Università degli Studi di Verona

Complessità

Riassunto dei principali argomenti

Matteo Danzi, Marco Colognese, Davide Bianchi

18 luglio 2018

Indice

Elenco delle figure

Listings

1 Introduzione

1.1 Cos'è la complessità computazionale

Nella teoria della complessità ci si pone la seguente domanda:

> *Come scalano le risorse necessarie per risolvere un problema all'aumentare delle dimensioni del problema?*

La teoria della *complessità computazionale* è una parte dell'informatica teorica che si occupa principalmente di classificare i problemi in base alla quantità di *risorse computazionali* (come il tempo di calcolo e lo spazio di memoria) che essi richiedono per essere risolti. Tale quantità è detta anche *costo computazionale* del problema.

1.2 Problemi *facili* e *difficili*

Vediamo quattro esempi di problemi che classificheremo come facili o difficili:

1. (**Eulerian Cycle**) Esiste un modo per attraversare ogni arco di un grafo una e una sola volta?

   - Il problema si può vedere anche nella forma più piccola del problema dei *sette ponti di Königsberg*:
     A Königsberg ci sono 7 ponti, esiste un percorso che attraversa tutti i ponti una e una sola volta per poi tornare al punto di partenza?
     Se avessi n ponti e su ogni riva partissero 2 ponti avrei 2^n possibili percorsi.
   - La **soluzione di Eulero** dice che un grafo connesso non orientato ha un percorso che parte e inizia esattamente nello stesso vertice e attraversa ogni arco esattamente una volta se e solo se ogni vertice ha grado dispari (grado = numero di archi uscenti).
     Se ci sono esattamente due vertici v e u, di grado dispari, allora esiste un percorso che parte da u e attraversa ogni arco esattamente una volta e finisce in v.
   - Seguendo quindi la soluzione di Eulero, *quanto costa decidere* se un grafo G ha un tour Euleriano?

     ```
     odd-vertex-num = 0;
     foreach vertex v of G
         if (deg(v) is odd)
             increment odd_vertex-num
     If(odd-vertex-num is neither 0 nor 2)
         output no Eulerian tour
     output Eulerian
     ```

     Questo algoritmo ha complessità: $O(|E| + |V|)$
     Il costo e l'algoritmo sono gli stessi se vogliamo *provare* che G *non* ha un tour Euleriano.

2. (**Hamiltonian Cycle**) Esiste un modo per attraversare ogni nodo di un grafo una e una sola volta?

   Esistono diverse soluzioni:

   - Provo tutte le possibilità ogni volta, costo: $O(2^n)$
   - Provo tutte le possibili permutazioni, costo: $O(n!)$
   - La soluzione migliore ad oggi è: $O(1.657^n)$

   Alla domanda: *Quanto costa decidere se un grafo ha un tour hamiltoniano?* Non sappiamo rispondere. Non sappiamo dire se il problema ha una soluzione non esponenziale. Per quanto ne sappiamo meglio di $O(1.657^n)$ non sappiamo fare.

   Non sappiamo nemmeno dire se Hamiltonian Cycle è più difficile di Eulerian Cycle.

3. N è un numero primo?

   Il migliore algoritmo conosciuto per decidere se N è un numero primo impiega $O((\log N)^{6+\epsilon})$

4. Quali sono i fattori primi di un numero?

   Ad oggi non conosciamo una procedure per fattorizzare un numero molto grande nei suoi divisori, che non sia provare tutte le possibilità.

1.3 Risolvere vs Verificare

La seguente tabella riassume in modo generico quanto detto nella sezione precedente riguardo alla difficoltà di risolvere problemi e verificare tali problemi su un istanza.

Tabella 1: Risolvere vs Verificare

Problema	**Risolvere**	**Verificare**
Eulerian Cycle	*facile*	*facile*
Hamiltonian Cycle	*difficile?*	*facile*
N è primo?	*facile*	*facile*
N ha un numero piccolo di fattori?	*difficile?*	*facile*

2 Problema computazionale

Un problema computazionale è una semplice relazione p che mappa l'insieme *infinito* di possibili input (domande o istanze) con un insieme *finito* (non vuoto) di output, cioè di risposte o soluzioni alle istanze.

$$\mathrm{p} : \text{ istanze infinite } \mapsto \text{ soluzioni finite alle istanze}$$

Un problema computazionale non è una singola domanda, ma è una **famiglia di domande**:

- Una domanda per ogni possibile istanza
- Ogni domanda è dello stesso tipo (appartiene alla stessa classe)

Esempio 2.0.1. Il seguente esempio è un problema computazionale:

- Input: Qualsiasi grafo G
- Domanda: Il grafo G contiene un ciclo Euleriano?

Esempio 2.0.2. Il seguente esempio *non* è un problema computazionale:

- Domanda: È vero che il bianco vince sempre a scacchi, sotto l'ipotesi della giocata perfetta?

Non è un problema computazionale perché non ho un insieme infinito di possibili partite in input.

2.1 Risolvere un problema computazionale

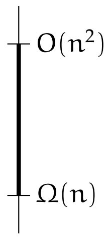
Risolvere un problema computazionale significa trovare un **algoritmo**, cioè una procedura che risolve il problema matematico in un numero finito di passi (di computazione elementare), che solitamente include la ripetizione di un operazione. È un procedimento deterministico che mappa l'input sull'output.

> Un algoritmo è una procedura *finita*, *definita*, *efficace* e con un input e un output.
>
> *Donald Knuth – The Art of Computer Programming*

2.2 Complessità di un problema computazionale

Misura della complessità. Come misuro la complessità di un problema computazionale? Come faccio a dire quanto è facile rispetto ad altri problemi?

- Do un **upper bound**: trovo un algoritmo qualsiasi che risolve il problema in modo da calcolare qual è il suo costo.
- Do un **lower bound**: trovo la minima quantità di risorse che ogni algoritmo utilizza per risolvere il problema. Tutti gli algoritmi sono *al minimo* complessi come il limite inferiore che abbiamo stabilito. Nessuno può fare di meglio.



2.3 Trattabilità di un problema.

La crescita della complessità di un problema è riducibile a 2 categorie fondamentali.

Crescita polinomiale. Un problema ha crescita polinomiale quando le risorse necessarie alla sua risoluzione sono limitate ad n^k, per qualche k. Se la taglia del problema aumenta, la sua complessità aumenta di un qualche fattore costante. Infatti, se la taglia dell'input va da n a $2n$ allora la complessità del problema si modifica in $(2n)^k = 2^k n^k$, ovvero aumenta di un fattore 2^k (costante). Raggruppiamo nella classe **P** i problemi di questo tipo.

Crescita esponenziale. Un problema ha crescita esponenziale la necessità di risorse necessarie alla sua risoluzione è proporzionale a c^n, per qualche costante $c > 1$. Se la taglia dell'input va da n a $2n$ c^n allora la richiesta di risorse si diventa $c^{2n} = c^n * c^n$, aumentando quindi di un fattore che cresce con l'aumentare di n. Raggruppiamo nella classe **Exp** i problemi di questo tipo.

3 Le classi di problemi computazionali

Notazione e idee di base. Formalmente definiamo un problema come un elemento $\mathbb{A}$ di una relazione

$$\mathcal{R} \subseteq \mathcal{I}(\mathbb{A}) \times \mathtt{Sol}$$

dove:

- $\mathcal{I}(\mathbb{A})$ è l'insieme delle istanze del problema $\mathbb{A}$
- $\mathtt{Sol}$ è l'insieme delle soluzioni delle istanze di $\mathbb{A}$

Si può quindi dire che

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathtt{Sol}(x) = \{\text{Soluzioni di } x\}$$

Non è restrittivo restringersi ai **problemi di tipo decisionale**, ovvero quei problemi che hanno come soluzione una risposta del tipo *si* o *no*, quindi i problemi del tipo

$$\mathbb{A} : \mathcal{I}(\mathbb{A}) \to \{yes, no\}$$

L'algoritmo $\mathcal{A}$ per un problema $\mathbb{A}$ è un algoritmo che dato il problema, $\forall x \in \mathcal{I}(\mathbb{A})$, $\mathcal{A}(x) = \mathbb{A}(x)$. Inoltre, dato un algoritmo $\mathcal{A}$, definiamo $T_{\mathcal{A}}(|x|)$ la sua **complessità**, cioè il *tempo che impiega* $\mathcal{A}$ sull'istanza di taglia $|x|$. Notare che $|x|$ è la taglia dell'istanza x.

3.1 Classe P

Intuitivamente la classe **P** è definita come la classe di problemi di **complessità polinomiale**. Introduciamo qui la definizione formale.

Definizione 3.1.1 (Classe P)**.** Definiamo la classe di problemi **P** come l'insieme dei problemi di complessità polinomiale, ovvero

$$\mathbf{P} = \left\{ \mathbb{A} \mid \exists \mathcal{A} \text{ t.c. } \exists c \text{ costante e } \forall x \in \mathcal{I}(\mathbb{A}),\ \ \mathcal{A}(x) = \mathbb{A}(x) \ \text{ e } \ T_{\mathcal{A}}(|x|) \leqslant |x|^c \right\}$$

Esempio 3.1.1 (Eulerian Cycle)**.** Un semplice esempio di problema appartenente alla classe **P** è il problema del tour euleriano. Per questo problema infatti abbiamo che è un problema computazionale di decisione:

- Input: grafo G
- Output: $yes \Leftrightarrow \exists$ Eulerian Cycle in G.

Come abbiamo già visto quindi:

$$\exists \mathcal{A} \text{ t.c. } T_{\mathcal{A}}(|G|) = O(|E| + |V|) = O(|G|)$$

Eulerian Cycle $\in$ **P** perché $\exists \mathcal{A}$ che impiega un tempo che è nell'ordine della taglia di G, in particolare $\exists c$ costante dove $c = 1$.

Esempio 3.1.2 (Hamiltonian Cycle)**.** Ci chiediamo allora se anche Hamiltonian Cycle $\in$ **P** ? La risposta è che non lo sappiamo dire. Quello che sappiamo per questo problema è che:

$$\exists \mathcal{A} \text{ t.c. } T_{\mathcal{A}}(|G|) = O(a^{|G|})$$

dove a è costante.

3.2 Classe Exp

Dal momento che non sappiamo se alcuni problemi stiano oppure no nella classe **P** (dal momento che non si conosce un algoritmo che li risolva in tempo polinomiale), si definisce la classe **Exp**, che racchiude tutte le istanze di questa tipologia di problemi di **complessità esponenziale**.

Definizione 3.2.1 (Classe **Exp**)**.** Definiamo la classe di problemi **Exp** come la classe di problemi di complessità esponenziale, ovvero

$$\mathbf{Exp} = \left\{ \mathbb{A} \mid \exists \mathcal{A} \text{ t.c. } \forall x \in \mathcal{I}(\mathbb{A}),\ \mathcal{A}(x) = \mathbb{A}(x) \ \text{ e } \ T_{\mathcal{A}}(|x|) \leqslant 2^{|x|^c} \right\}$$

Esempio 3.2.1 (Hamiltonian Cycle)**.** Ci chiediamo se Hamiltonian Cycle $\in$ **Exp** ? Se prendiamo l'algoritmo che prova tutte le combinazioni di archi cioè $\binom{|E|}{n}$ per vedere se formano un ciclo hamiltoniano. La complessità di quest'algoritmo è al massimo $2^{|E|^2}$.

Se invece prendiamo l'algoritmo che considera tutte le possibili permutazioni dei vertici del grafo abbiamo che la complessità è $n!$. Quindi il problema Hamiltonian Cycle $\notin$ **Exp**

Relazione tra P ed Exp. La domanda che sorge spontanea è **P** $\subseteq$ **Exp** ?

La risposta alla domanda è banalmente **si**, in quanto, dato un algoritmo $\mathcal{B}$ con complessità $T_{\mathcal{B}}(|x|)$, possiamo dire che

$$T_{\mathcal{B}}(|x|) = O(|x|^c) = O(2^{|x|^c}) \Rightarrow \mathbb{A} \in \mathbf{Exp}$$

Problema K-Graph-Colouring. Analizziamo ora il problema della K-colorabilità di un grafo G:

- Input: G non orientato.
- Output: $yes \Leftrightarrow \exists$ colorazione *propria* dei vertici di G ovvero:

$$\exists f :\ v \mapsto \{0, \dots, k-1\} \quad \text{t.c.} \quad \forall (u, v) \in E(G) \quad f(u) \neq f(v)$$

(a) Grafo con colorazione *non* propria (b) Grafo con colorazione propria

Figura 1: Esempi di grafi per Graph Colouring

Problema 2-Graph-Colouring. Consiste nel trovare se esiste una 2 colorazione del grafo dato in input in modo tale che un arco non si trovi tra due vertici dello stesso colore. Questo problema corrisponde a dire se il grafo è **bipartito**, cioè se *posso suddividere il grafo in due classi diverse*.
Per vedere se è bipartito si effettua una **BFS**, cioè una visita in ampiezza, e si controlla se c'è un ciclo dispari. Se c'è allora non è bipartito e quindi nemmeno 2-colorabile.

È 2-colorabile $\Leftrightarrow$ è Bipartito $\Leftrightarrow$ non contiene un ciclo dispari. La visita BFS ha una complessità pari a $O(|E| + |V|)$, perciò il problema è risolvibile in tempo polinomiale, perciò possiamo concludere che 2-Graph-Colouring $\in$ **P**.

Problema 3-Graph Colouring Il problema 3-Graph Colouring $\in$ **P**? Non sappiamo rispondere a questa domanda, poiché non sappiamo se esiste un algoritmo che lo svolga in tempo polinomiale. Il problema 3-Graph Colouring $\in$ **Exp**? Se consideriamo l'algoritmo che prova tutte le possibili colorazioni abbiamo che:

$$3^n \text{ sono le colorazioni dei vertici, dove } n = |V(G)|$$

Bisogna vedere se ci sono archi monocolore e quindi la complessità diventa:

$$O(3^n \cdot |E|) = O(3^{2n}) = O((2^{\log_2 3})^{2n}) = O(2^{2n \log_2 3})$$

Perciò possiamo concludere che il problema 3-Graph Colouring $\in$ **Exp**.

3.3 Classe Time(n)

Definizione 3.3.1 (Classe Time(n))**.** Definiamo la classe **Time**(**n**) come l'insieme dei problemi di complessità lineare, ovvero

$$\mathbf{Time}(\mathbf{n}) = \left\{ \mathbb{A} \mid \exists \mathcal{B} \text{ per } \mathbb{A} \quad \text{t.c.} \quad \forall x \in \mathcal{I}(\mathbb{A}) \quad T_{\mathcal{B}}(|x|) = O(n) = O(f(|x|)) \right\}$$

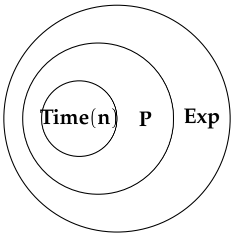
Teorema 3.3.1. $\forall \mathcal{B} \quad \text{t.c.} \quad \mathcal{B}(x) = \mathbb{A}(x) \quad T_{\mathcal{B}}(|x|) > |x|^c \quad \forall c$ *costante*

Teorema 3.3.2. *Qualsiasi **algoritmo di ordinamento** che usa confronti su n elementi ha tempo di esecuzione pari a*

$$\Omega(n \log n)$$

Possiamo dire quindi che:

- **Eulerian Cycle** $\in$ **Time**(**n**) perché esiste un problema che lo risolve in tempo lineare.
- **Sorting** $\notin$ **Time**(**n**) per il teorema 3.3.2.



Possiamo riassumere quindi che:

- **Eulerian Cycle** $\in$ **P**, **Eulerian Cycle** $\in$ **Time**$(\mathbf{n})$.
- **Hamiltonian Cycle** $\in$ **Exp**
- **Hamiltonian Cycle** $\in$ **P** ? non lo sappiamo dire.
- **K-Colouring** $\in$ **Exp**
- **K-Colouring** $\in$ **P**?
  per $k \geqslant 3$ non lo sappiamo dire
  per $k = 2$ sì.

Inoltre, con la definizione della classe **Time**$(\mathbf{n})$ si può dire che:

$$\mathbf{P} = \bigcup_{k \geqslant 0} \mathbf{Time}(\mathbf{n^k})$$

$$\mathbf{Exp} = \bigcup_{k \geqslant 0} \mathbf{Time}(\mathbf{2^{n^k}})$$

3.4 Classe NP

La classe **NP** (*non deterministic polinomial time*) è la classe di problemi tali che se la soluzione per un'istanza del problema è *yes*, allora è facile verificarlo.

Definizione 3.4.1. (Classe **NP**)

$$\begin{aligned} \mathbf{NP} = \{\mathbb{A} \quad \mid \quad & \exists \mathcal{B}(\overset{x}{\cdot}, \overset{w}{\cdot}) \quad \text{t.c.} \quad T_{\mathcal{B}}(|x| + |w|) = O((|x| + |w|)^c) \\ & \forall x \in \mathcal{I}(\mathbb{A}) \quad \mathbb{A}(x) = yes \Leftrightarrow \exists w \text{ t.c.} \quad |w| = O(|x|^d) \text{ e } \mathcal{B}(x, w) = yes\} \end{aligned}$$

dove:

- $\mathcal{B}(\overset{x}{\cdot}, \overset{w}{\cdot})$ è detto **verificatore** per $\mathbb{A}$. Se la risposta di $\mathbb{A}$ esiste, allora $\mathcal{B}$ dice *yes*. Il verificatore impiega **tempo polinomiale** nella taglia dell'istanza per rispondere.
- x è l'istanza
- w è il certificato.

Hamiltonian Cycle $\in$ **NP ?** Per vedere se il problema Hamiltonian cycle appartiene alla classe **NP** dobbiamo costruire un verificatore $\mathcal{B}$ che agisca in tempo polinomiale.

Algorithm 1: Verificatore per HamCycle

```
VerifyHamCycle(G = (V,E), C=x_1,...,x_n)
    if r != |V|: return no                      } O(|w|)
    foreach v∈ V                                }
        if v non appare in C: return no         } O(|V| · |C|)
    for i=1 to n-1                              }
        if (x_i, x_{i+1}) ∉ E: return no        } O(|C|)
    if (x_1, x_n) ∉ E: return no                } O(1)
    return yes
```

Il tempo di esecuzione del verificatore è polinomiale e quindi posso dire che Hamiltonian Cycle $\in$ **NP** .

K-Colouring $\in$ NP ? Per vederlo costruisco il verificatore:

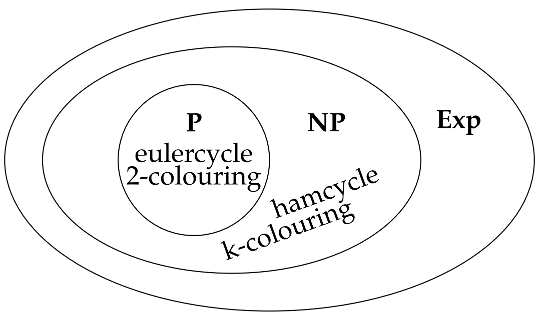
Algorithm 2: Verificatore per K-Colouring

```
VerifyK-Colouring(G = (V,E), f(v_1),...,f(v_n))
    foreach E(u,v)                              }
        if f(u) = f(v): return no               } O(|E|)
    for i=1 to n                                }
        if f(v_i) ⩾ K: return no                } O(|V|)
    return yes
```

Il tempo di esecuzione del verificatore è polinomiale e quindi posso dire che K-Colouring $\in$ **NP** .

P $\subseteq$ NP ? Vogliamo capire in che classe è **NP**. Se include la classe **P** allora significa che un problema che appartiene a quest'ultima, se lo sappiamo risolvere, lo sappiamo anche verificare. Infatti se $\mathbb{A} \in$ **P** dobbiamo dimostrare che esiste un verificatore. Tale verificatore per $\mathbb{A}$ sarà: $\mathcal{B}'(x, w) = \mathcal{B}(x)$ privo di certificato. Dobbiamo dimostrare che se l'istanza è *yes* allora $\mathcal{B}(x) = yes$ altrimenti $\mathcal{B}(x) = no$.

NP $\subseteq$ Exp ? Vogliamo capire in che classe è **NP**
Possiamo supporre che **P** $\subseteq$ **NP** $\subseteq$ **Exp**.



4 Riduzione alla Karp tra problemi di decisione

Definizione 4.0.1 (Riduzione alla Karp)**.** Un problema di decisione $\mathbb{A}$ si riduce alla Karp al problema $\mathbb{B}$: $\mathbb{A} \leqslant_K \mathbb{B}$ se esiste un algoritmo polinomiale $\mathcal{A}$ tale che

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathbb{B}(\mathcal{A}(x)) = yes \ \Leftrightarrow\ \mathbb{A}(x) = yes$$

Proposizione 4.0.1. Se $\mathbb{A} \leqslant_K \mathbb{B}$ e $\mathbb{B} \in$ **P** $\Rightarrow$ $\mathbb{A} \in$ **P**

Proposizione 4.0.2. Se $\mathbb{A} \leqslant_K \mathbb{B}$ e $\mathbb{B} \notin \mathbf{P} \Rightarrow \mathbb{A} \notin \mathbf{P}$

Come effettivamente svolgiamo le trasformazioni?

4.1 Problema SAT

Definizione 4.1.1 (SAT)**.** Il problema di soddisfacibilità di una formula booleana è definito nel seguente modo:

- Input: formula booleana : $\phi(x_1, \dots, x_n) = C_1 \wedge C_2 \wedge \dots \wedge C_n$
  Dove:
  - $C_i = l_{i1} \vee l_{i2} \vee \dots \vee l_{ik}$ (clausola)
  - $l_{ij} = x_k$ oppure $\bar{x}_k$ (letterale)
- Output: *yes* $\Leftrightarrow \quad \exists a_1 \dots a_n \in T, F^n \quad$ t.c. $\quad \phi(a_1, \dots, a_n) = T$

Esempio 4.1.1. $\phi(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3)$
Assegnamento che soddisfa la formula booleana $\phi(x_1, x_2, x_3)$:

$$\begin{array}{lll} x_1 = T & x_2 = F & x_3 = F \\ a_1 = T & a_2 = F & a_3 = F \end{array}$$

SAT $\in$ NP ? Ci chiediamo se il problema SAT sta nella classe **NP**. Vediamo dunque se esiste un certificato e un verificatore che attesta, dato una formula booleana, se essa è soddisfacibile in tempo polinomiale.

- Si può notare facilmente che il certificato è un assegnamento per la formula booleana, dunque è polinomialmente correlato alla grandezza delle variabili della formula, sarà al massimo n.
- Il verificatore viene costruito analizzando la formula booleana, controllando ogni letterale di ciascuna clausola. Ho quindi $m \times n \times n$ controlli, dove m = numero di clausole, n = numero di letterali. Il verificatore è quindi polinomiale.

Possiamo concludere che il problema SAT $\in$ **NP**. Questa affermazione si può tradurre con: *data una formula booleana di cui sappiamo essere soddisfacibile, allora è facile (polytime) costruire un verificatore che attesta che essa è SAT.*

Problema K-SAT: è il problema SAT in cui l'input ha come restrizione il vincolo che ogni clausola ha esattamente k letterali.

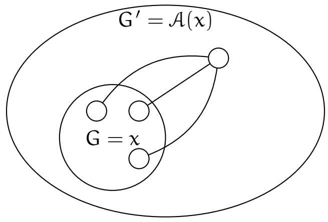
Esempio 4.1.2 (3-SAT)**.** $\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

4.2 Alcuni esempi di riduzioni tra problemi

K-colouring $\leqslant_K$ (K+1)-colouring Vediamo se il problema (K+1)-colouring non è più facile del problema K-colouring. Dobbiamo in sostanza dimostrare che decidere se possiamo colorare un grafo con $k+1$ colori non è più facile che decidere se possiamo colorare un grafo con k colori.
N.B.: da notare che i due grafi non sono necessariamente uguali, parliamo di qualsiasi grafo che appartiene al problema.

$$\begin{array}{rl} \mathcal{A} : & x \in \mathcal{I}(K - COL) \quad \mapsto \quad \mathcal{A}(x) \in \mathcal{I}((K+1) - COL) \\ & K - COL(x) = yes \quad \Leftrightarrow \quad (K+1) - COL(\mathcal{A}(x)) = yes \end{array}$$

Prendiamo quindi il grafo G':



per cui

$$G = (V, E)$$
$$G' = (V \cup \{v'\}, E \cup (v, u') \mid v \in V)$$

in tempo lineare e quindi sotto il polinomiale riesco a costruire il grafo G′.
Se G è K-colorabile allora G′ è (K+1)-colorabile. Mi basta assegnare a v' il colore k (il k+1-esimo colore) e mantenere la colorazione di G.
Se G non è K-colorabile allora G′ non è K+1-colorabile. Equivale a dire che se G′ è K+1-colorabile allora G è k-colorabile. Quindi se v' ha un colore $f(v') = x$ allora ogni $v \in V(G)$ ha un colore $f(v') \neq x$, al più usano k colori.

Da questa dimostrazione ricaviamo anche che 2-col $\leqslant_K$ 3-col $\leqslant_K$ 4-col $\leqslant_K$ 5-col

SAT $\leqslant_K$ 3-SAT Vogliamo dimostrare che data una formula booleana ϕ CNF esiste una trasformazione polytime che mi porta a una formula booleana ϕ' 3CNF (ogni clausola ha esattamente 3 letterali). E inoltre che ϕ è soddisfacibile se e solo se ϕ' è soddisfacibile.

Possiamo iniziare dicendo che $(x_1 \vee x_2) \equiv (x_1 \vee x_1 \vee x_2)$. Le clausole più piccole possono essere espanse. Seguendo questa intuizione arriviamo a dire che:

$$(l_1 \vee l_2 \vee l_3 \vee \cdots \vee l_k) \quad \rightsquigarrow$$
$$(l_1 \vee l_2 \vee z_1) \wedge (\bar{z}_1 \vee l_3 \vee z_2) \wedge (\bar{z}_2 \vee l_4 \vee z_3) \wedge (\bar{z}_3 \vee l_5 \vee z_4) \wedge \cdots \wedge (\bar{z}_{k-1} \vee l_{k+1} \vee z_k)$$

Dimostriamo che se ϕ non è soddisfacibile allora non lo è neanche ϕ'.

- Prendiamo $\phi = (x_1, \ldots, x_n)$. Per questa formula prendiamo un assegnamento $a_1, \ldots, a_n$ che non la rende soddisfacibile, quello in cui ogni letterale viene assegnato a F.
- Prendiamo dunque $\phi' = (x_1, \ldots, x_n, z_1, \ldots z_r)$. Per questa formula prendiamo lo stesso assegnamento di ϕ e vediamo cosa succede con i letterali z:

$$(\underset{F}{l_1} \vee \underset{F}{l_2} \vee \underset{V}{z_1}) \wedge (\underset{F}{\bar{z}_1} \vee \underset{F}{l_3} \vee \underset{V}{z_2}) \wedge (\underset{F}{\bar{z}_2} \vee \underset{F}{l_4} \vee \underset{V}{z_3}) \wedge (\underset{F}{\bar{z}_3} \vee \underset{F}{l_5} \vee \underset{V}{z_4}) \wedge \cdots \wedge (\underset{F}{\bar{z}_{k-1}} \vee \underset{F}{l_{k+1}} \vee \underset{F}{z_k})$$

  risulta che l'ultimo letterale z_k è falso, e quindi ϕ' non è soddisfacibile.

K-COL $\leqslant_K$ K-SAT Vogliamo dimostrare che il problema di colorare un grafo con k colori è riducibile al problema di soddisfacibilità di una formula booleana k-CNF.

Cerchiamo un modo per esprimere in modo logico il fatto che due nodi adiacenti non abbiano lo stesso colore. Supponiamo che il nodo v abbia colore i e il nodo u abbia colore i con $i = 0, 1, \ldots, k-1$. Per ogni $v \in V$: $x_0^{(v)}\ x_1^{(v)}\ x_2^{(v)} \ldots x_{k-1}^{(v)}$ dove $x_i^{(v)} = T$ se il vertice v ha colore i.

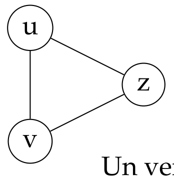
Ci chiediamo quindi quand'è che la formula è K-colorabile?

$$\forall v \in V \begin{cases} x_0^{(v)} \vee x_1^{(v)} \vee x_2^{(v)} \vee \cdots \vee x_{k-1}^{(v)} & \text{ogni vertice ha un colore} \\ \\ \overline{x_i^{(v)} \wedge x_j^{(v)}} = \overline{x_i^{(v)}} \vee \overline{x_j^{(v)}} & \forall i, j \end{cases}$$

$$\forall e = (u, v) \in E \text{ i due vertici non devono avere lo stesso colore}$$

$$\forall i \quad \overline{x_i^{(v)} \wedge x_i^{(u)}} = \overline{x_i^{(v)}} \vee \overline{x_i^{(u)}}$$

Esempio 4.2.1. Prendiamo per esempio il seguente grafo:



La formula booleana corrispondente sarà:

$$
\begin{aligned}
&\text{Ogni vertice ha un colore}\left\{
\begin{array}{l}
(x_0^{(u)} \vee x_1^{(u)} \vee x_2^{(u)}) \wedge \overbrace{(\overline{x_0^{(u)}} \vee \overline{x_1^{(u)}}) \wedge (\overline{x_0^{(u)}} \vee \overline{x_2^{(u)}}) \wedge (\overline{x_1^{(u)}} \vee \overline{x_2^{(u)}})}^{\text{Un vertice non può avere 2 colori}} \wedge \\
(x_0^{(v)} \vee x_1^{(v)} \vee x_2^{(v)}) \wedge (\overline{x_0^{(v)}} \vee \overline{x_1^{(v)}}) \wedge (\overline{x_0^{(v)}} \vee \overline{x_2^{(v)}}) \wedge (\overline{x_1^{(v)}} \vee \overline{x_2^{(v)}}) \wedge \\
(x_0^{(z)} \vee x_1^{(z)} \vee x_2^{(z)}) \wedge (\overline{x_0^{(z)}} \vee \overline{x_1^{(z)}}) \wedge (\overline{x_0^{(z)}} \vee \overline{x_2^{(z)}}) \wedge (\overline{x_1^{(z)}} \vee \overline{x_2^{(z)}}) \wedge
\end{array}\right. \\
&\text{Ogni arco ha colori diversi}\left\{
\begin{array}{l}
(\overline{x_0^{(v)}} \vee \overline{x_0^{(u)}}) \wedge (\overline{x_1^{(v)}} \vee \overline{x_1^{(u)}}) \wedge (\overline{x_2^{(v)}} \vee \overline{x_2^{(u)}}) \wedge \\
(\overline{x_0^{(v)}} \vee \overline{x_0^{(z)}}) \wedge (\overline{x_1^{(v)}} \vee \overline{x_1^{(z)}}) \wedge (\overline{x_2^{(v)}} \vee \overline{x_2^{(z)}}) \wedge \\
(\overline{x_0^{(z)}} \vee \overline{x_0^{(u)}}) \wedge (\overline{x_1^{(z)}} \vee \overline{x_1^{(u)}}) \wedge (\overline{x_2^{(z)}} \vee \overline{x_2^{(u)}})
\end{array}\right.
\end{aligned}
$$

La trasformazione è polinomiale? La complessità della trasformazione è:

$$|V| \cdot \left(K + 2\binom{k}{2}\right) + |E|K \cdot 2 \quad \leqslant \quad (|E| + |V|)K^2$$

Quindi è polinomiale.

4.3 Problema NAE-K-SAT

NAE-K-SAT (Not All Equivalent-K-SAT):

- Input: ϕ K-CNF $\quad \phi : \{T, F\}^n \mapsto \{T, F\}$
- Output: $yes \Leftrightarrow \exists \underline{a} \in \{T, F\}^n$ t.c. $\phi(\underline{a}) = T$ e, in ogni clausola $C_i = l_1^{(i)} \vee l_2^{(i)} \vee \cdots \vee l_k^{(i)}$ con $\underline{a}$, almeno un $l_j^{(i)}$ è vero e almeno un $l_j^{(i)}$ è falso.

Esempio 4.3.1.

$$\phi(x_1, x_2, x_3) = (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

$x_1 =$F $\quad x_2 = F \quad x_3 = F \quad$ *non* è NAE-K-SAT
$x_1 =$F $\quad x_2 = T \quad x_3 = F \quad$ è NAE-K-SAT

Proposizione 4.3.1. Se $\underline{a}$ è un assegnamento che soddisfa ϕ (è NAE), allora anche il negato $\overline{\underline{a}}$ soddisfa ϕ (è NAE).

3-SAT $\leqslant_K$ NAE-4-SAT Vogliamo dimostrare che data una qualsiasi formula ϕ 3-CNF la trasformo in una formula ψ 4-CNF in tempo polinomiale.

$$\phi \text{ 3-CNF} \longmapsto \psi \text{ 4-CNF}$$

$$
\begin{aligned}
\phi &= C_1 \wedge C_2 \wedge \cdots \wedge C_n \quad C_i = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)} \quad i = 1 \dots n \\
\psi &= C_1' \wedge C_2' \wedge \cdots \wedge C_n' \quad C_i' = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)} \vee z \quad i = 1 \dots n
\end{aligned}
$$

Per creare ψ espando le variabili e ne aggiungo sempre una. La trasformazione da ϕ a ψ è polinomiale nella taglia della formula ϕ, perché la scorro tutta per creare ψ.
Ora dobbiamo dimostrare che se ϕ è soddisfacibile allora anche ψ è soddisfacibile:

- ϕ è soddisfacibile $\Rightarrow$ $\exists \underline{a} \in \{T, F\}^n$ t.c. $\phi(\underline{a}) = T$.
- Se prendiamo l'assegnamento $\underline{b} = \underline{a}$ $z = F$ $\psi(\underline{b}) = T$ e ogni clausola ha un letterale a FALSE.
- Vogliamo dimostrare che se esiste un assegnamento $\underline{b}$ che soddisfa ψ allora esiste un assegnamento $\underline{a}$ che soddisfa ϕ.
- Se secondo $\underline{b}$ $z = F$ allora, la parte rimanente di $\underline{b}$ soddisfa ψ
- Se secondo $\underline{b}$ $z = T$ allora, lo nego e torno al primo caso. Perciò se ψ è nae-soddisfatta con $z = F$ allora ϕ è soddisfatta.

NAE-3-SAT $\leqslant_K$ 3-COL Vogliamo dimostrare che data la formula ϕ 3-CNF esiste una trasformazione polinomiale che la rende un grafo G tale che ϕ è NAE-soddisfacibile se e solo se il grafo G è 3-colorabile.

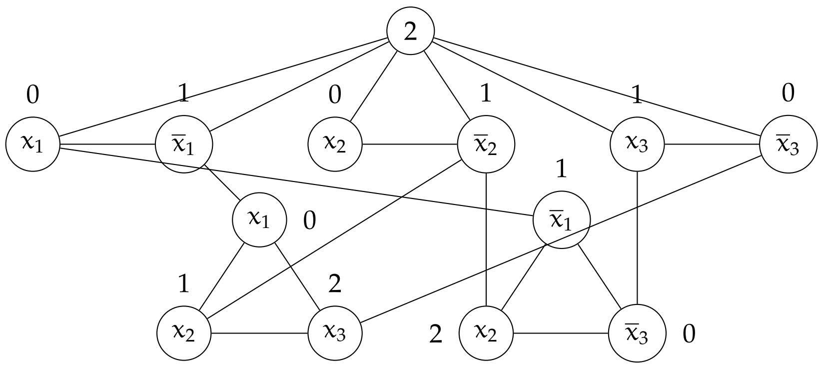
Mappo variabili (letterali) che possono valere T o F, su vertici (elementi del grafo) che hanno colore $0, 1, 2$.

$$x \begin{array}{l} \diagup\, T \\ \diagdown\, F \end{array} \qquad \longmapsto \qquad r \begin{array}{l} \diagup\, 0 \\ -\, 1 \\ \diagdown\, 2 \end{array}$$

Partendo dalla formula $\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$ costruiamo il grafo nel seguente modo:

- Creo un nodo per ogni letterale e per il suo negato, poi aggiungo un vertice perché per ogni vertice x uso la stessa coppia di colori.
- Per ogni clausola metto un triangolo che corrisponde ai letterali della clausola
- Se ho una 3-colorazione ho un assegnamento corrispondente per la clausola che mi mette un letterale T e uno F.
- Ora aggiungo gli archi, collego i letterali che hanno valori di verità opposti.

Se associamo $0 \mapsto T$, $1 \mapsto F$, e 2 libero, abbiamo il seguente risultato:



Perciò la trasformazione garantisce che se $\exists \underline{a}$ t.c. $\phi(\underline{a})$ è nae-soddisfatta allora esiste una 3-colorazione per il grafo G che associa ai valori di verità i colori in modo tale da rendere G 3-colorabile. È facile vedere anche l'implicazione nel verso opposto.

4.4 Transitività della riduzione alla Karp

La riduzione $\leqslant_K$ è transitiva, ciò implica che:

$$\mathbb{A} \leqslant_K \mathbb{B} \text{ e } \mathbb{B} \leqslant_K \mathbb{C} \Rightarrow \mathbb{A} \leqslant_K \mathbb{C}$$

in particolare abbiamo che:

$$\begin{array}{llll} \mathbb{A} \leqslant_K \mathbb{B} & \exists \mathcal{A} \text{ polytime } x \in \mathcal{I}(\mathbb{A}),\ \mathcal{A}(x) \in \mathcal{I}(\mathbb{B}) & \mathbb{A}(x) = yes \Leftrightarrow \mathbb{B}(\mathcal{A}(x)) = yes \\ \mathbb{B} \leqslant_K \mathbb{C} & \exists \mathcal{B} \text{ polytime } y \in \mathcal{I}(\mathbb{B}),\ \mathcal{B}(y) \in \mathcal{I}(\mathbb{C}) & \mathbb{B}(y) = yes \Leftrightarrow \mathbb{C}(\mathcal{B}(y)) = yes \end{array}$$

Perciò

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathcal{B}(\mathcal{A}(x)) \in \mathcal{I}(\mathbb{C}) \quad \mathbb{A}(x) = yes \Leftrightarrow \mathbb{C}(\mathcal{B}(\mathcal{A}(x))) = yes \quad \Rightarrow \quad \mathbb{C}(x) = \mathcal{B}(\mathcal{A}(x))$$

4.5 Problema Reachability

- Input: Grafo G diretto, due nodi s e t.
- Output: $yes \Leftrightarrow$ esiste un cammino che va da s a t.

Quanto costa risolvere Reachablity?
Una possibile soluzione potrebbe essere applicare BFS partendo da s. Se si trova t, allora ritorno yes, altrimenti no. Questo procedimento richiede $O(|V| + |E|)$. Quindi *Reachability* $\in$ **P**

5 Riduzione alla Turing tra problemi di decisione

Definizione 5.0.1 (Riduzione alla Turing)**.** $\mathbb{A} \leqslant_T \mathbb{B}$ se esiste un algoritmo con complessità polinomiale $\mathcal{A}$ che data un'istanza $x \in \mathcal{I}(\mathbb{A})$ utilizzando chiamate ad un *oracolo* per $\mathbb{B}$ che hanno costo $O(1)$, $\mathcal{A}(x) = \mathbb{A}(x)$.
2-SAT $\leqslant_T$ *Reachability*: non vale la riduzione alla *Karp* perché faccio più chiamate a *Reachability*.

6 Classe di problemi NP-Completi

Definizione 6.0.1. (Classe NPC) Un problema $\mathbb{A}$ è NP-completo (NPC) se

- $\mathbb{A} \in$ **NP**
- $\mathbb{A}$ è *NP-hard*, cioè se $\forall \mathbb{B} \in$ **NP** $\mathbb{B} \leqslant_K \mathbb{A}$

$$\textbf{NP-completo} = \big\{\exists p(x) = x^k,\ \exists V(\cdot,\cdot) \quad \text{t.c.} \quad T_V(a,b) = \mathcal{O}\big(p(|a| + |b|)\big) \\ \text{e } \forall x \in \mathcal{I}(\mathbb{A}),\ \mathbb{A}(x) = yes \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)},\ V(x,w) = yes\big\}$$

Se $\mathbb{A}$ è *NP-Completo* e $\mathbb{A} \in P$, allora *P* = *NP*.

6.1 Circuito Booleano

Definizione 6.1.1 (Circuito Booleano)**.** Un circuito booleano è un grafo aciclico orientato (DAG) C_n con n input e ha le seguenti caratteristiche:

- $\exists n$ vertici che hanno *in-degree* = 0
- $\exists 1$ vertice che ha *out-degree* = 0
- Ogni altro vertice ha *in-degree* = 1 o 2 ed è etichettato con and, or, not.
- La taglia di C_n è il numero di vertici.

Esempio 6.1.1. Per $n = 4$ abbiamo $C_4(x_1, x_2, x_3, x_4)$:

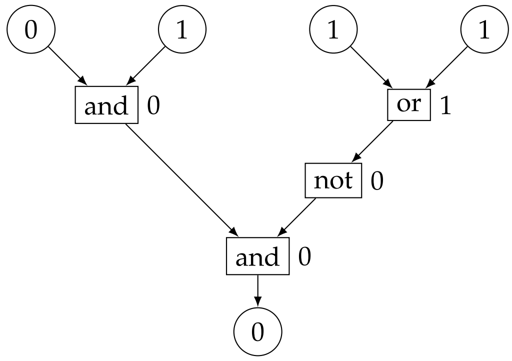


Figura 2: Esempio di circuito booleano con 4 input, il nodo finale di output è detto nodo *sink*.

6.2 Problema Circuit-SAT

- Input: Circuito booleano C_n
- Output: $yes \Leftrightarrow \exists \underline{x}$ t.c. $C(x) = 1$ (il circuito booleano è soddisfacibile).

Definiamo una famiglia di circuiti $C_{n \geqslant 0}$ (per ogni numero di input) di complessità $T(n)$ tale che la taglia di C_n è $O(T(n))$.

Vogliamo mappare il verificatore di ogni problema in **NP** in un circuito:

$$\mathbb{A} \quad \longmapsto \quad V(\cdot, \cdot)$$

$$\mathbb{A}(x) = yes \Leftrightarrow \exists w \text{ t.c. } V(x, w) = yes$$

Dove $V(x, w)$ è un circuito che prende x in input e che mi dice se esiste un certificato w tale che rende soddisfatto il circuito.

Teorema 6.2.1. *Se $\mathbb{A} \in \mathsf{TIME}(f(n))$ allora esiste una famiglia di circuiti $C_{n \geqslant 0}$ di complessità $T(n) = O(f(n)^2)$ tale che $\forall \underline{x} \in \mathcal{I}(\mathbb{A})$ e $n = |x|$ $C_n(x) = \mathbb{A}(x)$ e C_n è costruibile in tempo polinomiale.*

Corollario 6.2.1. *Se $\mathbb{A} \in$ **P** ($f(n)$ è un polinomio in $\mathsf{TIME}(f(n))$) allora esiste una famiglia di circuiti di complessità polinomiale ($T(n) = n^k$) tale che $\forall \underline{x} \in \mathcal{I}(\mathbb{A})$ e $n = |x|$ $C_n(\underline{x}) = \mathbb{A}(x)$ e C_n è costruibile in tempo polinomiale in $|x| = n$.*

Circuit SAT è NP-completo Dimostriamo prima a parole che Circuit-SAT $\in$ **NP**. Forniamo il verificatore $V(x, w)$ verifica se un'istanza soddisfa il problema. Il certificato w è l'assegnamento che soddisfa il circuito, mentre il verificatore scorre ogni nodo e ne valuta il valore, ritorna yes se il nodo finale (sink) è a 1, altrimenti no.

Ora dimostriamo che Circuit-SAT è NP-hard, ovvero che $\forall \mathbb{A} \in$ **NP** $\mathbb{A} \leqslant_K$ Circuit-SAT. Dobbiamo mostrare dunque che esiste tale trasformazione polinomiale:

$$\underline{x} \in \mathcal{I}(\mathbb{A}) \longmapsto C \in \mathcal{I}(\text{Circuit-SAT})$$

e vale anche che:

$$\mathbb{A} = yes \Leftrightarrow \exists w \text{ t.c. } C(w) = 1 (\text{C è soddisfacibile})$$

Sia $\mathbb{A} \in$ **NP** allora $\exists V_{\mathbb{A}}(x, w)$ per le istanze $x \in \mathcal{I}(\mathbb{A})$, tale che $V_{\mathbb{A}}$ ha complessità $O(p(|x|)) = |w|$ (polinomiale). Allora per il teorema 6.2.1 sappiamo che esiste una famiglia di circuiti C_m che fa esattamente ciò che fa il verificatore $V_{\mathbb{A}}$:

$$C_m = V_{\mathbb{A}} \quad m = |x| + p(|x|)$$

perciò, se consideriamo $C'_x(x) = C_m(x, w)$

$$\mathbb{A}(x) = yes \Leftrightarrow \exists w \text{ t.c. } V_{\mathbb{A}}(x, w) = yes \Leftrightarrow \exists w \text{ t.c. } C_m(x, w) = 1 \Leftrightarrow \exists w \text{ t.c. } C'_x(x) = 1$$

SAT è NP-completo Vogliamo dimostrare che dato un circuito booleano soddisfacibile esiste una riduzione che lo trasforma in tempo polinomiale in una formula booleana soddisfacibile.

$$\begin{aligned} \text{Circuit-SAT} &\leqslant_K \text{SAT} \\ \forall C \in \mathcal{I}(\text{Circuit-SAT}) &\longmapsto \phi(\dots) \\ C \text{ è soddisfacibile} &\Leftrightarrow \phi \text{ è soddisfacibile} \end{aligned}$$

Osservazione 6.2.1. Ogni funzione di gate ($and, or, not, \dots$) può essere espressa con una formula booleana CNF ϕ:

$$\begin{aligned} c = a \text{ and } b &\quad (\overline{c} \vee a) \wedge (\overline{c} \vee b) \wedge (c \vee \overline{a} \vee \overline{b}) \\ c = a \text{ or } b &\quad (\overline{c} \vee a \vee b) \wedge (c \vee \overline{b}) \wedge (c \vee \overline{a}) \\ c = \text{not } a &\quad (\overline{c} \vee \overline{a}) \wedge (c \vee a) \end{aligned}$$

Quindi un circuito booleano è soddisfatto quando ogni formula è soddisfatta e il nodo sink è soddisfatto (= 1).

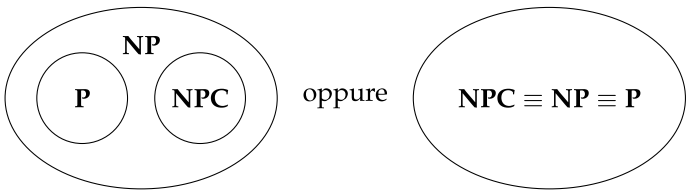
Perciò se ogni funzione di gate sottoforma di circuito booleano rappresenta ogni clausola della formula CNF ϕ, allora possiamo mettere in and tutte le clausole e dire che il circuito C è soddisfatto se e solo se ϕ è soddisfatta.
Con questo e con la dimostrazione che Circuit-SAT è NP-completo possiamo dire che

$$\forall \mathbb{B} \in \mathbf{NP} \quad \mathbb{B} \leqslant_K \text{Circuit-SAT} \leqslant_K \text{SAT}$$

Perciò, per la proprietà transitiva della riduzione alla Karp tra problemi di decisione, deduciamo che SAT è NP-completo.

6.3 Relazione tra P, NP, e NP-completo

Distinguiamo principalmente due casi che rappresentano le relazioni tra le classi di problemi **P**, **NP** e NP-completo:



Teorema 6.3.1. *Se* $\mathbf{NPC} \cap \mathbf{P} \neq \emptyset$ *e* $\mathbb{A} \in \mathbf{NP}$ t.c. $\mathbb{A}$ ***non è banale***, *ovvero*

$$\begin{aligned} \exists x \in \mathcal{I}(\mathbb{A}) &\quad \text{t.c. } \mathbb{A}(x) = yes \\ \exists y \in \mathcal{I}(\mathbb{A}) &\quad \text{t.c. } \mathbb{A}(y) = no \end{aligned}$$

Allora $\mathbb{A} \in \mathbf{NPC}$

Dimostrazione. Se $\mathbf{NPC} \cap \mathbf{P} \neq \emptyset \quad \exists \mathbb{B}$ Np-hard t.c. $\mathbb{B} \in \mathbf{P} \wedge \forall \mathbb{C} \in \mathbf{NP} \;\; \mathbb{C} \leqslant_K \mathbb{B}$. Perciò deduciamo che $\mathbb{C} \in \mathbf{P}$, quindi ogni problema che è in **NP** è anche in **P** e viceversa. Quindi $\mathbf{P} \equiv \mathbf{NP}$.

Dobbiamo quindi dimostrare che ogni problema in **NP** si riduce polinomialmente ad $\mathbb{A}$
Prendiamo come esempio il seguente problema *bit*:

- Input: Bit b
- Output: $yes \Leftrightarrow b = 1$

Sia $\mathbb{D}$ un problema $\mathbb{D} \in \mathbf{NP}$ e quindi $\mathbb{D} \in \mathbf{P}$ (c'è un risolutore polinomiale per $\mathbb{D}$). Dobbiamo trovare una trasformazione $f(x)$ tale che riduce il problema $\mathbb{D}$ al problema *bit*:

$$f(x) = \begin{cases} 1 & \text{se } \mathbb{D}(x) = yes \\ 0 & \text{altrimenti} \end{cases}$$

dove $x \in \mathcal{I}(\mathbb{D})$.
Sappiamo quindi risolvere $f(x)$ in tempo polinomiale perché sappiamo risolvere $\mathbb{D}$ in tempo polinomiale poiché $\mathbb{D} \in \mathbf{NP} \wedge \mathbb{D} \in \mathbf{P}$. Quindi siano x e y

$$x_{yes} \in \mathcal{I}(\mathbb{A}) \quad \text{t.c. } \mathbb{A}(x_{yes}) = yes$$
$$x_{no} \in \mathcal{I}(\mathbb{A}) \quad \text{t.c. } \mathbb{A}(x_{no}) = no$$

allora la trasformazione $f(x)$ sarà:

$$f(x) = \begin{cases} x_{yes} & \text{se } \mathbb{D}(x) = yes \\ x_{no} & \text{se } \mathbb{D}(x) = no \end{cases}$$

□

7 Classe di problemi CO-NP

Definizione 7.0.1. (Classe **CO-NP**) L'insieme dei problemi **CO-NP** è definito nel seguente modo:

$$\mathbf{CO\text{-}NP} = \{\mathbb{A} \mid \overline{\mathbb{A}} \in \mathbf{NP}\}$$

Sono quei problemi per cui è "facile" verificare le istanze no.

Di seguito forniamo un paio di esempi di problemi:

Esempio 7.0.1. Problema:

- Input: Grafo G
- Output: yes se G non è colorabile con 7 colori.

Questo problema è il complemento del problema 7-COL. Quest'ultimo appartiene alla classe **NP** quindi il problema in esempio è in **CO-NP**.

Esempio 7.0.2. Problema:

- Input: formula booleana ϕ
- Output: yes se $\forall \underline{a}\ \phi(a) = T$

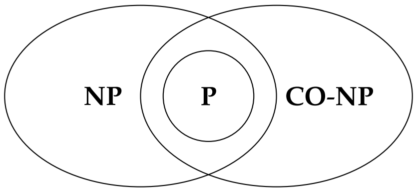
Per questo problema è facile vedere che esiste un'istanza no poiché basta che ci sia almeno una clausola con tutti i letterali a false. Quindi appartiene a **CO-NP**.

7.1 Relazione tra P,NP e CO-NP

Teorema 7.1.1. *Se* $\exists \mathbb{A}$ t.c. $\mathbb{A} \in \mathbf{NPC} \cap \mathbf{CO\text{-}NP}$ *allora* $\mathbf{NP} \equiv \mathbf{CO\text{-}NP}$.

$\mathbf{CO\text{-}NP} \subseteq \mathbf{NP}$. Supponiamo che $\mathbb{A} \in \mathbf{NPC}$ allora $\mathbb{A} \in \mathbf{NP}$ e $\forall \mathbb{C} \in \mathbf{NP} \quad \mathbb{C} \leqslant_K \mathbb{A}$.
Se prendiamo il problema $\mathbb{B} \in \mathbf{CO\text{-}NP} \quad \overline{\mathbb{B}} \in \mathbf{NP}$.
Allora esiste una riduzione alla Karp $\overline{\mathbb{B}} \leqslant_K \mathbb{A}$ che mappa le istanze yes di $\mathbb{B}$ alle istanze no di $\mathbb{A}$ ed esiste anche una riduzione $\mathbb{B} \leqslant_K \overline{\mathbb{A}}$ che è duale alla precedente.
Poiché $\mathbb{A} \in \mathbf{CO\text{-}NP}$ allora $\overline{\mathbb{A}} \in \mathbf{NP}$. Quindi $\mathbb{B}$ si riduce polinomialmente ad un problema in **NP**. Quindi $\mathbb{B} \in \mathbf{NP}$. Quindi per estensione $\mathbf{CO\text{-}NP} \subseteq \mathbf{NP}$. □

$\mathbf{NP} \subseteq \mathbf{CO\text{-}NP}$. Sia $\mathbb{C} \in \mathbf{NP} \quad \mathbb{C} \leqslant_K \mathbb{A} \quad \overline{\mathbb{C}} \leqslant_K \overline{\mathbb{A}}$. Poiché $\mathbb{A} \in \mathbf{CO\text{-}NP}$ allora $\overline{\mathbb{A}} \in \mathbf{NP}$. Quindi $\overline{\mathbb{C}} \in \mathbf{NP} \Rightarrow \mathbb{C} \in \mathbf{CO\text{-}NP} \Rightarrow \mathbf{NP} \subseteq \mathbf{CO\text{-}NP}$. □



Cosa succede se $\mathbf{P} \equiv$ CO-NP ? Se abbiamo l'equivalenza di queste due classi di problemi si ha che:

$$\mathbb{A}(x) \in \mathbf{NP} \qquad \mathcal{A}(x) = \exists w \; B(x, w) \in \mathbf{P}$$
$$\mathbb{A}(x) \in \mathbf{CO\text{-}NP} \quad \mathcal{A}(x) = \forall w \; B(x, w) \in \mathbf{P}$$

- Se $\mathbf{NP} \neq \mathbf{CO\text{-}NP} \;\Rightarrow\; \mathbf{P} \neq \mathbf{NP}$
- Se $\mathbf{P} = \mathbf{NP}$ siccome $\mathbf{P} = \mathbf{CO\text{-}NP}$ $\quad \forall \mathbb{A} \in \mathbf{NP}, \mathbb{A} \in \mathbf{P} \;\Rightarrow\; \overline{\mathbb{A}} \in \mathbf{P} = \mathbf{NP} \quad \Rightarrow\; \mathbf{NP} = \mathbf{CO\text{-}NP}$

Definizione 7.1.1 (Hardness del problema $\mathbb{A}$ nella classe **CO-NP**)**.** $\mathbb{A}$ è **CO-NP-completo** se $\mathbb{A} \in \mathbf{CO\text{-}NP}$ e $\forall \mathbb{B} \in \mathbf{CO\text{-}NP}$ $\mathbb{B} \leqslant_K \mathbb{A}$.

Teorema 7.1.2. *Se $\mathbb{A}$ è* **NP-*completo*** *allora $\overline{\mathbb{A}}$ è* **CO-NP-*completo*** *e viceversa.*

Dimostrazione. Se $\mathbb{A}$ è **NP-completo**, allora

- $\mathbb{A} \in \mathbf{NP}$
- $\forall \mathbb{B} \in \mathbf{NP}$ $\mathbb{B} \leqslant_K \mathbb{A}$

Dalla prima deduciamo che $\Rightarrow \quad \overline{\mathbb{A}} \in \mathbf{CO\text{-}NP}$
Dalla seconda invece, se $\mathbb{C} \in \mathbf{CO\text{-}NP}$, $\quad \overline{\mathbb{C}} \in \mathbf{CO\text{-}NP} \quad \Rightarrow \quad \overline{\mathbb{C}} \leqslant_K \mathbb{A} \quad \Rightarrow \mathbb{C} \leqslant_K \overline{\mathbb{A}}$
$\Rightarrow \; \forall \mathbb{C} \in \mathbf{CO\text{-}NP} \quad \Rightarrow \mathbb{C} \leqslant_K \overline{\mathbb{A}}$
Da queste due deduzioni abbiamo quindi la definizione di **CO-NP-completo** per $\overline{\mathbb{A}}$ □

1. Se vogliamo dimostrare che è **CO-NP-completo** possiamo dimostrare che *il complemento* è **NP-completo**.
2. Per dimostrare che $\mathbb{A}$ è **NP-completo**
   (a) $\mathbb{A} \in \mathbf{NP}$
   (b) $\forall \mathbb{B} \quad \mathbb{B} \leqslant_K \mathbb{A}$

Tautologia *(TAU)* è **CO-NP-completo**

- *Input*: una formula booleana ψ
- *Output*: yes $\Leftrightarrow \forall \underline{a} \in \{T, F\}^n \;\; \psi(\underline{a} = T)$

Si dimostra che $\overline{TAU}$ è **NPC** ($\exists \underline{a}$ t.c. $\psi(\underline{a}) = F$)

7.2 Problema Minimo circuito booleano

- Input: Circuito booleano C_n (con n input)
- Output: $yes \;\Leftrightarrow\; \nexists$ circuito C' $\quad$ t.c. $\quad \forall x \; C'(x) = C(x)$ con $|C'| < |C|$

Consideriamo l'algoritmo $\mathcal{A}$

$$\mathcal{A}(x) = \forall w_1 \exists w_2 \quad B(x, w_1, w_2) = yes \quad \text{ con } B \in \mathbf{P} \text{ e } |w_i| = O(p_i(|x|))$$

Se minimo circuito booleano $\in$ **NP** allora: $\forall w_1 \exists w_2 \quad B(x, w_1, w_2) \;\equiv\; \exists w' \quad B'(x, w')$.
Se minimo circuito booleano $\in$ **CO-NP** allora: $\forall w'' \quad B''(x, w'')$.

8 Gerarchia Polinomiale

Definizione 8.0.1 (Classe di problemi $\Pi_i P$)**.**
$\Pi_i \mathbf{P} = \{\mathcal{A}(x) = \forall w_1 \exists w_2 \forall w_3 \exists w_4 \dots Q_i w_i \quad B(x, w_1, \dots, w_i) \quad \text{dove } |w_i| = O(p_i(|x|)) \text{ e } B \in \mathbf{P}\}$

Definizione 8.0.2 (Classe di problemi $\Sigma_i P$)**.**
$\Sigma_i \mathbf{P} = \{\mathcal{A}(x) = \exists w_1 \forall w_2 \exists w_3 \forall w_4 \dots Q_i w_i \quad B(x, w_1, \dots, w_i) \quad \text{dove } |w_i| = O(p_i(|x|)) \text{ e } B \in \mathbf{P}\}$

Dalla definizione di queste classi di problemi deduciamo che:

$$\begin{aligned}
&\Pi_0 \mathbf{P} = \Sigma_0 \mathbf{P} = \mathbf{P} \qquad \mathcal{A}(x) = \mathcal{B}(x) \text{ non ho quantificatori} \\
&\Pi_1 \mathbf{P} = \mathbf{CO\text{-}NP} \\
&\Sigma_1 \mathbf{P} = \mathbf{NP} \\
&\text{Minimo circuito booleano } \in \Pi_2 \mathbf{P}
\end{aligned}$$

Osservazione 8.0.1. $\mathcal{A}(x) \in \Pi_i \mathbf{P} \quad \Leftrightarrow \quad \overline{\mathcal{A}(x)} \in \Sigma_i \mathbf{P}$.

Osservazione 8.0.2. $\Pi_i \mathbf{P} \subseteq \Sigma_{i+1} \mathbf{P} \quad \text{e} \quad \Sigma_i \mathbf{P} \subseteq \Pi_{i+1} \mathbf{P}$.
Infatti se aggiungo un quantificatore all'inizio, ho che

$$\begin{aligned}
\mathcal{A}(x) &\in \Pi_i \mathbf{P} \\
\mathcal{A}(x) &= \forall w_1 \exists w_2 \dots Q_i w_i \quad B(x, w_1, w_2, \dots, w_i) \\
\Sigma_{i+1} \mathbf{P} &= \exists w^* \forall w_1 \exists w_2 \dots Q_i w_i \quad B'(x, w^*, w_1, w_2, \dots, w_i)
\end{aligned}$$

Perciò $B'(\dots) = B(\dots)$

Osservazione 8.0.3. Per lo stesso motivo dell'osservazione precedente vale che:
$\Pi_i \mathbf{P} \subseteq \Pi_{i+1} \mathbf{P} \quad \text{e} \quad \Sigma_i \mathbf{P} \subseteq \Sigma_{i+1} \mathbf{P}$.

Osservazione 8.0.4. Se $\mathbf{P} \equiv \mathbf{NP} \quad \Rightarrow \quad \forall i \; \Sigma_i \mathbf{P} = \mathbf{P} \wedge \Pi_i \mathbf{P} = \mathbf{P}$
cioè abbiamo che:

$$B(x, w_1, w_2, \dots, w_i) = B'(x) \;\; \text{(elimino tutte le quantificazioni)}$$

Proposizione 8.0.1. Se $\mathbf{NP} = \mathbf{CO\text{-}NP} \quad \Rightarrow \quad \Sigma_1 \mathbf{P} = \Pi_1 \mathbf{P}$.
Quindi $\Sigma_i \mathbf{P} = \Pi_i \mathbf{P} = \Sigma_1 \mathbf{P} = \Pi_1 \mathbf{P} \quad \forall i \geqslant 1$.
Tutte le classi sopra collassano sulla classe 1.

Dimostrazione. Assumiamo che $\mathbf{NP} \equiv \mathbf{CO\text{-}NP}$:
$\mathcal{A}(x) = \exists w_1 \quad B(x, w_1) \;\Leftrightarrow\; \mathcal{A}(x) = \forall w_1' \quad B'(x, w_1)$
Sia $\mathcal{A}'(x) \in \Sigma_2 \mathbf{P} \qquad \mathcal{A}'(x) = \exists w_2 \forall w_1 \quad C(x, w_1, w_2) = \mathcal{D}_{w_2}(x)$.
$\mathcal{D}_{w_2}(x) \in \mathbf{CO\text{-}NP} \equiv \mathbf{NP}$ quindi $\mathcal{D}_{w_2}(x) = \exists w_1' \quad C'(x, w_1', w_2)$ perciò diventa:

$$\begin{aligned}
\mathbb{A}'(x) &= \exists w_2 \exists w_1'' \quad C'(x, w_1'', w_2) \\
&= \exists w_{12} \quad C'(x, w_{12}) \quad \in \mathbf{NP}
\end{aligned}$$

Quindi deduciamo che se $\mathbf{NP} \equiv \mathbf{CO\text{-}NP} \quad \Rightarrow \quad \Sigma_2 \mathbf{P} = \Sigma_1 \mathbf{P}$
Inoltre se $\mathbf{NP} \equiv \mathbf{CO\text{-}NP} \quad \Rightarrow \quad \Pi_2 \mathbf{P} = \Pi_1 \mathbf{P}$ □

Definizione 8.0.3 (Gerarchia Polinomiale)**.** Definiamo gerarchia polinomiale la classe **PH** delle proprietà $\mathbb{A}$ che possono essere espresse da una formula con quantificatori contenente un numero costante di quantificatori alternati:

$$\mathbf{PH} = \bigcup_k \Sigma_k \mathbf{P} = \bigcup_k \Pi_k \mathbf{P}$$

Teorema 8.0.1 (Collasso della gerarchia polinomiale)**.** *Se*

$$\mathbf{P} = \mathbf{NP} \quad \Rightarrow \quad \mathbf{NP} = \mathbf{CO\text{-}NP} = \mathbf{P} \quad \Rightarrow \quad \Sigma_i \mathbf{P} = \Pi_i \mathbf{P} = \mathbf{P} \quad \forall i$$

la gerarchia polinomiale collassa in **P**.
Se $\mathbf{NP} = \mathbf{CO\text{-}NP} \quad \Rightarrow \quad \mathbf{PH} = \mathbf{NP} = \mathbf{CO\text{-}NP}$.

Teorema 8.0.2. *Se* $\Pi_i \mathbf{P} = \Sigma_i \mathbf{P} \quad \Rightarrow \quad \mathbf{PH} = \Pi_i \mathbf{P} = \Sigma_i \mathbf{P}$

8.1 Funzione time-costruibile

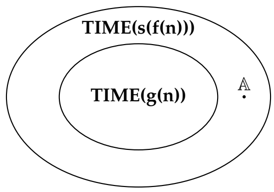
Proposizione 8.1.1. Nel modello computazionale in oggetto è possibile simulare t passi di un algoritmo (programma) mentre controlliamo che $\leqslant t$ passi sono fatti in $s(t)$ passi.

Esempio 8.1.1. Se il modello computazionale è la Macchina di Turing, allora $s(t) = O(t \log t)$.

Esempio 8.1.2. Se il modello computazionale è la RAM, allora $s(t) = O(t)$

Definizione 8.1.1. Diciamo che $f(n)$ è **Time-costruibile** se esiste un programma (algoritmo) che calcola $f(n)$ in $O(f(n))$.

Teorema 8.1.1. *Data l'assunzione precedente, per ogni funzione* $f(n)$ *time-costruibile e per ogni* $g(n) = o(f(n))$ *la classe* ***TIME(g(n))*** $\subset$ ***TIME(s(f(n)))***



8.2 Problema Catch 22

- Input: Π (programma)
- Output: se $\Pi(\Pi)$ termina in meno di $f(|\Pi|)$ passi allora ritorna $\overline{\Pi(\Pi)}$ altrimenti ritorna 0.

Supponiamo che esista un algoritmo Π_{22} tale che risolve il problema Catch 22 in $g(n)$ passi, dove $g(n) < f(n)$. Questo è equivalente a dire che Catch 22 $\in$ **TIME(g(n))**.
Se $\Pi_{22}(\Pi_{22}) =$ Catch $22(\Pi_{22})$ siccome ci mette meno di $f(\Pi_{22})$ passi, allora e uguale a $\overline{\Pi_{22}(\Pi_{22})}$. Questo è assurdo perché non può essere che $\Pi_{22}(\Pi_{22}) = \overline{\Pi_{22}(\Pi_{22})}$, quindi *non* esiste l'algoritmo Π_{22} che impiega $g(n) < f(n)$ passi.

Supponiamo che il programma Π risolve Catch 22 se e solo se $\forall x \in \mathcal{I}(\text{Catch } 22) \quad \Pi(x) =$ Catch $22(x)$. Se Π termina in $\leqslant f(n)$ passi per ogni x, allora $\exists x$ t.c. $\quad \Pi(x) \neq$ Catch $22(x)$.

Proposizione 8.2.1. Per ogni algoritmo esistono infiniti programmi Π che implementano l'algoritmo (fanno la stessa cosa) di lunghezza arbitrariamente grandi.

Proposizione 8.2.2. Per ogni $n \geqslant |\Pi_{22}|$ fissato esiste un altro Π'_{22} tale che $|\Pi'_{22}| = n$. Quindi $\Pi'_{22}(\Pi'_{22}) = \Pi_{22}(\Pi_{22})$.

9 Teorema di Ladner

Ci chiediamo se esiste un problema **NP** che non appartiene nè alla classe **P** nè alla classe **NPC**.

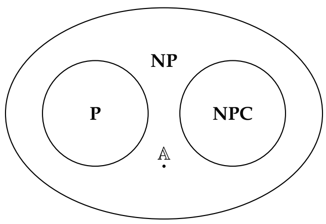


Figura 3: Esiste il problema $\mathbb{A}$?

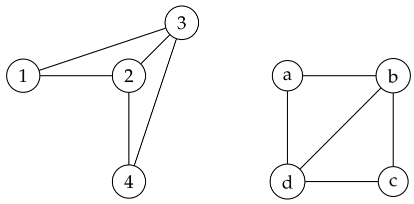
Teorema 9.0.1 (Teorema di Ladner)**.** *Se* $\mathbf{P} \neq \mathbf{NP}$ *allora esiste un problema* $\mathbb{A} \in \mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC})$.

Dimostrazione. Vediamo un problema esempio che soddisfa il *teorema di Ladner*:
Graph Isomorphism

- Input: G_1, G_2 grafi
- Output: $yes \Leftrightarrow$ G_1 è *isomorfo* a G_2.

Definizione 9.0.1 (Isomorfismo)**.** $\exists f: \quad V(G_1) \mapsto V(G_2)$
t.c. $(v, u) \in E(G_1) \Leftrightarrow (f(v), f(u)) \in E(G_2)$

Esempio 9.0.1 (Grafi isomorfi)**.** Ecco un esempio di due grafi isomorfi:



$$\begin{array}{ll} f(1) = a & f(2) = b \\ f(3) = c & f(4) = d \end{array}$$

$$\mathbb{A}(x) = \begin{cases} SAT(x) & \text{se } f(|x|) \text{ è pari} \\ 0 & \text{se } f(|x|) \text{ è dispari} \end{cases}$$

Vogliamo far vedere che:

1. $\mathbb{A} \in \mathbf{NP}$
2. $\mathbb{A} \notin \mathbf{P}$, cioè $\forall \Pi$ polinomiale $\exists x$ t.c. $\Pi(x) \neq \mathbb{A}(x)$.
3. $\mathbb{A} \notin \mathbf{NPC}$, cioè $\forall \Pi$ polinomiale $\exists x$ t.c. $SAT(x) \neq \mathbb{A}(\Pi(x))$.
   Se $\mathbb{A}$ è **NPC** sappiamo che $SAT \leqslant_K \mathbb{A}$

□

9.1 Problema Clique

- Input: grafo $G = (V, E), K$
- Output: $yes \Leftrightarrow$ G contiene una clique di taglia K

Clique è un insieme di vertici tutti connessi a due a due da un arco.

Clique $\in$ NPC Facciamo vedere che il problema Clique appartiene alla classe **NPC** e che quindi appartiene alla classe **NP** e che esiste la riduzione 3-SAT $\leqslant_K$ Clique che trasforma in tempo polinomiale una formula ϕ CNF in un grafo per il problema Clique.

Clique $\in$ NP Creiamo un verificatore per il problema Clique:

- Conta i vertici del grafo C. $[O(n)]$
- Per ogni $(u, v) \in C$ verifica che $(u, v) \in E$. $\left[O(|K|^2 \times |E|)\right]$

Questo verificatore è polinomiale.
Il certificato per il verificatore è una clique C di taglia K in G, tale clique ha taglia polinomiale perché K può essere al massimo n. Perciò Clique $\in$ **NP**.

3-SAT $\leqslant_K$ Clique Vediamo la seguente riduzione che mappa la formula

$$\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$$

in un grafo che soddisfa il problema Clique.

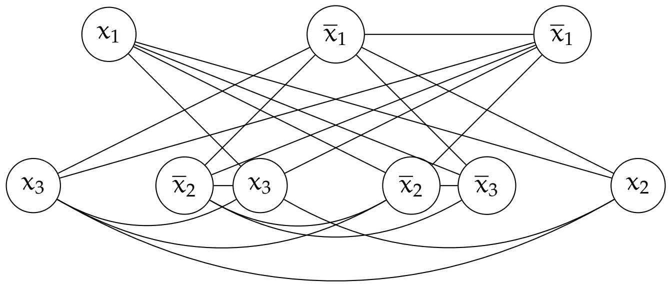


Figura 4: Grafo in cui c'è un arco per ogni letterale diverso dal proprio negato e che non appartiene alla stessa clausola

Sappiamo che se:

- se ϕ è soddisfacibile $\Rightarrow$ G ha una clique di taglia 3;
- se G ha una clique di taglia 3 $\Rightarrow$ ϕ è soddisfacibile.

Il grafo mostra le seguenti caratteristiche:

- Numero di vertici: $|V| = 3m$ con $\phi = C^{(1)} \wedge \cdots \wedge C^{(m)}$.
- Numero di archi: $|E| \leqslant 9m^2$

Quindi il grafo, e di conseguenza la riduzione, è costruibile in tempo polinomiale.

Dimostriamo ora che se ϕ è soddisfacibile allora esiste un assegnamento $a_1, a_2, \dots, a_n$ per $x_1, \dots, x_n$ tale che in ogni clausola un letterale è posto a T.
Siano $v_{i1}^{(1)} v_{i2}^{(2)} \cdots v_{in}^{(n)}$ i vertici corrispondenti ai letterali posti a T dell'assegnamento (uno per clausola). Tali vertici rappresentano nel grafo una clique.

Dimostriamo ora che se G ha una clique di taglia m allora ϕ è soddisfacibile. Supponiamo che G abbia una clique C è taglia m.

1. Gli m vertici di C sono uno per tripla. Le triple corrispondono alle clausole.
2. Due vertici in C non corrispondono a letterali opposti di ϕ.

Dall'ultimo punto in questione costruiamo un assegnamento che soddisfa ϕ. Se prendiamo i vertici di C e li assegniamo a T, gli altri vengono assegnati di conseguenza:

$$\begin{aligned} \overline{x}_2 = T \quad x_2 = F \\ \overline{x}_3 = T \quad x_3 = F \\ x_1 = F \end{aligned}$$

Perciò abbiamo che

$$\phi(F, F, F) = (\underset{F}{x_1} \vee \underset{T}{\overline{x}_2} \vee \underset{F}{x_3}) \wedge (\underset{T}{\overline{x}_1} \vee \underset{F}{x_2} \vee \underset{F}{x_3}) \wedge (\underset{T}{\overline{x}_1} \vee \underset{F}{x_2} \vee \underset{F}{x_3}) = T$$

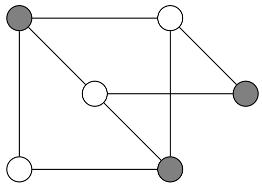
9.2 Problema Independent Set

- Input: Grafo $G = (V, E), k$
- Output: $yes \Leftrightarrow$ in G c'è un Indipendent Set di taglia $\geqslant k$.

Definizione 9.2.1 (Independent Set)**.** Un indipendent set è un insieme I:

$$I \subseteq V \quad \text{t.c.} \quad \forall (u, v) \in I \qquad (u, v) \notin E$$

Esempio 9.2.1 (Independent Set)**.** Vediamo un esempio di independent set:



IndSet $\in$ **NPC** Esiste una riduzione Clique $\leqslant_K$ IndSet tale che

$$\big(G = (V, E), k\big) \mapsto \big(G' = (V, \bar{E}), k\big)$$

Problema TreeIndependentSet Dimostriamo che il seguente problema appartiene alla classe **P**:

- Input: grafo *connesso* e *aciclico* $G = (V, E), k$.
- Output: $yes \Leftrightarrow$ G ha un Independent Set di taglia k.

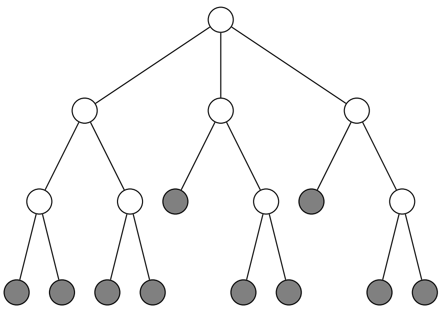


Figura 5: Esempio di Tree independent Set

Osservazione 9.2.1. Si può osservare che le *foglie* di un albero (grafo connesso e aciclico) rappresentano un independent set massimo.

Costruiamo quindi l'algoritmo che dimostra che il problema è in **P**:

Algorithm 3: Algoritmo che risolve TreeIndependentSet

```
TreeIndSetSolver(G = (V, E), k)
    I ← ∅
    while V ≠ ∅:
        foreach v t.c. d(v) ⩽ 1:
            I ← I ∪ {v}
            remove i vicini v da G
    if |I| ⩾ k return yes
    else return no
```

Problema Only Small Independent Set Vediamo ora il problema OSIS:

- Input: $G = (V, E), k$
- Output: $yes \Leftrightarrow$ ogni Independent Set I, $\left|I\right| \leqslant k$.

Se esiste un algoritmo $\mathcal{A}$ che risolve questo problema in tempo polinomiale allora

$$\mathbf{NP} \cap \mathbf{P} \neq \emptyset \quad \Rightarrow \quad \mathbf{P} = \mathbf{NP}$$

Perciò avremmo che

$$\forall (G, k) \quad \mathcal{A}(G, k) = yes \Leftrightarrow OSIS(G, k) = yes$$

dove la taglia di $\mathcal{A}$ è $T_{\mathcal{A}} = \Big(O\big(\left|G\right| + (\log\left|k\right|)^c\big)\Big)$.

Abbiamo dunque un algoritmo $\mathcal{B}^{\text{IndSet}} = \overline{\mathcal{A}(G, k-1)}$.

Osservazione 9.2.2. Osserviamo che è facile verificare il no di istanze del problema OSIS, inoltre si può vedere che tale problema è il duale di IndSet, il quale appartiene alla classe **NPC**. Concludiamo dunque dicendo che OSIS $\in$ **CO-NPC**.

10 Ricavare problemi di ottimizzazione e ricerca

10.1 Independent Set

Vediamo ora diverse formulazioni per il problema Independent Set:

- **Optimization Problem: IndSet-Opt**
  - Input: G
  - Output: un IndSet di massima cardinalità
- **Decision Problem: IndSet-Dec**
  - Input: $G, k \in \mathbb{N}$
  - Output: $yes \Leftrightarrow$ G ha un IndSet di cardinalità $\geqslant k$
- **Search Problem: IndSet-Search**
  - Input: $G, k \in N$
  - Output: un IndSet di G t.c. $|I| \geqslant k$ se esiste, altrimenti no.

Dimostriamo che se **P** = **NP** allora esiste un algoritmo che in tempo polinomiale trova un Independent Set di taglia massima in G.
Se **P** = **NP** allora esiste un algoritmo $\mathcal{A}$ polinomiale per **IndSet-Dec**:

$\Rightarrow$ $\forall(G, k)$ $\mathcal{A}(G, k) = yes \Leftrightarrow$ esiste in G un IndSet di taglia k.

$\Rightarrow$ In tempo polinomiale posso trovare k^* tale che esiste un IndSet in G di taglia k^* e ogni IndSet di G ha taglia al più

$$k^* = \max\{\, k \big| \exists I,\ \text{IndSet di } G,\ I = K\}$$

Per $v \in V$ se in $G - v - \{\, u \big| (u, v) \in E\}$ (i vicini di u) non esiste un IndSet di taglia $k^* - 1$ allora nessun IndSet di taglia k^* contiene v.

Per $v \in V$ se in $G - v - \{\, u \big| (u, v) \in E\}$ contiene un IndSet I' di taglia $k^* - 1$ allora $I' \cup \{v\}$ è un IndSet di G. Dove $\big| I \cup \{v\} \big| = k^*$
Vediamo ora l'algoritmo che permette di costruire un IndSet:

Algorithm 4: Algoritmo di Ottimizzazione per IndSet

```
CostruisciIndSet(G, k*)
    if A(G, k*) = no:
        return no
    else
        G̃ ← G, I ← ∅
        foreach v ∈ V:
            if A(G̃ − v − N(v), k − 1) = yes:
                I ← I ∪ {v}
                G̃ ← G̃ − v − N(v)
                k ← k − 1
        return I
```

Dove $N(v) = \{u \big| (u, v) \in E\}$
Se $\mathcal{A}$ utilizza tempo $T_{\mathcal{A}}(G)$, il tempo di `CostruisciIndSet` è $O(nT_{\mathcal{A}}(G))$
Quindi sapendo risolvere il problema di decisione in tempo polinomiale, riusciamo a risolvere il problema di ottimizzazione in tempo polinomiale.

10.2 Problema SAT-Search

- Input: ϕ CNF
- Output: assegnamento $\underline{a}$ t.c. $\phi(\underline{a}) = T$, se esiste, altrimenti no.

Vediamo ora che dato un algoritmo polinomiale $\mathcal{A}$ per il problema **SAT-Dec**, riusciamo a trovare un algoritmo polinomiale per **SAT-Search**.

L'idea è di procedere per passi. Prendiamo la seguente formula booleana CNF:

$$\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2\overline{x_3})$$

Assegniamo $x_1 = T$ ed eliminiamo così la prima clausola, poiché è sempre vera dato l'assegnamento:

$$\phi'(x_2, x_3) = (\overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3})$$

L'algoritmo procede facendo lo stesso per x_2 e x_3. Infine otteniamo la formula $\phi_{x_1=a_1 \dots x_i=a_i}$ ottenuta dopo aver fissato ogni variabile.

Algorithm 5: Algoritmo di Ricerca per SAT

```
SAT-Solver(ϕ)
    if A(ϕ) = no:
        return no
    for i = 1 to n:
        a_i ← T
        if A(ϕ_{x_1=a_1...x_i=a_i}) = no:
            a_i ← F
    return a_1, a_2, ..., a_i
```

Qual è la complessità? $T_{\texttt{SAT-Solver}}(|\phi|) = O\big(|\phi| \cdot T_{\mathcal{A}}(|\phi|)\big)$, è quindi polytime.
Abbiamo dimostrato quindi che se sappiamo risolvere il problema di decisione in tempo polinomiale, allora sappiamo risolvere anche il relativo problema di ricerca in tempo polinomiale.

10.3 Self Reduciblility

Proposizione 10.3.1. Abbiamo visto che per ogni problema **NPC**, se esiste un algoritmo polinomiale per il problema di *decisione*, esiste un algoritmo polinomiale per il problema di *ricerca* corrispondente.
Se **P** $\neq$ **NP** esiste un problema in **NP** per cui *non* vale "quanto sopra".

Decision e search per i problemi in NP Vediamo le definizioni dei problemi di decisione e di ricerca per i problemi della classe **NP**, cioè i problemi per cui

$$\mathbb{A} \in \mathbf{NP} \Leftrightarrow \exists V_{\mathbb{A}}(\cdot,\ \cdot) \text{ t.c. } \mathbb{A}(x) = yes \Leftrightarrow \exists w\ V_{\mathbb{A}}(x, w) = yes$$

Dato $\mathbb{A} \in$ **NP** e il verificatore $V_{\mathbb{A}}(\cdot,\ \cdot)$:

Definizione 10.3.1 (problema di decisione-$\mathbb{A}$)**.** Dato $x \quad \exists w \quad$ t.c. $\quad V_{\mathbb{A}}(x,\ w) = yes$

Definizione 10.3.2 (problema di ricerca-$\mathbb{A}$)**.** Dato x produci w, se esiste, t.c. $\quad V_{\mathbb{A}}(x,\ w) = yes$

Definizione 10.3.3 (Self Reducible)**.** $\mathbb{A} \in$ **NP** (rispetto a $V_{\mathbb{A}}$) è **self reducible** se, dato un **oracolo** per il problema di decisione-$\mathbb{A}$, esiste un algoritmo polinomiale per il problema di ricerca-$\mathbb{A}$.

Definizione 10.3.4 (Oracolo)**.** Un **oracolo** è una black box che prende in input un'istanza di decisione-$\mathbb{A}$ e ritorna in tempo costante $O(1)$ la soluzione (è specifico per il problema $\mathbb{A}$).

Abbiamo visto che **IndSet** è *Self Reducible* e **SAT** è *Self Reducible*.

Teorema 10.3.1. *Ogni problema* **NPC** *è* ***Self Reducible***

Con la seguente dimostrazione vediamo come sfruttare un algoritmo *"debole"* (decision) per costruirne uno *"forte"* (search).

Dimostrazione. **Assunzione:** assumiamo che esista un oracolo $\mathcal{O}_{\mathbb{A}}$ per il problema $\mathbb{A}$.
Data l'istanza $x \in \mathcal{I}(\mathbb{A})$ vogliamo un certificato w tale che $V_{\mathbb{A}}(x, w) = yes$, se w esiste.
Sappiamo che se $\mathbb{A} \in$ **NPC** allora $\mathbb{A} \leqslant_K \mathsf{SAT}$.

Partiamo dal teorema *Cook-Levin* per cui Circuit-Sat $\in$ **NPC** e SAT $\in$ **NPC**. Abbiamo che la riduzione da $\mathbb{A}$ a SAT è tale che il certificato per l'istanza prodotta di SAT è un certificato per il verificatore $V_{\mathbb{A}}$. Inoltre sappiamo che possiamo trovare un certificato per SAT se abbiamo un oracolo per SAT.

Se $\mathbb{A} \in$ **NPC** allora $\mathsf{SAT} \leqslant_K \mathbb{A}$ e quindi *un oracolo per* $\mathbb{A}$ *implica un oracolo per SAT*.

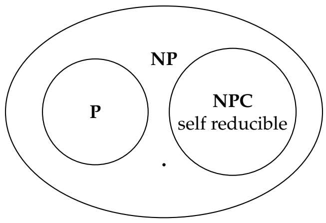
Prendiamo $x \in \mathcal{I}(\mathbb{A})$ e lo trasformiamo in $\phi^{(x)}$ di SAT utilizzando il teorema *Cook-Levin*. Sappiamo che

$$\mathsf{SAT}(\phi^{(x)}) = yes \iff \mathbb{A}(x) = yes$$
$$V_{\mathbb{A}}(x, w) = yes \iff V_{\mathsf{SAT}}(\phi^{(x)}, \underline{w}) = yes$$

Possiamo produrre w usando l'algoritmo `SAT-Solver` (5). La risposta di tale algoritmo sarà uguale alla risposta dell'oracolo

$$\mathcal{O}_{\mathbb{A}}(f(\phi_{x_1=a_1,\dots,x_i=a_i}))$$

dove f è la riduzione polinomiale da SAT a $\mathbb{A}$. In questo modo il certificato w che costruisce SAT-solver è lo stesso che serve a $V_{\mathbb{A}}$. □



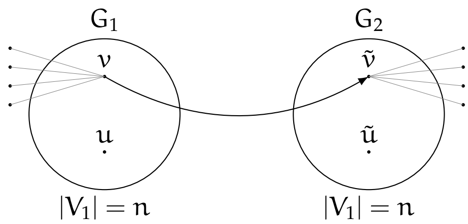
Vediamo ora un problema in **NP** che non crediamo sia in **NPC**.

10.4 Problema Graph Isomorphism

Versione **Graph Isomorphism-Search**:

- Input: $G_1 = (V_1, E_1),\ G_2 = (V_1, E_2)$ semplici e non diretti
- Output: una funzione $f : v_1 \mapsto v_2$ t.c. $\forall (v, u) \in V_1$
  $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$. Se esiste una tale f, altrimenti no.

Dato un oracolo $\mathcal{O}_{\text{GI-Dec}}$ per il problema Graph-Isomorphism-Decision, allora esiste un algoritmo polinomiale (che usa $\mathcal{O}_{\text{GI-Dec}}$) per il problema di ricerca Graph-Isomorphism-Search.



Algorithm 6: Graph Isomorphism Search

```
GraphIsomorphismSearch(G_1, G_2)
    if O^{GI-Decision}(G_1, G_2) = no:
        return no
    foreach v_i ∈ V_1: // Fissiamo v ∈ V_1, ṽ ∈ V_2
        foreach ṽ_i ∈ V_2:
            G̃_1 ← aggiungiamo n vertici a V_1 come vicini di v
            G̃_2 ← aggiungiamo n vertici a V_2 come vicini di ṽ
            if O^{GI-Decision}(G_1, G_2) = yes:
                f(v) = ṽ
                G_1 ← G̃_1, G_2 ← G̃_2
                break
```

Teorema 10.4.1. *Se* **NP** $\cap$ **CO-NP** $\neq$ **P** *allora esiste un problema non self-reducible di ricerca il cui problema di decisione è in* **NP**.

Dimostrazione. Partiamo dunque dall'ipotesi che

$$\exists \mathbb{A} \in \ (\mathbf{NP} \cap \mathbf{CO\text{-}NP}) \setminus \mathbf{P} \qquad \mathbb{A} \notin \mathbf{P},\ \mathbb{A} \in \mathbf{NP},\ \mathbb{A} \in \mathbf{CO\text{-}NP}$$

$\rightarrow$ $\mathbb{A} \in$ **NP** esiste un verificatore $V_{\text{yes}}(x, w)$ polinomiale per le istanze yes tale che

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathbb{A}(x) = yes \ \Leftrightarrow\ \exists w \ \ V_{\text{yes}}(x, w) = yes$$

$\rightarrow$ $\mathbb{A} \in$ **CO-NP** esiste un verificatore $V_{\text{no}}(x, w')$ polinomiale per le istanze no tale che

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathbb{A}(x) = no \ \Leftrightarrow\ \exists w' \ \ V_{\text{no}}(x, w') = yes$$

Definiamo $\forall x \in \mathcal{I}(\mathbb{A})$ un verificatore

$$V^*(x, w) = yes \ \Leftrightarrow\ V_{\text{yes}}(x, w) = yes \quad \text{OR} \quad V_{\text{no}}(x, w) = yes$$

V^* è polinomiale perché V_{yes} e V_{no} sono polytime. Questo verificatore è associato al problema $\mathbb{B} \in$ **NP** per cui $\mathcal{I}(\mathbb{A}) = \mathcal{I}(\mathbb{B})$, $\forall x \in \mathcal{I}(\mathbb{B})\ \mathbb{B}(x) = yes$.

Il problema di ricerca associato a V^* è dato per qualche w tale che $V^*(x, w) = yes$.

Se in tempo polinomiale, dato x, trovo un certificato w tale che $V^*(x, w) = yes$

$$\text{se } V_{\text{yes}}(x, w) = yes \ \text{ allora } \mathbb{A}(x) = yes$$
$$\text{se } V_{\text{no}}(x, w) = yes \ \text{ allora } \mathbb{A}(x) = no$$

Perciò risolvo $\mathbb{A}$ in tempo polinomiale. Questa è una *contraddizione* perché $\mathbb{A} \notin$ **P**. Perciò il problema non è self reducible. □

10.5 Problema No-small-Factor

- Input: due numeri interi q, r
- Output: $yes \Leftrightarrow q$ non ha un divisore $\leqslant r$

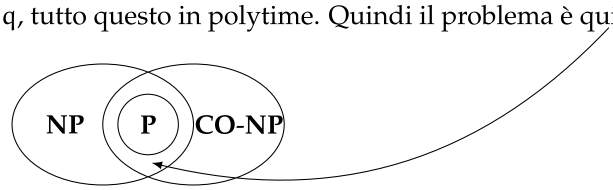
Se sappiamo risolvere No-small-Factor in tempo polinomiale allora sappiamo fattorizzare in tempo polinomiale.

Per trovare il minimo fattore di q ho un costo di $O(\log_{10} q \cdot \log q)$. Quindi è polinomiale in $|q|$.

Facciamo vedere che No-small-Factor $\in$ **NP** e No-small-Factor $\in$ **CO-NP**.
Nel primo caso il certificato è la fattorizzazione di q

$$q = a_1^{k_1} \times a_2^{k_2} \times \cdots \times a_r^{k_r} \qquad a_i \text{ sono numeri primi}$$

se per ogni $i \quad a_i < r$ e la fattorizzazione è giusta e a_i sono primi, allora ritorno yes. Tutto questo è fattibile in tempo polinomiale.
Per verificare che il problema è in **CO-NP** il verificatore semplicemente controlla che ci sia un divisore più piccolo di r dividendo q, tutto questo in polytime. Quindi il problema è qui



10.6 Problema Vertex Cover

- Input: grafo G non diretto, $k \in \mathbb{N}$
- Output: $yes \iff \exists U \subseteq v \quad |U| \leqslant k, \quad \forall (u,v) \in E \quad \{u,v\} \cap U \neq \emptyset$

(a) è vertex cover (b) *non* è vertex cover

Figura 6: Due esempi di grafi per vertex cover

Dimostriamo che il problema $\in$ **NPC**, partiamo col dimostrare che $\in$ **NP**:

- Certificato: $U, \ |U| \leqslant k,$ dove U è vertex cover.
- Verificatore:
  - Conta i vertici in U (tempo: $O(n)$)
  - $\forall (u,v) \in E$ controllo che $\{u,v\} \cap U \neq \emptyset$

  In tutto impiega $O(n^2) \times O(n) = O(n^3)$. Quindi è polinomiale.

Dimostriamo che il problema è **NP** $-$ $hard$: troviamo $\mathbb{A} \in$ **NPC** $\quad$ t.c. $\quad \mathbb{A} \leqslant_K$ VC.

Utilizziamo $\mathbb{A} =$ IndSet:

($\Leftarrow$) U è VC per $G, \ u,v \in U$ se $(u,v) \in E \ \Rightarrow \ U$ non è VC $\Rightarrow \ V \setminus U$ è un IndSet.

($\Rightarrow$) Sia I un IndSet per G.
Se $\exists (u,v) \in E \quad$ t.c. $\quad \forall w \in V \setminus I \ (w \neq u, w \neq v) \ \Rightarrow \ u,v \in I \ \Rightarrow \ I$ non è un IndSet.

10.7 Problema Hitting Set

- Input: $U, F = \{M_1, M_2, \ldots, M_k\} \quad M_i \subseteq U, \quad m \in \mathbb{N}$
- Output: $yes \iff \exists D \subseteq U, \quad |D| \leqslant m \quad$ t.c. $\quad D \cap M_i \neq \emptyset \quad \forall i.$

Facciamo vedere che Hitting Set $\in$ **NPC**.

Hitting Set $\in$ NP Il problema Hitting Set appartiene alla classe **NP**:

- Certificato: insieme D con $|D| \leqslant m$ t.c. $D \cap M_i \neq \emptyset \; \forall i$
- Verificatore:
  - conta gli elementi di D $O(|U|)$
  - $\forall M_i$ controlla che $M_i \cap D \neq \emptyset$ $O(k \times |U|)$

Hitting Set è NP-hard Dimostriamo che esiste la riduzione:

$$\textbf{Vertex Cover} \leqslant_K \textbf{Hitting Set}$$
$$G = (V, E), k \Leftrightarrow (U, F, m)$$

Dove $U \equiv V$, $F \equiv E$, $m \equiv k$. Possiamo dunque notare che Vertex Cover è un caso particolare di Hitting Set, il quale, invece di avere $M_1, M_2, \ldots$, ha un insieme di coppie.

11 Non determinismo e classe NTIME

Definizione 11.0.1 (Classe **NP**)**.** Diamo una definizione diversa della classe **NP**:

$\textbf{NP} = \{\mathbb{A} \mid$ esiste un algoritmo **non deterministico** che risolve istanze di $\mathbb{A}$ in tempo polinomiale$\}$

Definizione 11.0.2 (Algoritmo non deterministico)**.** Un algoritmo non deterministico è un programma (pseudocodice) che può usare un'istruzione (non deterministica) `goto both x, y`. Il programma, grazie a questa istruzione, si sdoppia in due vie in parallelo. Tale programma ritorna yes se esiste almeno una traccia di computazione che ritorna yes, altrimenti ritorna no.

Algorithm 7: Esempio di algoritmo non deterministico

```
SAT-Solver-ND(φ(x_1,...,x_n))
    for i = 1 to n
        a_i ← T
        goto both 4, 5
        a_i ← F
    end for
    if(φ(a_1,...,a_n) == T)
        return yes
    else
        return no
```

$O(|x|^k)$

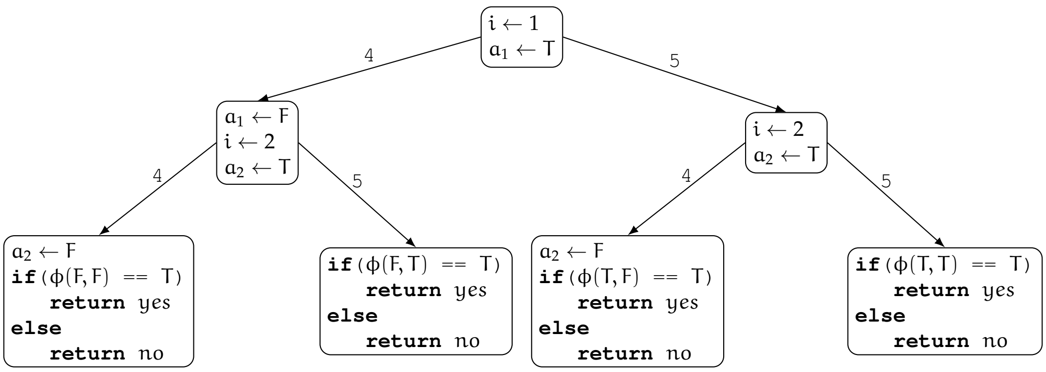


Figura 7: Esempio di albero delle tracce di esecuzione per l'algoritmo SAT-Solver

Osservazione 11.0.1. Possiamo vedere che:

- L'algoritmo precedentemente descritto si trasforma in diversi programmi. Ogni traccia di esecuzione, che viene rappresentata da un cammino, costituisce un assegnamento diverso.
- La complessità dell'algoritmo è polinomiale: $O(|x|^k)$. Ciò significa che la lunghezza massima di ogni cammino è polinomiale, ogni traccia di esecuzione termina in tempo polinomiale.
- Ogni programma non deterministico può essere trasformato in uno equivalente deterministico.
- Se abbiamo un algoritmo non deterministico, possiamo usarlo come *verificatore*, però deve essere deterministico. Il certificato w di tale verificatore è la scelta che deve fare ogni volta che c'è un'istruzione `goto both`.
  Quante sono le scelte che può fare? Sono polinomiali, quindi w è polinomiale nella taglia dell'input.

Definizione 11.0.3 (Classe NTIME)**.** Definiamo la classe $\mathbf{NTIME}(f(n))$ come:

$$\mathbf{NTIME}(f(n)) = \Big\{\mathbb{A} \;\Big|\; \text{t.c. esiste un algoritmo non deterministico che risolve istanze di } \mathbb{A} \text{ di taglia } n \text{ in tempo } O(f(n))\Big\}$$

Osservazione 11.0.2. Osserviamo che:

$$\mathbf{NP} = \bigcup_{k>0} \mathbf{NTIME}(n^k)$$

$$\mathbf{NEXP} = \bigcup_{k>0} \mathbf{NTIME}(2^{n^k})$$

Teorema 11.0.1. *Se* $\mathbf{NEXP} \neq \mathbf{Exp} \quad \Rightarrow \quad \mathbf{NP} \neq \mathbf{P}$

12 Alcuni problemi NP-Completi

In questa sezione vediamo alcuni problemi **NPC** con le relative dimostrazioni di appartenenza a tale classe.

12.1 Problema Max-Cut

- Input: grafo $G = (V, E)$ non diretto, k
- Output: $yes \;\Leftrightarrow\;$ esiste una *bicolorazione* dei vertici di G tale che almeno k archi *non* siano monocromatici.

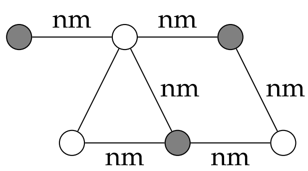


Figura 8: Esempio di Max Cut con $k = 5$ (nm = non monocromatico)

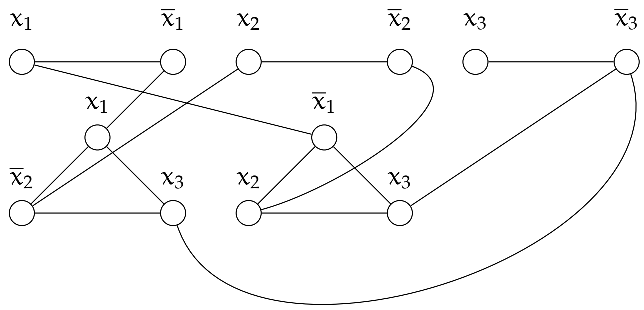
Dimostrazione. Max-Cut $\in$ **NPC**

1. Max-Cut $\in$ **NP**

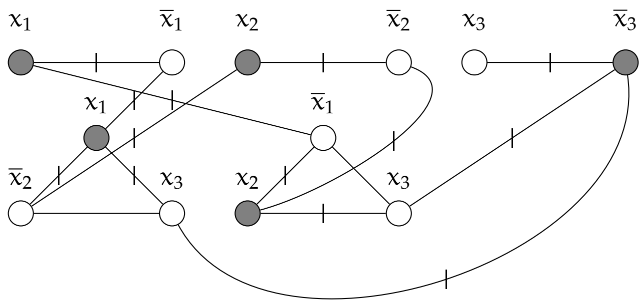
2. Max-Cut è **NP**-hard.

1. *Certificato*: cut o bicolorazione
   Verificatore: verifica arco per arco quanto sono non monocromatici ($O(|E| \times |V|)$).

2. Riduzione NAE-3-SAT $\leqslant_K$ Max-Cut:
   Data $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ vogliamo trovare (G, k).
   Per ogni variabile x di ϕ aggiungiamo un arco in G i cui vertici sono etichettati x e $\overline{x}$.
   Per ogni clausola aggiungiamo in G un triangolo con i vertici etichettati come i letterali. Colleghiamo l in un triangolo con $\overline{l}$ negli archi messi sopra.
   es.: $\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$



Dove m = numero di triangoli, n = numero di letterali. Quanti archi al massimo posso avere bicolorati? $k = n + 3m + 2m = n + 5m$.
ϕ è soddisfacibile $\Rightarrow$ G ha un cut di taglia $k = n + 5m$.
ϕ è soddisfacibile $\Leftrightarrow$ $\exists a_1, \ldots, a_n$ tale che in ogni clausola un letterale è T e un letterale è F.
Colora i vertici etichettati l nero se $l = T$, bianco se $l = F$.
$\phi(T, T, F) = (\overset{F}{\overline{x}_1} \vee \overset{T}{x_2} \vee \overset{F}{x_3}) \wedge (\overset{T}{x_1} \vee \overset{F}{\overline{x}_2} \vee \overset{F}{x_3})$



Esiste un cut di G di taglia $n + 5m$ $\Rightarrow$ esiste un assegnamento $a_1, \ldots, a_n$ t.c. $\phi(a_1, \ldots, a_n)$ è NAE soddisfatta.
Esiste un cut di G di taglia $n + 5m$ $\Rightarrow$

- Tutti gli archi variabile sono bicolorati.
- Tutti gli archi da variabile a triangolo sono bicolorati.
- In ogni triangolo 2 archi sono bicolorati.

Se scelgo

$$a_i = \begin{cases} T & \text{se } x_i \text{ è nero (tra gli archi variabile)} \\ F & \text{se } x_i \text{ è bianco (tra gli archi variabile)} \end{cases}$$

□

12.2 Problema Max-K-SAT

- Input: formula ϕ K-CNF, $t \in \mathbb{N}$
- Output: $yes \Leftrightarrow$ esiste un assegnamento che soddisfa almeno t clausole

esempio:

$$k = 3,\ t = 2 \qquad \phi = (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

Per $k \geqslant 3$ il problema è **NPC**, per questa condizione, con $t = m$ (m = numero di clausole), il problema è identico al problema k-SAT.
Max-2-SAT con $t = m$ è risolvibile in tempo polinomiale.
Max-2-SAT in generale è **NPC**:

Dimostrazione. Dimostriamo la *hardness* del problema Max-2-SAT, con la riduzione:

$$\begin{aligned} \text{Max-Cut} &\leqslant_K \text{Max-2-SAT} \\ (G, k) &\mapsto \phi \text{ 2-CNF}, k' \end{aligned}$$

- Idea di fondo: Vero = colore bianco, Falso = colore nero.
- Per ogni $v \in V$ definiamo la variabile x_v. Quindi

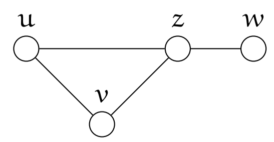
$$\phi(x_{v_1}, \dots, x_{v_n}) \text{ dove } \{v_1, \dots, v_n\} = V$$

- Per ogni arco $(u, v) \in E$ aggiungiamo in ϕ la formula che mi rende diversi i nodi, cioè le clausole

$$x_u \neq x_v \equiv (x_u \vee x_v) \wedge (\overline{x}_u \vee \overline{x}_v)$$

- Dato G otteniamo

$$\phi(x_{v_1}, \dots, x_{v_n}) = \bigwedge_{l=(u,v)\in E} (x_u \vee x_v) \wedge (\overline{x}_u \vee \overline{x}_v)$$



$$\begin{aligned} \phi(x_u, x_v, x_z, x_w) =& (x_v \vee x_u) \wedge (\overline{x}_v \vee \overline{x}_u) \wedge (x_v \vee x_z) \wedge (\overline{x}_v \vee \overline{x}_z) \\ & (x_z \vee x_u) \wedge (\overline{x}_z \vee \overline{x}_u) \wedge (x_z \vee x_w) \wedge (\overline{x}_z \vee \overline{x}_w) \end{aligned}$$

Osservazione 12.2.1. Per un qualsiasi assegnamento, in ϕ almeno la metà delle clausole è soddisfatta, ed in particolare almeno una per "coppia" $(x_a \vee x_b)$.
k' non posso prenderlo più piccolo della metà del numero di clausole.

Osservazione 12.2.2. Per ogni coppia entrambe le clausole sono soddisfatte se e solo se le due variabili hanno valore diverso.
k' deve essere almeno $|E| + k$.
Quindi per l'esempio sopra $k = 2,\ k' = |E| + k = 4 + 2 = 6$

Facciamo vedere che:

1. La riduzione è polinomiale:
   $|\phi| = 2 \times 2 \cdot |E|$ letterali quindi $k' = O(k + |E|)$, perciò è polinomiale.
2. $\overset{\text{Max-cut}}{yes} \Leftrightarrow \overset{\text{Max-k-SAT}}{yes}$:

2.1. $yes \Rightarrow yes$: Se G ha un cut di taglia k allora esiste una colorazione che bicolora k archi $\forall u \in V \quad x_u = T \Leftrightarrow u$ è bianco.

$$\begin{aligned}
&\Rightarrow \forall(u,v) \in E \text{ se } col(u) \neq col(v) \equiv (u,v) \in cutC && (1)\\
&\Rightarrow (x_v \vee x_u) \text{ è soddisfatta e } (\overline{x}_v \vee \overline{x}_u) \text{ è soddisfatta.} && (2)\\
&\Rightarrow \forall(u,v) \in E \text{ se } (u,v) \notin cutC && (3)\\
&\Rightarrow (x_v \vee x_u) \text{ è soddisfatta oppure } (\overline{x}_v \vee \overline{x}_u) \text{ è soddisfatta.} && (4)\\
&\Rightarrow 2|C| + |E| - |C| = |E| + |C| \text{ sono soddisfatte.} && (5)
\end{aligned}$$

$2|C|$ dalla implicazione (1) e (2), $|E| - |C|$ da (3) e (4).

2.2. $yes \Leftarrow yes$: Assumiamo che in ϕ $|E| + k$ clausole siano soddisfatte dall'assegnamento $a_1, \dots, a_n$ $(n = |V|)$.
$\Rightarrow$ almeno k coppie sono soddisfatte,
$\Rightarrow$ per almeno k coppie le variabili hanno assegnato un valore opposto.
Creiamo quindi la bicolorazione del grafo:

$$\forall v \quad col(v) = \begin{cases} \text{bianco} & \text{se } x_v = T \\ \text{nero} & \text{se } x_v = F \end{cases}$$

Per almeno k archi i vertici hanno colore diverso $\Rightarrow$ esiste un cut di taglia k.

□

12.3 Problema Set-Splitting

- Input: $(\mathcal{S}, C)$, $\quad C = \{C_1, C_2, \dots, C_k\}$, con $C_i \subseteq \mathcal{S} \quad i = 1 \dots k$
- Output: $yes \Leftrightarrow$ possiamo colorare gli elementi di $\mathcal{S}$ rosso o blu in modo tale che ogni C_i *non* è monocromatico.

Set-Splitting $\in$ NP Dimostriamo che esiste un verificatore e un certificato che in tempo polinomiale decidono, data un'istanza, se questa appartiene al problema o meno in tempo polinomiale: esempio di istanza yes:

$$\mathcal{S} = \{\underset{r}{1}, \underset{r}{2}, \underset{b}{3}, \underset{b}{4}, \underset{b}{5}\} \quad C = \Big\{\{\underset{r}{1}, \underset{b}{3}, \underset{b}{5}\}, \{\underset{r}{2}, \underset{b}{4}\}, \{\underset{r}{1}, \underset{b}{5}\}\Big\}$$

Il verificatore scorre tutto l'insieme C, e lo fa al massimo $|\mathcal{S}| = n$ volte, quindi la complessità è $O(k \times n)$. Perciò è polinomiale.

Nae-3-SAT $\leqslant_K$ Set-Splitting Dimostriamo che esiste una riduzione

$$\begin{gathered}\text{Nae-3-SAT} \leqslant_K \text{Set-Splitting}\\ \phi \text{ 3-CNF} \mapsto (\mathcal{S}, C)\end{gathered}$$

1. $yes \Rightarrow yes$:
   Mappo ogni clausola di $\phi \quad \forall i \quad C^{(i)} = (l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)})$ in
   $$C_i = \{l_1^{(i)}, l_2^{(i)}, l_3^{(i)})\} \quad \forall j = 1 \dots n \quad C_j = \{x_j, \overline{x}_j\}$$
   Quindi se abbiamo la formula $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$ diventa:
   $$\begin{aligned}&\mathcal{S} = \{x_1, x_2, x_3, \overline{x}_1, x_2, \overline{x}_3\}\\ &C = \Big\{\{x_1, x_2, x_3\}, \{\overline{x}_1, x_2, \overline{x}_3\}, \{x_1, \overline{x}_1\}, \{x_2, \overline{x}_2\}, \{x_3, \overline{x}_3\}\Big\}\end{aligned}$$
   Perciò ad un assegnamento di ϕ che soddisfa la formula in termini NAE corrisponde una bicolorazione non monocromatica di C_i.
2. $yes \Leftarrow yes$:
   Ogni colorazione di $\mathcal{S}$ che bicolora i vari C implica un assegnamento che soddisfa ϕ in termini NAE. Devo imporre che colori di letterali uguali opposti siano opposti.

12.4 Problema Set-Cover

- Input: $(\mathcal{S}, C, k)$, $\quad C$ famiglia di sottoinsiemi di $\mathcal{S}$, $k \in \mathbb{N}$
- Output: $yes \iff \exists C_{i1}, \ldots, C_{ik} \in C \quad$ t.c. $\quad \bigcup_{j=1}^{k} C_{ij} = \mathcal{S}$.

esempio di istanza yes:

$$\mathcal{S} = \{1,2,3,4,5\}, \quad C = \big\{\underset{C_1}{\{1,2\}}, \underset{C_2}{\{2,3,5\}}, \underset{C_3}{\{1,2,4\}}, \underset{C_4}{\{1,3,5\}}\big\}, \quad k = 2$$

Esistono 2 insiemi di C la cui unione è uguale a $\mathcal{S}$? Sì, sono $C_2 \cup C_3 = \{1,2,3,4,5\} = \mathcal{S}$.

Vertex-Cover $\leqslant_K$ Set-Cover Dimostriamo che esiste tale riduzione:

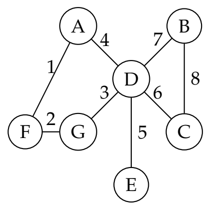


Figura 9: Per questo grafo esistono $\tilde{k}$ vertici che toccano tutti gli archi

Perciò il grafo in esempio si traduce in istanza di Set-Cover nel seguente modo:

$$(\mathcal{S}, C, k) = \big(\{1,2,3,\ldots,8\}, \overset{A}{\{1,4\}}, \overset{B}{\{7,8\}}, \overset{C}{\{6,8\}}, \overset{D}{\{3,4,5,6,7\}}, \overset{E}{\{5\}}, \overset{F}{\{1,2\}}, \overset{G}{\{2,3\}}, \tilde{k}\big)$$

Osservazione 12.4.1. Possiamo osservare che

$$\text{istanze di Vertex-Cover} \subseteq \text{istanze di Set-Cover}$$

Quindi la riduzione ha costo unitario.

12.5 Classe di problemi DP e Problema Clique-No-Clique

Definizione 12.5.1 (classe di problemi **DP**).

$$\mathbf{DP} = \big\{\mathbb{A} \;\big|\; \exists \mathbb{B}, \mathbb{C} \in \mathbf{NP} \quad \mathcal{I}(\mathbb{A}) = \mathcal{I}(\mathbb{B}) = \mathcal{I}(\mathbb{C}) \text{ t.c. } \mathbb{A}(x) = yes \iff \mathbb{B}(x) = yes \wedge \mathbb{C}(x) = no\big\}$$

Problema Clique-No-Clique

- Input: Due grafi G_1, G_2, $\quad k_1, k_2 \in \mathbb{N}$
- Output: $yes \iff G_1$ ha una clique di taglia $\geqslant k_1$ e G_2 *non* ha alcuna clique di taglia $\geqslant k_2$.

Clique-No-Clique $\in$ DP Prendiamo i due problemi $\mathbb{B}$ e $\mathbb{C}$. Dove:

- $\mathbb{B}$ = problema che prende in input G_1, G_2, k_1, k_2 e che ritorna in output $yes \iff G_1$ ha una clique di taglia $\geqslant k_1$.
- $\mathbb{C}$ = input uguale a $\mathbb{B}$: G_1, G_2, k_1, k_2 tranne che ritorna in output $yes \iff G_2$ ha una clique di taglia $\geqslant k_2$.

Quindi $\mathbb{B}, \mathbb{C} \in$ **NP** e Clique-no-Clique$(x) = yes \iff \mathbb{B}(x) = yes \wedge \mathbb{C}(x) = no$.

$\forall \mathbb{A} \in \mathbf{DP}$, $\mathbb{A} \leqslant_k$ **Clique-no-Clique** È vero che esiste tale trasformazione?

Dimostrazione.

$$\begin{aligned} \mathbb{A} \in \mathbf{DP} &\Leftrightarrow \exists \mathbb{B}, \mathbb{C} \in \mathbf{NP} \quad \text{t.c.} \quad \mathbb{A}(x) = yes \Leftrightarrow \mathbb{B}(x) = yes \land \mathbb{C}(x) = no \\ &\Leftrightarrow \exists \mathbb{B} \in \mathbf{NP}, \overline{\mathbb{C}} \in \mathbf{CO\text{-}NP} \quad \text{t.c.} \quad \mathbb{A}(x) = yes \Leftrightarrow \mathbb{B}(x) = yes \land \overline{\mathbb{C}}(x) = yes \end{aligned}$$

Perciò nella riduzione

$$\begin{aligned} &\text{Clique-no-Clique}(G_1, G_2, k_1, k_2) = yes \Leftrightarrow \\ &\text{Clique}(G_1, k_1) = yes \land \text{Clique}(G_2, k_2) = no \Leftrightarrow \\ &\text{Clique}(G_1, k_1) = yes \land \overline{\text{Clique}(G_2, k_2)} = yes \end{aligned}$$

Abbiamo dunque che:

$$\begin{array}{cc} \text{Clique} \in \mathbf{NPC}, & \overline{\text{Clique}} \in \mathbf{CO\text{-}NPC} \\ \Downarrow & \Downarrow \\ \mathbb{B} \leqslant_K \text{Clique} & \overline{\mathbb{C}} \leqslant_K \overline{\text{Clique}} \\ \Rightarrow \exists f_1\ \forall x \in \mathcal{I}(\mathbb{B}) & \Rightarrow \exists f_2\ \forall x \in \mathcal{I}(\overline{\mathbb{C}}) \\ \text{t.c.} \quad \mathbb{B} = yes \Leftrightarrow \text{Clique}\big(f_1(x)\big) = yes & \text{t.c.} \quad \mathbb{B} = yes \Leftrightarrow \overline{\text{Clique}}\big(f_2(x)\big) = yes \end{array}$$

Quindi $\forall x \in \mathcal{I}(\mathbb{A}) = \mathcal{I}(\mathbb{B}) = \mathcal{I}(\mathbb{C})$ abbiamo la funzione:

$$f(x) = \big(f_1(x), f_2(x)\big)$$

$$\begin{aligned} \mathbb{A}(x) = yes &\Leftrightarrow \mathbb{B}(x) = yes \land \overline{\mathbb{C}}(x) = yes \Leftrightarrow \text{Clique}\big(f_1(x)\big) = yes \land \overline{\text{Clique}}\big(f_2(x)\big) = yes \\ &\Leftrightarrow \text{Clique-no-Clique}\big(f(x)\big) = yes \end{aligned}$$

□

12.6 Problema D-Ham-Path

- Input: grafo diretto $G = (V, E)$
- Output: $yes \Leftrightarrow$ esiste in G un cammino Hamiltoniano (che percorre tutti i nodi).

(a) Istanza yes (b) Istanza no

Figura 10: Istanze del problema D-Ham-Path

3-SAT $\leqslant_K$ D-Ham-Path Dimostriamo che esiste tale riduzione:

Dimostrazione. Dobbiamo costruire il grafo a partire dalla formula ϕ 3-CNF:

- Per ogni variabile x di ϕ definiamo un cammino che va in tutte e due le direzioni, con un numero di nodi pari al numero di letterali $x, \overline{x}$ in $\phi + 2$.
- Aggiungiamo due vertici s, t e vertici $y_1, \dots, y_n$ tra i cammini. Colleghiamo y_i agli estremi del cammino di x_{i+1} $\quad i = 1 \dots n$. Stabilendo che attraversando il cammino x_i da destra a sinistra (sinistra verso destra) significa $x_i = T$ $(x_i = F)$, abbiamo una corrispondenza tra assegnamenti e cammini hamiltoniani.

- Per ogni clausola $C^{(i)} = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)}$ aggiungiamo un vertice.
  esempio: $\phi(x_1, x_2, x_3) = (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3)$

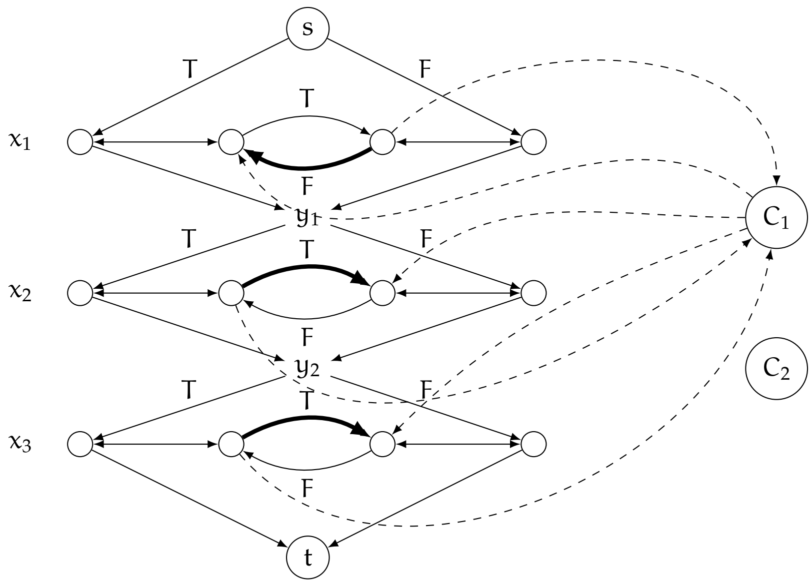


Figura 11: Rappresentazione per la clausola C_1 del grafo con cammino Hamiltoniano

Quindi abbiamo che ci sono:

- $n + 1$ vertici $(s, t, y_1, \ldots, y_n)$
- $2n$ vertici (gli estremi per i cammini per gli x)
- $2 * 3m$ vertici nei cammini
- n vertici (1 per clausola)

Se esiste un assegnamento $a_1, \ldots, a_n$ t.c. $\phi(a_1, \ldots, a_n) = T$ il cammino che corrisponde a prendere ogni cammino variabile (x) nella direzione corrispondente a a_i permette di toccare ogni nodo clausola ($yes \Rightarrow yes$).

Se esiste un cammino Hamiltoniano in G, esso inizia da s e termina in t, e ogni cammino variabile x_i è fatto da destra verso sinistra o viceversa. Per toccare i vertici clausola il cammino Hamiltoniano deve aver attraversato almeno uno dei cammini dei letterali della clausola nella direzione corrispondente a rendere il letterale vero. L'assegnamento corrispondente alle direzioni scelte nei cammini variabile soddisfa ogni clausola ($yes \Leftarrow yes$).

□

Problema D-Ham-Cycle

- Input: grafo diretto $G = (V, E)$
- Output: $yes \Leftrightarrow$ esiste in G un ciclo Hamiltoniano (che percorre tutti i nodi).

Si tratta di un problema **NPC** come il precedente (dimostrazione simile).

13 Complessità di Spazio e la classe SPACE

In questa sezione osserviamo quali problemi sono risolvibili sotto limitazioni di memoria (spazio).

Modello: l'istanza viene data in sola lettura e fuori dalla memoria centrale di lavoro. Diciamo che abbiamo limite $f(n)$ se per istanze lunghe n possiamo usare al massimo $f(n)$ bit di memoria di lavoro ($O(f(n))$).

Definizione 13.0.1 (Classe **SPACE**)**.** La classe **SPACE** è definita come segue:

$$\mathbf{SPACE}(f(n)) = \Big\{ \mathbb{A} \;\Big|\; \text{esiste un programma/algoritmo che risolve istanze di } \mathbb{A} \text{ usando al più } O(f(n)) \text{ bit di memoria di lavoro e accede all'istanza in sola lettura. } n \text{ è la taglia dell'istanza.} \Big\}$$

Problema Palindroma $\in$ L Osserviamo quanto spazio di memoria occupa l'algoritmo che risolve il problema Palindroma.

```
Palindroma(s)
    for i = 1 to n:
        if s[i] ≠ s[n - i + 1]
            return no
    return yes
```

Lo spazio utilizzato è $O(\log n)$. Quindi Palindroma $\in$ **SPACE**$(\log n)$.

Problema SAT $\in$ PSPACE Osserviamo quanto spazio di memoria occupa l'algoritmo che risolve il problema SAT.

```
SAT(ϕ(x_1,...,x_n))
    for i = 0 to 2^n - 1
        for j = 1 to n
            x_j ← j-esimo bit di i
            if (ϕ(x_1,...,x_n) = T)
                return yes
    return no
```

Questo algoritmo, come già visto, impiega tempo esponenziale, mentre occupa spazio lineare.

- Per i: n bit
- Per j: $\log n$ bit
- Per `x[]`: n bit

Complessità di spazio: $O(n)$, dove $n =$ numero di variabili di ϕ. $n \leqslant |\phi|$.

13.1 Classe PSPACE, L e NTIME

Definizione 13.1.1 (Classe PSPACE)**.**

$$\mathbf{PSPACE} = \bigcup_{k>0} \mathbf{SPACE}(n^k)$$

Abbiamo visto che: **SAT** $\in$ **PSPACE**

Definizione 13.1.2 (Classe L)**.**

$$\mathbf{L} = \mathbf{SPACE}(\log n)$$

Abbiamo visto che: **Palindroma** $\in$ **L**
Inoltre sappiamo che **NP** $\subseteq$ **PSPACE**

Proposizione 13.1.1. $\forall \mathbb{A} \in$ **NP** esiste una riduzione $\mathbb{A} \leqslant_K$ SAT che trasforma l'istanza $x \in \mathcal{I}(\mathbb{A})$, in tempo polinomiale ($n^k$), in una formula ϕ_x tale che $\mathbb{A} = yes \Leftrightarrow \text{SAT}(\phi_x) = yes$.
Tale riduzione utilizza spazio $O(n^k)$, in più sappiamo che SAT(ϕ_x) utilizza spazio $O(n^{k'})$. Quindi lo spazio totale utilizzato è $O(n^{k+k'})$. Perciò è polinomiale in $n = |x|$.

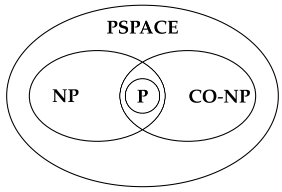


Figura 12: **P** $\subseteq$ **NP** $\subseteq$ **PSPACE**

Definizione 13.1.3 (Classe NTIME)**.** Ridefiniamo la classe **NTIME** nel seguente modo:

$$\textbf{NTIME} = \Big\{ \mathbb{A} \;\Big|\; \begin{array}{l} \text{esiste un verificatore } V_{\mathbb{A}}(\cdot,\cdot) \\ \mathbb{A} = yes \Leftrightarrow V_{\mathbb{A}}(x,w) = yes \\ V_{\mathbb{A}} \text{ impiega tempo } O(f(|x|)) \\ |w| = O(f(|x|)) \Big\} \end{array}$$

$\textbf{NTIME}\big(f(n)\big) \subseteq \textbf{SPACE}\big(f(n)\big)$ Dimostriamo questa affermazione:

Dimostrazione. Sia $\mathbb{A} \in \textbf{NTIME}\big(f(n)\big)$ allora esiste un k tale che

$$\begin{array}{l} \forall x \in \mathcal{I}(\mathbb{A}) \quad \mathbb{A} = yes \Leftrightarrow \exists w \in \{0,1\}^{k \cdot f(|x|)} \quad \text{t.c.} \\ V_{\mathbb{A}}(x,w) = yes \;\text{ e } V_{\mathbb{A}}(x,w) \text{ impiega tempo al più } k \cdot f(n) \end{array}$$

Facciamo vedere che $\mathbb{A} \in \textbf{SPACE}\big(f(n)\big)$: dobbiamo produrre l'algoritmo Π che risolve le istanze di $\mathbb{A}$ e usa al massimo $f(n)$ bit.
Π costruisce tutti i certificati $w \in \{0,1\}^{k \cdot f(|x|)}$ e per ognuno di questi chiama il programma $V_{\mathbb{A}}(x,w)$.

- Se per uno dei $V_{\mathbb{A}}$ dice yes allora esiste il certificato, perciò tale istanza è yes.
- Se per tutti i $V_{\mathbb{A}}$ dice no allora è un'istanza no.

Quanto spazio utilizza Π?

- Per produrre tutti i certificati w usa spazio $k \cdot f(|x|) \Rightarrow O\big(f(n)\big)$ bit.
- Poi utilizza lo spazio che utilizza il verificatore $V_{\mathbb{A}}(x,w)$, il quale termina in tempo $O\big(f(n)\big) \Rightarrow$ utilizza $O\big(f(n)\big)$ spazio di memoria (ogni operazione usa una quantità fissata di spazio).

$\Rightarrow O\big(f(n)\big) + O\big(f(n)\big) \Rightarrow O\big(f(n)\big)$

Perciò $\mathbb{A} \in \textbf{SPACE}\big(f(n)\big)$ ed è quindi risolvibile in tempo deterministico. □

Ora quindi sappiamo che

$$\textbf{TIME}\big(f(n)\big) \subseteq \textbf{NTIME}\big(f(n)\big) \subseteq \textbf{SPACE}\big(f(n)\big)$$

$\textbf{SPACE}\big(f(n)\big) \subseteq \textbf{TIME}\big(2^{O(f(n))}\big)$ Dimostriamo questa affermazione:

Dimostrazione. Esiste Π t.c. $\Pi(x) = \mathbb{A}(x)$ con $\mathbb{A} \in \textbf{SPACE}\big(f(n)\big)$ e utilizza spazio $O\big(f(|x|)\big)$. Cosa significa che Π utilizza spazio $O\big(f(|x|)\big)$? Significa che tutta la memoria che contiene/usa Π è limitata superiormente da $f(|x|)$.

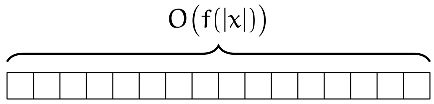


Figura 13: In questa memoria i bit cambiano a seconda delle istruzioni dell'algoritmo

In quanti modi questi bit possono cambiare? $O\big(f(|x|)\big) = k \cdot f(n)$. Ci sono al più $2^{k \cdot f(n)}$ stati in cui la memoria si può trovare durante l'esecuzione. È deterministico, la memoria mi dice quale operazione bisogna svolgere successivamente, quindi non avremo mai il caso in cui la memoria sarà uguale più di una volta.

$\Rightarrow$ Il numero di stati/passi/istruzioni del programma/algoritmo è al più $2^{k \cdot f(n)}$.

$\Rightarrow$ $\mathbb{A} \in \textbf{TIME}\big(2^{O(f(n))}\big)$

Perciò $\textbf{SPACE}\big(f(n)\big) \subseteq \textbf{TIME}\big(2^{O(f(n))}\big)$ □

Quindi osserviamo l'evoluzione del programma osservando *l'evoluzione della memoria*. La conseguenza immediata di questo è che:

$$\textbf{PSPACE} = \bigcup_{k>0} \textbf{SPACE}(n^k) \subseteq \bigcup_{k>0} \textbf{TIME}(2^{n^k}) = \textbf{Exp}$$

$$\Rightarrow \textbf{PSPACE} \subseteq \textbf{Exp}$$

$$\Rightarrow \textbf{L} = \textbf{SPACE}(\log n) \subseteq \textbf{TIME}\underbrace{\big(2^{O(f(n))}\big)}_{2^{k \log n} = n^k = \textbf{P}}$$

Perciò abbiamo che:

$$\textbf{L} \subseteq \textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subseteq \textbf{Exp}$$

13.2 Non determinismo e classe NSPACE

Definizione 13.2.1 (Classe NSPACE)**.**

$$\textbf{NSPACE} = \Big\{ \mathbb{A} \;\Big|\; \text{esiste un algoritmo/programma } \Pi \textit{ non deterministico} \\ \text{che risolve } x \in \mathcal{I}(\mathbb{A}) \text{ usando memoria di lavoro } O(f(|x|)) \quad \Pi(x) = \mathbb{A}(x) \Big\}$$

Nel caso del tempo con limitazione polinomiale in algoritmi non deterministici abbiamo la classe dei problemi **NP**. Cosa succede nel caso dello spazio? Quali sono i problemi in **NPSPACE** ?

$\textbf{NSPACE} \subseteq \textbf{TIME}\big(2^{O(f(n))}\big)$ Se ammettiamo non determinismo, dimostriamo che sappiamo risolvere lo stesso problema in tempo $2^{O(f(n))}$.

Dimostrazione. Prendiamo un problema $\mathbb{A} \in \textbf{NSPACE}\big(f(n)\big)$. Sappiamo che la memoria dell'algoritmo Π può avere $2^{k \cdot f(n)}$ configurazioni.

- Nel determinismo: ogni configurazione mi porta per forza alla successiva.
- Nel *non* determinismo: ogni configurazione mi può portare al massimo in 2 configurazioni diverse. Rappresentiamo lo spazio di configurazioni con un grafo G_x^{Π} in cui
  - Ogni vertice è lo stato della memoria.
  - Ogni vertice ha *out-degree* 0, 1, 2.

– Ha $2^{k \cdot f(n)}$ vertici.
– Parte da uno stato iniziale e termina in uno stato finale in cui dice yes o no.

Ciò significa che $\mathbb{A} = yes \Leftrightarrow$ esiste in G_x^Π un cammino dallo stato $start$ allo stato $finale$.

Se conosco Π posso costruire il grafo, poiché conosco l'evoluzione della memoria del programma e quindi simulo il programma e vedo i possibili stati della memoria. Il grafo è costruibile in tempo $2^{k \cdot f(n)}$.

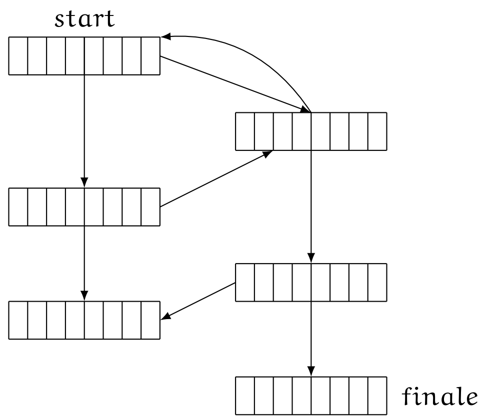


Figura 14: Esempio di grafo G_x^Π

Sia $\mathbb{A} \in \mathbf{NSPACE}(f(n)) \Rightarrow \exists \Pi \Rightarrow \forall x \, \exists G_x^\Pi = (V, E)$ con $|V| = 2^{O(f(|x|))}$ ed E è costruibile in tempo $2^{O\left(f(|x|)\right)}$

$$\Rightarrow G_x^\Pi \text{ è costruibile in } 2^{O\left(f(|x|)\right)}$$

Una volta costruito eseguiamo il solutore del problema **Reachability** su $\left(G_x^\Pi, start, finale\right)$ e ritorna il valore ritornato. Il solutore è una BFS, la quale viene eseguita in tempo $2^{O\left(f(|x|)\right)}$.

Perciò $\mathbf{NSPACE} \subseteq \mathbf{TIME}\left(2^{O(f(n))}\right)$.

- Se limitiamo il tempo $\Rightarrow$ limitiamo la lunghezza del cammino di reachability di G_x^Π.
- Se limitiamo lo spazio $\Rightarrow$ limitiamo la dimensione del grafo G_x^Π.

$\square$

13.3 Problema Reachability in termini di spazio

- Input: grafo G, nodi s e t.
- Output: $yes \Leftrightarrow$ esiste un cammino da s a t.

Sappiamo che il problema appartiene alla classe **P** poiché utilizzando BFS sappiamo risolverlo in tempo polinomiale, quindi vale anche che Reachability $\in$ **PSPACE** (**P** $\subseteq$ **PSPACE**).

Reachability $\in$ $\mathbf{SPACE}\left((\log n)^2\right)$ Utilizzando BFS non utilizziamo spazio logaritmico ma ne utilizziamo sicuramente di più.

Dimostrazione. Dimostriamo l'appartenenza a tale classe utilizzando un algoritmo ricorsivo che utilizza l'induzione:

→ Se esiste un cammino $s \rightsquigarrow t$ allora esiste un cammino che utilizza al più $|V|$ vertici che ha lunghezza $\leqslant n$ ($n = |V|$).

→ Se esiste un cammino $s \rightsquigarrow t$ di lunghezza al più n allora esiste un vertice u tale che:

- Esiste un cammino $s \rightsquigarrow u$ di lunghezza $\leqslant \lceil n/2 \rceil$.
- Esiste un cammino $u \rightsquigarrow t$ di lunghezza $\leqslant \lceil n/2 \rceil$.

→ Se esiste un cammino $s \rightsquigarrow t$ di lunghezza $\leqslant 1$ allora $s = t$ oppure $(s, t) \in E$.

Scriviamo quindi l'algoritmo ricorsivo:

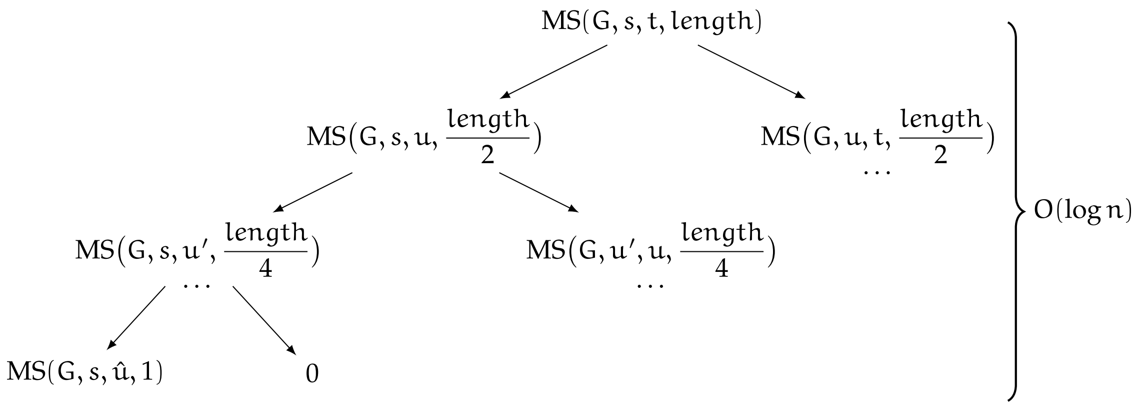
Algorithm 8: Algoritmo ricorsivo per risolvere Reachability

```
MiddleSearch(G,s,t, length)
    if length ⩽1:
        if s = t or (s,t) ∈ E:
            return yes
        else:
            return no
    else:
        risposta ← no
        foreach u ∈ V:
            if (MiddleSearch(G,s,u, length/2) = yes and MiddleSearch(G,u,t, length/2) = yes):
                risposta ← yes
        return risposta
```

Quanta memoria utilizza?
Quanta memoria occupano le variabili?

- Per s, t e u abbiamo $O(\log n)$ bit.
- Per `risposta` abbiamo 1 bit.

Cosa succede nelle chiamate ricorsive?



In tutto viene utilizzata $O(\log n \cdot \log n) = O(\log^2 n)$ memoria:

- $\log n$ bit per ogni cammino radice foglia, poiché è la profondità dell'albero.
- $\log n$ bit per memorizzare ogni nodo lungo il cammino.

Perciò il problema Reachability $\in$ **SPACE**$\big((\log n)^2\big)$.

□

13.4 Classe NPSPACE

Definizione 13.4.1. (Classe NPSPACE) Definiamo la classe **NPSPACE** in modo analogo alla classe **PSPACE** come:

$$\mathbf{NPSPACE} = \bigcup_{k>0} \mathbf{NSPACE}(n^k)$$

Nella successiva sezione vogliamo dimostrare che **PSPACE** $\equiv$ **NPSPACE**. Lo facciamo vedere attraverso il teorema di Savitch.

13.5 Teorema di Savitch

Teorema 13.5.1 (Teorema di Savitch). *Per ogni funzione* $f(n) \geqslant \log n$ *si ha che*

$$\mathbf{NSPACE}\big(f(n)\big) \subseteq \mathbf{SPACE}\Big(\big(f(n)\big)^2\Big)$$

Dimostrazione. Sappiamo che **PSPACE** $\subseteq$ **NPSPACE**. Facciamo vedere che

$$\mathbf{NPSPACE} = \bigcup_{k>0} \mathbf{NSPACE}(n^k) \subseteq \bigcup_{k>0} \mathbf{SPACE}(n^{2k}) \subseteq \bigcup_{k>0} \mathbf{SPACE}(n^k) = \mathbf{PSPACE}$$

Sia $\mathbb{A} \in \mathbf{NPSPACE}\big(f(n)\big)$:

- $\Rightarrow$ Esiste un algoritmo Π non deterministico tale che per ogni istanza $x \in \mathcal{I}(\mathbb{A})$ abbiamo che $\Pi(x) = \mathbb{A}(x)$ e Π usa spazio $O\big(f(n)\big)$.
- $\Rightarrow$ Detto G_x^Π il grafo degli stati di Π su x, sappiamo che tale grafo avrà $|V| = 2^{k\cdot f(n)}$ e dati 2 stati $u, v \in V$ abbiamo che $(u, v) \in E$ se e solo se Π nello stato u ha v tra le possibili transizioni.

$\Pi(x) = yes$

- $\Leftrightarrow$ In G_x^Π esiste un cammino dallo stato $start$ allo stato $finale$.
- $\Leftrightarrow$ Reachability$(G_x^\Pi, start, finale, 2^{k\cdot f(n)}) = yes$.
- $\Leftrightarrow$ Middlesearch$(G_x^\Pi, start, finale, 2^{k\cdot f(n)}) = yes$. Con al più

$$O\big(\log^2(2^{k\cdot f(n)})\big) = O\Big(k^2\big(f(n)\big)^2\Big) = O\big(f^2(n)\big)$$

Abbiamo dunque dimostrato che **PSPACE** $=$ **NPSPACE** poiché il non-determinismo non aggiunge potenzialità nello spazio. Abbiamo inoltre che **NL** $\subseteq$ **SPACE**$(\log^2(n))$

□

Abbiamo dato una definizione di non-determinismo in termini di spazio. Ne diamo una definizione in termini di verifica.

$$\begin{aligned}\mathbf{NSPACE}(f(n)) = \big\{\mathbb{A} \mid \exists V(\cdot,\cdot)\ &\text{deterministico tale che } \forall x \in \mathcal{I}(\mathbb{A}) \quad \mathbb{A}(x) = yes \Leftrightarrow V(x, w) = yes\\ &\text{e } V \text{ usa al più } O(f(n)) \text{ di memoria di lavoro (escludendo } x \text{ e } w),\\ &V \text{ accede a } x \text{ in maniera Read-Only},\\ &V \text{ accede a } w \text{ in maniera Read-Only e Left-to-Right}\big\}\end{aligned}$$

Algorithm 9: SAT-Solver-ND $\in$ **NPSPACE**(n)

```
SAT-Solver-ND(φ(x₁...xₙ))
    for i=1 to n:
        aᵢ ← T
        GotToBoth 4,5
        aᵢ ← F
    endfor
    if φ(a₁...aₙ) = T
        return yes
    return no
```

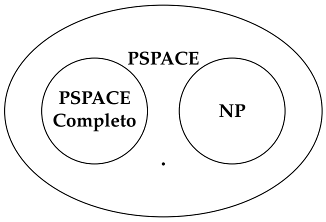
Algorithm 10: SAT-Verifier

```
SAT-Verifier(φ(C₁...Cₙ), a)
    risposta ← yes
    for i=1 to n:
        if l₁ⁱ = F in a ∧ l₂ⁱ = F in a ∧ l₃ⁱ = F in a
            return no
    return yes
```

$$
\begin{aligned}
&\textbf{PSPACE} = \textbf{NPSPACE} = \\
&\{\mathbb{A} \mid \exists V(\cdot,\cdot) \text{ deterministico tale che } \forall x \in \mathcal{I}(\mathbb{A}) \quad \mathbb{A}(x) = yes \Leftrightarrow V(x,w) = yes \\
&\text{e V usa al più } O(|x|^k) \text{ di memoria di lavoro (escludendo } x \text{ e } w), \\
&\text{V accede a } x \text{ in maniera Read-Only,} \\
&\text{V accede a } w \text{ in maniera Read-Only e Left-to-Right}\}
\end{aligned}
$$

13.6 Classe di problemi PSPACE-completi

Definizione 13.6.1. $\mathbb{A}$ è **PSPACE**-completo se:



- $\mathbb{A} \in$ **PSPACE**
- $\forall \mathbb{B} \in$ **PSPACE** $\quad \mathbb{B} \leqslant_K \mathbb{A}$ (hardness)

$\Rightarrow$ se $\mathbb{A}$ è **PSPACE**-completo e $\mathbb{A} \in$ **P** allora **P** $\equiv$ **PSPACE** (implica anche che **P** $\equiv$ **NP**)

13.7 Problemi Q-SAT e 2-Player-SAT

Q-SAT *(Quantified SAT)* $\in$ **PSPACE-Completo**

- Input: $\phi(x_1 \dots x_n) = \exists x_1 \forall x_2 \dots \exists x_{n-1} \forall x_n \phi(x_1 \dots x_n)$
- Output: $yes \Leftrightarrow \phi$ è vera

2-Player-SAT

- Input: $\phi(x_1 \dots x_n)$ CNF
- Output: $yes \Leftrightarrow P_1$ vince nel seguente gioco:
  P_1 e P_2 si alternano scegliendo i valori delle varie x_i. P_1 vince se ϕ è soddisfatta dai valori scelti, altrimenti vince P_2 (P_1 deve fare scelte che valgono $\forall$ mossa di P_2).

13.8 Problema Geography

- Input: grafo diretto $G = (V, E)$ diretto, $s \in V$
- Output: $yes \Leftrightarrow P_1$ vince nel seguente gioco:
  P_1 e P_2 si alternano scegliendo un vertice collegato da un arco nell'ultimo vertice scelto e non scelto ancora. Perde il primo che non ha più mosse.

Il problema **Geography** è **PSPACE-Completo** (si dimostra con *Q-SAT* $\leqslant$ *Geography*).

13.9 Problema Alternating Hamiltonian Path

- Input: grafo diretto orientato $G = (V, E)$, $s \in V$
- Output: $yes \Leftrightarrow P_1$ vince nel seguente gioco:
  P_1 e P_2 si alternano come in Geography. P_1 vuole completare un HamPath e P_2 vuole bloccare P_1.

Il problema **Geography** è **PSPACE-Completo** (si dimostra con *Q-SAT* $\leqslant$ *A-HamPath*).

14 Approssimazione

Per ogni *problema di ottimizzazione* è possibile definire il corrispondente problema di *decisione*. Diciamo che $\mathbb{A}^{opt}$ è **NP**-*hard* se il corrispondente problema di decisione è **NPC**.

14.1 Algoritmo di Approssimazione per un problema A^{opt}

Definizione 14.1.1. Sia $\mathbb{A}$ un problema di **minimizzazione**, $\mathcal{A}$ è un algoritmo di *k-approssimazione* per $\mathbb{A}$ se:

$$\forall x \in \mathcal{I}(\mathbb{A}) \quad \frac{\mathsf{VAL}(\mathcal{A}(x))}{\mathsf{VAL}(\mathsf{OPT}(x))} \leqslant k$$

Definizione 14.1.2. Sia $\mathbb{A}$ un problema di **massimizzazione**, $\mathcal{A}$ è un algoritmo di *k-approssimazione* per $\mathbb{A}$ se:

$$\forall x \in \mathcal{I}(\mathbb{A}) \quad \frac{\mathsf{VAL}(\mathsf{OPT}(x))}{\mathsf{VAL}(\mathcal{A}(x))} \leqslant k$$

dove

- *OPT(x)*: è una soluzione di valore massimo/minimo per l'istanza x.
- $\mathcal{A}(x)$: è la soluzione ritornata dall'algoritmo $\mathcal{A}$ sull'istanza x.

Un algoritmo di *1-approssimazione* è un algoritmo ottimo.

14.2 Approssimazione per il problema Makespan

- Input: n job/task $\{1, 2, \dots, n\}$ di taglia $j_1, j_2, \dots, j_n$
- Output: Partizione di $\{1 \dots n\}$, $M_1 \dots M_n$ tale che

$$\max 1 \leqslant k \leqslant m \sum_{i \in M_k} j_i \text{ è } \textit{minimo}$$

Il problema di minimizzazione *Makespan* è **NP-hard** (il problema di decisione è **NPC**).

14.3 Il problema SubSetSum (decisione)

- Input: $A = \{a_1 \dots a_n\}, S$
- Output: $yes \Leftrightarrow \exists A' \subseteq A t.c. \Sigma_{x \in A'} x = S$

Il problema di decisione *SubSetSum* è **NPC**

14.4 Il problema Partition (decisione)

- Input: $A = \{a_1 \dots a_n\}$
- Output: $yes \Leftrightarrow \exists A' \subseteq A t.c. \Sigma_{x \in A'} x = \Sigma_{x \notin A'} x$

Il problema di decisione *Partition* è **NPC** e sappiamo che *Partition* $\leqslant$ *Makespan* (con $m = 2$).

14.5 PTAS e FPTAS

PTAS è un algoritmo che data $x \in \mathcal{I}(\mathbb{A})$ e ϵ, fornisce una soluzione s per x t.c. $\dfrac{\mathsf{VAL}(\mathcal{A}(x))}{\mathsf{VAL}(\mathsf{OPT}(x))} \leqslant (1 + \epsilon)$.
Il running-time del *PTAS* $\mathcal{A}$ è $\mathcal{O}(|x|^k)$ (polinomiale in $|x|$ ma può essere esponenziale in $(\frac{1}{\epsilon})$).

FPTAS è un algoritmo che data $x \in \mathcal{I}(\mathbb{A})$ e ϵ, fornisce una soluzione s per x t.c. $\dfrac{\mathsf{VAL}(\mathcal{A}(x))}{\mathsf{VAL}(\mathsf{OPT}(x))} \leqslant (1 + \epsilon)$.
Il running-time del *PTAS* $\mathcal{A}$ è $\mathcal{O}(|x|^k(\frac{1}{\epsilon})^k)$ (polinomiale in $|x|$ e in $(\frac{1}{\epsilon})$).

14.6 Problema di Traveling Salesman

- Input: Grafo completo diretto pesato $G = (V, E)$, $w : E \mapsto \mathbb{N}$
- Output: ciclo hamiltoniano di peso minimo.

Il problema *TSP* è **NP-hard** (il problema di decisione è **NPC**, infatti *HamCycle* $\leqslant$ *TSP*$_{Decision}$).
TSP non è approssimabile (se P $\neq$ NP): non esiste un algoritmo polinomiale per TSP che garantisce $f(n)$ approssimazione, altrimenti risolverei *HamCycle* in tempo polinomiale.

14.7 Problema Knapsack

- Input: $v_1, \dots, v_n$ (valori) $\quad w_1, \dots, w_n$ (pesi)
- Output: $A \subseteq \{1 \dots n\}$ t.c. $\sum\limits_{i \in A} w_i \leqslant W, \quad \sum\limits_{i \in A} v_i$ è massima.

Il problema *Knapsack* è **NP-hard** (il problema di decisione è **NPC**, infatti *Partition* $\leqslant$ *Knapsack*).
Si risolve con programmazione dinamica: soluzione non polinomiale per istanze grandi.

14.8 Classe APX

Definizione 14.8.1 (Classe APX)**.**

$$\mathbf{APX}(r(n)) = \Big\{ \mathbb{A} \text{ di ottimizzazione} \;\Big|\; \text{esiste un algoritmo polinomiale per } \mathbb{A} \text{ che è } r(n)\text{-approssimabile} \Big\}$$

TSP $\notin$ APX, IndependentSet $\notin$ APX.
VertexCover $\in$ APX, Makespan $\in$ APX, Knapsack $\in$ APX.

14.9 Tecnica del GAP

Teorema 14.9.1. *Dato un problema di ottimizzazione $\mathbb{A}$ definiamo il problema $(a, b) - \mathbb{A}$ il problema **di decisione** che consiste nel dire se la soluzione ottima ad un'istanza di $\mathbb{A}$ è $\leqslant a$ o $\geqslant b$.*

Se esiste un problema $\mathbb{B} \in$ **NPC** t.c. $\mathbb{B} \leqslant (a, b) - \mathbb{A}$, allora *non esiste* un algoritmo di approssimazione polinomiale per $\mathbb{A}$ che garantisce approssimazione $k < \frac{b}{a}$ se P$\neq$ NP.

14.10 Problema Max-k-xor-SAT

- Input: formula k-xor-CNF, esempio: $\phi = (x_1 \oplus x_2 \oplus x_2) \wedge (\overline{x}_1 \oplus x_2 \oplus \overline{x}_3)$.
- Output: il massimo numero di clausole soddisfacibili.

Il problema è **NP-hard** $\forall k \geqslant 2$, altrimenti è inapprossimabile (problema di decisione **NPC**).
Il problema (a, b)-*gap-k-max-xor-sat* è **NPC** per ogni (a, b) t.c. $a < 1$ e $b > \frac{1}{2}$.

14.11 Max-3-SAT

- Input: formula ψ 3 CNF
- Output: assegnamento che soddisfa il massimo numero di clausole.

Il problema generale *Max-k-SAT* è **NP-hard** $\forall k \geqslant 2$, altrimenti è inapprossimabile.
Max-3-SAT non ammette approssimazione migliore di $\frac{8}{7}$.

14.12 Inapprossimabilità

VertexCover $\notin$ FPTAS.
IndependetSet $\notin$ FPTAS.