Algoritmi e complessità

Indice

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

Teorema (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 $\textit{NP-C}$ lo possiamo definire come "roba probabilmente difficile".

Corollario: 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 $\textit{NP-C}$.

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

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

  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 $\forall y \in \text{Amm}_\Pi(x)$ 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: Se $\Pi \in PO$ allora il suo problema di decisione associato $\widetilde{\Pi} \in P$.

Corollario: 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.

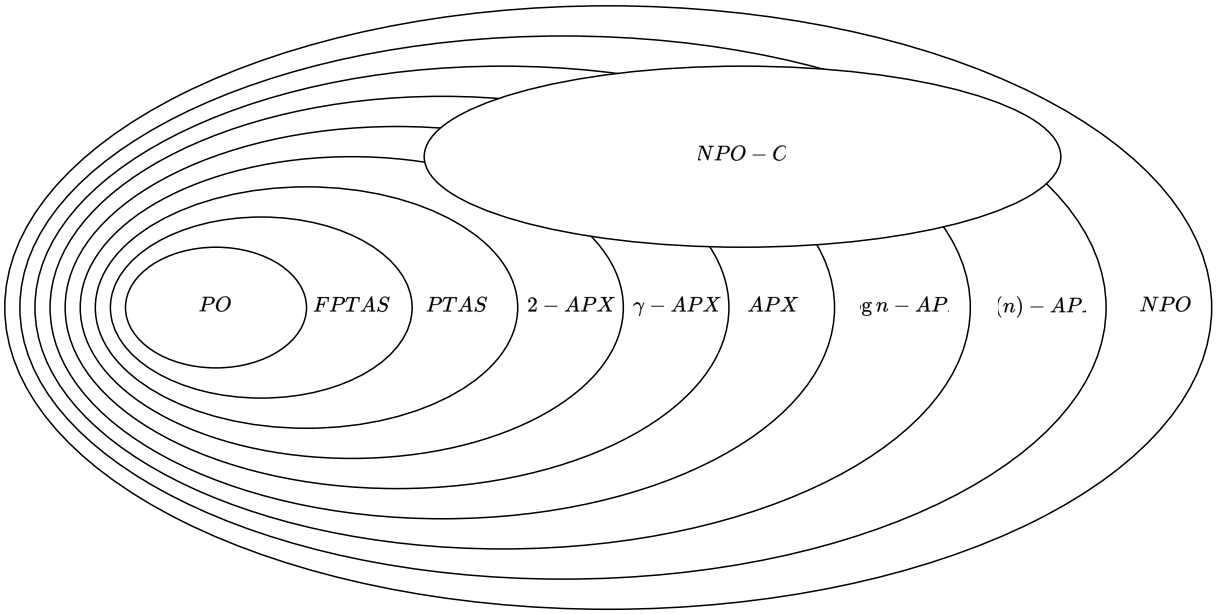
Se $R_\Pi = 1$ allora l'algoritmo su quell'input è ottimo e mi dà la soluzione migliore. Se, ad esempio, $R_\Pi = 2$, allora ho ottenuto

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

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

Dimostrazione: 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

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

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

Teorema: Bipartite Max Matching è in PO.

Corollario: 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: 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: Greedy Load Balancing è una 2-approssimazione per Load Balancing.

Dimostrazione: 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

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