.

Dimostrazione 3.2.1: Chiamiamo L^* il valore della funzione obiettivo nella soluzione ottima.

Osserviamo che:

1. vale la relazione
$$L^* \geq \frac{1}{m} \sum_j t_j,$$
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 la relazione

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

ovvero una macchina deve impiegare almeno il tempo più grande tra quelli disponibili.

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

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

Sommiamo rispetto al numero di macchine, quindi

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

Dividiamo tutto per m e otteniamo

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

Sappiamo che

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

quindi

$$R_{\text{GLB}} = \frac{L}{L^*} \leq 2.$$

■

4. Lezione 04 [10/10]

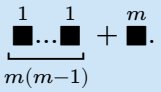
4.1. Ancora Load Balancing

L'algoritmo Greedy Load Balancing non è **tight**, ovvero posso costruire un input che si avvicini molto all'approssimazione alla quale appartiene il problema.

Teorema 4.1.1: $\forall \varepsilon > 0$ esiste un input per Load Balancing tale che Greedy Load Balancing produce un output

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

Dimostrazione 4.1.1: 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 4.1.2: Sorted Greedy Load Balancing è una $\frac{3}{2}$-approssimazione per Load Balancing.

Dimostrazione 4.1.2: 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 $\widetilde{i}$ la macchina tale che $L_{\widetilde{i}} = L$ indice della macchina con carico massimo. Se $\widetilde{i}$ ha un compito solo, la soluzione è ottima.

Consideriamo allora $\widetilde{i}$ con più di un compito: sia $\widetilde{j}$ l'ultimo compito assegnato a quella macchina. So che $\widetilde{j} \geq m$, perché le prime m task le do ad ogni macchina i distinta, allora

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

Ma allora

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

■

Graham nel 1969 ha poi dimostrato che questo algoritmo è una $\frac{4}{3}$-approssimazione.

Hochbaum e Shmoys hanno dimostrato, nel 1988, che:

- Load Balancing è un problema *PTAS*;
- Load Balancing non è un problema *FPTAS*.

4.2. 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 e un 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:
  $S \subseteq \Omega \mid |S| = n > 0$
  $k > 0$
  $r \in \mathbb{R}^{>0}$

$C \leftarrow \varnothing$ insieme dei centri scelti
while $S \neq \varnothing$:
  $\overline{s} \leftarrow$ take-any-from(S)
  $C \leftarrow C \cup \{\overline{s}\}$
  for s in S tali che $d(s, \overline{s}) \leq 2r$:
    $S \leftarrow S / \{s\}$
if $|C| \leq k$
  output C
else
  output «Impossibile»

Teorema 4.2.1: Se Center Selection Plus emette un output allora esso è una $\frac{2r}{\rho^*}$-approssimazione.

Dimostrazione 4.2.1: 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

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

ma allora

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

Teorema 4.2.2: Se $r \geq \rho^*$ l'algoritmo emette un output.

Dimostrazione 4.2.2: 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 la prossima lezione per dimostrare alcune proprietà.

Center Selection Plus v2

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

1: $C \leftarrow \{\overline{s}\}$ per un certo $\overline{s} \in S$
2: while true:
3: if $\exists \overline{s} \in S \mid d(\overline{s}, C) > 2r$:
4: $C \leftarrow C \cup \{\overline{s}\}$

Center Selection Plus v2

```
    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 $|S| \leq k$:
    output S
$C \leftarrow \{\overline{s}\}$ per un certo $\overline{s} \in S$
while $|C| < k$:
    seleziona $\overline{s} \in S$ che massimizzi $d(\overline{s}, C)$
    $C \leftarrow C \cup \left\{\overline{S}\right\}$
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.

5. Lezione 05 [11/10]

5.1. Ancora Center Selection

Ricordiamo il nostro obiettivo: scegliere dei centri che minimizzino il raggio di copertura, ovvero il massimo tra tutte le distanze punto-centro più vicino.

Vediamo come l'algoritmo della scorsa lezione approssima il mio Center Selection.

Teorema 5.1.1: Greedy Center Selection è una 2-approssimazione per Center Selection.

Dimostrazione 5.1.1: 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 $\widetilde{s} \in S$ tale che $d(\widetilde{s}, C) > 2\rho^*$.

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

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

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 $\widetilde{s}$ tale che $d(\widetilde{s}, C) > 2\rho^*$.

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

Teorema 5.1.2: Se $P \neq NP$ non esiste un algoritmo polinomiale che α-approssimi Center Selection per qualche $\alpha < 2$.

Dimostrazione 5.1.2: Usiamo il problema NP-C Dominating Set (*addome degli insetti*):

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

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

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.

Quando ho distanza 1? Devo scegliere C con $|C| \leq k$ tale che tale che

$$\forall s \in S \quad d(s,C) \leq 1,$$

ovvero tutti i punti sono a distanza 1 dai loro centri. Ma allora $\exists C^* \subseteq S$ tale che

$$\min_{c \in C^*} d(s,c) = 1.$$

Essendo a distanza possiamo dire che $\exists C^* \subseteq S$ tale che

$$\forall s \in S \quad \exists c \in C^* \mid sc \in E.$$

Questa è esattamente la definizione di Dominating Set, e il nostro Dominating Set è esattamente C^*.

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 dico devo rispondere *SI* al problema di Dominating Set, nel secondo caso devo rispondere *NO*.

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

5.2. Set Cover

5.2.1. Funzione armonica

Il prof dell'aula accanto è un chad:

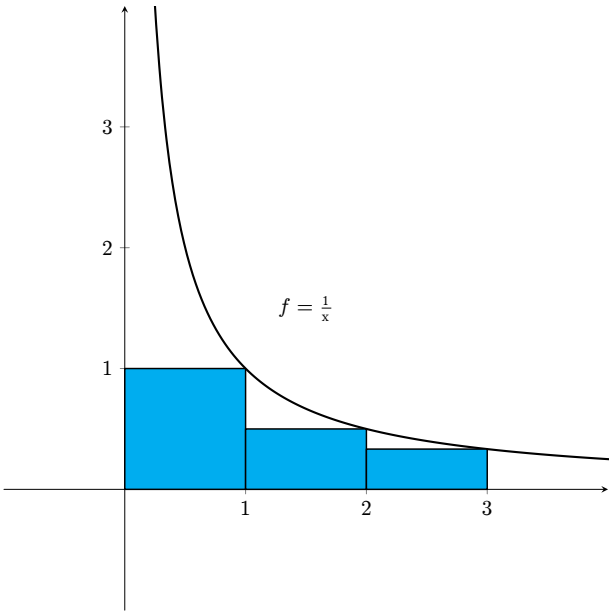
non è qui per trasmettere informazioni, ma per trasmettere emozioni

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

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

Questa la possiamo capire dal grafico: tutta l'area tra i rettangoli e la funzione $\frac{1}{x}$ la andiamo a prendere nella maggiorazione.

La seconda proprietà afferma che

$$\int_t^{t+1} \frac{1}{x}\,\mathrm{d}x \leq \int_t^{t+1} \frac{1}{t}\,\mathrm{d}x = \frac{1}{t}\int_t^{t+1} 1\,\mathrm{d}x = \frac{1}{t}\cdot[x]_t^{t+1} = \frac{1}{t}(t+1-t) = \frac{1}{t}.$$

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:

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

5.2.2. Definizione del problema

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 tanti 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

$R \leftarrow U$
$I \leftarrow \varnothing$
while $R \neq \varnothing$:
  scegli S_i che minimizzi $\frac{w_i}{|S_i \cap R|}$
  $I \leftarrow I \cup \{i\}$
  $R \leftarrow R / S_i$
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 5.2.2.1: Alla fine di Greedy Set Cover abbiamo

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

Dimostrazione 5.2.2.1: 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 5.2.2.2: Per ogni k vale

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

Dimostrazione 5.2.2.2: 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

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

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

Non ho capito la prima minorazione. ■

6. Lezione 06 [17/10]

6.1. Ancora Set Cover

Teorema 6.1.1: Sia $M = \max_i |S_i|$, allora Greedy Set Cover è una $H(M)$-approssimazione per Set Cover.

Dimostrazione 6.1.1: Sia I^* una soluzione ottima, ovvero un insieme di indici tali che

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

Per il lemma 2 della scorsa lezione 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

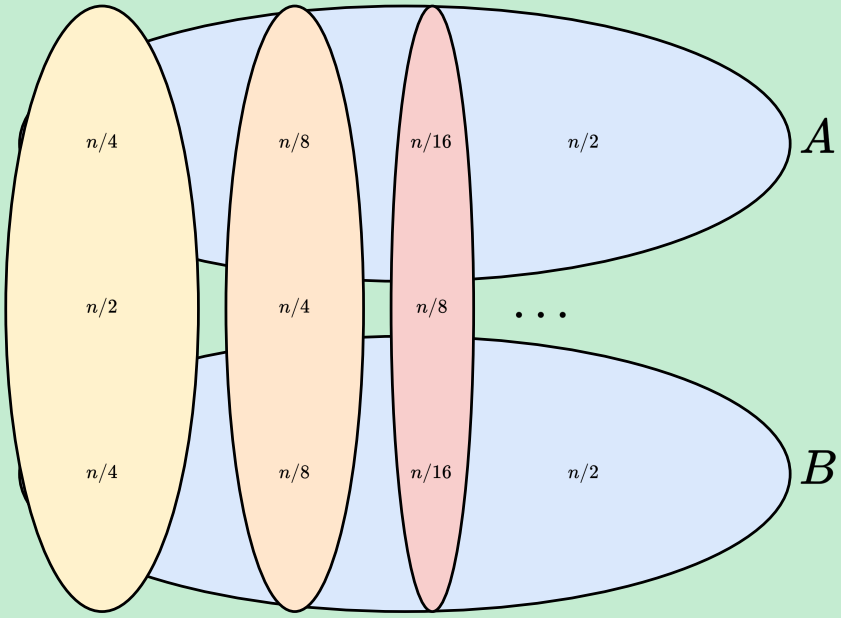
$$H(M) \leq H(n) = O(\log(n)).$$

Corollario 6.1.1.1: Greedy Set Cover è una $O(\log(n))$-approssimazione per Set Cover.

Questo qui 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 6.1.1: 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 6.1.2: Se $P \neq NP$ non esiste un algoritmo che approssimi Set Cover meglio di

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

Visti questi risultati, Set Cover finisce in $\log(n)$-APX.

6.2. Vertex Cover

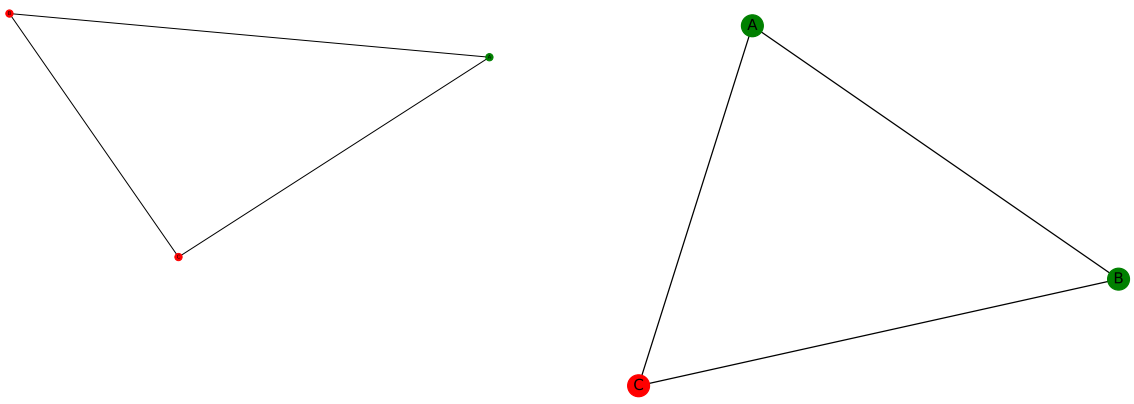
Il problema di **Vertex Cover** assomiglia spaventosamente a Dominating Set.

Vediamo da cosa 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 un arco che va in almeno una guardia
- in Vertex Cover andiamo a scegliere una serie di vertici X e tutti i *lati* hanno almeno un vertice nell'insieme scelto.



Questa differenza

mi manda in sbattimento (*cit. Boldi*)

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.

I vertici sono una gang, una mafia, vogliono massimizzare quello che portano a casa

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

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

Dimostrazione 6.2.1: 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

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

La prima disuguaglianza è vera perché selezionando tutti i vertici nella soluzione ottima X^* e poi tutti i lati che incidono su questi vertici potrei sommare più volte lo stesso lato. ■

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 $w_i \in \mathbb{R}^{>0} \quad \forall i \in V$

$p_e \leftarrow 0 \quad \forall e \in E$

while $\exists e = (i, j)$ tale che $(p_e)_{e \in E}$ è stretto su i e su j:

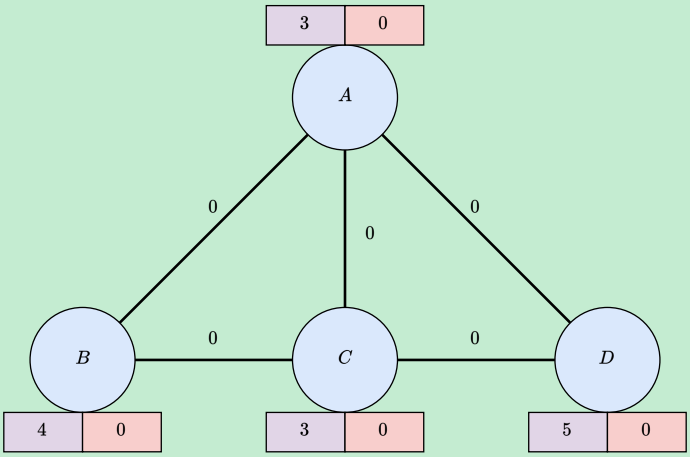
sia $\overline{e}$ il lato che minimizza $\Delta = \min\left(\left(w_i - \sum_{e \in E \wedge i \in e} p_e\right), \left(w_j - \sum_{e \in E \wedge j \in e} p_e\right)\right)$

$p_{\overline{e}} \leftarrow p_{\overline{e}} + \Delta$, ovvero alzo l'offerta del minimo indispensabile

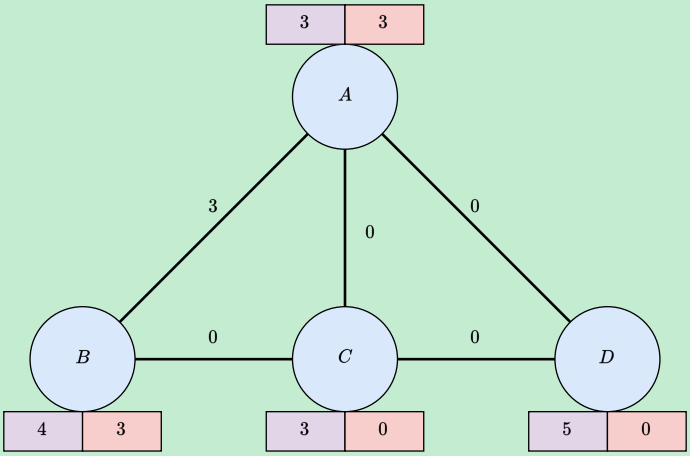
output $\{i \in V \mid p_e \text{ non è stretto su } i\}$ insieme dei vertici contenti

Esempio 6.2.1: 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.

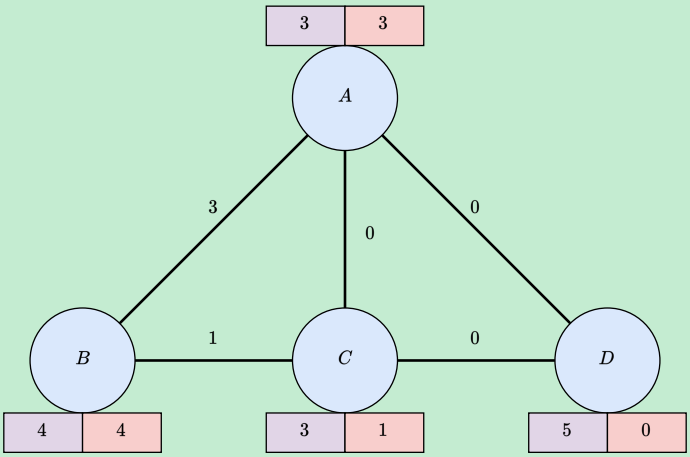
All'inizio dell'algoritmo siamo nella seguente configurazione, con il pricing banale.



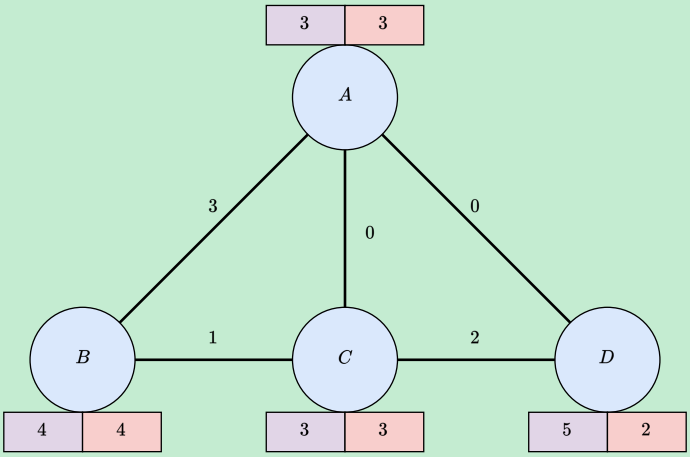
Alla prima iterazione del while tutti i lati hanno entrambi i vertici con pricing stretto e tutti i lati verrebbero aggiornati con il valore 3. Scegliamo di modificare il lato AB.



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.

7. Lezione 07 [18/10]

7.1. Ancora Vertex Cover

Chiudiamo la porta, c'è il prof che regala emozioni (*cit. Boldi*)

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 7.1.1: Alla fine di Pricing Vertex Cover abbiamo

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

Dimostrazione 7.1.1: Noi paghiamo

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

ma in output ho tutti i vertici non stretti sul pricing, quindi

$$w = \sum_{\substack{i \in V \mid p_e \text{ non} \\ \text{è stretto su } i}} w_i$$

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

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

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 7.1.1: Pricing Vertex Cover è una 2-approssimazione per Vertex Cover.

Dimostrazione 7.1.2: 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 esiste un PTAS, ovvero non si conosce una γ-approssimazione con $\gamma < 2$. Siamo quindi in un caso di inapprossimabilità, ci sarà un minimo tasso ma non sappiamo quanto è.

Networkx riempie di emozioni (*cit. Boldi*)

7.2. Problema dei cammini disgiunti (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.

Diamo una definizione a 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 $\pi_i : s_i \to t_i \quad \forall i \in I$ 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
  $I \leftarrow I \cup \{i\}$
  $P \leftarrow P \cup \{\pi_i\}$
  $\forall a \in \pi_i$:
    $l(a) \leftarrow \beta \cdot l(a)$

Greedy Paths

13: | | if $l(a) = \beta^c$:
14: | | | rimuovi a dal grafo
15: output I e P

Definizione 7.2.1 (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 7.2.2 (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 7.2.1: Sia $i \in I^*/I_o$, allora

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

Dimostrazione 7.2.1: 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 7.2.2: Sia $m = |A|$, allora

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

Dimostrazione 7.2.2: 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

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

per il primo lemma di questa lezione.

Osserviamo anche 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

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

L'ultima minorazione è per il lemma 2 di questa lezione.

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

Teorema 7.2.1: Greedy Path fornisce una $\left(2cm^{\frac{1}{c+1}} + 1\right)$-approssimazione per Disjoint Paths.

Dimostrazione 7.2.3: Osserviamo anche 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 abbastanza sus: 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.

8. Lezione 08 [24/10]

8.1. Vertex Cover, il ritorno

8.1.1. Programmazione lineare, intera e non

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 $\in$ *NPO-C* solo per l'imposizione di soluzioni intere.

8.1.2. Vertex Cover con 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**:

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

- **obiettivo**: $\min_x \left(\sum_{i \in V} w_i x_i \right)$.

Con il primo vincolo imponiamo che ogni lato abbia almeno un vincolo selezionato nella variabile x, mentre gli altri due vincoli impongono che $x_i \in \{0, 1\} \quad \forall i \in V$.

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

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

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

Lemma 8.1.2.1:

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

Dimostrazione 8.1.2.1: 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 8.1.2.2: $\overline{x}$ è una soluzione ammissibile di $\mathrm{ILP}(\pi)$.

Dimostrazione 8.1.2.2: 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 8.1.2.3:

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

Dimostrazione 8.1.2.3: Dobbiamo controllare i due possibili valori di $\overline{x}_i$.

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

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

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

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

■

Lemma 8.1.2.4:

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

Dimostrazione 8.1.2.4: Molto facile:

$$w = \sum_i w_i \overline{x}_i \underset{3}{\leq} 2 \sum_i w_i x_i^* = 2w_{\text{LP}}^*.$$

■

Teorema 8.1.2.1: L'algoritmo basato su rounding è una 2-approssimazione per Vertex Cover.

Dimostrazione 8.1.2.5: Dopo quattro estenuanti lemmi possiamo affermare che

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

■

9. Lezione 09 [25/10]

9.1. Implementazione di Vertex Cover

Citazioni più importanti della prima parte della lezione.

Mi faccio prendere dal vento autistico (*cit. Boldi*)

Implementiamo, non so se sarò in grado di comunicarvi delle emozioni come il docente della porta accanto (*cit. Boldi*)

Mi hanno fornito un'emozione (*cit. Boldi*)

Sto facendo delle osservazioni assurde (*cit. Boldi*)

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

Sarebbe bello avere algoritmi che, oltre a darti un valore con una data approssimazione (*teorica*) ti dica anche quanto è l'approssimazione vera del risultato fornito. Così posso capire al massimo quanto sono lontano dall'approssimazione teorica.

9.2. La nascita della 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.

IMMAGINE

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

Dimostrazione 9.2.1: Dimostriamo solo una delle due implicazioni in modo non formale.

$\{\Longleftarrow\}$

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.

  ■

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. Come si formalizza?

Lemma 9.2.1 (Handshaking lemma): In un grafo, il numero di vertici di grado dispari è pari.

Dimostrazione 9.2.2: 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. ■

Questo proprietà sono tipo delle Ramsey-type, ovvero teorie trovare per il meme che cercano la regolarità nel caos.

10. Lezione 10 [07/11]

10.1. Problema del commesso viaggiatore [TSP]

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 $G = (V, E)$ con delle 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 10.1.1: TSP $\equiv$ TSP su cricche.

Dimostrazione 10.1.1: Da chiedere, non l'ho capita.

■

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. Un **albero** in teoria dei grafi è un grafo connesso aciclico. Se il grafo non è connesso si chiama foresta. Gli alberi in informatica sono invece **alberi radicati**, ovvero viene scelto un vertice come radice e il grafo viene *«appeso»* a questo vertice e il resto cade per gravità. 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**.

10.2. Algoritmo di Christofides

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

$T =$ minimum spanning tree di G

$D =$ insieme dei vertici di grado dispari in T

Per l'handshaking lemma, $|D|$ ha cardinalità pari

$M =$ minimum-weight perfect matching su D

Può succedere che $T \bigcap M \neq \varnothing$, ovvero dei lati sono usati per entrambi gli insiemi

$H = T \bigsqcup M$ 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

$\pi =$ cammino euleriano, la cui esistenza è garantita dal teorema di Eulero

$\widetilde{\pi}$ = 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

output $\widetilde{\pi}$

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

Lemma 10.2.1:

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

Dimostrazione 10.2.1: 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 10.2.2:

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

Dimostrazione 10.2.2: Sia π^* il TSP ottimo. Dentro questo TSP ho anche i vertici di D, nodi che nell'albero 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 10.2.1: L'algoritmo di Christofides è una $\frac{3}{2}$-approssimazione per il TSP metrico.

Dimostrazione 10.2.3: Osserviamo che $\delta(\pi) = \delta(T) + \delta(M)$, ma per i due lemmi precedenti sappiamo che

$$\delta(\pi) \underset{l_1+l_2}{\leq} \delta^* + \frac{1}{2}\delta^* = \frac{3}{2}\delta^*.$$

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

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

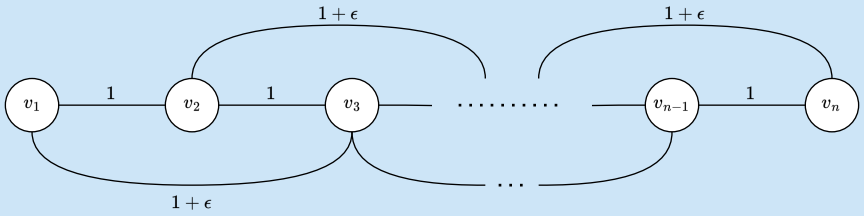
■

Teorema 10.2.2: L'analisi fatta è stretta.

Dimostrazione 10.2.4: 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

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

■

Si può dimostrare che TSP metrico è $\frac{3}{2}$-approssimabile, e che l'algoritmo di Christofides è il migliore disponibile.

11. Lezione 11 [14/11]

L'algoritmo di Christofides è di fine anni '70, ed è il migliore. 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.

11.1. Inapprossimabilità di TSP

Lemma 11.1.1: Il problema di decidere se un grafo ammetta un circuito hamiltoniano è *NP-C*.

Teorema 11.1.1: Non esiste $\alpha > 1$ tale che TSP sia α-approssimabile.

Dimostrazione 11.1.1: Supponiamo per assurdo di avere un algoritmo α-approssimante per TSP.

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

Calcolo 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 sono quindi non sovrapposti. Allora riesco a decidere se G ha un CH in tempo polinomiale, ma per il Lemma1 questo non è possibile in tempo polinomiale. ■

11.2. Un PTAS per 2-LoadBalancing

Ricordiamo cosa sono i *PTAS*: essi 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 11.2.1: L'algoritmo magico è una $(1+\varepsilon)$-approssimazione per 2-LoadBalancing.

Dimostrazione 11.2.1: 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

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

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 11.2.2: 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 11.2.2: 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. ■

Gli FPTAS, a differenza dei PTAS, hanno una parte *Fully* polinomiale, ovvero anche la parte di problema esaustiva è polinomiale.

11.3. Introduzione a 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.

12. Lezione 12 [14/11]

12.1. Knapsack

Abbiamo dato una breve introduzione del problema nella scorsa lezione, ora vediamo la sua 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* perché 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 individuano quella cella.

Questa tecnica è molto comoda quando i problemi sono di natura esponenziale. Il numeri 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 12.1.1: 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 facile 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**.

Dobbiamo dire se è polinomiale anche nei bit dell'input, che sono logaritmici in x. Ma una cosa lineare nel logaritmo è esponenziale (????).

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

12.1.1. 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

1: **Strategia di riempimento**

2: - La prima riga è 0 perché la capienza w è 0

3: - La prima colonna è 0 perché sto considerando 0 oggetti
   - L'algoritmo riempie per righe tutte quelle successive alla prima

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

5: - Considero la cella $v[w, i-1]$, ovvero a parità di peso prendo i primi $i-1$ oggetti

6: - Se non prendo l'oggetto i-esimo allora $v[w, i] = v[w, i-1]$

7: - 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:

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

8: **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.

12.1.2. 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 $0, ..., n$

Seconda versione di PD (focus valori)

- 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.

12.1.3. FPTAS per Knapsack

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**.

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

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 12.1.3.1: Sia X una soluzione ammissibile. Allora

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

Dimostrazione 12.1.3.1: 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

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

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 12.1.3.1: Vale inoltre

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

Dimostrazione 12.1.3.2: 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 12.1.3.1.1: 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.

13. Lezione 13 [21/11]

Oggi si implementa e basta.

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.

Per le soluzioni esatte, visto che uso sempre la colonna prima, posso tenere in memoria solo questa e costruire mano a mano quella corrente. Non mi cambia il tempo ma lo spazio.

14. Lezione 14 [22/11]

14.1. Algoritmi probabilistici

Introduciamo gli **algoritmi probabilistici**: quello che cambia è il **paradigma** che utilizziamo.

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

Quello che fanno gli algoritmi probabilistici è avere 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 solo 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.

14.2. 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 $k = \left|\left\{e \in E \mid e \cap X \neq \varnothing \wedge e \cap X^C \neq \varnothing\right\}\right|$, 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.

Il problema è *NPO-C* anche se non sembra.

Lemma 14.2.1: Il taglio minimo è $\leq$ del grado minimo dei vertici.

Questo è un bound molto molto ampio: 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. Infatti, se isolo in G_i un taglio e vedo la sua dimensione, essa è uguale in G perché tutti i lati che non ho contratto e che sono in G_i li trovo anche in G.

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. Vogliamo sapere la probabilità che all'i-esimo passo noi contraiamo uno dei lati *«preziosi»*.

Lemma 14.2.2: Vale

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

Dimostrazione 14.2.1: Valutiamo

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

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

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

Dimostrazione 14.2.2: Il taglio minimo si ottiene quando non contraggo mai i lati preziosi, quindi

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

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

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

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 14.2.1.1: 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 14.2.3: 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

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

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.

I bound che abbiamo dato sono veri se utilizziamo un generatore random ottimo, ma questo lo vedremo nella prossima lezione.