

UNIVERSITÀ DEGLI STUDI DI VERONA

---

# **Complessità**

---

RIASSUNTO DEI PRINCIPALI ARGOMENTI

*Matteo Danzi, Davide Bianchi*

29 giugno 2018

# Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	Cos'è la complessità computazionale . . . . .	4
1.2	Problemi <i>facili</i> e <i>difficili</i> . . . . .	4
1.3	Risolvere vs Verificare . . . . .	5
<b>2</b>	<b>Problema computazionale</b>	<b>5</b>
2.1	Risolvere un problema computazionale . . . . .	5
2.2	Complessità di un problema computazionale . . . . .	6
2.3	Trattabilità di un problema. . . . .	6
<b>3</b>	<b>Le classi di problemi computazionali</b>	<b>6</b>
3.1	Classe <b>P</b> . . . . .	7
3.2	Classe <b>Exp</b> . . . . .	7
3.3	Classe Time(n) . . . . .	8
3.4	Classe <b>NP</b> . . . . .	9
<b>4</b>	<b>Riduzione alla Karp tra problemi di decisione</b>	<b>10</b>
4.1	Problema SAT . . . . .	11
4.2	Alcuni esempi di riduzioni tra problemi . . . . .	11
4.3	Problema NAE-K-SAT . . . . .	13
4.4	Transitività della riduzione alla Karp . . . . .	14
4.5	Problema Reachability . . . . .	15
<b>5</b>	<b>Riduzione alla Turing tra problemi di decisione</b>	<b>15</b>
<b>6</b>	<b>Classe di problemi NP-Completi</b>	<b>15</b>
6.1	Circuito Booleano . . . . .	15
6.2	Problema Circuit-SAT . . . . .	16
6.3	Relazione tra <b>P</b> , <b>NP</b> , e <b>NP-completo</b> . . . . .	17
<b>7</b>	<b>Classe di problemi CO-NP</b>	<b>18</b>
7.1	Relazione tra <b>P</b> , <b>NP</b> e <b>CO-NP</b> . . . . .	18
7.2	Problema Minimo circuito booleano . . . . .	19
<b>8</b>	<b>Gerarchia Polinomiale</b>	<b>20</b>
8.1	Funzione time-costruibile . . . . .	21
8.2	Problema Catch 22 . . . . .	21
<b>9</b>	<b>Teorema di Ladner</b>	<b>21</b>
9.1	Problema Clique . . . . .	22
9.2	Problema Independent Set . . . . .	24
<b>10</b>	<b>Ricavare problemi di ottimizzazione e ricerca</b>	<b>26</b>
10.1	Independent Set . . . . .	26
10.2	Problema SAT-Search . . . . .	27
10.3	Self Reducibility . . . . .	27
10.4	Problema Graph Isomorphism . . . . .	28
10.5	Problema No-small-Factor . . . . .	29
10.6	Problema Vertex Cover . . . . .	30
10.7	Problema Hitting Set . . . . .	30
<b>11</b>	<b>Non determinismo e classe NTIME</b>	<b>31</b>

<b>12 Alcuni problemi NP-Completi</b>	<b>32</b>
12.1 Problema Max-Cut . . . . .	32
12.2 Problema Max-K-SAT . . . . .	34
12.3 Problema Set-Splitting . . . . .	35
12.4 Problema Set-Cover . . . . .	36
12.5 Classe di problemi DP e Problema Clique-No-Clique . . . . .	36
12.6 Problema D-Ham-Path . . . . .	37
<b>13 Complessità di Spazio e la classe SPACE</b>	<b>39</b>
13.1 Classe PSPACE, L e NTIME . . . . .	39
13.2 Non determinismo e classe NSPACE . . . . .	41
13.3 Problema Reachability in termini di spazio . . . . .	42
13.4 Classe NPSPACE . . . . .	43
13.5 Teorema di Savitch . . . . .	44
13.6 Classe di problemi PSPACE-completi . . . . .	45
13.7 Problemi Q-SAT e 2-Player-SAT . . . . .	45
13.8 Problema Geography . . . . .	45
13.9 Problema Alternating Hamiltonian Path . . . . .	45
<b>14 Approssimazione</b>	<b>45</b>
14.1 Approssimazione per il problema Makespan . . . . .	45
14.2 PTAS e FPTAS . . . . .	45
14.3 Problema di Traveling Salesman . . . . .	45
14.4 Problema Knapsack . . . . .	46
14.5 Classe APX . . . . .	46
14.6 Tecnica del GAP . . . . .	46
14.7 Problema Max-k-xor-SAT . . . . .	46
14.8 Inapprossimabilità . . . . .	46

## Elenco delle figure

1	Esempi di grafi per Graph Colouring . . . . .	8
2	Esempio di circuito booleano con 4 input, il nodo finale di output è detto nodo <i>sink</i> . . . . .	16
3	Esiste il problema A? . . . . .	21
4	Grafo in cui c'è un arco per ogni letterale diverso dal proprio negato e che non appartiene alla stessa clausola . . . . .	23
5	Esempio di Tree independent Set . . . . .	24
6	Due esempi di grafi per vertex cover . . . . .	30
7	Esempio di albero delle tracce di esecuzione per l'algoritmo SAT-Solver . . . . .	31
8	Esempio di Max Cut con $k = 5$ ( $nm$ = non monocromatico) . . . . .	32
9	Per questo grafo esistono $\tilde{k}$ vertici che toccano tutti gli archi . . . . .	36
10	Istanze del problema D-Ham-Path . . . . .	37
11	Rappresentazione per la clausola $C_1$ del grafo con cammino Hamiltoniano . . . . .	38
12	$P \subseteq NP \subseteq PSPACE$ . . . . .	40
13	In questa memoria i bit cambiano a seconda delle istruzioni dell'algoritmo . . . . .	41
14	Esempio di grafo $G_x^\Pi$ . . . . .	42

## Listings

1	Verificatore per HamCycle . . . . .	9
2	Verificatore per K-Colouring . . . . .	10
3	Algoritmo che risolve TreeIndependentSet . . . . .	24
4	Algoritmo di Ottimizzazione per IndSet . . . . .	26
5	Algoritmo di Ricerca per SAT . . . . .	27
6	Graph Isomorphism Search . . . . .	29

---

7	Esempio di algoritmo non deterministico . . . . .	31
8	Algoritmo ricorsivo per risolvere Reachability . . . . .	43
9	SAT-Solver-ND $\in \mathbf{NPSPACE}(n)$ . . . . .	44
10	SAT-Verifier . . . . .	44

## 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 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:

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

$\text{Eulerian Cycle} \in \mathbf{P}$  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  $\text{Hamiltonian Cycle} \in \mathbf{P}$ ? 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  $\text{Hamiltonian Cycle} \in \mathbf{Exp}$ ? 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  $\text{Hamiltonian Cycle} \notin \mathbf{Exp}$

**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:  $\text{yes} \Leftrightarrow \exists \text{ colorazione propria dei vertici di } G \text{ ovvero:}$

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



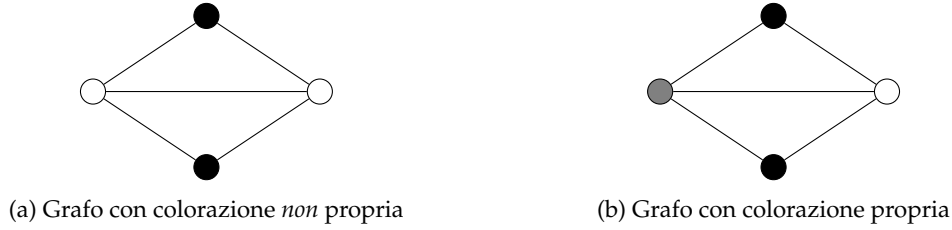


Figura 1: Esempi di grafi per Graph Colouring

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

**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

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

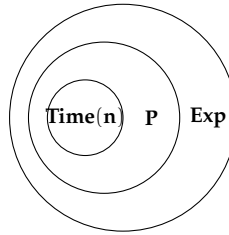
**Teorema 3.3.1.**  $\forall \mathbb{B}$  t.c.  $\mathbb{B}(x) = \mathbb{A}(x) \quad T_{\mathbb{B}}(|x|) > |x|^c \quad \forall c \text{ costante}$

**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)

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

dove:

- $\mathcal{B}(\cdot, \cdot)$  è detto **verificatore** per  $\mathbb{A}$ . 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**  $\in \text{NP}$  ? 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 ( $G = (V, E)$ ,  $C = x_1, \dots, x_n$ )
  if  $r \neq |V|$  : return no
  foreach  $v \in V$ 
    if  $v$  non appare in  $C$  : return no
  for  $i=1$  to  $n-1$ 

```

```

    if  $(x_i, x_{i+1}) \notin E$ : return no
    if  $(x_1, x_n) \notin E$ : return no
    return yes

```

$\left. \begin{array}{l} \\ \end{array} \right\} O(|C|)$   
 $\left. \begin{array}{l} \\ \end{array} \right\} O(1)$

---

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

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

Algorithm 2: Verificatore per K-Colouring

```

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

```

$\left. \begin{array}{l} \\ \end{array} \right\} O(|E|)$   
 $\left. \begin{array}{l} \\ \end{array} \right\} O(|V|)$

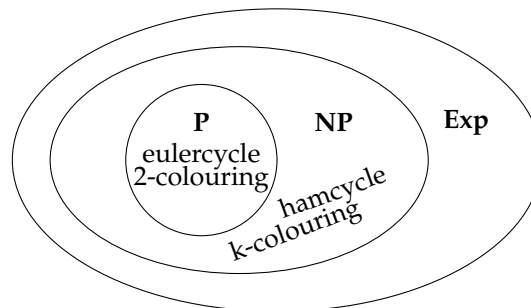
---

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

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

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

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

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

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

Come effettivamente svolgiamo le trasformazioni?  $\left. \begin{array}{l} \\ \end{array} \right\} O(|w|)$   
 $\left. \begin{array}{l} \\ \end{array} \right\} O(|V| \cdot |C|)$

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

**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.

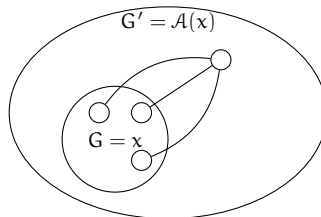
**Esempio 4.1.2 (3-SAT).**  $\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

## 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

$$G = (V, E)$$

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

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

**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  $i$ .

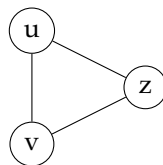
Ci chiediamo quindi quand'è che la formula è  $K$ -colorabile?

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

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

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

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.

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

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

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

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  $\psi$ .

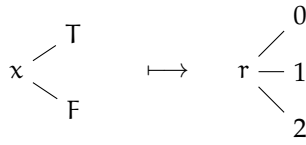
Ora dobbiamo dimostrare che se  $\phi$  è soddisfacibile allora anche  $\psi$  è soddisfacibile:

- $\phi$  è soddisfacibile  $\Rightarrow \exists \underline{a} \in \{T, F\}^n$  t.c.  $\phi(\underline{a}) = T$ .

- Se prendiamo l'assegnamento  $\underline{b} = \underline{a}$   $z = F$   $\psi(\underline{b}) = T$  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  $\underline{b}$   $z = F$  allora, la parte rimanente di  $\underline{b}$  soddisfa  $\psi$
- Se secondo  $\underline{b}$   $z = T$  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.

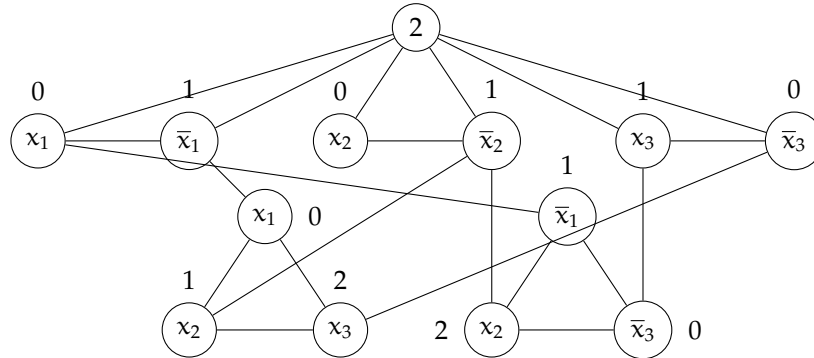
Mappo variabili (letterali) che possono valere T o F, su vertici (elementi del grafo) che hanno colore 0, 1, 2.



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:

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

Perciò

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

## 4.5 Problema Reachability

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

Quanto costa risolvere Reachability?

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*  $\in \mathbf{P}$

## 5 Riduzione alla Turing tra problemi di decisione

**Definizione 5.0.1** (Riduzione alla Turing).  $\mathbb{A} \leq_T \mathbb{B}$  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  $O(1)$ ,  $\mathcal{A}(x) = \mathbb{A}(x)$ .

$2\text{-SAT} \leq_T \text{Reachability}$ : 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

- $\mathbb{A} \in \mathbf{NP}$
- $\mathbb{A}$  è NP-hard, cioè se  $\forall \mathbb{B} \in \mathbf{NP} \quad \mathbb{B} \leq_K \mathbb{A}$

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

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

### 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  $C_4(x_1, x_2, x_3, x_4)$ :



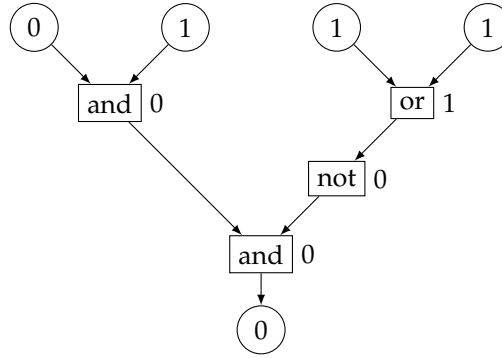


Figura 2: Esempio di circuito booleano con 4 input, il nodo finale di output è detto nodo *sink*.

## 6.2 Problema Circuit-SAT

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

Definiamo una famiglia di circuiti  $C_{n \geq 0}$  (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:

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

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

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

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

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

**Circuit SAT è NP-completo** Dimostriamo prima a parole che Circuit-SAT  $\in \mathbf{NP}$ . Forniamo il verificatore  $V(\underline{x}, w)$  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  $\forall \mathbb{A} \in \mathbf{NP} \quad \mathbb{A} \leq_K \text{Circuit-SAT}$ . 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:

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

Sia  $\mathbb{A} \in \mathbf{NP}$  allora  $\exists V_{\mathbb{A}}(\underline{x}, w)$  per le istanze  $\underline{x} \in \mathcal{I}(\mathbb{A})$ , tale che  $V_{\mathbb{A}}$  ha complessità  $O(p(|\underline{x}|)) = |w|$  (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  $V_{\mathbb{A}}$ :

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

perciò, se consideriamo  $C'_x(\underline{x}) = C_m(\underline{x}, w)$

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

**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.

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

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

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

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  $\phi$ , allora possiamo mettere in and tutte le clausole e dire che il circuito  $C$  è soddisfatto se e solo se  $\phi$  è soddisfatta.

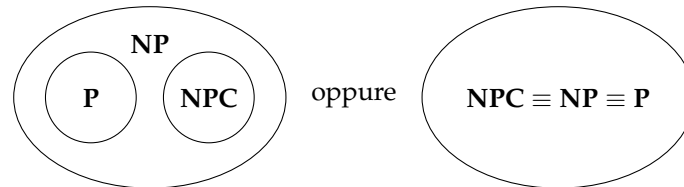
Con questo e con la dimostrazione che Circuit-SAT è NP-completo possiamo dire che

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

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

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

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

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

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:  $\text{yes} \Leftrightarrow b = 1$

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

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

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  $y$

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

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:

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

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  $\phi$
- Output: yes se  $\forall a \phi(a) = T$

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.  $\mathbb{A} \in \mathbf{NPC} \cap \mathbf{CO-NP}$  allora  $\mathbf{NP} \equiv \mathbf{CO-NP}$ .

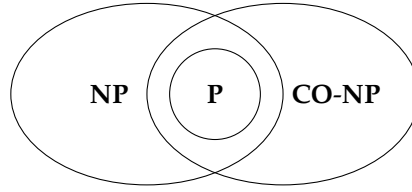
**CO-NP  $\subseteq$  NP.** Supponiamo che  $\mathbb{A} \in \mathbf{NPC}$  allora  $\mathbb{A} \in \mathbf{NP}$  e  $\forall \mathbb{C} \in \mathbf{NP} \quad \mathbb{C} \leq_K \mathbb{A}$ .

Se prendiamo il problema  $\mathbb{B} \in \mathbf{CO-NP}$   $\overline{\mathbb{B}} \in \mathbf{NP}$ .

Allora esiste una riduzione alla Karp  $\overline{\mathbb{B}} \leq_K \mathbb{A}$  che mappa le istanze yes di  $\mathbb{B}$  alle istanze no di  $\mathbb{A}$  ed esiste anche una riduzione  $\mathbb{B} \leq_K \overline{\mathbb{A}}$  che è duale alla precedente.

Poiché  $\mathbb{A} \in \mathbf{CO-NP}$  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 **CO-NP  $\subseteq$  NP**. □

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



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

$$\begin{aligned} A(x) \in NP & \quad A(x) = \exists w B(x, w) \in P \\ A(x) \in CO-NP & \quad A(x) = \forall w B(x, w) \in P \end{aligned}$$

- Se  $NP \neq CO-NP \Rightarrow P \neq NP$
- Se  $P = NP$  siccome  $P = CO-NP \quad \forall A \in NP, A \in P \Rightarrow \bar{A} \in P = NP \Rightarrow NP = CO-NP$

**Definizione 7.1.1** (Hardness del problema  $A$  nella classe  $CO-NP$ ).  $A$  è **CO-NP-completo** se  $A \in CO-NP$  e  $\forall B \in CO-NP \quad B \leq_K A$ .

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

*Dimostrazione.* Se  $A$  è **NP-completo**, allora

- $A \in NP$
- $\forall B \in NP \quad B \leq_K A$

Dalla prima deduciamo che  $\Rightarrow \bar{A} \in CO-NP$

Dalla seconda invece, se  $C \in CO-NP, \quad \bar{C} \in CO-NP \Rightarrow \bar{C} \leq_K A \Rightarrow C \leq_K \bar{A}$   
 $\Rightarrow \forall C \in CO-NP \Rightarrow C \leq_K \bar{A}$

Da queste due deduzioni abbiamo quindi la definizione di **CO-NP-completo** per  $\bar{A}$  □

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

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

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

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*:  $\text{yes} \Leftrightarrow \nexists$  circuito  $C'$  t.c.  $\forall x \quad C'(x) = C(x)$  con  $|C'| < |C|$

Consideriamo l'algoritmo  $A$

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

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  $\Pi_i P$ ).

$$\Pi_i P = \{A(x) = \forall w_1 \exists w_2 \forall w_3 \exists w_4 \dots Q_i w_i \quad B(x, w_1, \dots, w_i) \quad \text{dove } |w_i| = O(p_i(|x|)) \text{ e } B \in P\}$$

**Definizione 8.0.2** (Classe di problemi  $\Sigma_i P$ ).

$$\Sigma_i P = \{A(x) = \exists w_1 \forall w_2 \exists w_3 \forall w_4 \dots Q_i w_i \quad B(x, w_1, \dots, w_i) \quad \text{dove } |w_i| = O(p_i(|x|)) \text{ e } B \in P\}$$

Dalla definizione di queste classi di problemi deduciamo che:

$$\Pi_0 P = \Sigma_0 P = P \quad A(x) = B(x) \text{ non ho quantificatori}$$

$$\Pi_1 P = \mathbf{CO-NP}$$

$$\Sigma_1 P = \mathbf{NP}$$

$$\text{Minimo circuito booleano} \in \Pi_2 P$$

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

**Osservazione 8.0.2.**  $\Pi_i P \subseteq \Sigma_{i+1} P$  e  $\Sigma_i P \subseteq \Pi_{i+1} P$ .

Infatti se aggiungo un quantificatore all'inizio, ho che

$$A(x) \in \Pi_i P$$

$$A(x) = \forall w_1 \exists w_2 \dots Q_i w_i \quad B(x, w_1, w_2, \dots, w_i)$$

$$\Sigma_{i+1} P = \exists w^* \forall w_1 \exists w_2 \dots Q_i w_i \quad B'(x, w^*, w_1, w_2, \dots, w_i)$$

Perciò  $B'(\dots) = B(\dots)$

**Osservazione 8.0.3.** Per lo stesso motivo dell'osservazione precedente vale che:

$$\Pi_i P \subseteq \Pi_{i+1} P \quad \text{e} \quad \Sigma_i P \subseteq \Sigma_{i+1} P.$$

**Osservazione 8.0.4.** Se  $P \equiv \mathbf{NP} \Rightarrow \forall i \Sigma_i P = P \wedge \Pi_i P = P$

cioè abbiamo che:

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

**Proposizione 8.0.1.** Se  $\mathbf{NP} = \mathbf{CO-NP} \Rightarrow \Sigma_1 P = \Pi_1 P$ .

Quindi  $\Sigma_i P = \Pi_i P = \Sigma_1 P = \Pi_1 P \quad \forall i \geq 1$ .

Tutte le classi sopra collassano sulla classe 1.

*Dimostrazione.* Assumiamo che  $\mathbf{NP} \equiv \mathbf{CO-NP}$ :

$$A(x) = \exists w_1 \quad B(x, w_1) \Leftrightarrow A(x) = \forall w'_1 \quad B'(x, w_1)$$

$$\text{Sia } A'(x) \in \Sigma_2 P \quad A'(x) = \exists w_2 \forall w_1 \quad C(x, w_1, w_2) = \mathcal{D}_{w_2}(x).$$

$$\mathcal{D}_{w_2}(x) \in \mathbf{CO-NP} \equiv \mathbf{NP} \quad \text{quindi } \mathcal{D}_{w_2}(x) = \exists w'_1 \quad C'(x, w'_1, w_2) \text{ perciò diventa:}$$

$$\begin{aligned} A'(x) &= \exists w_2 \exists w'_1 \quad C'(x, w'_1, w_2) \\ &= \exists w_{12} \quad C'(x, w_{12}) \in \mathbf{NP} \end{aligned}$$

Quindi deduciamo che se  $\mathbf{NP} \equiv \mathbf{CO-NP} \Rightarrow \Sigma_2 P = \Sigma_1 P$

Inoltre se  $\mathbf{NP} \equiv \mathbf{CO-NP} \Rightarrow \Pi_2 P = \Pi_1 P$  □

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

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

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

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

la gerarchia polinomiale collassa in  $P$ .

Se  $\mathbf{NP} = \mathbf{CO-NP} \Rightarrow \mathbf{PH} = \mathbf{NP} = \mathbf{CO-NP}$ .

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

## 8.1 Funzione time-costruibile

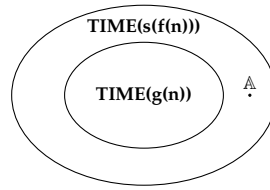
**Proposizione 8.1.1.** Nel modello computazionale in oggetto è possibile simulare  $t$  passi di un algoritmo (programma) mentre controlliamo che  $\leq t$  passi sono fatti in  $s(t)$  passi.

**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  $\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$



## 8.2 Problema Catch 22

- Input:  $\Pi$  (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  $\Pi_{22}$  tale che risolve il problema Catch 22 in  $g(n)$  passi, dove  $g(n) < f(n)$ . Questo è equivalente a dire che  $\text{Catch 22} \in \mathbf{TIME}(g(n))$ .

Se  $\Pi_{22}(\Pi_{22}) = \text{Catch 22}(\Pi_{22})$  siccome ci mette meno di  $f(\Pi_{22})$  passi, allora è 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  $\Pi_{22}$  che impiega  $g(n) < f(n)$  passi.

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

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

**Proposizione 8.2.2.** Per ogni  $n \geq |\Pi_{22}|$  fissato esiste un altro  $\Pi'_{22}$  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 né alla classe **P** né alla classe **NPC**.

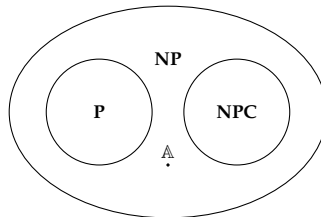


Figura 3: Esiste il problema  $A^*$ ?

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

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

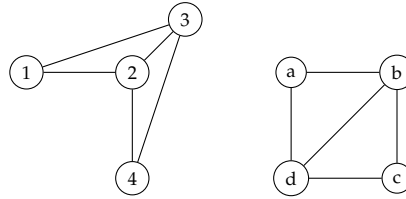
### Graph Isomorphism

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

**Definizione 9.0.1** (Isomorfismo).  $\exists f: V(G_1) \mapsto V(G_2)$

t.c.  $(v, u) \in E(G_1) \Leftrightarrow (f(v), f(u)) \in E(G_2)$

**Esempio 9.0.1** (Grafici isomorfi). Ecco un esempio di due grafici isomorfi:



$$f(1) = a \quad f(2) = b$$

$$f(3) = c \quad f(4) = d$$

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

Vogliamo far vedere che:

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

□

## 9.1 Problema Clique

- Input: grafo  $G = (V, E), K$
- Output:  $\text{yes} \Leftrightarrow G$  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  $3\text{-SAT} \leq_K \text{Clique}$  che trasforma in tempo polinomiale una formula  $\phi$  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  $(u, v) \in E$ .  $[O(|K|^2 \times |E|)]$

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  $\leq_K$  Clique** Vediamo la seguente riduzione che mappa la formula

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

in un grafo che soddisfa il problema Clique.

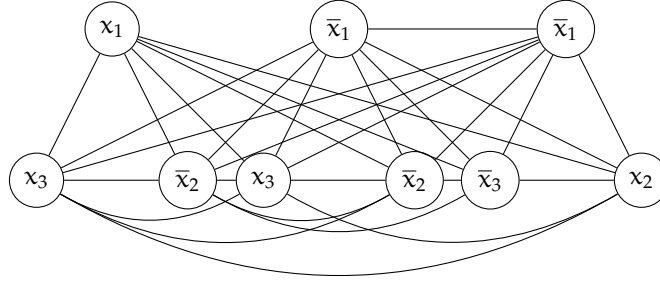


Figura 4: Grafo in cui c'è un arco per ogni letterale diverso dal proprio negato e che non appartiene alla stessa clausola

Sappiamo che se:

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

Il grafo mostra le seguenti caratteristiche:

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

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

Dimostriamo ora che se  $\phi$  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  $v_{i1}^{(1)}, v_{i2}^{(2)}, \dots, v_{in}^{(n)}$  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  $\phi$  è soddisfacibile. Supponiamo che G abbia una clique C di 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  $\phi$ .

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

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

Perciò abbiamo che

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



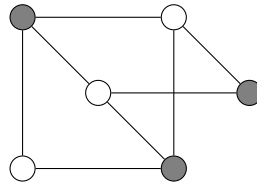
## 9.2 Problema Independent Set

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

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

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

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



**IndSet**  $\in$  **NPC** Esiste una riduzione  $\text{Clique} \leq_K \text{IndSet}$  tale che

$$(G = (V, E), k) \mapsto (G' = (V, E), k)$$

**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  $k$ .

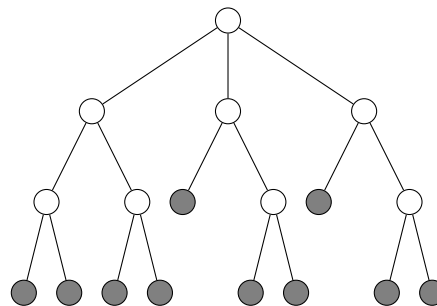


Figura 5: Esempio di Tree independent Set

**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( $G = (V, E), k$ )
   $I \leftarrow \emptyset$ 
  while  $V \neq \emptyset$ :
    foreach  $v$  t.c.  $d(v) \leq 1$ :
       $I \leftarrow I \cup \{v\}$ 
      remove i vicini  $v$  da  $G$ 
  if  $|I| \geq k$  return yes
  else return no

```

---

**Problema Only Small Independent Set** Vediamo ora il problema OSIS:

- Input:  $G = (V, E), k$
- Output:  $\text{yes} \Leftrightarrow$  ogni Independent Set  $I$ ,  $|I| \leq k$ .

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

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

Perciò avremmo che

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

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

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  $\text{OSIS} \in \mathbf{CO-NPC}$ .

## 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:  $\text{yes} \Leftrightarrow G$  ha un IndSet di cardinalità  $\geq k$
- **Search Problem: IndSet-Search**
  - Input:  $G, k \in \mathbb{N}$
  - Output: un IndSet di  $G$  t.c.  $|I| \geq k$  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**:

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

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

Per  $v \in V$  se in  $G - v - \{u \mid (u, v) \in E\}$  (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  $G - v - \{u \mid (u, v) \in E\}$  contiene un IndSet  $I'$  di taglia  $k^* - 1$  allora  $I' \cup \{v\}$  è un IndSet di  $G$ . Dove  $|I \cup \{v\}| = k^*$

Vediamo ora l'algoritmo che permette di costruire un IndSet:

Algorithm 4: Algoritmo di Ottimizzazione per IndSet

---

```

CostruisciIndSet( $G, k^*$ )
  if  $\mathcal{A}(G, k^*) = \text{no}$ :
    return no
  else
     $\tilde{G} \leftarrow G, I \leftarrow \emptyset$ 
    foreach  $v \in V$ :
      if  $\mathcal{A}(\tilde{G} - v - N(v), k - 1) = \text{yes}$ :
         $I \leftarrow I \cup \{v\}$ 
         $\tilde{G} \leftarrow \tilde{G} - v - N(v)$ 
         $k \leftarrow k - 1$ 
    return  $I$ 

```

---

Dove  $N(v) = \{u \mid (u, v) \in E\}$

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:  $\phi$  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( $\phi$ )
  if  $\mathcal{A}(\phi) = \text{no}$ :
    return no
  for  $i = 1$  to  $n$ :
     $a_i \leftarrow T$ 
    if  $\mathcal{A}(\phi_{x_1=a_1 \dots x_i=a_i}) = \text{no}$ :
       $a_i \leftarrow F$ 
  return  $a_1, a_2, \dots, a_i$ 

```

---

Qual è la complessità?  $T_{\text{SAT-Solver}}(|\phi|) = O(|\phi| \cdot T_{\mathcal{A}}(|\phi|))$ , 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 Reducibility

**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

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

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

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

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

**Definizione 10.3.3** (Self Reducible).  $\mathbb{A} \in \mathbf{NP}$  (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  $V_{\mathbb{A}}(x, w) = \text{yes}$ , se  $w$  esiste.

Sappiamo che se  $\mathbb{A} \in \mathbf{NPC}$  allora  $\mathbb{A} \leq_K \text{SAT}$ .

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

Se  $\mathbb{A} \in \mathbf{NPC}$  allora  $\text{SAT} \leq_K \mathbb{A}$  e quindi un oracolo per  $\mathbb{A}$  implica un oracolo per  $\text{SAT}$ .

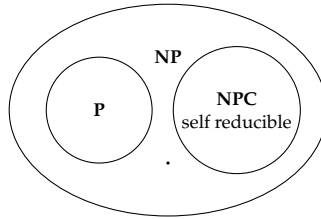
Prendiamo  $x \in \mathcal{I}(\mathbb{A})$  e lo trasformiamo in  $\phi^{(x)}$  di  $\text{SAT}$  utilizzando il teorema *Cook-Levin*. Sappiamo che

$$\begin{aligned} \text{SAT}(\phi^{(x)}) = \text{yes} &\Leftrightarrow \mathbb{A}(x) = \text{yes} \\ V_{\mathbb{A}}(x, w) = \text{yes} &\Leftrightarrow V_{\text{SAT}}(\phi^{(x)}, \underline{w}) = \text{yes} \end{aligned}$$

Possiamo produrre  $w$  usando l'algoritmo  $\text{SAT-Solver}$  (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  $\text{SAT}$  a  $\mathbb{A}$ . In questo modo il certificato  $w$  che costruisce  $\text{SAT-solver}$  è lo stesso che serve a  $V_{\mathbb{A}}$ .  $\square$



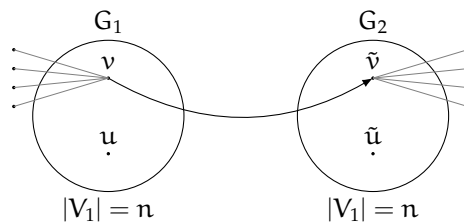
Vediamo ora un problema in  $\mathbf{NP}$  che non crediamo sia in  $\mathbf{NPC}$ .

**10.4 Problema Graph Isomorphism**

Versione **Graph Isomorphism-Search**:

- Input:  $G_1 = (V_1, E_1)$ ,  $G_2 = (V_2, E_2)$  semplici e non diretti
- Output: una funzione  $f : v_1 \mapsto v_2$  t.c.  $\forall (v, u) \in E_1$   
 $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$ . 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  $\mathcal{O}^{\text{GI-Decision}}(G_1, G_2) = \text{no}$ :
    return no
  foreach  $v_i \in V_1$ : // Fissiamo  $v \in V_1, \tilde{v} \in V_2$ 
    foreach  $\tilde{v}_i \in V_2$ :
       $\tilde{G}_1 \leftarrow$  aggiungiamo  $n$  vertici a  $V_1$  come vicini di  $v$ 
       $\tilde{G}_2 \leftarrow$  aggiungiamo  $n$  vertici a  $V_2$  come vicini di  $\tilde{v}$ 
      if  $\mathcal{O}^{\text{GI-Decision}}(G_1, G_2) = \text{yes}$ :
         $f(v) = \tilde{v}$ 
         $G_1 \leftarrow \tilde{G}_1, G_2 \leftarrow \tilde{G}_2$ 
        break

```

---

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

*Dimostrazione.* Partiamo dunque dall'ipotesi che

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

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

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

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

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

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

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

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

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

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

se  $V_{\text{yes}}(x, w) = \text{yes}$  allora  $\mathbb{A}(x) = \text{yes}$

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

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

## 10.5 Problema No-small-Factor

- Input: due numeri interi  $q, r$
- Output:  $\text{yes} \Leftrightarrow q$  non ha un divisore  $\leq r$

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

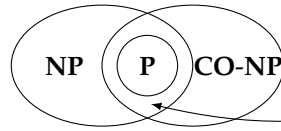
Per trovare il minimo fattore di  $q$  ho un costo di  $O(\log_{10} q \cdot \log q)$ . Quindi è polinomiale in  $|q|$ .

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

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

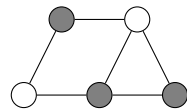
se per ogni  $i$   $a_i < r$  e la fattorizzazione è giusta e  $a_i$  sono primi, allora ritorno yes. Tutto questo è fattibile in tempo polinomiale.

Per verificare che il problema è in  $\mathbf{CO-NP}$  il verificatore semplicemente controlla che ci sia un divisore più piccolo di  $r$  dividendo  $q$ , tutto questo in polytime. Quindi il problema è qui

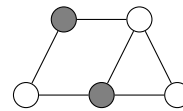


## 10.6 Problema Vertex Cover

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



(a) è vertex cover



(b) non è vertex cover

Figura 6: Due esempi di grafi per vertex cover

Dimostriamo che il problema  $\in \mathbf{NPC}$ , partiamo col dimostrare che  $\in \mathbf{NP}$ :

- Certificato:  $U$ ,  $|U| \leq k$ , 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  $\mathbf{NP-hard}$ : troviamo  $\mathbb{A} \in \mathbf{NPC}$  t.c.  $\mathbb{A} \leq_K \text{VC}$ .

Utilizziamo  $\mathbb{A} = \text{IndSet}$ :

$(\Leftarrow)$   $U$  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  $\exists (u, v) \in E$  t.c.  $\forall w \in V \setminus I$  ( $w \neq u, w \neq v$ )  $\Rightarrow u, v \in I \Rightarrow I$  non è un IndSet.

## 10.7 Problema Hitting Set

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

Facciamo vedere che Hitting Set  $\in \mathbf{NPC}$ .

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

- Certificato: insieme  $D$  con  $|D| \leq m$  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:

$$\text{Vertex Cover} \leq_K \text{Hitting Set}$$

$$G = (V, E), k \Leftrightarrow (U, F, m)$$

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  $M_1, M_2, \dots$ , ha un insieme di coppie.

## 11 Non determinismo e classe NTIME

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

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

**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

```

0 SAT-Solver-ND( $\phi(x_1, \dots, x_n)$ )
1   for  $i = 1$  to  $n$ 
2      $a_i \leftarrow T$ 
3     goto both 4, 5
4      $a_i \leftarrow F$ 
5   end for
6   if ( $\phi(a_1, \dots, a_n) == T$ )
7     return yes
8   else
9     return no
10

```

}  $O(|x|^k)$

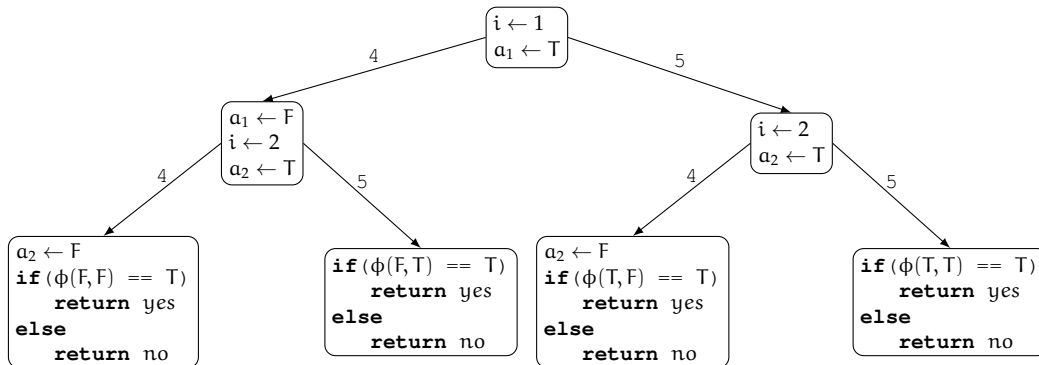


Figura 7: Esempio di albero delle tracce di esecuzione per l'algoritmo SAT-Solver



**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  $\text{NTIME}(f(n))$  come:

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

**Osservazione 11.0.2.** Osserviamo che:

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

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

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

## 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.

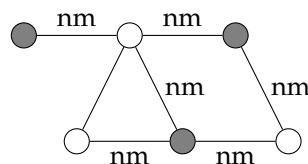


Figura 8: Esempio di Max Cut con  $k = 5$  (nm = non monocromatico)

*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  $\leq_K$  Max-Cut:

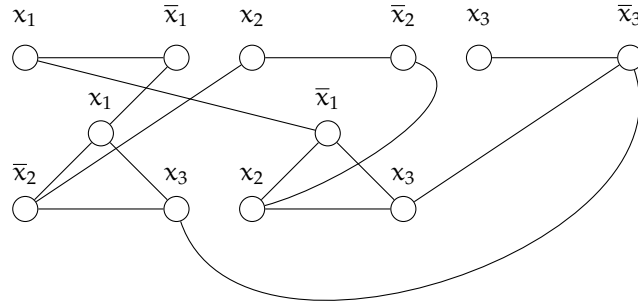
Data  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_m$  vogliamo trovare  $(G, k)$ .

Per ogni variabile  $x$  di  $\phi$  aggiungiamo un arco in  $G$  i cui vertici sono etichettati  $x$  e  $\bar{x}$ .

Per ogni clausola aggiungiamo in  $G$  un triangolo con i vertici etichettati come i letterali.

Collegiamo  $l$  in un triangolo con  $\bar{l}$  negli archi messi sopra.

es.:  $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$



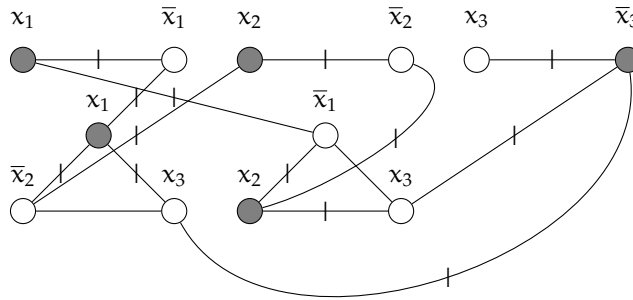
Dove  $m$  = numero di triangoli,  $n$  = numero di letterali. Quanti archi al massimo posso avere bicolorati?  $k = n + 3m + 2m = n + 5m$ .

$\phi$  è soddisfacibile  $\Rightarrow G$  ha un cut di taglia  $k = n + 5m$ .

$\phi$  è soddisfacibile  $\Leftrightarrow \exists a_1, \dots, a_n$  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  $l = F$ .

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



Esiste un cut di  $G$  di taglia  $n + 5m \Rightarrow$  esiste un assegnamento  $a_1, \dots, a_n$  t.c.  $\phi(a_1, \dots, a_n)$  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  $\phi$  K-CNF,  $t \in \mathbb{N}$
- Output: yes  $\Leftrightarrow$  esiste un assegnamento che soddisfa almeno  $t$  clausole

esempio:

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

Per  $k \geq 3$  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:

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

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

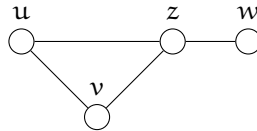
$$\phi(x_{v_1}, \dots, x_{v_n}) \text{ dove } \{v_1, \dots, v_n\} = V$$

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

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

- Dato  $G$  otteniamo

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



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

**Osservazione 12.2.1.** Per un qualsiasi assegnamento, in  $\phi$  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.  $\overset{\text{Max-cut}}{\text{yes}} \Leftrightarrow \overset{\text{Max-k-SAT}}{\text{yes}}$ :

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

$$\Rightarrow \forall (u, v) \in E \text{ se } col(u) \neq col(v) \equiv (u, v) \in cutC \quad (1)$$

$$\Rightarrow (x_v \vee x_u) \text{ è soddisfatta e } (\bar{x}_v \vee \bar{x}_u) \text{ è soddisfatta.} \quad (2)$$

$$\Rightarrow \forall (u, v) \in E \text{ se } (u, v) \notin cutC \quad (3)$$

$$\Rightarrow (x_v \vee x_u) \text{ è soddisfatta oppure } (\bar{x}_v \vee \bar{x}_u) \text{ è soddisfatta.} \quad (4)$$

$$\Rightarrow 2|C| + |E| - |C| = |E| + |C| \text{ sono soddisfatte.} \quad (5)$$

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

2.2.  $yes \Leftarrow yes$ : Assumiamo che in  $\phi$   $|E| + k$  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 bicolazione 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:  $(S, C)$ ,  $C = \{C_1, C_2, \dots, C_k\}$ , con  $C_i \subseteq S \quad i = 1 \dots k$
- Output:  $yes \Leftrightarrow$  possiamo colorare gli elementi di  $S$  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  $yes$ :

$$S = \{1_r, 2_r, 3_b, 4_b, 5_b\} \quad C = \{\{1_r, 3_b, 5_b\}, \{2_r, 4_b\}, \{1_r, 5_b\}\}$$

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

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

$$\begin{aligned} \text{Nae-3-SAT} &\leq_K \text{ Set-Splitting} \\ \phi \text{ 3-CNF} &\mapsto (S, C) \end{aligned}$$

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

$$C_i = \{l_1^{(i)}, l_2^{(i)}, l_3^{(i)}\} \quad \forall j = 1 \dots n \quad C_j = \{x_j, \bar{x}_j\}$$

Quindi se abbiamo la formula  $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$  diventa:

$$S = \{x_1, x_2, x_3, \bar{x}_1, x_2, \bar{x}_3\}$$

$$C = \{\{x_1, x_2, x_3\}, \{\bar{x}_1, x_2, \bar{x}_3\}, \{x_1, \bar{x}_1\}, \{x_2, \bar{x}_2\}, \{x_3, \bar{x}_3\}\}$$

Perciò ad un assegnamento di  $\phi$  che soddisfa la formula in termini NAE corrisponde una bicolazione non monocromatica di  $C_i$ .

2.  $yes \Leftarrow yes$ :

Ogni colorazione di  $S$  che bicolore i vari  $C$  implica un assegnamento che soddisfa  $\phi$  in termini NAE. Devo imporre che colori di letterali uguali opposti siano opposti.

## 12.4 Problema Set-Cover

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

esempio di istanza  $\text{yes}$ :

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

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

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

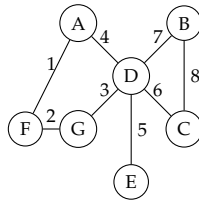


Figura 9: Per questo grafo esistono  $\tilde{k}$  vertici che toccano tutti gli archi

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

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

**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).

$$\mathbf{DP} = \{A \mid \exists B, C \in \mathbf{NP} \quad \mathcal{I}(A) = \mathcal{I}(B) = \mathcal{I}(C) \text{ t.c. } A(x) = \text{yes} \Leftrightarrow B(x) = \text{yes} \wedge C(x) = \text{no}\}$$

**Problema Clique-No-Clique**

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

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

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

Quindi  $B, C \in \mathbf{NP}$  e  $\text{Clique-no-Clique}(x) = \text{yes} \Leftrightarrow B(x) = \text{yes} \wedge C(x) = \text{no}$ .

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

*Dimostrazione.*

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

Perciò nella riduzione

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

Abbiamo dunque che:

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

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

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

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

□

## 12.6 Problema D-Ham-Path

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



Figura 10: Istanze del problema D-Ham-Path

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

*Dimostrazione.* Dobbiamo costruire il grafo a partire dalla formula  $\phi$  3-CNF:

- Per ogni variabile  $x$  di  $\phi$  definiamo un cammino che va in tutte e due le direzioni, con un numero di nodi pari al numero di letterali  $x, \bar{x}$  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  $x_{i+1}$   $i = 1 \dots n$ . 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:  $\phi(x_1, x_2, x_3) = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$

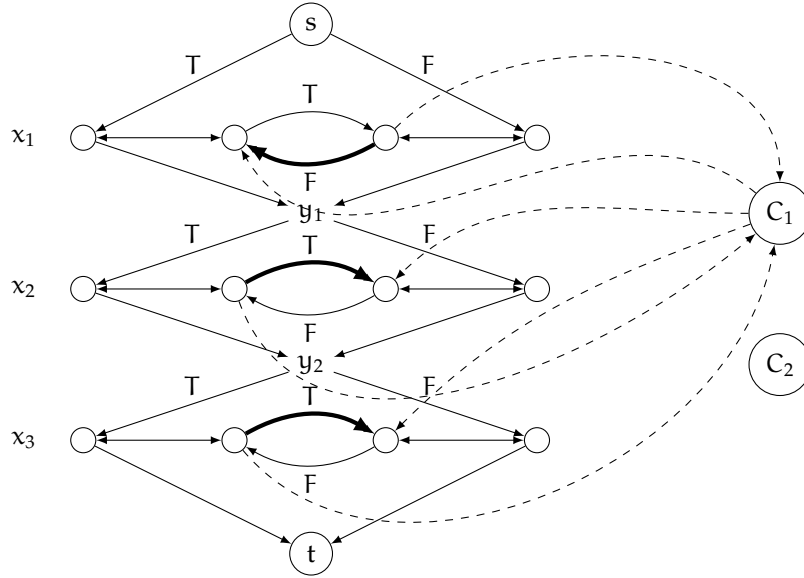


Figura 11: Rappresentazione per la clausola  $C_1$  del grafo con cammino Hamiltoniano

Quindi abbiamo che ci sono:

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

Se esiste un assegnamento  $a_1, \dots, a_n$  t.c.  $\phi(a_1, \dots, a_n) = T$  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:

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

**Problema Palindroma**  $\in \mathbf{L}$  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  $\in \mathbf{SPACE}(\log n)$ .

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

---

```

SAT( $\phi(x_1, \dots, x_n)$ )
  for i = 0 to  $2^n - 1$ 
    for j = 1 to n
       $x_j \leftarrow$  j-esimo bit di i
    if ( $\phi(x_1, \dots, x_n) = \text{T}$ )
      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  $\phi$ .  $n \leq |\phi|$ .

### 13.1 Classe PSPACE, L e NTIME

**Definizione 13.1.1** (Classe **PSPACE**).

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

Abbiamo visto che: **SAT**  $\in \mathbf{PSPACE}$

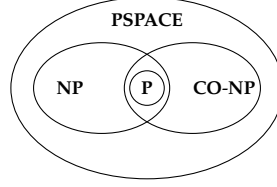
**Definizione 13.1.2** (Classe **L**).

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

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



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

Figura 12:  $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$ 

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

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

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

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

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

Facciamo vedere che  $\mathbb{A} \in \mathbf{SPACE}(f(n))$ : dobbiamo produrre l'algoritmo  $\Pi$  che risolve le istanze di  $\mathbb{A}$  e usa al massimo  $f(n)$  bit.

$\Pi$  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  $\Pi$ ?

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

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

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

Ora quindi sappiamo che

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

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

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

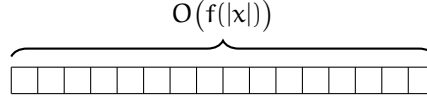


Figura 13: In questa memoria i bit cambiano a seconda delle istruzioni dell'algoritmo

In quanti modi questi bit possono cambiare?  $O(f(|x|)) = k \cdot f(n)$ . 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  $2^{k \cdot f(n)}$ .

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

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

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

$$\begin{aligned} \text{PSPACE} &= \bigcup_{k>0} \text{SPACE}(n^k) \subseteq \bigcup_{k>0} \text{TIME}(2^{n^k}) = \text{Exp} \\ &\Rightarrow \text{PSPACE} \subseteq \text{Exp} \\ &\Rightarrow \text{L} = \text{SPACE}(\log n) \subseteq \underbrace{\text{TIME}(2^{O(f(n))})}_{2^{k \log n} = n^k = \text{P}} \end{aligned}$$

Perciò abbiamo che:

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

## 13.2 Non determinismo e classe NSPACE

**Definizione 13.2.1** (Classe NSPACE).

$\text{NSPACE} = \{ \mathbb{A} \mid \text{esiste un algoritmo/programma } \Pi \text{ non deterministico}$

che risolve  $x \in \mathbb{J}(\mathbb{A})$  usando memoria di lavoro  $O(f(|x|))$   $\Pi(x) = \mathbb{A}(x)$  }

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 NSPACE?

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

*Dimostrazione.* Prendiamo un problema  $\mathbb{A} \in \text{NSPACE}(f(n))$ . Sappiamo che la memoria dell'algoritmo  $\Pi$  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  $G_x^\Pi$  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  $A = \text{yes} \Leftrightarrow$  esiste in  $G_x^\Pi$  un cammino dallo stato start allo stato finale.

Se conosco  $\Pi$  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  $2^{k \cdot f(n)}$ .

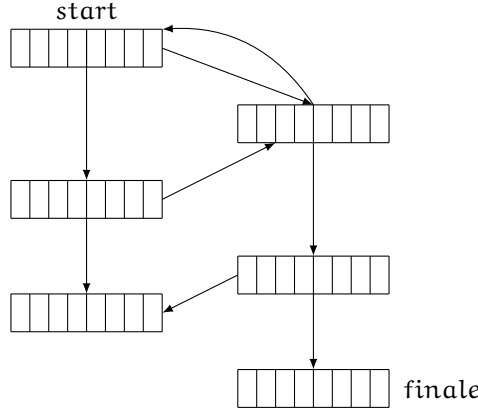


Figura 14: Esempio di grafo  $G_x^\Pi$

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

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

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

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

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

□

### 13.3 Problema Reachability in termini di spazio

- Input: grafo  $G$ , nodi  $s$  e  $t$ .
- Output:  $\text{yes} \Leftrightarrow$  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  $\in \mathbf{PSPACE}$  ( $\mathbf{P} \subseteq \mathbf{PSPACE}$ ).

**Reachability**  $\in \mathbf{SPACE}((\log n)^2)$  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:

- $\rightarrow$  Se esiste un cammino  $s \rightsquigarrow t$  allora esiste un cammino che utilizza al più  $|V|$  vertici che ha lunghezza  $\leq n$  ( $n = |V|$ ).
- $\rightarrow$  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  $\leq \lceil n/2 \rceil$ .
  - Esiste un cammino  $u \rightsquigarrow t$  di lunghezza  $\leq \lceil n/2 \rceil$ .
- Se esiste un cammino  $s \rightsquigarrow t$  di lunghezza  $\leq 1$  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,  $\frac{\text{length}}{2}$ ) = yes and MiddleSearch(G, u, t,  $\frac{\text{length}}{2}$ ) = yes) :
        risposta ← yes
    return risposta

```

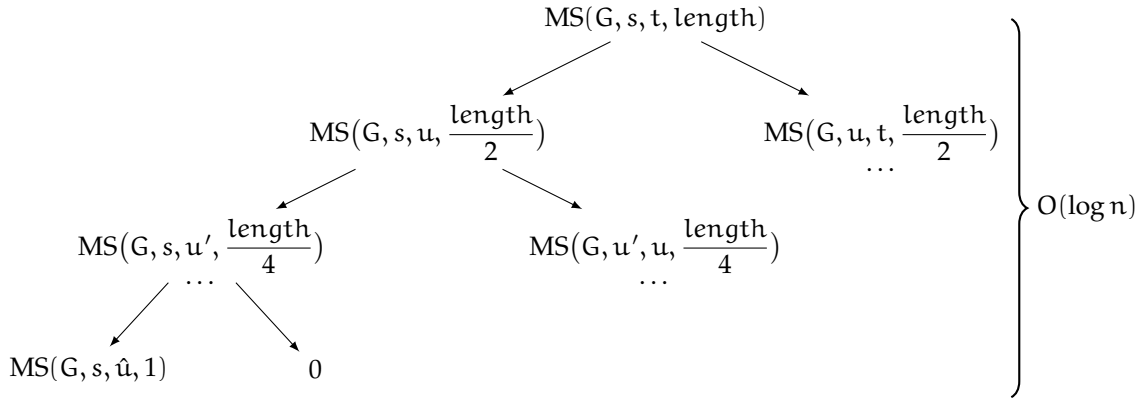
---

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  $\in \mathbf{SPACE}((\log n)^2)$ .

□

## 13.4 Classe NPSPACE

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

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

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

### 13.5 Teorema di Savitch

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

$$\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}\left((f(n))^2\right)$$

*Dimostrazione.* Sappiamo che  $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$ . 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  $\mathbb{A} \in \mathbf{NPSPACE}(f(n))$ :

$\Rightarrow$  Esiste un algoritmo  $\Pi$  non deterministico tale che per ogni istanza  $x \in \mathcal{I}(\mathbb{A})$  abbiamo che  $\Pi(x) = \mathbb{A}(x)$  e  $\Pi$  usa spazio  $O(f(n))$ .

$\Rightarrow$  Detto  $G_x^\Pi$  il grafo degli stati di  $\Pi$  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  $\Pi$  nello stato  $u$  ha  $v$  tra le possibili transizioni.

$\Pi(x) = \text{yes}$

$\Leftrightarrow$  In  $G_x^\Pi$  esiste un cammino dallo stato start allo stato finale.

$\Leftrightarrow \text{Reachability}(G_x^\Pi, \text{start}, \text{finale}, 2^{k \cdot f(n)}) = \text{yes}$ .

$\Leftrightarrow \text{Middlesearch}(G_x^\Pi, \text{start}, \text{finale}, 2^{k \cdot f(n)}) = \text{yes}$ . Con al più

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

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

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

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

Algorithm 9: SAT-Solver-ND  $\in \mathbf{NPSPACE}(n)$

---

```

SAT-Solver-ND( $\phi(x_1 \dots x_n)$ )
  for  $i=1$  to  $n$ :
     $a_i \leftarrow \text{T}$ 
    GotToBoth 4, 5
     $a_i \leftarrow \text{F}$ 
  endfor
  if  $\phi(a_1 \dots a_n) = \text{T}$ 
    return yes
  return no

```

---

Algorithm 10: SAT-Verifier

---

```

SAT-Verifier( $\phi(C_1 \dots C_n), \underline{a}$ )
  risposta  $\leftarrow \text{yes}$ 
  for  $i=1$  to  $n$ :
    if  $l_1^i = \text{F in } \underline{a} \wedge l_2^i = \text{F in } \underline{a} \wedge l_3^i = \text{F in } \underline{a}$ 
      return no
  return yes

```

---

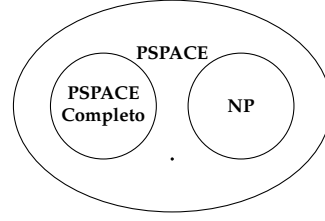
$\mathbf{PSPACE} = \mathbf{NPSPACE} = \{\mathbb{A} \mid \exists V(.,.) \text{ deterministico tale che } \forall x \in \mathcal{I}(\mathbb{A}) \mathbb{A}(x) = \text{yes sse } V(x, w) = \text{yes e } V \text{ usa al più } O(|x|^k) \text{ di memoria di lavoro (escludendo } x \text{ e } w), V \text{ accede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

### 13.6 Classe di problemi PSPACE-completi

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

- $\mathbb{A} \in \text{PSPACE}$
- $\forall \mathbb{B} \in \text{PSPACE} \quad \mathbb{B} \leq_K \mathbb{A}$  (hardness)

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



### 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  $\phi$  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  $Q\text{-SAT} \leq \text{Geography}$ ).

### 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  $Q\text{-SAT} \leq A\text{-HamPath}$ ).

## 14 Approssimazione

### 14.1 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

$$\max_{1 \leq k \leq m} \sum_{i \in M_k} j_i \text{ è minimo}$$

## 14.2 PTAS e FPTAS

### 14.3 Problema di Traveling Salesman

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

### 14.4 Problema Knapsack

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

### 14.5 Classe APX

**Definizione 14.5.1** (Classe APX).

$$\text{APX}(r(n)) = \left\{ \mathbb{A} \text{ di ottimizzazione} \mid \begin{array}{l} \text{esiste un algoritmo polinomiale per } \mathbb{A} \\ \text{che è } r(n)\text{-approssimabile} \end{array} \right\}$$

### 14.6 Tecnica del GAP

**Teorema 14.6.1.** 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  $\geq a$  o  $\leq b$ .

### 14.7 Problema Max-k-xor-SAT

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

### 14.8 Inapprossimabilità