

UNIVERSITÀ DEGLI STUDI DI VERONA

---

# **Complessità**

---

RIASSUNTO DEI PRINCIPALI ARGOMENTI

*Matteo Danzi, Davide Bianchi*

21 aprile 2018

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Cos'è la complessità computazionale . . . . .	2
1.2	Problemi <i>facili</i> e <i>difficili</i> . . . . .	2
1.3	Risolvere vs Verificare . . . . .	3
<b>2</b>	<b>Problema computazionale</b>	<b>3</b>
2.1	Risolvere un problema computazionale . . . . .	3
2.2	Complessità di un problema computazionale . . . . .	4
2.3	Trattabilità di un problema. . . . .	4
<b>3</b>	<b>Le classi di problemi computazionali</b>	<b>4</b>
3.1	Classe P . . . . .	5
3.2	Classe Exp . . . . .	5
3.3	Classe Time(n) . . . . .	6
3.4	Classe NP . . . . .	7
<b>4</b>	<b>Riduzione alla Karp tra problemi di decisione</b>	<b>8</b>
4.1	Problema SAT . . . . .	8
4.2	Alcuni esempi di riduzioni tra problemi . . . . .	9
4.3	Problema NAE-K-SAT . . . . .	11
4.4	Transitività della riduzione alla Karp . . . . .	12
4.5	Problema Reachability . . . . .	12
4.6	Classe di problemi NP-Completi . . . . .	12

## 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;
For each 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 procedura 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	<i>facile</i>	<i>facile</i>
Hamiltonian Cycle	<i>difficile?</i>	<i>facile</i>
N è primo?	<i>facile</i>	<i>facile</i>
N ha un numero piccolo di fattori?	<i>difficile?</i>	<i>facile</i>

## 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.

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

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  $2n$  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

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

dove:

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

Si può quindi dire che

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

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

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

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

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

**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:  $\text{yes} \Leftrightarrow \exists \text{ Eulerian Cycle in } G$ .

Come abbiamo già visto quindi:

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

Eulerian Cycle  $\in \mathbf{P}$  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  $\in \mathbf{P}$ ? 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

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

**Esempio 3.2.1** (Hamiltonian Cycle). Ci chiediamo se Hamiltonian Cycle  $\in \mathbf{Exp}$ ? 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  $\notin \mathbf{Exp}$

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

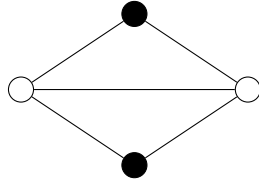
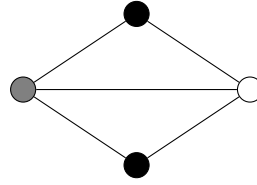
La risposta alla domanda è banalmente **si**, in quanto, dato un algoritmo  $\mathcal{B}$  con complessità  $T_{\mathcal{B}}(|x|)$ , possiamo dire che

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

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

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

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

(a) Grafo con colorazione *non* propria

(b) Grafo con colorazione propria

**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  $\in \mathbf{P}$ .

**Problema 3-Graph Colouring** Il problema 3-Graph Colouring  $\in \mathbf{P}$ ? Non sappiamo rispondere a questa domanda, poiché non sappiamo se esiste un algoritmo che lo svolga in tempo polinomiale. Il problema 3-Graph Colouring  $\in \mathbf{Exp}$ ? 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 monocolori 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  $\in \mathbf{Exp}$ .

### 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

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

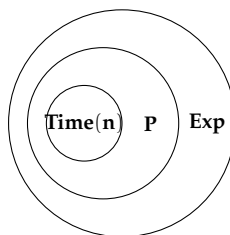
**Teorema 3.3.1.**  $\forall \mathbb{B} \text{ t.c. } \mathbb{B}(x) = \mathbb{A}(x) \quad T_{\mathbb{B}}(|x|) > |x|^c \quad \forall c \text{ costante}$

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



Possiamo riassumere quindi che:

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

Inoltre, con la definizione della classe **Time**(n) si può dire che:

$$P = \bigcup_{k \geq 0} \text{Time}(n^k)$$

$$\text{Exp} = \bigcup_{k \geq 0} \text{Time}(2^{n^k})$$

### 3.4 Classe NP

La classe **NP** (*non deterministic polynomial 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)

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

dove:

- $\mathcal{B}(\cdot, \cdot)$  è detto **verificatore** per  $A$ . Se la risposta di  $A$  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**  $\in \text{NP}$  ? Per vedere se il problema Hamiltonian cycle appartiene alla classe **NP** dobbiamo costruire un verificatore  $\mathcal{B}$  che agisca in tempo polinomiale.

```

VerifyHamCycle ( $G = (V, E)$ ,  $C = x_1, \dots, x_n$ )
  if  $r \neq |V|$ : return no                                }  $O(|w|)$ 
  for each  $v \in V$                                          }
  if  $v$  non appare in  $C$ : return no                         }  $O(|V| \cdot |C|)$ 
  for  $i=1$  to  $n-1$                                          }
  if  $(x_i, x_{i+1}) \notin E$ : return no                       }  $O(|C|)$ 
  if  $(x_1, x_n) \notin E$ : return no                         }  $O(1)$ 
  return yes

```

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



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

```

VerifyK-Colouring( $G = (V, E), f(v_1), \dots, f(v_n)$ )
  for each  $E(u, v)$ 
    if  $f(u) = f(v)$ : return no
  for  $i=1$  to  $n$ 
    if  $f(v_i) \geq K$ : return no
  return yes

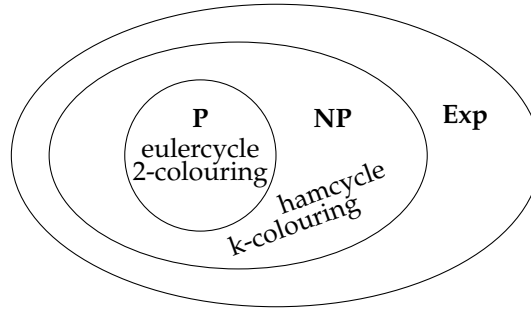
```

$\left. \begin{array}{l} \text{for each } E(u, v) \\ \text{if } f(u) = f(v): \text{ return no} \end{array} \right\} O(|E|)$   
 $\left. \begin{array}{l} \text{for } i=1 \text{ to } n \\ \text{if } f(v_i) \geq K: \text{ return no} \end{array} \right\} O(|V|)$

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  $A \in P$  dobbiamo dimostrare che esiste un verificatore. Tale verificatore per  $A$  sarà:  $B'(x, w) = B(x)$  privo di certificato. Dobbiamo dimostrare che se l'istanza è *yes* allora  $B(x) = \text{yes}$  altrimenti  $B(x) = \text{no}$ .

**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 4.0.2** (Riduzione alla Karp). Un problema di decisione  $A$  si riduce alla Karp al problema  $B$ :  $A \leq_K B$  se esiste un algoritmo polinomiale  $A$  tale che

$$\forall x \in J(A), B(A(x)) = \text{yes} \Leftrightarrow A(x) = \text{yes}$$

**Proposizione 4.0.1.** Se  $A \leq_K B$  e  $B \in P \Rightarrow A \in P$

**Proposizione 4.0.2.** Se  $A \leq_K B$  e  $B \notin P \Rightarrow A \notin P$

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:  $\text{yes} \Leftrightarrow \exists a_1 \dots a_n \in T, F^n$  t.c.  $\phi(a_1, \dots, a_n) = T$

**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.

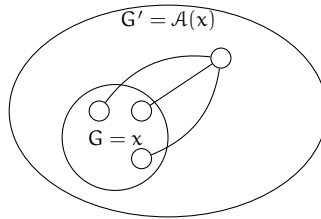
**Esempio 4.1.2 (3-SAT).**  $\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

## 4.2 Alcuni esempi di riduzioni tra problemi

**K-colouring  $\leq_K$  (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.

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

Prendiamo quindi il grafo  $G'$ :



per cui

$$\begin{aligned} G &= (V, E) \\ G' &= (V \cup \{v'\}, E \cup \{(v, v') \mid v \in V\}) \end{aligned}$$

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  $2\text{-col} \leq_K 3\text{-col} \leq_K 4\text{-col} \leq_K 5\text{-col}$

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

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

Dimostriamo che se  $\phi$  non è soddisfacibile allora non lo è neanche  $\phi'$ .

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

$$\begin{matrix} (l_1 \vee l_2 \vee z_1) \wedge (\bar{z}_1 \vee l_3 \vee z_2) \wedge (\bar{z}_2 \vee l_4 \vee z_3) \wedge (\bar{z}_3 \vee l_5 \vee z_4) \wedge \dots \wedge (\bar{z}_{k-1} \vee l_{k+1} \vee z_k) \\ \text{F} \quad \text{F} \quad \text{V} \quad \text{F} \quad \text{F} \quad \text{V} \quad \text{F} \quad \text{F} \quad \text{V} \quad \text{F} \quad \text{F} \quad \text{V} \quad \text{F} \quad \text{F} \quad \text{V} \quad \text{F} \quad \text{F} \quad \text{V} \end{matrix}$$

risulta che l'ultimo letterale  $z_k$  è falso, e quindi  $\phi'$  non è soddisfacibile.

**K-COL  $\leq_K$  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  $i = 0, 1, \dots, k-1$ . Per ogni  $v \in V$ :  $x_0^{(v)} x_1^{(v)} x_2^{(v)} \dots x_{k-1}^{(v)}$  dove  $x_i^{(v)} = T$  se il vertice  $v$  ha colore  $i$ .

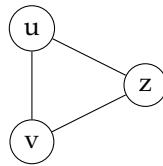
Ci chiediamo quindi quand'è che la formula è  $K$ -colorabile?

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

$\forall e = (u, v) \in E$  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:

Un vertice non può avere 2 colori

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

La trasformazione è polinomiale? La complessità della trasformazione è:

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

Quindi è polinomiale.

### 4.3 Problema NAE-K-SAT

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

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

**Esempio 4.3.1.**

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

$$x_1 = F \quad x_2 = F \quad x_3 = F \quad \text{non è NAE-K-SAT}$$

$$x_1 = F \quad x_2 = T \quad x_3 = F \quad \text{è NAE-K-SAT}$$

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

**3-SAT  $\leq_K$  NAE-4-SAT** Vogliamo dimostrare che data una qualsiasi formula  $\phi$  3-CNF la trasformo in una formula  $\psi$  4-CNF in tempo polinomiale.

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

$$\phi = C_1 \wedge C_2 \wedge \dots \wedge C_n \quad C_i = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)} \quad i = 1 \dots n$$

$$\psi = C'_1 \wedge C'_2 \wedge \dots \wedge C'_n \quad C'_i = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)} \vee z \quad i = 1 \dots n$$

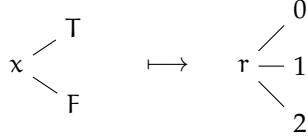
Per creare  $\psi$  espando le variabili e ne aggiungo sempre una. La trasformazione da  $\phi$  a  $\psi$  è polinomiale nella taglia della formula  $\phi$ , perché la scorro tutta per creare  $\psi$ .

Ora dobbiamo dimostrare che se  $\phi$  è soddisfacibile allora anche  $\psi$  è soddisfacibile:

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

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

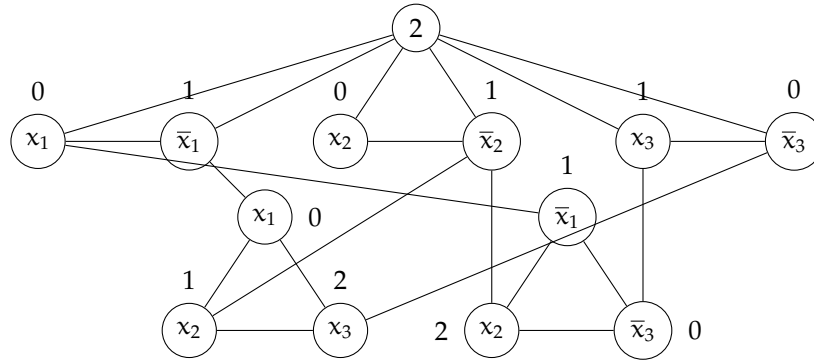
Mappo variabili (letterali) che possono valere T o F, su vertici (elementi del grafo) che hanno colore 0, 1, 2.



Partendo dalla formula  $\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$  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  $\leq_K$  è transitiva, ciò implica che:

$$A \leq_K B \text{ e } B \leq_K C \Rightarrow A \leq_K C$$

in particolare abbiamo che:

$$\begin{aligned} A \leq_K B \quad \exists A \text{ polytime } x \in J(A), A(x) \in J(B) \quad A(x) = \text{yes} &\Leftrightarrow B(A(x)) = \text{yes} \\ B \leq_K C \quad \exists B \text{ polytime } y \in J(B), B(y) \in J(C) \quad B(y) = \text{yes} &\Leftrightarrow C(B(y)) = \text{yes} \end{aligned}$$

Perciò

$$\forall x \in J(A), B(A(x)) \in J(C) \quad A(x) = \text{yes} \Leftrightarrow C(B(A(x))) = \text{yes} \Rightarrow C(x) = B(A(x))$$

#### 4.5 Problema Reachability

#### 4.6 Classe di problemi NP-Completi