Algoritmi Euristici

Manuel Pagliuca

22 ottobre 2021

Indice

1 Introduzione

L'obiettivo di questo corso è quello di mostrare che gli **algoritmi euristici** non sono ricette per problemi specifici: le euristiche e i problemi possono essere combinati liberamente. Una qualsiasi euristica può essere utilizzata su un qualsiasi problema. *Euristica* è una parola che deriva dal Greco, e sta per *"metodologia di ricerca della verità o dei fatti"*, deriva da *heurìsko* che significa *io trovo*.

Il termine deriva da una storia molto famosa di Archimede, che aveva da risolvere il problema di imparare se la data corona d'oro era effettivamente d'oro o se fosse solo placcata in oro. Questo era possibile da dedurre conoscendo il rapporto tra peso e volume, però il peso era facile da ottenere, mentre il volume è molto più difficile.

L'idea di Archimede consisteva nell'utilizzo di un secchio colmo di acqua ed immergendo la corona in quest'ultimo, il quantitativo di acqua che esce dal secchio consiste nel volume della corona.

2 Algoritmi euristici

2.1 Cenni sulla parola

La parola *euristica* può cambiare di significato in base al campo in cui viene utilizzata, in alcuni settori scientifici la parola *algoritmo euristico* è considerata come **ossimoro** in quanto le due parole vengono considerate opposti.

- **Algoritmo** ha un significato che sta per *procedura deterministica* e *formale*, la quale consiste in una sequenza finita di step elementari.
- **Euristico** ha un significato che sta per *informale, creativo* e *metodo a "regola aperta"* per trovare una soluzione.

Ogni algoritmo ha (od è) una **dimostrazione di correttezza** mentre un algoritmo euristico non ne ha nessuna. Essi sono trasformazioni meccanico-simboliche che partono da uno *starting point* (chiamato *ipotesi*) e giungono ad un *end point*(chiamato *tesi*).

2.2 Definizione

Gli algoritmi euristici sono procedure formali dove la soluzione non è garantita essere quella **corretta**. Questo potrebbe sembrare inutile, ma al contrario potrebbe essere utile per diversi motivi:

1. **Costa** molto meno di un algoritmo corretto, in termini di *spazio* e *tempo*.
2. Frequentemente **restituisce** qualcosa di vicino alla soluzione corretta.

Per definire la *vicinanza* della soluzione, lo **spazio delle soluzioni** sarà dotato di una *metrica* per esprimere una *distanza soddisfacente* della soluzione corrente dalla soluzione corretta. Inoltre sarà dotato di una **distribuzione probabilistica** per esprimere la frequenza soddisfacente delle soluzioni che si trovano ad una distanza soddisfacente dalla soluzione corretta (ovvero esprime quanto spesso l'algoritmo restituisce una soluzione soddisfacente).

2.3 Motivazioni per gli algoritmi euristici

Le euristiche sono la costruzione sia delle **dimostrazioni** che degli **algoritmi**, in caso di successo l'euristica viene abbandonata e la *dimostrazione* viene preservata. Altrimenti, una buona euristica solitamente porta ad un buon risultato, seppur non perfetto.

3 Classificazione dei problemi

Questo corso è incentrato su gli algoritmi euristici applicati ai problemi di **ottimizzazione combinatoria** che sono **basati su soluzioni** (contrapposti a quelli **basati sui modelli**).

Un problema è una domanda che viene effettuata su un **sistema matematico**, la tipologia dei problemi viene classificata in base alla natura delle loro soluzioni.

1. **problemi di decisione**: la loro soluzione è un booleano.
2. **problemi di ricerca**: la loro soluzione è un qualsiasi sottoinsieme *fattibile*.
3. **problemi di ottimizzazione**: la loro soluzione è un numero il quale è il *minimo* o *massimo* valore di una **funzione oggettiva** definita su dei sottoinsiemi fattibili.
4. **problemi di conteggio**: la loro soluzione è il *numero* di sottoinsiemi *fattibili*.
5. **problemi di enumerazione**: la loro soluzione è la collezione di tutti i sottoinsiemi *fattibili*.

I problemi di ottimizzazione possono essere combinati con i problemi di ricerca, noi ci concentreremo su questo tipo di problemi, ovvero siamo alla ricerca del **valore ottimale** e del **sottosistema che assume** quel valore.

3.1 Problemi di ottimizzazione e ricerca

Un problema di ottimizzazione e ricerca può essere rappresentato con:

$$opt_{x \in X} f(x)$$

Dove x rappresenta una **sottosistema fattibile** che è una delle soluzioni, la quale soddisfa le condizioni fornite dal problema. Invece, X è lo **spazio delle soluzioni fattibili**. Invece la funzione, si chiama **funzione oggettiva** ed è mappata in questa maniera $f : X \rightarrow R$, il suo compito è quello di *misurare quantitativamente* la qualità di ogni sottosistema (o soluzione). Generalmente, in quanto problema di ottimizzazione, la funzione oggettiva è *massimizzabile* o *minimizzabile*, questo viene denotato con $opt \in min, max$.

Il problema consiste nel determinare il **valore ottimale della funzione oggettiva** assieme alla **soluzione ottima** tale che sia un sottoinsieme. Il valore ottimale della funzione oggettiva viene chiamato f^*, ed è il risultato della seguente equazione:

$$f^* = opt_{x \in X} f(x)$$

Ovvero un valore che consiste nel minimo o massimo della funzione oggettiva. Mentre, la soluzione ottima tale che sia un sottoinsieme si chiamerà x^*:

$$x^* \in X^* = arg\ opt_{x \in X} f(x) = x^* \in X : f(x^*) = opt_{x \in X} f(x)$$

Ovvero, vogliamo trovare una soluzione ottima nell'intero insieme di tutte le soluzioni ottime, anche se solitamente una è abbastanza, la notazione *arg* sta per l'intero insieme delle soluzioni (ne basta una).

3.2 perché siamo interessati nei problemi di ottimizzazione e ricerca ?

I problemi di ottimizzazione e ricerca sono di forte interesse poiché diversi campi applicativi richiedono oggetti o strutture caratterizzati da valori molto alti o molto bassi rispetto ad una propria funzione di valutazione.

- Bioinformatica
- Social networks
- Machine learning
- Hardware design
- Stima dei parametri
- Finanza

L'**ottimizzazione esatta** è costosa da un punto di vista computazionale e non sempre desiderabile (per questo gli algoritmi euristici sono favoriti); perciò, solitamente, le funzioni di valutazione sono delle *approssimazioni* di quello che realmente accade. In questo corso assumeremo il punto di vista dell'ottimizzazione, cercando di ottimizzare al meglio possibile la funzione oggettiva.

Diversi problemi possono spesso essere ridotti in problemi di ottimizzazione e ricerca

- *Problemi di ricerca* possono essere ridotti rilassando le condizioni da soddisfare, in maniera da allargare la **regione di fattibilità** da X a $X' \supset X$ ed ottenere un problema di ricerca semplice. Si introduce una funzione $d(x)$ per quantificare la distanza di ogni soluzione $x \in X'$ da X. Infine, minimizzando $d(x)$ per trovare x^* tale che $d(x^*) = 0 \Leftrightarrow x^* \in X$.
- Alcuni *problemi di decisione* riguardano l'esistenza di sottosistemi fattibili, e sono identici ai problemi di ricerca (trovare il sottosistema dimostra la sua esistenza).
- Alcuni *problemi di numerazione* riguardano la ricerca di sottosistemi con *"buoni"* valori di funzioni oggettive in conflitto e permettono *adattamenti diretti* ad algoritmi di ottimizzazione e ricerca.

Tali riduzioni sono spesso possibili ed utili, ma non sempre.

3.3 Problema di ottimizzazione combinatoria

Un problema è un problema CO (*Combinatorial Optimization*) quando la regione di fattibilità X è un insieme finito, e quindi, che abbia un numero finito di *soluzioni fattibili*. Questa sembra un'assunzione molto restrittiva, ma sono presenti molti problemi che hanno un numero infinito di soluzioni che possono essere ridotti a problemi che hanno un numero finito di soluzioni.

Per esempio:

- I problemi infiniti nel discreto possono avere un insieme finito di soluzioni interessanti.
- Alcuni problemi continui possono essere ridotti a problemi di ottimizzazione combinatoria: *Programmazione lineare, Flusso massimo, Costo minimo di flusso, ...*
- I problemi continui possono essere ridotto in discreto utilizzando il campionamento (solitamente non è molto efficace).
- Le idee concepite per i problemi CO possono essere estese ad altri problemi.

3.4 Euristiche basate sui modelli

In questo corso parleremo solamente di **euristiche basate su soluzioni**, ma è importante conoscere anche la controparte. L'euristiche basate sui modelli descrivono la regione di fattibilità X con un "modello", un esempio tipico è una funzione matematica.

$$opt_{x \in X} f(x) \rightarrow \min_{g_i(\xi) \leq 0} \phi(\xi) \text{ per } i = 1, ..., m$$

Dove $\xi \in \mathbb{R}^n$, il quale è il vettore delle soluzioni di n numeri reali. Mentre $X = \xi \in \mathbb{R}^n : g_i(\xi) \leq 0, i = 1, ..., m$, ovvero che la regione di fattibilità è l'insieme dei vettori che soddisfanno tutte le disuguaglianze. L'euristiche basate su

modelli estrapolano l'informazione derivata dal modello, che sono le proprietà analitiche della funzione ϕ e g_i per $i = 1, ...m$.

3.5 Definizione alternativa di CO

Una problema è un problema CO() quando:

1. Il numero di soluzioni fattibili è finito (prima definizione).
2. La regione di fattibilità è $X \subseteq 2^B$ per un dato **ground set** B, ovvero, le *soluzioni fattibili* sono tutte sottoinsiemi del ground set che soddisfa le condizioni adeguate.

Entrambe le definizioni sono equivalenti:

- 2 $\implies$ 1: se il ground set B è finito, ogni collezione $X \subseteq 2^B$ è finita.
- 1 $\implies$ 2: se il numero di soluzioni fattibili è finito, definire B come il loro sovrainsieme ed X la *regione fattibile* sarà la collezione dei singoli elementi di B (una *"soluzione"* è un insieme contenente una singola soluzione).

In generale, la definizione sofisticata permette un analisi più profonda, perché X non viene semplicemente numerato e viene definito in una maniera *compatta* e *significativa*.

3.6 Euristiche basate su soluzioni

L'euristiche basate su soluzioni considerano le soluzioni come sottoinsiemi del ground set, esse possono essere classificate in:

1. **Euristiche costruttive/distruttive**, iniziano da un sottoinsieme estremamente semplice (può essere $\emptyset$ o B), poi, esse aggiungono/rimuovono gli elementi fino a che non ottengono la soluzione desiderata.
2. **Euristiche di scambio**, iniziano da un sottoinsieme ottenuto in una qualsiasi maniera, poi, scambiano gli elementi fino a che non ottengono la soluzione desiderata.
3. **Euristiche di ricombinazione**, iniziano da una popolazione di sottoinsiemi ottenuta in una qualsiasi maniera, poi, ricombinando differenti sottoinsiemi produrranno una *nuova* popolazione.

I progettisti delle euristiche possono combinare in maniera creativa gli elementi delle diverse classi delle euristiche.

3.6.1 Randomizzazione e memoria

Sono presenti due cose importanti che intervengono nella progettazione di un algoritmo:

- **Randomizzazione**
- **Memoria**

Puoi avere algoritmi che usano o non usano la randomizzazione o la memoria. Questi due elementi sono ortogonali (indipendenti) rispetto alla classificazione delle euristiche basate sulle soluzioni, per ognuna di esse possiamo dire che abbiamo quattro sottoclassi.

1. Utilizzo della randomizzazione:
    - **Euristiche puramente deterministiche**
    - **Euristiche *"randomizzate"***, essenzialmente sono algoritmi che utilizzano come input numeri pseudo-casuali.
2. Utilizzo della memoria:
    - Euristiche dove l'input include solamente i **dati del problema**.
    - Euristiche dove l'input include anche **soluzioni precedentemente generate**.

Comunemente si utilizza il termine *metaeuristiche* (dal Greco, *"oltre le euristiche"*) per descrivere gli algoritmi euristici che vanno utilizzano la randomizzazione e/o la memoria.

3.7 Rischi da cui stare attenti

1. **Attitudine reverenziale o alla tendenza**, ovvero, nello scegliere un algoritmo basato sul contesto sociale, anziché sul problema.
2. **Attitudine magica**, ovvero, *credere* in un metodo sulla base di un analogia con un fenomeno fisico e naturale.
3. **Integralismo euristico**, ovvero, utilizzare un euristica per un problema che ammette l'utilizzo di un algoritmo esatto.
4. **Sgranocchiare numeri**, ovvero, eseguire sofisticati e complessi calcoli con numeri inaffidabili.
5. **Attitudine SUV**, ovvero, affidarsi alla potenza dell'hardware.
6. **Complicare ulteriormente**, ovvero, introdurre componenti e parametri *ridondanti*, per cercare (fallendo) di migliorare il risultato.
7. **Overfitting**, ovvero, adattare i componenti ed i parametri dell'algoritmo ad un dataset specifico utilizzato nella valutazione sperimentale.

Inoltre è fondamentale:

- Liberarsi dai pregiudizi.
- Valutare le prestazioni dell'algoritmo in una maniera scientifica.
- Distinguere il contributo di ogni componente dell'algoritmo.
- Implementare efficientemente ogni componente dell'algoritmo.

4 Problemi di ottimizzazione combinatoria

Il ground set è la base sul quale si costruisce l'algoritmo, abbiamo visto che sono presenti molteplici possibilità con le euristiche basate sulle soluzioni, la loro classe cambierà in base al ground set utilizzato. Quindi per prima cosa dobbiamo capire che cosa è (*tipologia*) il **ground set**.

Visiteremo inizialmente un certo numero di problemi, questo sarà utile perché:

- Le idee astratte devono essere applicate concretamente su diversi algoritmi per diversi problemi.
- La stessa idea può avere differente efficacia su diversi problemi.
- Alcune idee funzionano solamente su problemi con una specifica struttura.
- Diversi problemi potrebbero non avere un apparente relazione, cosa che può essere sfruttata per progettare algoritmi.

Una buona conoscenza di diversi problemi ci insegna ad applicare le idee astratte a nuovi problemi e ci insegna come trovare e sfruttare le relazioni tra problemi conosciuti e nuovi.

4.1 Insieme dei problemi pesati

4.1.1 Knapsack Problem (KP)

Il problema dello zaino, *Knapsack Problem*. Il problema consiste nell'avere a disposizione uno zaino che ha una *capacità limitata* ed un insieme di oggetti con differenti *volumi* e *valori*, si vuole riempire lo zaino con gli oggetti di valore massimo (ovviamente non si può mettere dentro tutti gli oggetti).

Dati:

- Insieme elementare E di oggetti.
- Una funzione $v : E \to \mathbb{N}$ che descrive il **volume** di ogni oggetto.
- Un numero $V \in \mathbb{N}$ che descrive la **capacità** dello zaino.

- Una funzione $\phi : E \to \mathbb{N}$ che descrive il **valore** di ogni oggetto.

Banalmente, il **ground set** è l'insieme degli oggetti $B \equiv E$. La **regione di fattibilità** include tutti i sottoinsiemi degli oggetti il cui volume totale non eccede la capacità V dello zaino.

$$X = \left\{ x \subseteq B : \sum_{j \in x} v_j \leq V \right\}$$

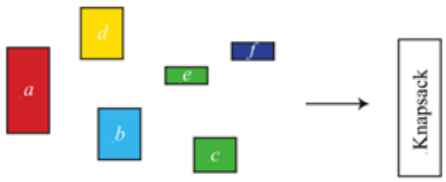
L'obiettivo è quello di massimizzare il valore totale degli oggetti scelti:

$$max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Per esempio, nella seguente tabella sono mostrati tutti gli elementi di E con i relativi valori e volumi. Sapendo che lo zaino ha una capacità massima $V = 8$, consideriamo due **soluzioni candidate**:

E	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

Dataset



Raffigurazione problema dello zaino

$x' = \{c, d, e\} \in X$ $\quad f(x') = 13$ $\qquad x'' = \{a, c, d\} \notin X$ $\quad f(x'') = 16$

Soluzioni candidate

- La prima soluzione candidata considera tre elementi la cui somma è $\leq V$, questo significa che la soluzione è un sottoinsieme preso dalla regione di fattibilità X ed è una **soluzione fattibile**.
- La seconda soluzione candidata ha una somma di elementi pari a 10, la quale è $> V$. Quindi questo sotto insieme non è appartenente alla regione di fattibilità X (ma solo al ground set), perciò verrà chiamata **soluzione infattibile**.

Tra le soluzioni fattibili proposte la **funzione oggettiva** propone di prendere la soluzione *massima*, ma visto che x'' non è fattibile, prenderemo come soluzione x'.

4.1.2 Maximum Diversity Problem (MDP)

Il problema della diversità massima, *Maximum Diversity Problem*, è un problema importante per il corso e verrà utilizzato come esempio per la parte di laboratorio. Questo è un problema definito su uno spazio metrico, quindi uno spazio con la nozione di *distanza*.

Dati:

- Un insieme di punti P.
- Una funzione $d : P \times P \to \mathbb{N}$, la quale provvede la distanza tra le coppie di punti.
- Un numero $k \in 1, ..., |P|$, il quale è il numero di punti che si vuole selezionare.

Il problema chiede di selezionare da un insieme di punti P un sottoinsieme di k punti la cui sommatoria delle distanze tra le coppie dei punti sia massima. Questo è un problema CO, perché il numero di sottoinsiemi possibili è finito, ed in particolare è un problema CO perché le soluzioni sono sottoinsiemi del

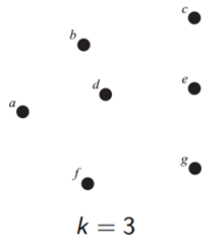
ground set. Il ground set, banalmente, è l'insieme dei punti $B \equiv P$, mentre la regione di fattibilità include tutti i sottoinsiemi composti da k punti.

$$X = \{x \subseteq B : |x| = k\}$$

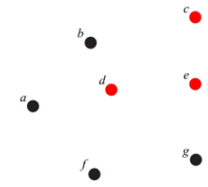
La funzione oggettiva è la sommatoria di tutte le distanze tra le coppie di punti in x:

$$\max_{x \in X} f(x) = \sum_{(i,j):i,j \in x} d_{ij}$$

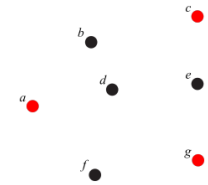
Per esempio, consideriamo un dataset costituito da 7 punti e considerando un $k = 3$, questo significa che vogliamo trovare un sottoinsieme costituito da 3 punti tale che le coppie abbiano distanza massima.



Dataset



Prima soluzione candidata del MDP



Seconda soluzione candidata del MDP

La prima soluzione x', considerando una metrica *non fornita* ha come valutazione della funzione oggettiva $f(x') = 24$, ed è un sottoinsieme tale per cui la sua cardinalità sia $\leq k$, e quindi appartenente alla regione di fattibilità X.

La seconda soluzione ha come soluzione della funzione oggettiva $f(x'') = 46$, questo è anche fattibile visto che $|x''| = k$, ed è una soluzione preferibile alla prima visto che stiamo cercando il sottoinsieme che soddisfi massimo della funzione oggettiva.

4.1.3 Interludio 1: la funzione oggettiva

Fermiamoci un attimo e pensiamo qualcosa a riguardo di questo problema, in particolare soffermiamoci sulla funzione oggettiva. Questa viene data come funzione che ha come dominio la regione di fattibilità e giunge al codominio all'insieme dei numeri naturali.

$$f : X \to \mathbb{N}$$

Il calcolo di questa funzione potrebbe essere molto complesso ed estenuante, ogni singola soluzione ha il proprio valore rispetto alla funzione oggettiva e si dovrebbe andare a controllare ogni volta in una tabella per svolgere il calcolo, non è una cosa molto interessante da fare.

Questo ovviamente, non è il caso dei precedenti problemi in quanto semplici da svolgere.

In particolare il problema KP una funzione oggettiva **additiva**, nel calcolare la f dobbiamo effettuare delle somme con il valore della funzione ausiliaria ϕ definita sul ground set, ricordando che nel problema KP, $B \equiv E$.

$$\phi : B \to \mathbb{N} \text{ induce } f(x) = \sum_{j \in x} \phi_j : X \to \mathbb{N}$$

Questo è interessante perché significa che si deve memorizzare solamente il valore della funzione ausiliaria ϕ, i quali sono $|B|$ valori, e non 2^B, come sarebbe per i valori della funzione oggettiva che è definita su X.

Lo stesso accade per il problema MDP e la sua funzione ausiliaria d, anche se quest'ultima ha una funzione oggettiva quadratica, poi fornito un $n = |B|$ di punti nel caso peggiore si dovranno sommare $\frac{n(n-1)}{2}$ (numero di archi in un grafo completo) distanze, tuttavia però il calcolo rimane *"solamente"* una somma, quindi avente una **complessità polinomiale**.
Nel problema KP una volta che il valore viene fornito per uno specifico sottoinsieme, rimane possibile modificare gli elementi e ricalcolare il valore della funzione oggettiva facilmente. Il valore della funzione oggettiva nel caso del MDP va trattato in maniera differente per essere calcolato in tempo lineare (poiché quadratica).

Un'altra importante osservazione è che il problema KP e MDP sono definiti sull'intero insieme delle possibili (non fattibili) soluzioni 2^B, e questo è generalmente inutile visto che stiamo cercando una soluzione fattibile (in alcuni casi questo però sarà utile).

Per riassumere, quando guardiamo un problema cerchiamo di capire come la funzione oggettiva è costituita:

- È una funzione additiva?
- È una funzione quadratica?
- È una funzione semplice da calcolare?
- È una funzione semplice da aggiornare?
- Su cosa è definita la funzione oggettiva?

4.2 Insieme dei problemi di partizionamento

In questi problemi un insieme di oggetti viene fornito, l'obiettivo consiste nel dividerlo in sottoinsiemi ottenendo una partizione con alcune peculiarità.

4.2.1 Bin Packing Problem (BPP)

Il Bin Packing Problem (BPP), si ha un insieme di oggetti con un *volume*, e si vuole mettere questi oggetti all'interno di container con una *capacità fissa* (fornita) utilizzando il **minimo numero di container**.

Dati:

- Insieme E di oggetti.
- Una funzione $v : E \to \mathbb{N}$ che fornisce il volume per un dato oggetto $e \in E$.
- Un insieme C di containers.
- Un numero $V \in \mathbb{N}$ il quale rappresenta la capienza massima dei container (volume massimo contenibile).

La prima domanda che ci si vuole porre è: *è un problema di ottimizzazione combinatoria?*

Il ground set è definito come $B = E \times C$, dove ogni elemento di B è definito da una coppia $\langle oggetto, container \rangle$. Una soluzione per questo problema è un sottoinsieme formato da oggetti di questo tipo, il prodotto cartesiano è necessario poiché si deve selezionare un oggetto ed inserirlo in un determinato container.

Una volta che la lista di coppie contenenti gli elementi di E è costruita, una soluzione candidata sarà ottenuta (un sottoinsieme del ground set, ma a noi questo non basta, voglia che sia un sottoinsieme della regione di fattibilità).

Consideriamo B_e come il sottoinsieme del ground set dove gli oggetti delle coppie provengono da E (i container sono tutti i possibili), e B_c come il sottoinsieme del ground set dove i container nelle coppie degli elementi provengono da C (gli elementi sono tutti i possibili).

$$B_e = \{(i, j) \in B : i = e\}$$

$$B_c = \{(i,j) \in B : i = c\}$$

La **regione di fattibilità** include tutte le partizione degli oggetti tra i container tale per cui non ecceda la capacità di un qualsiasi container.

$$X = \left\{ x \subseteq B : |x \cap B_e| = 1 \; \forall e \in E, \sum_{(e,c) \in B^c} v(e) \leq V \; \forall c \in C \right\}$$

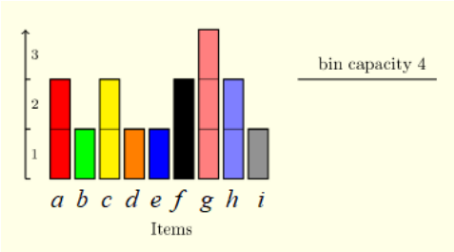
La prima parte dell'espressione è un vincolo sul sottoinsieme delle soluzioni fattibili x, dice che l'intersezione tra il sottoinsieme delle soluzioni fattibili ed il ground set deve avere modulo 1. Questo significa che gli elementi all'interno di x devono essere presenti solamente una volta rispetto a gli elementi E del ground set, ovvero $\in B_e$.

La seconda parte dell'espressione anch'essa è un vincolo ma rispetto al volume massimo dei container, dice che la somma dei volumi di ogni singolo elemento del sottoinsieme fattibile non deve eccedere la capacità massima dei container V.

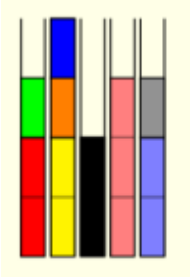
L'obiettivo è quello di minimizzare il numero di container utilizzati:

$$\min_{x \in X} f(x) = |c \in C : x \cap B^c \neq \emptyset|$$

Per esempio, consideriamo degli oggetti con diversi volumi e sia data una capacità massima dei container pari a $V = 4$.



Dataset

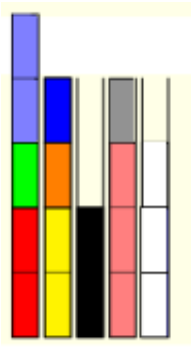


Prima soluzione candidata del Bin Packing Problem

Consideriamo la prima soluzione proposta, visto che la lista dei prodotti cartesiano rispetta i due vincoli nella definizione della regione di fattibilità la soluzione x' è una **soluzione fattibile**.

$$x' = \{(a,1),(b,1),(c,2),(d,2),(e,2),(f,3),(g,4),(h,5),(i,5)\} \in X$$

$$f(x') = 5$$



Seconda soluzione candidata del Bin Packing Problem

Invece, la seconda soluzione proposta è una **soluzione infattibile**, questo perché gli elementi non stanno rispettando il secondo vincolo sul volume, e quindi questo sottoinsieme non è compreso all'interno della regione di fattibilità.

$$x'' = \{(a,1),(b,1),(c,2),(d,2),(e,2),(f,3),(g,4),(h,1),(i,4)\} \notin X$$

$$f(x'') = 4$$

Considerando il caso in cui x'' fosse una soluzione fattibile avremmo scelto quella tra le soluzioni proposte, poiché la funzione oggettiva effettua l'ottimizzazione sul minimo.

4.2.2 Parallel Machine Scheduling Problem (PMSP)

Il Parallel Machine Scheduling Problem (PMSP), è un problema nel quale un insieme di attività (tasks) deve essere diviso lungo un set di macchine in modo che il *tempo di completamento* sia minimizzato.

Dati:

- Un insieme T di tasks (o *attività*).
- Una funzione $d : T \to \mathbb{N}$ che descrive la lunghezza (temporale) di ogni task.
- Un insieme di M macchine.

Come prima, il ground set è dato dal prodotto cartesiano di due set forniti:

$$B \equiv T \times M$$

Significa che la soluzione deve essere una coppia

$$\langle task, macchina \rangle$$

. È importante sottolineare che la sequenza in cui i task sono eseguiti non è rilevante, invece è rilevante il **tempo di completamento** , ovvero il tempo con cui l'ultimo task termina (o il tempo in cui una macchina completa l'esecuzione dei suoi tasks).

La regione di fattibilità include tutte le partizioni delle attività nella macchine:

$$X = \{x \subseteq B : |x \cap B_t| = 1 \; \forall t \in T\}$$

La **funzione oggettiva** ha come obiettivo quello di minimizzare il massimo della sommatoria delle lunghezze di tempo per ogni task di ogni macchina:

$$\min_{x \in X} f(x) = \max_{m \in M} \sum_{t:(t,m) \in x} d_t$$

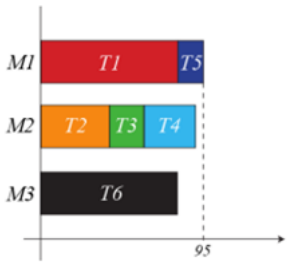
In parole povere, vogliamo trovare il sottoinsieme x che minimizza il tempo di completamento di ciascuna macchina, dove il tempo di completamento per un singolo task è $\sum_{t:(t,m) \in x} d_t$.
Per esempio, consideriamo il seguente insieme di dati per tre macchine, $|M| = 3$, e sette task differenti $|T| = 7$.

$$T = \{T1, T2, T3, T4, T5, T6\}$$

$$M = \{M1, M2, M3\}$$

task	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$
d	80	40	20	30	15	80

Dataset del Parallel Machine Scheduling Problem



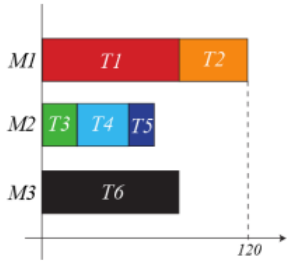
Prima soluzione Parallel Machine Scheduling Problem

Consideriamo la prima soluzione proposta:

$$x' = \{(T1, M1), (T2, M2), (T3, M2), (T4, M2), (T5, M1), (T6, M3)\} \in X$$

$$f(x') = 95$$

Possiamo notare che questa è una **soluzione fattibile**, visto che ogni task accade al meno ed al massimo una volta. Notiamo che il valore assunto dalla funzione oggettiva è 95, questo proprio perché l'ultimo task ha un tempo di completamento pari a 95.



Seconda soluzione Parallel Machine Scheduling Problem

Consideriamo la seconda soluzione proposta:

$$x'' = \{(T1, M1), (T2, M1), (T3, M2), (T4, M2), (T5, M2), (T6, M3)\} \in X$$

$$f(x'') = 120$$

Notiamo che anche questa è una **soluzione fattibile** visto che ogni task accade al minimo ed al massimo una volta, in questo casa la funzione oggettiva assume come valore 120. Questo significa che fra le due soluzioni proposte la prima è quella **ottima**.

4.2.3 Interludio 2: la funzione oggettiva, ancora

È necessario familiarizzare con il fatto che il ground set B non è sempre uno degli insiemi forniti nel problema, ma può essere formato dalla combinazione (come il prodotto cartesiano) di diversi insiemi forniti. Ora affrontiamo la domande proposte nell'ultimo interludio.

Le funzioni oggettivi di che tipo sono? (additive, quadratiche,...) Questa volta le funzioni oggettive per il BPP e PMSP **non** sono additive, e non sono neanche banali. È presente un algoritmo polinomiale per calcolare la funzione oggettiva, seppur non complesso, non è semplice come per i problemi precedenti.

Notiamo che piccole modifiche alle soluzioni hanno un impatto *variabile* sull'obiettivo, per esempio consideriamo la seconda soluzione x'' del PMSP:

- Spostare il task $T5$ sulla macchina $M1$, allunga il tempo di completamento complessivo$M1$, il risultato della funzione oggettiva cambia perché viene incrementato del task spostato (***impatto corrispondente al tempo del task spostato***).
- Spostare il task $T5$ sulla macchina $M3$, non modifica il tempo di completamento complessivo delle macchine, il risultato della funzione oggettiva rimane lo stesso (***impatto zero***).
- Spostare il task $T2$ sulla macchina $M2$, comporta una modifica dei tempi di completamento complessivi, il risultato della funzione oggettiva cambia poiché l'ultimo task viene spostato (***impatto intermedio***)

In fatti, l'impatto di una modifica di una soluzione dipende :

- Da entrambi gli elementi modificati.
- E dagli elementi non modificati (questo è contrario alle cose dette nell'interludio 1).

Un punto interessante è che la funzione oggettiva del PMSP tende ad essere **piatta**, ovvero che sono presenti molteplici soluzioni all'interno del problema dove il valore della funzione rimane lo stesso anche se avvengono delle modifiche (l'esempio precedente, la soluzione x'' rimane fissa per diverse combinazioni su 120).

4.3 Problemi delle funzioni logiche

4.3.1 The Max-SAT problem

Il problema del Max-Sat, sia da una formula in **forma normale congiuntiva** (CNF, *Conjunctive Normal Form*), si vogliono fornire in ingresso dei valori di verità alle variabili logiche della CNF tali per cui la formula venga soddisfatta (valutata come vera).

Dati:

- Un insieme V di **variabili logiche** x_j con valori in $\mathbb{B} \in \{0, 1\}$.
- Un **letterale** $l_j(x) \in x_j, \overline{x}_j$ che è una funzione che consiste in una variabile logica *affermata* o *negata*.
- Una **formula logica** $C_i(x) = l_{i,1} \vee ... \vee l_{i,n_i}$, la quale è una disgiunzione o *somma logica* (OR) di letterali. Soddisfare una formula logica significa fargli assumere valore 1.
- Una formula in **forma normale congiuntiva** $CNF(x) = C_1 \wedge ... \wedge C_n$ è una congiunzione di *prodotti logici* di formule logiche.
- Una funzione w che provvede dei *pesi* per la formula CNF. La funzione associa ogni formula logica della CNF ad un rispettivo peso.

Visto che la soluzione consiste in un sottoinsieme caratterizzato dall'assegnamento di valori di verità a variabili logiche, il **ground set** sarà il prodotto cartesiano fra le variabili logiche e l'insieme dei numeri booleani:

$$B = V \times \mathbb{B} = \{(x_1, 0), (x_1, 1), ..., (x_n, 0), (x_n, 1)\}$$

La **regione di fattibilità** è l'insieme delle soluzioni fattibili tali che una *variabile* venga considerata al più una volta. Essa include tutti sottoinsiemi costituenti gli assegnamenti semplici che sono:

- **completi**, ovvero che ad ogni variabile corrisponde *almeno* un letterale.
- **consistenti**, ovvero che per ogni variabile corrisponde *al massimo* un letterale.

$$X = \{x \subseteq B : |x \cap B_v| = 1 \; \forall v \in V\}$$
$$B_{xj} = \{(x_j, 0), (x_j, 1)\}$$

La **funzione oggettiva** (come sempre ottimizzata):

$$max f(x) = \sum_{i:C_i(x)=1} w_i$$

L'obiettivo è quello di massimizzare il peso totale della *formula logica soddisfatta* segnata come $C_i(x) = 1$ per $i = 1, ..., n$ (dove n è il numero di formule logiche presenti).

Consideriamo il seguente esempio:

$$V = \{x_1, x_2, x_3, x_4\}$$
$$L = \{x_1, \overline{x}_1, x_2, \overline{x}_3, x_3, \overline{x}_4, x_4\}$$
$$C_1 = \overline{x}_1 \vee x_2 \; ... \; C_7 = x_2$$
$$CNF = (\overline{x}_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_4) \wedge x_1 \wedge x_2$$
$$w_i = 1 \forall C_i$$

Consideriamo adesso la seguente soluzione:

$$x = \{(x_1, 0), (x_2, 0), (x_3, 1), (x_4, 1)\}$$

La funzione oggettiva per questa soluzione assume valore $f(x) = 5$, significa che soddisfa 5 formule delle 7.

Risulta semplice trovare il valore della funzione oggettiva? Non proprio, la complessità della funzione oggettive è polinomiale

In caso di pesi uniformi sono presenti un campo ristretto di valori, che vanno da 0 a n, il numero di formule logiche, anche se sono presenti $2^{|V|}$ combinazioni che possono essere considerate.

4.4 Problemi con matrici numeriche

4.4.1 Set covering problem (SCP)

Data una **matrice binaria** ed una **funzione costo** definita per ogni colonna della matrice (come vettore), si vuole selezionare il sottoinsieme di colonne che coprono tutte le righe di costo minimo.

Dati:

- Matrice binaria $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C.
- La colonna $j \in C$ copre la riga $i \in R$ quando $a_{ij} = 1$.
- Una funzione $c : C \to \mathbb{N}$ provvede il costo di ogni colonna.

Il **ground set** è l'insieme delle colonne.

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsiemi delle colonne che coprono tutte le righe.

$$X = \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \geq 1 \; \forall i \in R \right\}$$

L'obiettivo è *minimizzare* il costo totale delle colonne selezionate, la **funzione oggettiva** è additiva, molto veloce da calcolare ed aggiornare, però la *fattibilità* non è semplice da ottenere.

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Consideriamo il seguente esempio di una matrice con il relativo vettore dei costi:

c	4	6	10	14	5	6

	0	1	1	1	1	0
	0	0	1	1	0	0
A	1	1	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	1	0

Dataset del Set Covering Problem (SCP)

Notiamo che la terza e la quinta riga (iniziando dall'alto) sono coperte dalla prima colonna. Infatti, *"set covering"*, si coprono i *set* (righe) con *subset* (colonne).

Consideriamo adesso una prima soluzione proposta:

	0	1	1	1	1	0	2
	0	0	1	1	0	0	1
A	1	1	0	0	0	1	1
	0	0	0	1	1	1	1
	1	1	1	0	1	0	3

Prima soluzione proposta del SCP

$$x' = c_1, c_2, c_5 \in X$$
$$f(x') = 19$$

La prima soluzione x' è una **soluzione fattibile**, perché ogni riga ha almeno un elemento $a_{ij} \geq 1$.

	0	1	1	1	1	0	1
	0	0	1	1	0	0	0
A	1	1	0	0	0	1	2
	0	0	0	1	1	1	2
	1	1	1	0	1	0	2

Seconda soluzione proposta del SCP

$$x'' = c_1, c_5, c_6 \notin X$$
$$f(x'') = 15$$

La seconda soluzione x'' **non è una soluzione fattibile**, dato che la Seconda riga non è coperta da almeno un elemento delle colonne. Comunque, nel caso in cui la seconda soluzione sia una soluzione fattibile, allora quest'ultima sarebbe una soluzione migliore della prima.

4.4.2 Interludio 3: il test di fattibilità

Solitamente, gli algoritmi euristici **richiedono** di risolvere il seguente problema: *Dato un sottoinsieme x, è fattibile o infattibile?*, in breve $x \in X$?. Come risolvere questo problema? Innanzitutto, è un **problema di decisione**.

Consideriamo SCP, la fattibilità può essere decisa passando per ogni riga e sommando gli 1 che appaiono nella colonna selezionata: se una qualsiasi riga ha una somma complessiva pari a 0 la soluzione sarà **infattibile**.

Nel caso del KP, il test di fattibilità richiede di calcolare dalla soluzione e testare un singolo numero (il peso totale rispetto alla capacità dello zaino) proprio come nel MDP, dove la cardinalità della soluzione si trovava sotto una restrizione k.

Altri problemi come il Max-SAT ed il PMSP richiedono di testare la fattibilità su un singolo insieme di numeri (numero di variabili logiche non si ripeta nella soluzione $x \cap B_v$), mentre problemi come il BPP richiedono di testare su diversi insiemi di numeri (i volumi degli oggetti rispetto alla capacità dei container).

Alcune modifiche alle soluzioni vengono vietate *a priori* per evitare l'inammissibilità delle soluzioni. Supponiamo di avere una soluzione fattibile per il MDP, una qualsiasi modifica in cui il numero di punti non è uguale al numero di punti aggiunti rende la soluzione **infattibile**.

Alcune modifiche non garantiscono **inammissibilità** (unfeasible) della soluzione, le quali richiedono un test *a posteriori* come nel PMSP.

4.5 Set Packing Problem

Il Set Packing Problem è un problema molto simile al precedente SCP, questo perché appartiene alla stessa classe di problemi e provvede un valore per ogni colonna.

Dati:

- Una **matrice binaria** $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C.
- Sia definito un conflitto tra due colonne $j', j'' \in C$ quando $a_{ij'} = a_{ij''} = 1$.
- Una funzione $\phi : C \to \mathbb{N}$ che provvede il valore di ogni colonna.

Il **ground set** è ancora l'insieme delle colonne:

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsieme di colonne che non sono in conflitto:

$$X = \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \leq 1 \; \forall i \in R \right\}$$

L'obiettivo consiste nello scegliere le colonne di valore massimo senza che siano presenti *conflitti*.

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Prendiamo in considerazione la seguente matrice:

ϕ	4	6	10	14	5	6
A	0	1	0	0	1	0
	0	0	1	1	0	0
	1	0	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	0	0

Dataset Set Packing Problem

A	0	1	0	0	1	0	1
	0	0	1	1	0	0	1
	1	0	0	0	0	1	1
	0	0	0	1	1	0	1
	1	1	1	0	0	0	1

Prima soluzione proposta del SPP (*Set Packing Problem*)

La prima soluzione proposta è $x' = \{c_2, c_4\} \in X$, con valutazione della funzione oggettiva $f(x') = 20$. Questa è una soluzione **fattibile**, visto che non presenti conflitti sulle righe delle colonne selezionate.

A	0	1	0	0	1	0	1
	0	0	1	1	0	0	0
	1	0	0	0	0	1	2
	0	0	0	1	1	0	1
	1	1	1	0	0	0	1

Seconda soluzione proposta del SPP (*Set Packing Problem*)

In questo caso **non è una soluzione fattibile**, visto che avviene il conflitto sulla terza riga, anche quando la valutazione della funzione oggettiva era meglio della prima soluzione, le somme valutate non sono ottenute dalla regione di fattibilità X.

4.6 Set Partitioning Problem

Il problema di Set Partitioning (non lo chiameremo con l'acronimo inglese), è una fusione dei due precedenti problemi SCP e SPP.

Dati:

- Una **matrice binaria** $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C.

- Una funzione $c : C \to \mathbb{N}$ che fornisce il costo di ogni colonna.

La risoluzione del problema prevede di selezionare il sottoinsieme di *costo minimo* delle colonne che non sono in conflitto. Il **ground set**, anche questa volta, è l'insieme delle colonne:

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsiemi di colonne che coprono tutte le righe che non sono in conflitto:

$$X = \left\{ x \subseteq C : \sum_{j \in x} a_{ij} = 1 \; \forall i \in R \right\}$$

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Consideriamo la seguente matrice binaria:

c	4	6	10	14	5	6
	0	1	0	0	1	0
	0	0	1	1	0	0
A	1	0	0	0	0	1
	0	0	0	1	1	0
	1	1	1	0	0	0

Dataset del Set Partitioning Problem

Ora prendiamo in considerazione la prima soluzione proposta:

	0	1	0	0	1	0	1
	0	0	1	1	0	0	1
A	1	0	0	0	0	1	1
	0	0	0	1	1	0	1
	1	1	1	0	0	0	1

Prima soluzione del SPP (*Set Partitioning Problem*)

$$x' = \{c_2, c_3, c_6\} \in X$$
$$f(x') = 26$$

Notiamo che la soluzione x' è un **soluzione fattibile**, visto che gli elementi appartenenti alle selezionati si trovano colonne si trovano in una maniera da non generare conflitti lungo le righe (la sommatoria degli elementi lungo le righe non porta ad un risultato diverso da 1).

Ora consideriamo la seconda soluzione candidata:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Seconda soluzione del SPP (*Set Partitioning Problem*)

$$x'' = \{c_1, c_5, c_6\} \notin X$$
$$f(x'') = 15$$

Anche se il risultato della funzione oggettiva ha un valore migliore (più piccolo) rispetto a quello della prima soluzione, la soluzione x'' **non è una soluzione fattibile**. Questo perché, gli elementi sulla terza riga si trovano in conflitto ($\sum_{j \in x} a_{ij} \geq 1$), ed anche perché la seconda riga non viene *coperta* da alcuna colonna ($\sum_{j \in x} a_{ij} = 0$).

4.7 Interludio 4: ricerca di soluzioni fattibili

Gli algoritmi euristici spesso richiedono di risolvere un altro problema: *trovare una soluzione che fattibile* $x \in X$, questo è un **problema di ricerca**. Chiaramente dato che le soluzioni sono definite da una soluzione iniziale, le euristiche di scambio e ricombinazione hanno bisogno di partire da un sottoinsieme valido tale per cui esso stesso sia una soluzione fattibile.

In base al problema la soluzione può essere banale:

- Alcuni insiemi sono sempre fattibili: $x = \emptyset$ (come nel KP, SPP) o $x = B$ (nel SCP).
- Alcune soluzioni casuali soddisfano un vincolo come $|x| = k$ (nel MDP).
- Alcune soluzioni casuali soddisfano vincoli consistenti, come assegnare un task per ogni macchina come nel PMSP, o un valore ad ogni variabile logica come nel Max-SAT.

Oppure può essere difficile:

- Nel BPP il numero di container deve essere sufficientemente grande.
- Nel SPP non è conosciuto alcun algoritmo polinomiale per risolvere il problema.

Alcuni algoritmi ingrandiscono al regione di fattibilità da X a X' (processo detto *"rilassamento"*), la funzione oggettiva f deve essere estesa anch'essa da X a X', ma spesso $X' \setminus X$ provvede delle soluzioni migliori.

4.8 Problemi sui grafi

4.8.1 Vertex Cover Problem

Dato un grafo indiretto $G = (V, E)$, selezionare un sottoinsieme di vertici di cardinalità minima tale che ogni arco del grafo sia incidente a quest'ultimo.

Il **ground set** è l'insieme dei vertici:

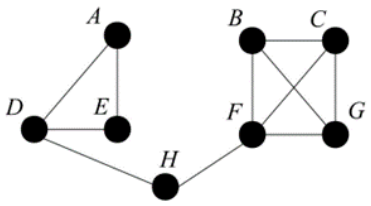
$$B \equiv V$$

La **regione di fattibilità** include tutti i sottoinsiemi dei vertici tali che gli archi del grafo siano incidenti ad essi:

$$X = \{x \subseteq V : x \cap (i, j) \neq \emptyset \, \forall (i, j) \in E\}$$

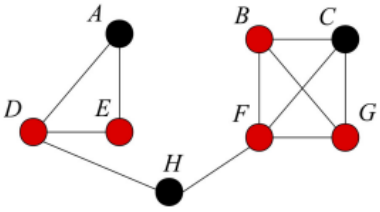
L'obiettivo è minimizzare il numero di vertici selezionati:

$$\min_{x \in X} f(x) = |x|$$



Dataset del Vertex Covering Problem

Prima soluzione proposta:

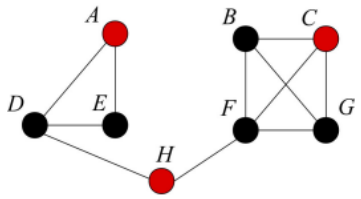


Prima soluzione proposta per il VCP

$$x' = \{B, D, E, F, G\} \in X$$
$$f(x') = 5$$

Notiamo che x' è una **soluzione fattibile**, questo perché il sottoinsieme di vertici selezionato interseca ogni arco del grafo (appartiene alla regione di fattibilità).

Guardiamo la seconda soluzione proposta:



Seconda soluzione proposta per il VCP

Invece, la seconda soluzione **non è una soluzione fattibile**, anche se la funzione oggettiva porta ad un risultato che è più convincente della soluzione precedente, questo sottoinsieme non appartiene alla regione di fattibilità (il sottoinsieme di vertici selezionato non è incidente a tutti gli archi del grafo, e.g.: (d, e)).

4.8.2 Maximum Clique Problem (MCP)

Dati:

- Un **grafo indiretto** $G = (V, E)$.
- Una funzione $w : V \to \mathbb{N}$ che provvede il peso di ogni vertice.

Selezionare il sottoinsieme di coppie di vertici adiacenti di peso massimo. Il **ground set** è l'insieme dei vertici.

$$B \equiv V$$

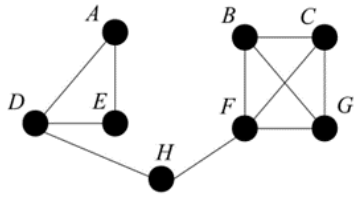
La **regione di fattibilità** include tutti i sottoinsiemi di coppie di vertici adiacenti.

$$X = \{x \subseteq V : (i, v) \in E \; \forall i \in x, \forall j \in x \setminus \{i\}\}$$

L'obiettivo è quello di massimizzare il peso dei vertici selezionati:

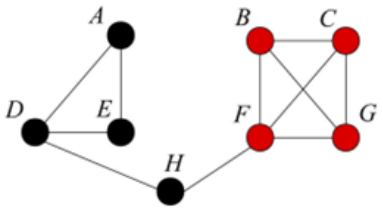
$$f(x) = \sum_{j \in x} w_j$$

Consideriamo il seguente grafo indiretto costituito da **pesi uniform** $w_i = 1 \forall i \in V$.



Dataset per il Maximum Clique Problem

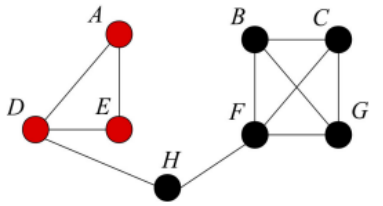
Consideriamo la prima soluzione proposta:



Prima soluzione proposta per il MCP

$$x' = \{B, C, F, G\} \in X$$
$$f(x') = 4$$

La prima soluzione proposta è una soluzione **fattibile**, visto che ogni coppia nel sottoinsieme di vertici presenta un arco tra di loro.



Seconda soluzione proposta per il MCP

$$x' = \{A, D, E\} \in X$$
$$f(x') = 3$$

La seconda soluzione proposta è anch'essa una soluzione **fattibile**, per lo stesso motivo precedente.

4.8.3 Maximum Independent Set

Questo problema è opposto al MCP, vogliamo trovare un sottoinsieme di vertici di peso massimo che non è connesso da archi. Dati:

- Un **grafo indiretto** $G = (V, E)$.
- Una funzione $w : V \to \mathbb{N}$ che provvede un peso per ogni arco.

Il **ground set** è l'insieme dei vertici.

$$B \equiv V$$

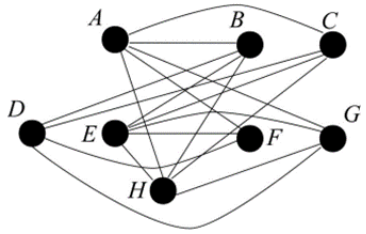
La **regione di fattibilità** include tutti i sottoinsiemi di vertici i cui archi *non sono adiacenti*.

$$X = \{x \subseteq B : (i,j) \notin E \; \forall i \in x, \forall j \in x \setminus \{i\}\}$$

L'obiettivo è quello di massimizzare il peso dei vertici selezionati.

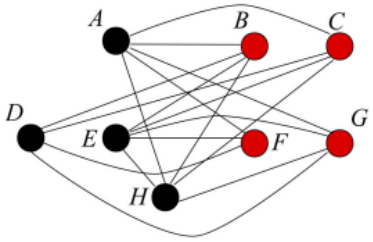
$$\max_{x \in X} f(x) = \sum_{j \in x} w_j$$

Consideriamo il seguente grafo indiretto con *pesi uniformi*.



Dataset per il Maximum Independent Set

Adesso consideriamo la prima soluzione proposta:

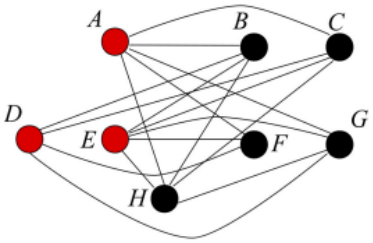


Prima soluzione per il MIS

$$x' = \{B, C, F, G\} \in X$$
$$f(x') = 4$$

La soluzione x' è una soluzione **fattibile**, ogni vertice del sottoinsieme proposto *non è connesso* con un altro vertice dello stesso sottoinsieme.

Consideriamo adesso una seconda proposta:



Seconda soluzione per il MIS

$$x'' = \{A, D, E\} \in X$$
$$f(x'') = 3$$

Anche la seconda soluzione è una **soluzione fattibile**.

4.8.4 Interludio 5: le relazioni tra i problemi

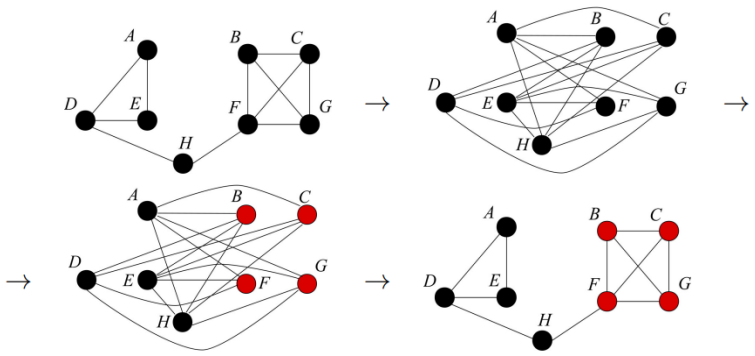
Come al solito le domande che uno si dovrebbe porre per un qualsiasi problema sono le solite :

- *Come si calcola la funzione oggettiva?*
- *Cosa succede se togliamo un vertice?*
- *Come si verifica la fattibilità?*
- *Cosa succede se aggiungiamo o rimuoviamo un vertice ad una soluzione fattibile?*

Questi ultimi tre problemi sui grafi che abbiamo affrontato erano molto simili. Dovrebbe essere già noto dalla **teoria delle complessità computazionali** che alcuni problemi possono essere **ridotti** ad altri problemi, e che si possa *utilizzare* un problema per risolverne un altro.

Un chiaro esempio è il seguente:

- Si parte dall'istanza iniziale del MCP, con un grafo $G = (V, E)$.
- Si costruisce il **grafo complementare** $\overline{G} = (V, (V \times V) \setminus E)$ (un grafo tale per cui i vertici adiacenti nel grafo originario ora non lo sono più e viceversa).
- Si cerca una soluzione ottima per il MISP su $\overline{G}$.
- I vertici corrispondenti danno una soluzione ottimale del MCP su G.

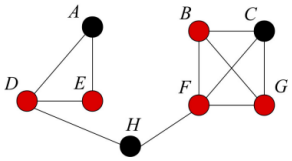


Una soluzione euristica MISP che da una soluzione euristica MCP

Questo processo può essere applicato anche nel verso opposto.

Anche i problemi SCP (*Set Covering Problem*) e VCP (*Vertex Covering Problem*) P hanno una relazione tra di loro, ma in maniera differente; ogni istanza del VCP può essere trasformata in un istanza del SCP.

- Ogni arco i corrisponde ad una riga della matrice A.
- Ogni vertice j corrisponde ad una colonna A.
- Se l'arco i tocca il vertice j, l'insieme $a_{ij} = 1$, altrimenti $a_{ij} = 0$.
- Una soluzione ottimale del SCP da una soluzione ottimale del VCP.



	A	B	C	D	E	F	G	H
(A, D)	1	0	0	1	0	0	0	0
(A, E)	1	0	0	0	1	0	0	0
(B, C)	0	1	1	0	0	0	0	0
(B, F)	0	1	0	0	0	1	0	0
(B, G)	0	1	0	0	0	0	1	0
(C, F)	0	0	1	0	0	1	0	0
(C, G)	0	0	1	0	0	0	1	0
(D, E)	0	0	0	1	1	0	0	0
(D, H)	0	0	0	1	0	0	0	1
(F, G)	0	0	0	0	0	1	1	0
(F, H)	0	0	0	0	0	1	0	1

Una soluzione euristica SCP che da una soluzione euristica VCP

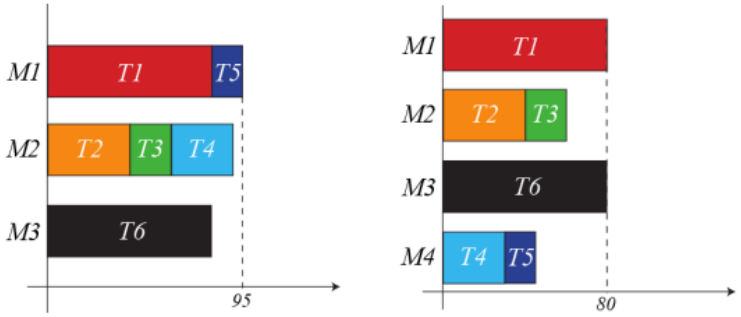
In questo caso non è semplice effettuare il procedimento inverso.

I problemi BPP (*Bin Packing Problem*) e PMSP (*Parallel Machine Scheduling Problem*) sono equivalenti, ma in una maniera più sofisticata:

- I task corrispondono a gli oggetti.
- Le macchine corrispondono ai container, ma ricordiamo che il BPP cerca di ottimizzare il numero di container (data una capacità), mentre il PMSP dato un numero di macchine cerca di ottimizzare il *tempo di completamento*.

Partiamo da un istanza del BPP:

- Facciamo un assunzione sul numero di container ottimali, per esempio 3.
- Costruiamo una corrispondente istanza nel PMSP.
- Calcoliamo il tempo di completamento ottimale, per esempio 95; se eccede la capacità, incrementa l'assunzione fatta precedentemente (tipo 4 o 5). Nel caso contrario, decrementa l'assunzione fatta (2 o 1).



Una soluzione euristica PMSP che da una soluzione euristica BPP

Il processo inverso è possibile. I due problemi sono equivalenti, ma ognuno dei due deve venire risolto molteplici volte.

È importante sottolineare il fatto che in caso di **riducibilità**, una soluzione euristica all'interno di un istanza ridotta è una soluzione euristica per il problema originale; studiare le relazioni tra i problemi è importante anche senza pensare agli algoritmi.

4.8.5 Travelling Salesman Problem (TSP)

Dato:

- Un grafo **diretto** $G = (N, A)$
- Una funzione $c : A \to \mathbb{N}$ che provvede i costi per ogni arco.

Si vuole selezionare un ciclo di costo minimo che visiti tutti i nodi del grafo. Il **ground set** è l'insieme degli archi.

$$B \equiv A$$

La **regione di fattibilità** include i cicli che visitano tutti i nodi del grafo (**cicli Hamiltoniani**).

L'obiettivo è minimizzare il costo totale degli archi selezionati.

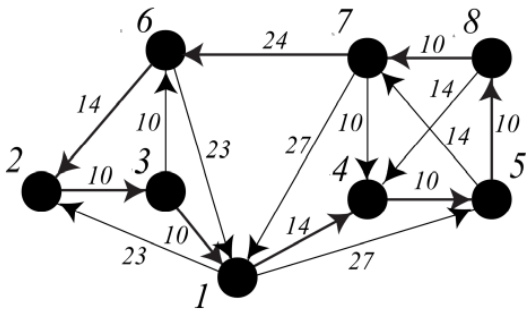
$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Come determinare quando un sottoinsieme è una soluzione fattibile? Quel sottoinsieme deve identificare un ciclo sul grafo ed ogni nodo deve avere esattamente un arco entrante ed uno uscente (ma sempre facente parte del sottoinsieme). Inoltre, una visita del grafo utilizzando gli archi del sottoinsieme dovrebbe visitare tutti i nodi, in altre parole non sono previste sotto visite di G.

Cosa accade se si effettua una modifica di una soluzione fattibile? Questo dipende dal tipo della modifica effettuata sulla soluzione, potrebbe essere necessario ricalcolare la funzione oggettiva.

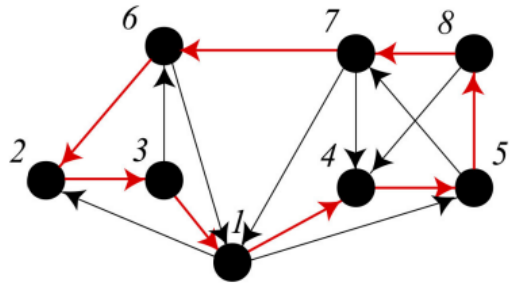
Risulta difficile trovare una soluzione fattibile? Trovare una soluzione che sia fattibile potrebbe essere altrettanto difficile; in generale un grafo Hamiltoniano (ovvero che presenta un ciclo Hamiltoniano) è un problema NP-completo, è di risoluzione banale solo nei grafi **completi**.

Consideriamo il seguente grafo diretto e pesato:



Dataset per il Travelling Salesman Problem

Consideriamo la prima soluzione proposta:

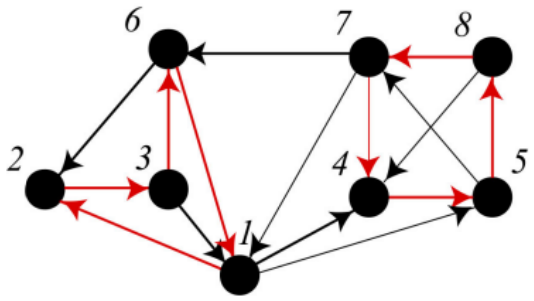


Prima soluzione del TSP

$$x' = \{(1,4),(4,5),(5,8),(8,7),(7,6),(6,2),(2,3),(3,1)\} \in X$$
$$f(x') = 102$$

La soluzione x' è una soluzione **fattibile**.

Consideriamo la seconda soluzione proposta:



Seconda soluzione del TSP

$$x'' = \{(4,5),(5,8),(8,7),(7,4),(1,2),(2,3),(3,6),(6,1)\} \notin X$$
$$f(x'') = 106$$

La soluzione x'' è una soluzione **fattibile**.

4.8.6 Minimum Capacitated Spanning Tree Problem (MCSTP)

Dati:

- Un grafo indiretto $G = (V, E)$ con un vertice radice $r \in V$
- Una funzione $c : E \to \mathbb{N}$ che provvede il *costo* di ogni arco.
- Una funzione $w : V \to \mathbb{N}$ che provvede il peso di ogni vertice.
- Un numero $W \in \mathbb{N}$ che è la capacità di ogni sotto albero.

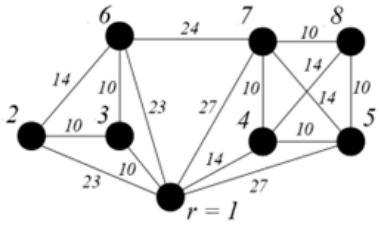
Si vuole selezionare un **minimo albero ricoprente** tale che ogni ramo (sotto albero rispetto alla radice) rispetti la capacità massima W. Il **ground set** è l'insieme degli archi.

$$B \equiv E$$

La **regione di fattibilità** include tutti gli alberi ricoprenti tali che il peso costituito dai vertici rispetti la capacità W. L'obiettivo è quello di minimizzare il costo totale degli archi selezionati.

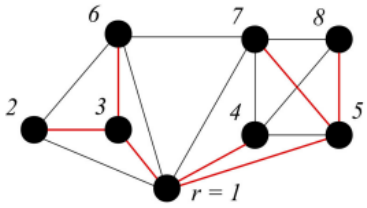
$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Consideriamo il seguente grafo indiretto di pesi uniformi $w_i = 1 \forall i \in V$, e con una capacità massima dei sotto alberi $W = 3$.



Dataset del Minimum Capacitated Spanning Tree Problem

La prima soluzione candidata:

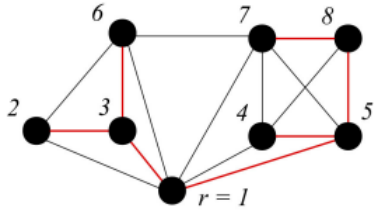


Prima soluzione del MCSTP

$$x' = \{(r,3),(3,2),(3,6),(r,4),(r,5),(5,7),(5,8)\} \in X$$
$$f(x') = 95$$

Questa è una soluzione **fattibile**, visto che ogni sotto albero non eccede la capacità W e i sottoinsiemi sono un minimo albero ricoprente del grafo.

La seconda soluzione candidata:



Seconda soluzione del MCSTP

$$x'' = \{(r,3),(3,2),(3,6),(r,4),(r,5),(5,7),(5,8)\} \notin X$$
$$f(x'') = 87$$

Questa **non è** una soluzione fattibile, visto che il sotto albero destro non rispetta la capacità massima (nonostante il sottoinsieme sia un albero ricoprente).

Il costo delle operazioni principali cambia, il testi di fattibilità richiede solamente la somma dei pesi, calcolare la funzione oggettiva richiede risolvere un problema MST (*Minimum Spanning Tree*).

La funzione oggettiva risulta:

- **Lenta da valutare**, calcolare un MST per ogni sottoinsieme.
- **Lenta da aggiornare**, ricalcolare un MST per ogni sottoinsieme modificato.

Se il grafo è **completo**, i test di ammissibilità sono:

- **Veloci da eseguire**, sommare i pesi dei vertici per ogni sotto albero.
- **Veloci da aggiornare**, sommare i pesi aggiunti e sottrarre quelli rimossi.

Visto che la funzione oggettiva è **additiva**, è abbastanza semplice da valutare, ma mentre è semplice da ricalcolare, è più difficile verificarne la fattibilità, eccetto per situazioni banali.

Trovare uno **albero ricoprente capacitato** è un problema fortemente NP-completo, quindi spesso è difficile trovare una soluzione fattibile a meno che il grafo sia completo. Dato un insieme di vertici, in ordine si controlla se la soluzione è una soluzione fattibile o meno, è necessario costruire la corretta rappresentazione dell'albero e poi visitare il sotto albero, sommare i pesi dei vertici, in una maniera simile ad una visita DFS.

Descrizione alternativa

4.8.7 Vehicle Routing Problem (VRP)

4.8.8 Interludio 6: combinare rappresentazioni alternative

Il CMSTP ed il VRP condividono una complicazione interessante: differenti definizioni del ground set B sono possibili e naturali.

- La descrizione come insieme di archi sembra preferibile per gestire la funzione oggettiva.
- La descrizione che utilizza un insieme di coppie del tipo $(vertice, albero)/(nodo/ciclo)$ sembra migliore per generare soluzioni ottime ed ha a che fare con la fattibilità.

Quale descrizione dovrebbe essere adottata? Quella che rende più semplice le operazioni più frequenti, oppure una possibilità risulta quella di utilizzare entrambe le rappresentazioni se le operazioni sono usate molto più frequentemente che quanto sono aggiornate, in maniera che il fardello di mantenerle aggiornate e consistenti sia accettabile.

Per riassumere, le domande che ci si deve chiedere a riguardo di un problema sono:

- Come si calcola la funzione oggettiva?
- Come provo che un sottoinsieme sia fattibile?
- Come trovo una soluzione fattibile?
- Come valutare il test di fattibilità?
- Cosa succede quando un cambiamento alla soluzione fattibile viene effettuato: è ancora fattibile? Sicuramente non lo è più? È necessario rivalutare da capo la funzione oggettiva o c'è una soluzione migliore?
- Quale è la definizione corretta di ground set? Sono qui presenti più definizioni possibili?
- Sono presenti relazioni tra questo ed un altro problema?

5 Efficienza teorica

La seconda parte di questo corso è dedicata alle caratteristiche degli algoritmi euristici: abbiamo precedentemente descritto gli algoritmi euristici come algoritmi che non provvedono sempre soluzioni corrette, ma che sono caratterizzati da due aspetti:

- Costa molto meno degli algoritmi corretti.
- *Spesso* restituisce qualcosa che è *vicino* alla soluzione corretta.

Considereremo questi due aspetti, **costi** e **qualità**, significa la distanza e la probabilità di ottenere una certa qualità e considereremo essi da due punti di vista:

- *Analisi a priori*, basata sulla teoria.
- *Analisi a posteriori*, basata sull'evidenza e sui dati empirici ottenuti dall'esecuzione dell'algoritmo su dataset appositi per test.

5.1 Problemi

Informalmente, un problema è una domanda su un sistema costituito da oggetti matematici. La stessa domanda può essere spesso posta su diversi sistemi simili.

- Un istanza $I \in \mathcal{I}$ consiste in ogni specifico sistema riguardante la domanda.
- Una soluzione $S \in \mathcal{S}$ è una risposta corrispondente ad una delle istanze.

Per esempio: *n è un numero primo?*, questo è un problema con infinite istanze e due soluzioni.

$$\mathcal{I} = \mathbb{N}^+ \setminus \{1\} \text{ and } \mathbb{S} = \{\text{yes,no}\}$$

Formalmente, un problema è una funzione che relaziona le istanze e le soluzioni:

$$P : \mathcal{I} \to \mathcal{S}$$

Definire una funzione non significa sapere come calcolarla.

5.2 Algoritmi

Un algoritmo è una **procedura formale**, composta da passi elementari posti in una sequenza finita, ogni uno è determinato da un input e dai risultati dei passi precedenti.

Un algoritmo per un problema P è un algoritmo tale per cui un input $I \in \mathcal{I}$ restituisce una soluzione $S_I \in \mathcal{S}$.

$$A : \mathcal{I} \to \mathcal{S}$$

Un algoritmo definisce una funzione ed il modo per calcolarla, questo può essere:

- **Esatto**, se la funzione associata coincide con il problema.
- Altrimenti **euristico**.

Un algoritmo euristico è utile se risulta:

- **Efficiente**, significa che costa molto meno dell'algoritmo esatto.
- **Efficace**, significa che restituisce frequentemente la soluzione *"vicina"* a quella corretta.

5.3 Costi di un algoritmo euristico

Il costo di un algoritmo (entrambi i tipi) denotano il costo di computazione durante esecuzione:

- **Tempo**, richiesto per terminare la sequenza finita di passi elementari.
- **Spazio**, quello occupato in memoria dai risultati dei passi precedenti.

Il costo in tempo viene molto più discusso perché lo spazio è una risorsa rinnovabile, mentre il tempo non lo è. Utilizzare lo spazio richiede di utilizzare meno tempo possibile, in oltre è tecnicamente più semplice distribuire l'utilizzo di spazio che quello del tempo. Lo spazio ed il tempo sono parzialmente intercambiabili, è possibile ridurre il costo di uno incrementando l'utilizzo dell'altro.

5.3.1 Il tempo

Il tempo richiesto per risolvere un problema dipende da diversi aspetti:

- L'**istanza** specifica da risolvere.
- L'**algoritmo** utilizzato.
- La **macchina** che sta eseguendo l'algoritmo.

La nostra misura di tempo computazionale dovrebbe essere:

- **Sconnessa** dalla tecnologia, che sia la stessa su macchine differenti.
- **Coincisa**, che viene riassunta in una semplice espressione simbolica.
- **Ordinale**, che sia sufficiente per essere comparata con diversi algoritmi.

Il tempo computazionale in secondo per ogni istanza viola tutti i requisiti.

5.3.2 Complessità asintotica nel caso peggiore

La complessità asintotica di un algoritmo nel **caso peggiore** provvede una tale misura attraverso i seguenti passaggi:

1. Definire il tempo come un numero T di operazioni elementari eseguite.
2. Definire la dimensione di un istanza come opportuno valore di n.
3. Trovare il caso peggiore, ovvero il massimo valore di T su tutte le istanze di dimensione n.
$$T(n) = max_{I \in \mathcal{I}_n} T(I) \text{ , } n \in \mathbb{N}$$
Ora la complessità in tempo è una funzione $T : \mathbb{N} \to \mathbb{N}$.
4. Approssimare $T(n)$ da sotto e/o da sopra con una funzione più semplice $f(n)$, questo considerando solamente il loro comportamento asintotico (per $n \to \infty$).
5. Collezionare le funzioni in **classi** con la stessa *funzione di approssimazione*.

5.3.3 Gli spazi funzionali Θ

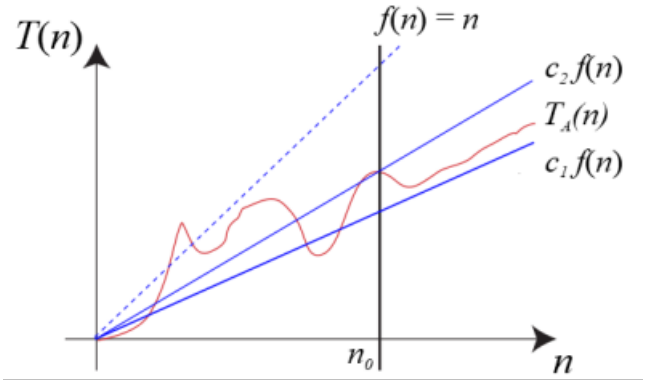
$$T(n) \in \Theta(f(n))$$

Formalmente significa:

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 f(n) \leq T(n) \leq c_2 f(n) \; \forall n \geq n_0$$

Dove c_1, c_2 e n_0 sono indipendenti da n. $T(n)$ è *"racchiusa"* tra $c_1 f(n)$ e $c_2 f(n)$, per:

- Alcuni *"piccoli"* valori di c_1.
- Alcuni *"grandi"* valori di c_2.
- Alcuni *"grandi"* valori di n_0.



$$f(n) = \Theta(T(n))$$

Asintoticamente, la funzione $f(n)$ stima la funzione $T(n)$ per un fattore moltiplicativo: per grandi istanze, il tempo computazionale è al meno ed al più il valore della funzione $f(n)$.

5.3.4 Gli spazi funzionali O

$$T(n) \in O(f(n))$$

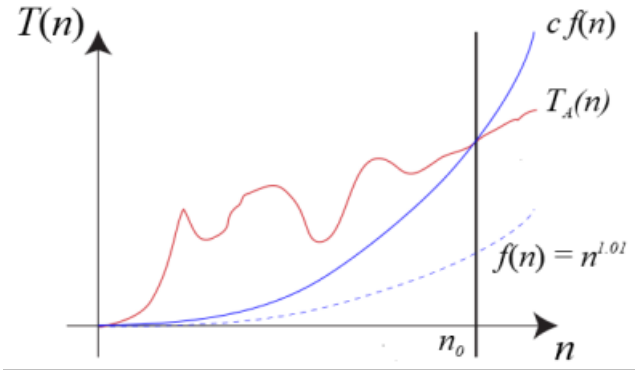
Formalmente significa:

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : T(n) \leq cf(n)\ \forall n \geq n_0$$

Dove c e n_0 sono indipendenti da n. $T(n)$ è *"dominata"* da $cf(n)$, per:

- Alcuni *"grandi"* valori di c.
- Alcuni *"grandi"* valori di n_0.

Asintoticamente, la funzione $f(n)$ sovrastima la funzione $T(n)$ per un fattore moltiplicativo: per grandi istanze, il tempo computazionale è al più proporzionale al valore della funzione $f(n)$



$$f(n) = O(T(n))$$

5.3.5 Gli spazi funzionali Ω

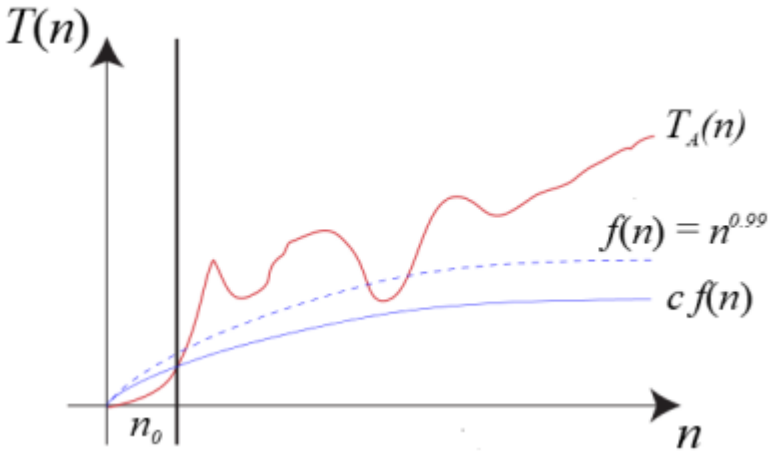
$$T(n) \in \Omega(f(n))$$

Formalmente significa:

$$\exists c > 0, n_0 \in \mathbb{N} : T(n) \geq cf(n)\ \forall n \geq n_0$$

Dove c e n_0 sono indipendenti da n. $T(n)$ *"domina"* $cf(n)$, per:

- Alcuni *"grandi"* valori di c.
- Alcuni *"grandi"* valori di n_0.



$$f(n) = \Omega(T(n))$$

Asintoticamente, $f(n)$ sovrastima $T(n)$ per un fattore moltiplicativo: per alcune grandi istanze, il tempo computazionale è almeno proporzionale al valore della funzione $f(n)$.

5.3.6 L'algoritmo esaustivo

Per i problemi di ottimizzazione combinatoria la dimensione di un istanza può essere misurata dalla cardinalità del ground set.

$$n = |B|$$

L'**algoritmo esaustivo**:

- Considera ogni sottoinsieme $x \subseteq B$ tale che $x \in 2^{|B|}$.
- Testi la fattibilità (se $x \in X$) in un tempo $\alpha(n)$.
- In caso positivo, risolve la funzione oggettiva $f(x)$ in tempo $\beta(n)$.
- Se necessario, aggiorna il miglior valore trovato finora.

La complessità in tempo degli algoritmi esaustivi è :

$$T(n) \in \Theta\left(2^n\left(\alpha(n) + \beta(n)\right)\right)$$

Questa risulta lo stesso esponenziale, anche se $\alpha(n)$ e $\beta(n)$ sono dei polinomi (caso più frequente). La maggior parte delle volte l'algoritmo esaustivo è **impraticabile**.

5.3.7 Complessità polinomiale ed esponenziale

Nei problemi di ottimizzazione combinatoria, la principale distinzione si trova tra:

- **Complessità polinomiale**, $T(n) \in O(n^d)$, dove $d > 0$ è costante.
- **Complessità esponenziale**, $T(n) \in \Omega(d^n)$, dove $d > 1$ è costante.

Nella prima famiglia fanno parte gli **algoritmi efficienti**, invece nella seconda quelli **inefficienti**. In generale, gli algoritmi euristici sono algoritmi polinomiali per problemi dove l'algoritmo esatto risulta esponenziale.

5.3.8 Problemi di trasformazione e riduzione

Una relazione tra problemi permette la progettazione di algoritmi (*interludio 5*).

Progettare un algoritmo per **trasformazione**:

1. Data un istanza di un problema P, detta I_P, trasformarla in una istanza di un altro problema Q, I_Q.
2. Data I_Q, applicare l'algoritmo A_Q per ottenere la soluzione S_Q.

3. Data S_Q, trasformiamola nella soluzione del problema P, S_P.

Progettare un algoritmo per **riduzione**:

1. Ripetere le trasformazioni $1, 2, 3$ diverse volte correggendo I_Q basato sulle soluzioni $\{S_Q\}$ già ottenute.

I due algoritmi spesso hanno complessità simili: se A_Q è polinomiale (o esponenziale) e:

- Costruire I_Q impiegherà tempo polinomiale/esponenziale.
- Il numero di iterazioni sarà polinomiale/esponenziale.
- Costruire S_P impiegherà tempo polinomiale/esponenziale.

Quindi A_P è un algoritmo *polinomiale/esponenziale*.
Il punto è che se si ha una trasformazione nella quale la modifica dell'istanza è polinomiale, ed il tempo in cui la trasformazione è applicata è polinomiale, allora la complessità di tutto l'algoritmo dipenderà dalla complessità di A_Q.

Oltre alla complessità del caso peggiore La complessità nel caso peggiore ha diversi svantaggi:

- Cancella tutte le informazioni delle istanze più semplici, poiché considera solamente le istanza con grandi dimensioni.
- Fornisce una approssimativa sovrastima del tempo computazionale, che in alcuni rari casi è inutile. Per esempio, i problemi di *programmazione lineare*, nei quali i problemi hanno infinite soluzioni ma *finite* soluzioni base, ed ammettono un algoritmo che ha complessità esponenziale.

  Tuttavia nella maggioranza dei casi gli algoritmi sono polinomi di bassa complessità.

Cosa se le istanze difficili sono rare nelle applicazioni pratiche? Per compensare, uno può investigare:

- La **complessità parametrizzata**, la quale introduce un nuovo rilevante parametro k (che è basato sulla dimensione di n), ed esprime il tempo come $T(n, k)$.
- La **complessità nel caso medio**, la quale si assume una probabilità di distribuzione su $\mathcal{I}$ ed esprime il tempo come il valore atteso.

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

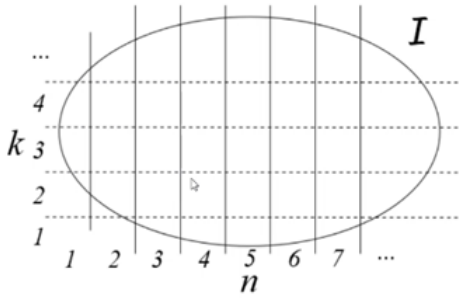
5.4 Complessità parametrizzata

L'idea è che alcuni algoritmi non sono esponenziali in n ma sono esponenziali in un altro parametro k che è più piccolo di n.

Se entrambi k e n sono grandi, allora ovviamente l'algoritmo è inefficiente, ma se k è piccolo, allora l'algoritmo sarà efficiente e possibilmente anche polinomiale.

Perciò:

- Efficiente su istanze con k piccolo.
- Inefficiente su istanze con k grande.



Insieme delle istanza infinitamente diviso dai parametri k ed n

Abbiamo tutti gli insiemi delle istanze del nostro problema, possiamo partizionare queste istanze con un numero infinito di fette per n e per l'altro parametro k. Possiamo considerare il nostro algoritmo.

Il parametro k può fare parte dell'input come la capacità, il massimo numero di letterali per formula (SAT), il numero di elementi non-zero (problema matrici numeriche), il massimo grado o il diametro (problemi con i grafi).

Stranamente il parametro aggiuntivo k può essere parte della soluzione (come la *cardinalità* nel VCP): in tale caso non puoi conoscere *a priori* se l'algoritmo è efficiente o meno, ma solo un approssimazione può essere disponibile. Quindi se k è facente parte della soluzione, solo *a posteriori* sarà possibile sapere se si tratta di un algoritmo efficiente.

5.4.1 Bounded tree search (algoritmo con parametro)

Per esempio, il VCP
L'algoritmo esaustivo: per ogni uno dei 2^n sottoinsiemi di vertici, testare se coprono tutti gli archi, calcolare la cardinalità e mantenere il valore più piccolo.

$$T(n,m) \in \Theta(2^n(m+n)), \text{ dove } n = |V| \text{ e } m = |E|$$

Ma se noi già sappiamo una soluzione con $f(x) = |x| = k+1$, possiamo cercare per una soluzione di k vertici, e progressivamente decrementare k (o ancora meglio utilizzare la **ricerca binaria** su k).

L'algoritmo naive: per ogni sotto insieme di k vertici, testare se copre tutti gli archi.

$$T(n, m, k) \in \Theta(n^k m)$$

Per una dato k fissato, l'algoritmo risulta di complessità polinomiale. Detto questo è possibile fare di meglio.

Bounded Tree Search per il VCP , un algoritmo migliore può essere basato sulla seguente proprietà:

$$x \cap (u, v) \neq \emptyset \; \forall x \in X, (u, v) \in E$$

Ovvero, una qualsiasi soluzione fattibile include almeno un estremo (vertice) di ogni arco.

Allora l'algoritmo **Bounded tree search** (*ricerca delimitata da gli alberi*), vuole trovare x tale che $x \leq k$:

1. Scegliere un qualsiasi arco (u, v), sia che $u \in x$ che $u \notin x$ e che $v \in x$ (ovvero che uno degli estremi non faccia per forza parte dalla soluzione).
2. Per ogni caso *"aperto"*, rimuovere i vertici di x e gli archi che essi coprono.

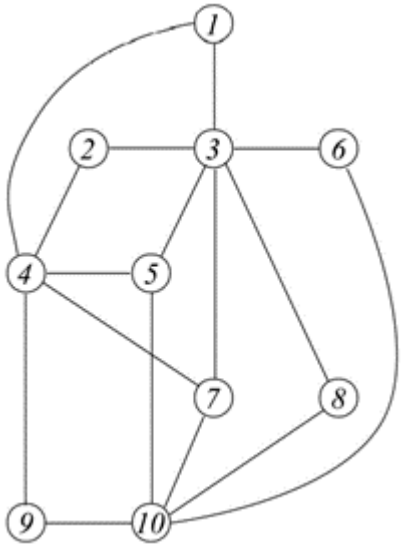
$$V := V \setminus x, E := E \setminus \{e \in E : e \cap x \neq \emptyset\}$$

3. Se $|x| \leq k$ e $E = \emptyset$, x è la soluzione richiesta.
4. Se $|x| = k$ e $E \neq \emptyset$, non è presente alcuna soluzione.
5. Altrimenti ritorna al passo 1.

La complessità è $T(n, m, k) \in \Theta(2^k m)$, polinomiale in n ($m < n^2$). Per $n >> 2$, questo algoritmo è molto più efficiente di quello *naive*.

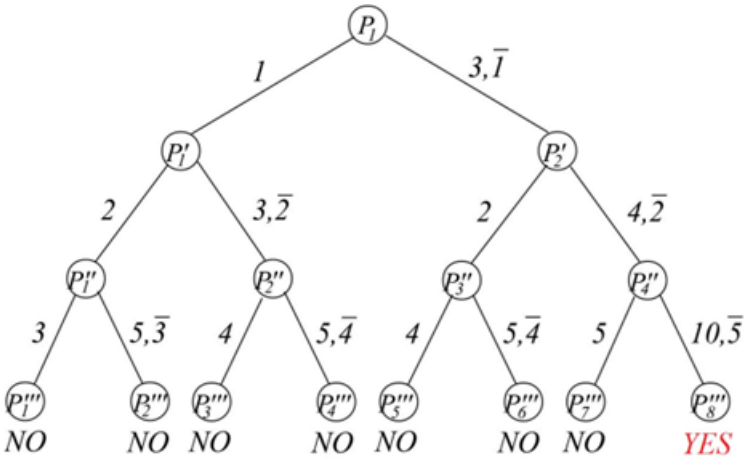
Esempio del Bounded tree search

Nel seguente grafo con $n = 10, m = 16$, *è presente una soluzione con* $|x| \leq 3$*?* (quindi $k = 3$).



Stato iniziale del grafo (*Bounded Tree Search*)

Algoritmo esaustivo: $\Theta(2^n(m+n))$, con $2^n(m+n) = 1024 \cdot (16+10)$
Algoritmo naive: $\Theta(n^k m)$, con $n^k m = 1000 \cdot 16$
Bounded tree search algorithm: $\Theta(2^k m)$ con $2^k m = 8 \cdot 16$



Soluzione del Bounded Tree Search (*ordinamento lessicografico*)

5.4.2 Kernelization - "Problem reduction" (algoritmo con parametro)

La *kernelizzazione* trasforma tutte le istanze di P in istanze più semplici di P, anziché istanze di un altro problema Q. Questa pratica è anche conosciuta con il nome di **problem reduction** (*riduzione del problema*).

Abbastanza spesso, in fatti, sono presenti delle proprietà molto utili che lo dimostrano:

- Esiste una soluzione ottimale che non include certi elementi di B (tali elementi possono essere rimossi).
- Esiste una soluzione ottimale che include certi elementi di B (tali elementi possono essere messi da parte ed aggiunti dopo).

In breve rimuove elementi di B senza modificare la soluzione. Possibili risultati utili sono:

- Un algoritmo esatto e polinomiale in n (complessità parametrizzata).
- Algoritmi esatti più veloci ed algoritmi euristici.
- Migliori soluzioni euristiche.
- **Heuristic kernelization**, applica condizione rilassata sacrificando l'ottimalità.

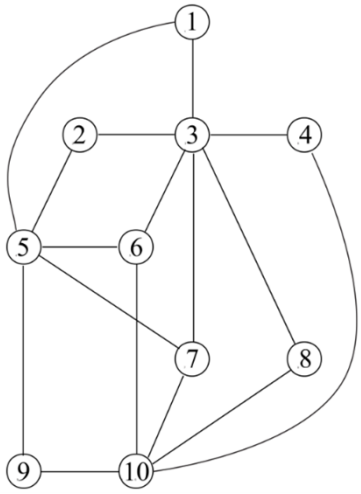
Tornando al problema del VCP, vediamo la **Kernelization del VCP**:
Se $\delta_v \geq k+1$, il vertice v appartiene ad una qualsiasi soluzione possibile di valore $\leq k$. Considerare che v ha $k+1$ archi incidenti e che dovrebbe essere coperto da altrettanti vertici.
Algoritmo di kernelizzazione per mantenere solo i vertici della soluzione x con $|x| \leq k$:

- Iniziare allo step $t = 0$ con $k_0 = k$ ed un sottoinsieme di vertici vuoto $x_t := \emptyset$
- Impostare $t = t+1$ ed aggiungere alla soluzione i vertici di grado $\geq k_t + 1$.
$$\delta_v \geq k_t + 1 \implies x_t := x_{t-1} \cup \{v\}$$
- Aggiornare k_t: $k_t := k_0 - |x_t|$
- Rimuovere i vertici con grado zero, quelli di x e degli archi coperti.
$$V := \{v \in V : \delta_v > 0\} \setminus x_t$$
$$E := \{e \in E : e \cap x_t = \emptyset\}$$
- Se $|E| > k_t^2$, non è presente alcuna soluzione fattibile (k_t vertici non sono abbastanza).

- Se $|E| \leq k_t^2 \implies |V| \leq 2k_t^2$, si applica l'algoritmo esaustivo.

La complessità è $T(n,k) \in \Theta(n + m + 2^{2k^2}k^2)$.

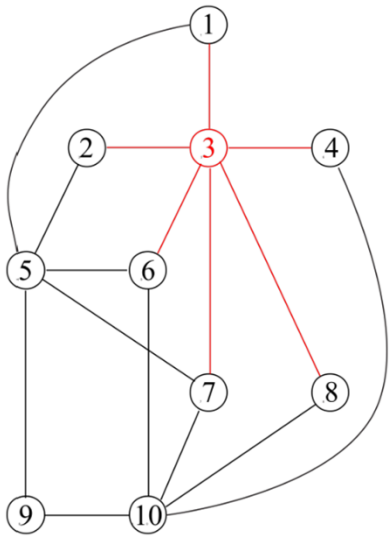
Per esempio, consideriamo il seguente grafo con $n = 10, m = 16$, *è presente una soluzione con* $|x| \leq k_0 = 5$*?*



Grafo d'esempio per la Kernelization del VCP

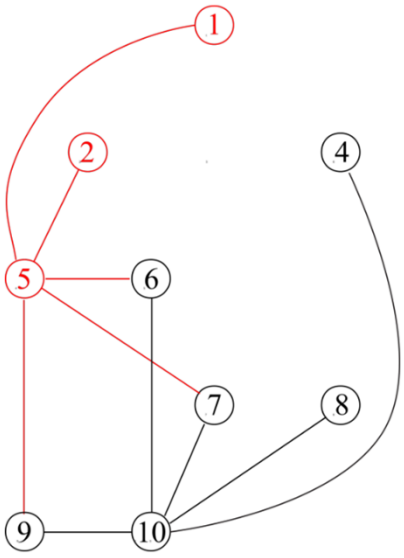
Algoritmo esaustivo:$\Theta(2^n(m+n)) \implies T \approx 2^{10}(10+16) = 26624$
Algoritmo naive: $\Theta(n^k m) \implies T \approx 10^5 \cdot 16 = 16000000$

$\delta_3 \geq k_0 + 1 \implies x_1 := \{3\}$ rimozione degli archi incidenti ed avremo $k_1 = 4$.

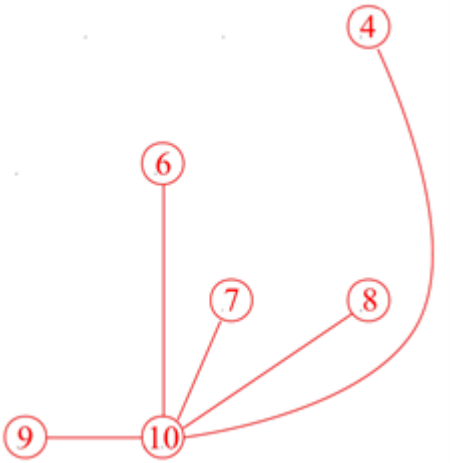


Rimozione degli archi incidenti con $k_1 = 4$

$\delta_1 0 \geq k_2 + 1 \implies x_3 := \{5, 10\}$ rimozione degli archi incidenti ed avremo $k_3 = 2$.



Rimozione degli archi incidenti con $k_2 = 3$



Rimozione degli archi incidenti con $k_3 = 2$

5.5 Complessità nel caso medio (average-case complexity)

Alcuni algoritmi sono inefficienti solo in un insieme di istanze ridotte, per esempio il *simplex algorithm* nel caso della programmazione lineare.

Gli studi teorici:

- Definiscono un modello probabilistico del problema. Il quale è una distribuzione probabilistica su $\mathcal{I}_n$ $\forall n \in \mathbb{N}$
- Calcolano il valore stimato di $T(I)$

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

Gli studi empirici:

- Costruiscono un *modello di simulazione* del problema, il quale è una distribuzione probabilistica di $\mathcal{I}_n$ $\forall n \in \mathbb{N}$ teorico o empirico (costruito).
- Costruire un benchmark di istanze casuali relativo alla distribuzione.
- Applicare l'algoritmo e misurare il tempo richiesto.

5.6 Modelli probabilistici per matrici numeriche

Data una matrice binaria (m righe e n colonne):

- **Equiprobabilità**, elencare tutte le 2^{mn} matrici binarie e selezionare una delle matrici con probabilità uniforme.

- **Probabilità uniforme**, impostare ogni cella ad 1 con una data probabilità p

  $$Pr[a_{ij} = 1] = p \ (i = 1, ..., m; j = 1, ..., n)$$

  Se $p = 0.5$, esso coincide con il modello di equiprobabilità, per un qualsiasi altro valore alcune istanze sono più simili di altre.

- **Densità fissa**, estrae δmn celle da mn con una probabilità uniforme e le imposta ad 1. Se $\delta = p$, esso assomiglierà al modello di probabilità uniforme, ma alcune istanze non potranno essere generate.

5.7 Modelli probabilistici per grafi

Dato un grafo casuale con un numero n di vertici:

- **Equiprobabilità**, elenca tutti i $2^{\frac{n(n-1)}{2}}$ grafi e seleziona quello con la probabilità uniforme.

- **Modello di Gilbert** o **probabilità uniforme** $G(n, p)$

  $$Pr[(i, j) \in E] = p \ (i \in V, j \in V \setminus \{i\})$$

  Tutti i grafi con lo stesso numero di archi m hanno la stessa probabilità $p^m(1-p)^{\frac{n(n-1)}{2}-m}$ (diverso per ogni m). Se $p = 0.5$, esso coincide con il modello di equiprobabilità.

- **Modello di Erdos-Renyi** $G(n, m)$: estrae m coppie di vertici non ordinate su $\frac{n(n-1)}{2}$ con probabilità uniforme e crea un arco per ognuna. Se $p = \frac{2m}{n(n-1)=p}$, esso assomiglierà al modello con probabilità uniforme, ma alcune istanze non verranno generate.

5.8 Modelli probabilistici per funzioni logiche

Considerando una CNF casuale con un dato numero di variabili n:

- **Insieme con probabilità fissa**: elenca tutte le $\binom{n}{k}2^k$ formule con k letterali distinti e consistenti, ed aggiungo ognuno alla CNF con probabilità p.

- **Insieme con dimensione fissa**: costruisco m formule, aggiungo ad ognuna k letterali distinti e consistenti, estratti con probabilità uniforme. Se $p = \frac{m}{\binom{n}{k}2^k}$, esso assomiglierà al modello con probabilità fissa, ma alcune istanze non potranno essere generate.

5.9 Transizione di fase

Differenti valori (deterministici o probabilistici) dei parametri corrispondono a diverse regioni dell'insieme delle istanze.

Per i grafi:

- $m = 0$ e $p = 0$ corrisponde al grafo vuoto.
- $m = \frac{n(n-1)}{2}$ e $p = 1$ corrisponde al grafo completo.
- Valori intermedi corrispondono al grafo di densità intermedia (deterministicamente per m, probabilisticamente per p).

Per molti problemi la **performance** dell'algoritmo è decisamente differente in regioni diverse, riguardo a:

- **Tempo computazionale** (per algoritmi esatti ed euristici).
- **Qualità della soluzione** (per algoritmi euristici).

Spesso, la *variazione* di performance avviene all'improvviso in piccole regioni dello spazio dei parametri, come la *transizione di fase* all'interno di un sistema fisico.

Questo è utile per predire il comportamento di un algoritmo data un istanza.

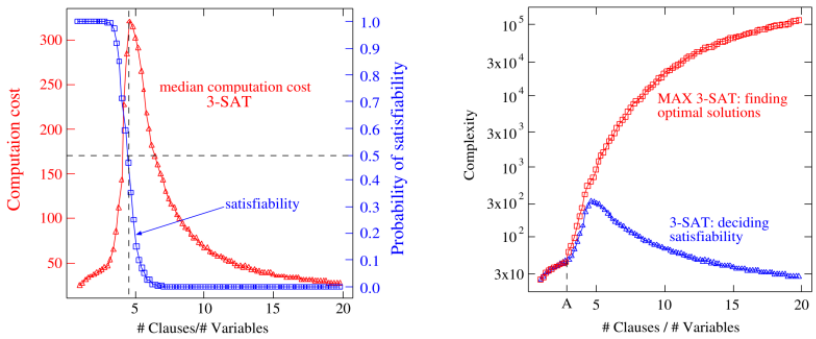
5.10 Transizione di fare per 3-SAT e Max-3-SAT

Data una CNF di n variabili e con formule logiche contenenti 3 letterali.

- 3-SAT: *è qui presente un assegnamento di verità che soddisfi tutte le formule ?*
- Max-3-SAT: *quale è il massimo numero di formule soddisfacibili?*

Con l'aumentare del rateo formule-variabili $\alpha = \frac{m}{n}$:

- Le istanze soddisfacibili decrementano da quasi tutte (tante variabili per poche formule) a quasi nessuna (poche variabili per tante formule).
- Il tempo di calcolo per il 3-SAT incrementa fortemente e poi subito decresce, invece nel Max-SAT incrementa ulteriormente dopo la transizione.



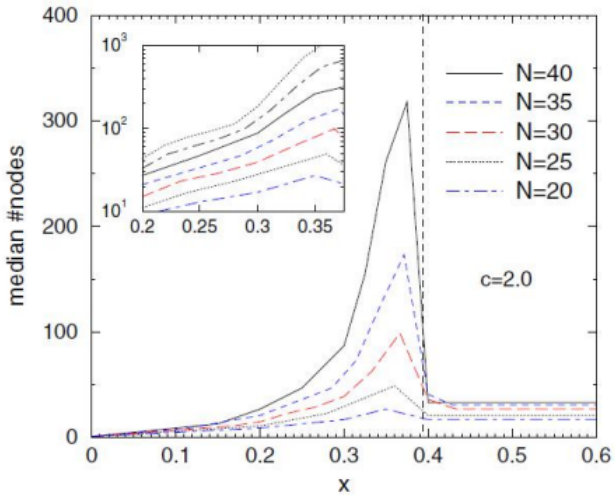
Plot delle fasi

Per $n \to \infty$ la transizione si concentra attorno $\alpha_c \approx 4.26$

5.11 La transizione di fase per il VCP

Il VCP esibisce una transizione di fase simile alla precedente quando $\frac{|x|}{|V|}$ incrementa.

- Il tempo computazionale prima esplode, dopo cade.
- Per $n \to \infty$ la transizione si concentra attorno al valore critico.



Fase del VCP

Quando $\frac{|x|}{|V|}$ è piccolo, alcuni vertici sono chiaramente necessari, problema risolto. Quando $\frac{|x|}{|V|}$ è grande, molti vertici sono chiaramente necessari, problema risolto.

5.12 Costo computazionale degli algoritmi euristici

La complessità in tempo di un algoritmo euristico è solitamente:

- **Strettamente polinomiale** (con esponenti bassi).
- **Abbastanza robusta** rispetto ai parametri secondari.

Perciò, la stima nel caso peggiore è buona nella media dei casi. Le **metaeuristiche** utilizzano passi casuali o memoria:

- La complessità è ben definita per componenti singoli dell'algoritmo.
- La complessità globale non è chiaramente definita, in teoria potrebbe essere estesa *indefinitamente*, in pratica, è definita da una condizione imposta dall'utente.

Perché discutere ciò in un corso di algoritmi euristici ?

- Per guidare la ricerca dell'algoritmo corretto, un algoritmo corretto può essere efficiente in un caso specifico, ed inefficiente in un caso peggiore.
- Per mostrare che gli algoritmi euristici possono interagire proficuamente: poiché gli algoritmi euristici provvedono informazioni per migliorare gli algoritmi esatti (rendendoli più efficienti).
- Per mostrare che la kernelization migliora gli algoritmi euristici (diventano più efficienti ed efficaci).
- Per identificare *a priori* le istanze più ardue, certamente non tutti gli algoritmi hanno le stesse istanze *"complesse"*.

6 Efficacia teorica

6.1 Efficacia di un algoritmo euristico

Un algoritmo euristico è utile se è:

- **Efficiente**, ovvero *"costa"* molto meno di un algoritmo esatto.
- **Efficace**, ovvero restituisce *"frequentemente"* una soluzione che è vicina a quella esatta.

Adesso vogliamo discute dell'efficacia degli algoritmi euristici, possiamo formalmente discutere questo concetto introducendo il concetto di distanza di una soluzione rispetto ad una ottimale e la frequenza (probabilistica) di ottenere una soluzione ottimale o quasi (rispetto ad una data distanza) rispetto ad una soluzione ottima. Queste caratteristiche possono essere combinate in una distribuzione a frequenza di soluzioni più o meno vicine alla soluzione ottimale (andrà introdotto un concetto di distanza).

L'efficacia di un algoritmo euristico può essere investigata in due modi:

- **Analisi teorica** (*a priori*), provando che un algoritmo trova sempre o con una data frequenza soluzioni con una certa garanzia di qualità.
- **Analisi sperimentale** (*a posteriori*), misurando le prestazioni dell'algoritmo su istanze di benchmark campionate per dimostrarne la presenza.

6.2 Distanza tra le soluzioni

Come detto precedentemente, dobbiamo dare un significato al concetto di distanza, e possiamo fornire molteplici interpretazioni differenti.
L'efficacia di un algoritmo di ottimizzazione euristico A è misurata dalla **differenza** tra il valore $f_A(I)$ (la funzione oggettiva sull'istanza I calcolata da A) e il valore ottimale $f*(I)$ (la soluzione migliore per l'istanza I).

6.2.1 Differenza assoluta

Una possibile definizione iniziale tra due valori:

$$\widetilde{\delta}_A(I) = |f_A(I) - f^*(I)|$$

Seppure questa definizione è molto naturale ed ovvia, è raramente utilizzata in pratica poiché dipende dall'unità di misura della funzione oggettiva: basa pensare di provare a minimizzare il tempo complessivo nel caso del TSP e che lo si misuri in giorni. Si potrebbe provare ad utilizzare ore, minuti o secondi anziché ed il valore di $\widetilde{\delta}_A(I)$ dipenderà fortemente da questa scelta. Questo sarà valido quando la funzione oggettiva è un numero puro.

6.2.2 Differenza relativa

La seconda definizione è basata sull'idea che calcolare la differenza relativa rispetto alla soluzione ottimale.

$$\delta_A(I) = \frac{|f_A(I) - f^*(I)|}{f^*(I)}$$

Questo è un approccio molto frequente nell'analisi sperimentale ed il suo vantaggio è che nel caso in cui la funzione oggettiva abbia un unità di misura, allora il rapporto (*rateo*) ottenuto sarà un numero *puro*.

6.2.3 Rapporto di approssimazione

Un terzo approccio, ampiamente utilizzato, è il rapporto di approssimazione:

$$\rho_A(I) = max\left[\frac{f_A(I)}{f^*(I)}, \frac{f^*(I)}{f_A(I)}\right] \geq 1$$

Il quale include sia il problema di minimizzazione che di massimizzazione: nel caso in cui sia un problema di minimizzazione, probabilmente $f*(I) < f_A(I)$ e viceversa per i problemi di massimizzazione. Chiaramente è collegato alla differenza relazionale, in fatti nel caso dei problemi di minimizzazione:

$$\delta_A(I) = \rho_A(I) - 1$$

La relazione esiste anche per i problemi di massimizzazione.

6.3 Analisi teorica: garanzia di approssimazione

Per garantire *a priori* le prestazioni dell'algoritmo, l'idea è ancora una volta quella di considerare il caso peggiore, proprio come per l'efficienza. In generale quando si applica un algoritmo euristico il risultato $f_A(I)$ potrebbe essere un pessimo (distante) risultato rispetto alla soluzione ottimale $f*(I)$, ma se l'algoritmo è buono la differenza non sarà molto grande;

La differenza tra $f_A(I)$ e $f^*(I)$ è generalmente illimitata, ma per alcuni algoritmi è limitata:

- **Approssimazione assoluta**:

  $$\exists\widetilde{\alpha}_A \in \mathbb{N} : \widetilde{\delta}_A(I) \leq \widetilde{\alpha}\ \forall I \in \mathcal{I}$$

  Significa che la la distanza assoluta rispetto alla soluzione ottimale è limitata da un intero costante; un esempio è l'algoritmo di Vizing per la colorazione del grafo ($\widetilde{\alpha} = 1$).

- **Approssimazione relativa**:

  $$\exists\alpha_A \in \mathbb{R}^+ : \rho_A(I) \leq \alpha_A\ \forall I \in \mathcal{I}$$

  Significa che il rapporto di approssimazione $\rho_A(I)$ ha un limite superiore delimitato da una costante reale α_A. La cosa interessante è che la definizione può essere estesa a casi in cui una garanzia di approssimazione costante non può essere trovata considerando l'approssimazione come un valore ma come un'apposita funzione parametrizzata rispetto alla dimensione dell'istanza:

  $$\rho_A(I) \leq \alpha_A(n)\ \forall I \in \mathcal{I}_n, n \in \mathbb{N}$$

In conclusione, è importante sottolineare che mentre l'efficienza e è necessariamente dipendente dalla dimensione dell'istanza, l'efficacia *potrebbe* non esserlo.

6.3.1 Come ottenere una garanzia di approssimazione?

Fornite le basi teoriche, ora possiamo descrivere in maniera generale un metodo per introdurre e dimostrare un algoritmo con garanzia di approssimazione. Per i problemi di minimizzazione, si vuole dimostrare:

$$\exists \alpha_A \in \mathbb{R} : f_A(I) \leq \alpha_A f^*(I) \; \forall I \in \mathcal{I}$$

Uno schema generalmente astratto è:

- Trovare un modo per costruire una sottostima $LB(I)$

$$LB(I) \leq f^*(I), I \in \mathcal{I}$$

- Trovare un modo per costruire una sovrastima $UB(I)$ che sia in relazione con il coefficiente α_A

$$UB(I) = \alpha_A LB(I), I \in \mathcal{I}$$

- Trovare un algoritmo A la cui soluzione non sia peggiore di $UB(I)$

$$f_A(I) \leq UB(I) \; I \in \mathcal{I}$$

Se i tre passi sono stati completati e vengono trovati valori ed algoritmi adatti

$$f_A(I) \leq UB(I) = \alpha LB(I) \leq \alpha f^*(I) \; \forall I \in \mathcal{I} \implies f_A(I) \leq \alpha_A f^*(I) \; \forall I \in \mathcal{I}$$

La parte più complicata potrebbe essere il secondo passo, quindi affrontiamo mostrando un esempio.

Garanzia di approssimazione per il VCP: algoritmo 2-approssimato : dato un grafo indiretto $G = (V, E)$ trovare un sottoinsieme di vertici di cardinalità minima tale per cui ogni arco dell'arco del grafo sia incidente.
Definiamo:

- Un insieme **"matching"**, tale per cui sia l'insieme degli archi non adiacenti.
- Un insieme **"maximal matching"**, è un insieme di tipo *matching* tale che ogni altro arco del grafo è adiacente ad uno dei suoi archi.

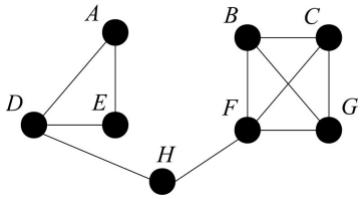
Algoritmo di Matching:

1. Costruire un *maximal matching* $M \subseteq E$ scorrendo gli archi di E e includendo in M quelli che non sono adiacenti ad M (ora ogni arco di $E \setminus M$ è adiacente ad un arco di M).

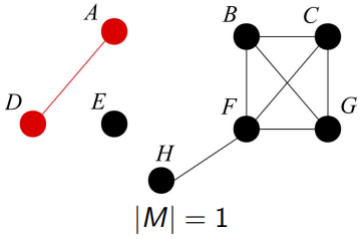
2. L'insieme di vertici *estremi* degli archi in *matching* è la soluzione del VCP

$$x_A := \bigcup_{(u,v)\in M} \{u, v\}$$

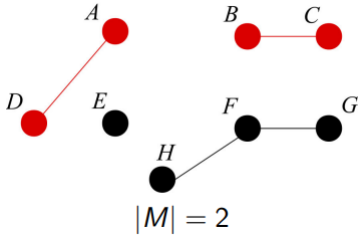
Può essere migliorata rimuovendo i vertici ridondanti.



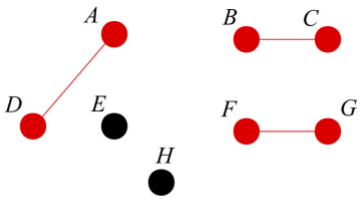
Algoritmo di Matching - Passo 0 (stato iniziale del grafo)



Algoritmo di Matching - Passo 1



Algoritmo di Matching - Passo 2



Algoritmo di Matching - Passo 3

Torniamo adesso alla garanzia di approssimazione. Un possibile limite inferiore è quello di includere nella soluzione solamente uno dei vertici dell'arco nel *maximal matching*: considero solo gli archi nel *maximal matching* M.

È chiaro che il grafo consiste nei soli archi $e \in M$ ($G' = (V, M)$) solo *metà* dei vertici inclusi nella soluzione x_A copre gli archi in M. In altre parole, la copertura ottimale dei vertici degli archi in M è certamente più piccola della copertura ottimale di tutti gli archi in E.

Per fornire un esempio numerico, supponiamo che un certo grafo per il VCP abbia la soluzione ottimale $f^* = 5$ (quindi 5 vertici possono coprire tutti gli archi) e che l'algoritmo di matching trovi il matching massimale con 3 archi, quindi la soluzione dell'algoritmo sarà $f_A = 2 \cdot |M| = 6$. Però coprire tutti gli archi in M (considerare questo insieme come un grafo), basteranno $\frac{|M|}{2} = 3$ vertici!

Il valore $\frac{|M|}{2}$ è un possibile limite inferiore (o *lower bound*) per il nostro problema VCP:

$$LB(I) = \frac{|M|}{2} \leq f^*(I)$$

Ma c'è dell'altro: sappiamo che x_A viene calcolato prima che sia una soluzione attualmente fattibile ed è collegata a $LB(I)$:

$$f_A = 2 \cdot |M| = 2 \cdot LB(I) = UB(I)$$

Quindi abbiamo trovato sia il limite inferiore che quello superiore. Adesso forniamo una dimostrazione formale che l'algoritmo di matching è **2-approssimato**:

Dimostrazione algoritmo di Matching 2-approssimato , per prima cosa dimostriamo che la cardinalità dell'insieme "matching" sia una sottostima $LB(I)$: la cardinalità di una copertura ottimale per un qualsiasi sottoinsieme di archi $E' \subseteq E$ non supera quella di una copertura ottimale per E

$$|x^*_{E'}| \leq |x^*_E|$$

e la copertura ottimale di un qualsiasi insieme di matching M ha cardinalità $\frac{|M|}{2}$, quindi

$$|M| \leq |E| \implies |x^*_M| \leq |x^*_E| \implies \frac{|M|}{2} \leq |x^*_E| \; \forall M$$

$$LB(I) = \frac{|M|}{2} \leq |x_e^*| \text{ è un limite inferiore valido}$$

In parole, visto che l'insieme M ha una cardinalità minore o uguale a quella di E, la soluzione ottimale (la quale ha cardinalità esatta di $\frac{|M|}{2}$) ha una cardinalità minore o uguale a quella della soluzione ottimale per l'insieme E, rendendola un valido limite inferiore.

Il fatto che $|M|$ stesso consiste in una sovrastima è provato dal fatto che la definizione dei vertici degli archi in M copre tutti gli archi in E, significa che l'insieme dei vertici M consiste in una soluzione fattibile (il che significa che è un limite superiore ed è un valore ottimale) e copre sia il "matching" che gli archi adiacenti. Perciò $|M| = UB(I) = 2 \cdot LB(I)$ è una sovrastima.

La prova si conclude con il fatto che l'algoritmo restituisce soluzioni di valore $f_A(I) \leq UB(I)$, visto che in M potrebbero essere presenti dei vertici *ridondanti* che possono essere rimossi. Questo implica che $f_A(I) \leq 2 \cdot f^*(I)\ \forall I \in \mathcal{I}$, che è $\alpha_A = 2$

Lo stretto legame tra stime inferiori e superiori , *sono presenti delle istanze $\overline{I}$ per il VCP tali che $f_A(\overline{I}) = 2 \cdot f^*(\overline{I})$ utilizzando l'algoritmo di Matching?* Più in generale, sono presenti delle istanze $\overline{I}$ tali che:

$$f_A(\overline{I}) = \alpha_A f^*(\overline{I})$$

e se così, *come sono le istanze $\overline{I}$?* In altre parole, dopo aver dimostrato che $f_A(I)$ è in relazione con $f^*(I)$ attraverso α_A, questo è **costante**, **preciso** e **importante** o e *"solo"* un limite superiore? Lo studio delle istanze $\overline{I}$ è utile per valutare se sono rare o frequenti e per introdurre modifiche *ad hoc* per migliorare l'algoritmo.

Garanzia di approssimazione per il TSP (disuguaglianza triangolare) , consideriamo il TSP con delle assunzione aggiuntive, ovvero che $G = (N, A)$ sia **completo** e che la funzione c sia simmetrica e soddisfi la **disuguaglianza triangolare**:

$$c_{ij} = c_{ij}\ \forall i, j \in \mathbb{N}$$

e che

$$c_{ij} + c_{jk} \geq\ \forall i, j, k \in \mathbb{N}$$

Il problema in generale, è fortemente NP-completo. Ma è anche fortemente NP-completo capire se le istanze hanno una soluzione fattibile (non quale sia fattibile), ovvero se è presente un ciclo Hamiltoniano che visiti tutti i nodi.

Sotto le precedenti assunzioni un algoritmo euristico può essere costruito con una garanzia di approssimazione associata ad esso.

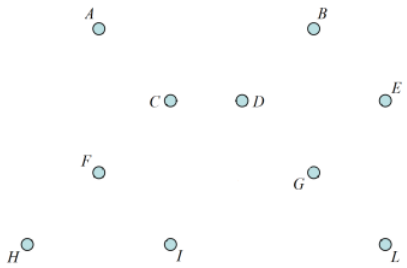
Algoritmo Double-tree

1. Considera il grafo completo e indiretto G
2. Costruisci un minimo albero ricoprente $T^* = (N, X^*)$

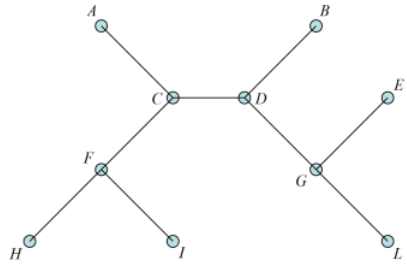
3. Fai una visita in pre-ordine di T^* e costruisci due liste di archi:

   (a) x sarà la lista degli archi utilizzata sia per la visita che per il backtracking: questo è un ciclo che visita ogni nodo, possibilmente molteplici volte.

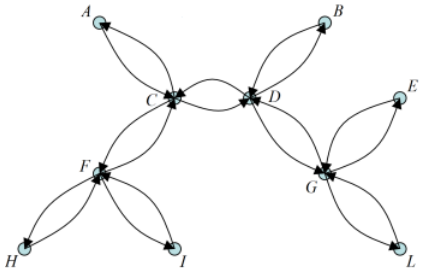
   (b) x' sarà la lista degli archi che connettono i nodi in pre-ordine con il primo nodo: questo è un ciclo che visita ogni nodo esattamente una volta.



Double-tree per TSP - Passo 1 (grafo completo con archi omessi)

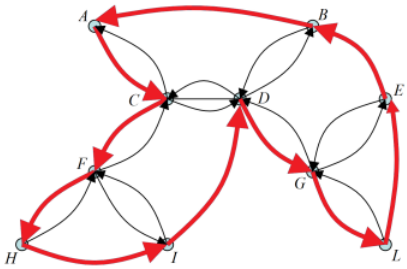


Double-tree per TSP - Passo 2 (*Minimo Albero Ricoprente* T^*)



Double-tree per TSP - Passo 3.a (costruzione x)

$$x = \{A, C, F, H, F, I, F, C, D, G, L, G, E, G, D, B, D, C, A\}$$



Double-tree per TSP - Passo 3.b (costruzione x')

$$x' = \{A, C, F, H, U, D, G, L, E, B, A\}$$

La struttura è la stessa del VCP: una sottostima è trovata assieme al fattore di moltiplicazione per costruire un limite superiore e quindi è provato che un algoritmo euristico può essere dominato da questo limite superiore, il che significa che l'algoritmo euristico è quasi sicuramente un fattore moltiplicato per la soluzione ottima. Andiamo a dimostrare che l'algoritmo double-tree è 2-approssimato.

Dimostrazione algoritmo Double-tree 2-approssimato ,