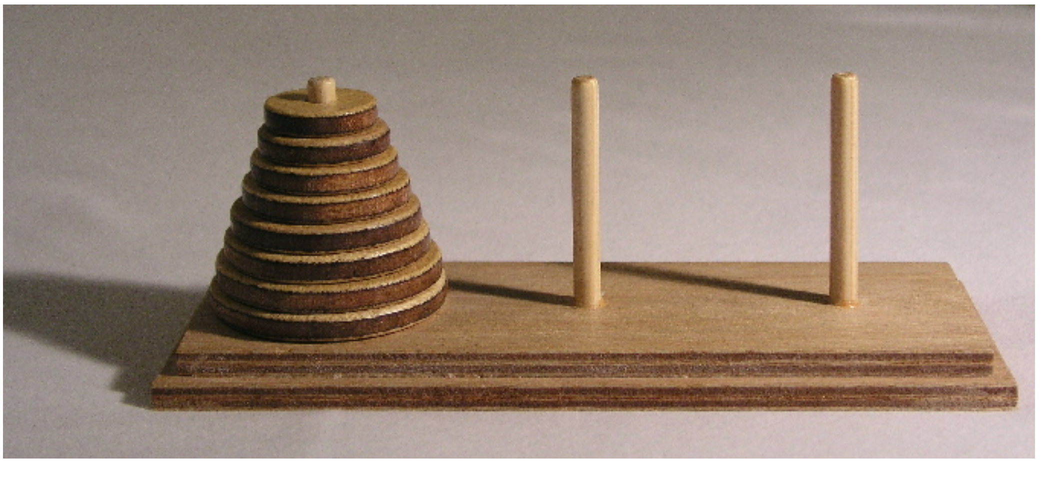
Problemi computazionali

Intrattabilità e classi computazionali

Decidibilità e Trattabilità

Problemi decidibili possono richiedere tempi di risoluzione elevati: **Torri di Hanoi**



Decidibilità e Trattabilità

Problemi decidibili possono richiedere tempi di risoluzione elevati: **Torri di Hanoi**

- 3 pioli
- n = 64 dischi sul primo piolo (vuoti gli altri due), tutti di dimensione diversa
- Disco grande non può stare su disco più piccolo
- Ogni mossa sposta un disco
- Obiettivo: spostarli tutti dal primo all'ultimo piolo

Soluzione ricorsiva

```
TorriHanoi( n, primo, secondo, terzo ):
  IF (n = 1) {
    PRINT primo ⟼ terzo;
  } ELSE {
    TorriHanoi( n - 1, primo, terzo, secondo );
    PRINT primo ⟼ terzo;
    TorriHanoi( n - 1, secondo, primo, terzo );
  }
```

Numero di mosse: 2^n-1

```
TorriHanoi( n, primo, secondo, terzo ):
  IF (n = 1) {
    PRINT primo ⟼ terzo;
  } ELSE {
    TorriHanoi( n - 1, primo, terzo, secondo );
    PRINT primo ⟼ terzo;
    TorriHanoi( n - 1, secondo, primo, terzo );
  }
```

- Caso base: n = 1, 2^1-1 = 1
- Passo induttivo: (2^{n-1}-1)+1+(2^{n-1}-1) = 2^n-1
- n = 64 ➔ 2^{64}-1 = 18 446 744 073 709 551 615
- 1 mossa/sec ➔ circa 585 miliardi di anni!
- NOTA: Tempo esponenziale, ed è stato dimostrato che non si può risolvere in meno mosse

Tempo esponenziale 2^n-1 (1 operazione/sec)

n	5	10	15	20	25	30	35	40	45
tempo	31s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a	1115689 a

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora **solo** di un fattore **additivo** $\log_2$ X

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

Nell’esempio: dischi gestibili nel tempo necessario a gestire 64 dischi

Tempo polinomiale

Torri di Hanoi generalizzate con $k > 3$ pioli

- Pioli numerati da 0 a k-1
- Obiettivo: spostare i dischi dal piolo 0 al k-1
- Hp. semplificativa: n è multiplo di k-2

```
TorriHanoiGen( n, k ):
  FOR (i = 1; i <= k-2; i = i+1)
    TorriHanoi(n/(k-2), 0, k-1, i);
  FOR (i = k-2; i >= 1; i = i-1)
    TorriHanoi(n/(k-2), i, 0, k-1);
```

Torri di Hanoi generalizzate

- Il codice richiede $2 \times (k-2) \times (2^{n/(k-2)}-1)$ mosse
- Al più n^2 mosse, fissando $k = n/\log n$ e $n \geq 5$
- $n = 64$ ➔ al più $64^2 = 4096$ mosse


```
TorriHanoiGen( n, k ):
  FOR (i = 1; i <= k-2; i = i+1)
    TorriHanoi(n/(k-2), 0, k-1, i);
  FOR (i = k-2; i >= 1; i = i-1)
    TorriHanoi(n/(k-2), i, 0, k-1);
```

Tempo polinomiale n^2
(1 operazione/sec)

n	5	10	15	20	25	30	35	40	45
tempo	25 s	100 s	225 s	7 m	11 m	15 m	21 m	27 m	34 m

Aumentare di un fattore **moltiplicativo** X (ossia X operazioni/sec) migliora di un fattore **moltiplicativo** $\sqrt{X}$

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	202	640	2023	6400	20238	64000	2023857

SUDOKU

- Tabella 9 X 9 contenente numeri in [1..9]
- Divisa in 3 X 3 sottotabelle (di taglia 3 X 3)
- Alcune celle contengono numeri, altre vuote
- Riempire le celle vuote in modo che

1. ogni riga contenga una permutazione di 1,2,...,9
2. ogni colonna contenga una perm. di 1,2,...,9
3. ogni sottotabella contenga una perm. di 1,2,...,9

SUDOKU

3	9							8
	7	1			3			
		8		4	9		6	
1			2	7				9
6								3
5				3	6			4
	4		1	5		9		
			9			8	2	
9							4	7

3	9	6	5	1	2	4	7	8
4	7	1	6	8	3	5	9	2
2	5	8	7	4	9	3	6	1
1	3	4	2	7	5	6	8	9
6	8	7	4	9	1	2	5	3
5	2	9	8	3	6	7	1	4
8	4	2	1	5	7	9	3	6
7	1	3	9	6	4	8	2	5
9	6	5	3	2	8	1	4	7

- Soluzione ottenibile in questo caso attraverso implicazioni logiche
- Es. Sottotabella in alto a destra: il 3 può stare solo qui

		8
	6	

BACKTRACK con scelte non uniche

			6		2		9	
								6
			7	3	1	5		8
4		9	3			6		5
		3				1		
5		8			7	9		2
		1	5	2	3			
7								
	6	2	9		4			

			6		2		9	
			8					6
			7	3	1	5		8
4	2	9	3	1	8	6	7	5
6	7	3	2			1	8	4
5	1	8	4	6	7	9	3	2
		1	5	2	3			
7			1	8	6			
	6	2	9	7	4			

Partendo dalla configurazione a sinistra, giungiamo nella configurazione a destra che ammette diverse scelte per ogni casella

Backtrack: *algoritmo che esplora tali scelte, annullando gli effetti nel caso che non conducano a soluzione*

Backtrack per SUDOKU

- Esamina le m caselle vuote nell'ordine indicato da **`PrimaVuota`**, **`SuccVuota`**, **`UltimaVuota`**

```
Sudoku( casella ):                                    ⟨pre: casella vuota⟩
  elenco = insieme delle cifre ammissibili per casella;
  FOR (i = 0; i < |elenco|; i = i+1) {
    Assegna( casella, elenco[i] );
    IF (!UltimaVuota(casella) && !Sudoku(SuccVuota(casella))) {
      Svuota( casella );
    } ELSE {
      RETURN TRUE;
    }
  }
  RETURN FALSE;
```

Invocata con **`casella = PrimaVuota()`**

- Nel caso pessimo, esplora circa 9^m scelte ($m \leq 9^2$)
- In generale, tabella n x n: circa $n^m \leq n^{n^2} = 2^{n^2 \log n}$

SUDOKU: quale complessità?

- L'algoritmo di backtrack è quindi esponenziale ...ma il SUDOKU n x n è trattabile o meno?
- Dipende dall'esistenza di un algoritmo polinomiale: ad oggi, tale algoritmo è ignoto
- SUDOKU sembra avere una natura computazionale diversa da quella delle Torri di Hanoi:
  - *Torri di Hanoi: esiste dimostrazione formale che non si può risolvere in meno di 2^n-1 mosse*
  - *SUDOKU: è possibile verificare la correttezza di una data soluzione in tempo polinomiale* (non possibile con Torri di Hanoi).

SUDOKU: verifica polinomiale della correttezza di una soluzione

```
VerificaSudoku( sequenza ):        ⟨pre: sequenza di m cifre, con 0 < m ⩽ n²⟩
  casella = PrimaVuota( );
  FOR (i = 0; i < m; i = i+1) {
    cifra = sequenza[i];
    IF (cifra appare in casella.riga) RETURN FALSE;
    IF (cifra appare in casella.colonna) RETURN FALSE;
    IF (cifra appare in casella.sotto_tabella) RETURN FALSE;
    Assegna( casella, cifra );
    casella = SuccVuota(casella);
  }
  RETURN TRUE;
```

- Richiede circa m x n = $O(n^3)$ passi

Problemi

Un *problema* è una relazione

$$\Pi \subseteq I \times S$$

I: insieme delle istanze in ingresso

S: insieme delle soluzioni

Tipologie di problemi

- Problemi di decisione
  - Richiedono una risposta binaria ($S = \{0,1\}$)
  - Un grafo è connesso? Un numero è primo?
  - Istanze positive $(x,1) \in \Pi$ o negative $(x,0) \in \Pi$
- Problemi di ricerca
  - Richiedono di restituire una soluzione s tale che $(x,s) \in \Pi$
  - Trovare un cammino tra due vertici, trovare il mediano di un insieme di elementi

Tipologie di problemi

- Problemi di ottimizzazione
  - Data un'istanza x, si vuole trovare la migliore soluzione s tra tutte le possibili s per cui $(x,s) \in \Pi$
  - Ricerca della massima sottosequenza comune, ricerca del cammino minimo fra due nodi di un grafo

Problemi decisionali

- La teoria della complessità computazionale è definita principalmente in termini di problemi di decisione
  - Essendo la risposta binaria, non ci si deve preoccupare del tempo richiesto per restituire la soluzione e tutto il tempo è speso esclusivamente per il calcolo

Problemi decisionali

- Molti problemi di interesse pratico sono però di ottimizzazione
- È però possibile esprimere un problema di ottimizzazione in forma decisionale
  - Massima sottosequenza comune: date due stringhe, esiste una SC di lunghezza almeno k?
  - Problema non più difficile di quello di ottimizzazione
    - Se sappiamo trovare la soluzione ottima, la confrontiamo con k!

Problemi decisionali

- Il problema di ottimizzazione è quindi almeno tanto difficile quanto il corrispondente problema decisionale
  - Caratterizzare la complessità di quest'ultimo permette quindi di dare almeno una limitazione inferiore alla complessità del primo

Classi di complessità

- Dato un problema Π ed un algoritmo A, diciamo che A risolve Π se

$$\text{A restituisce 1 su } x \Leftrightarrow (x,1) \in \Pi$$

- A risolve Π in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono rispettivamente $t(n)$ e $s(n)$

Classi Time e Space

- Data una qualunque funzione f(n), chiamiamo

  Time(f(n)) e Space(f(n))

  gli insiemi dei problemi decisionali che possono essere risolti rispettivamente in tempo e spazio O(f(n))

Classi Time e Space

- Il problema di verificare se un certo elemento è presente in un dizionario ordinato realizzato tramite array e contenente *n* elementi appartiene alla classi

  Time(????) e Space(????)

Classi Time e Space

- Il problema di verificare se un certo elemento è presente in un dizionario ordinato realizzato tramite array e contenente *n* elementi appartiene alla classi

  Time(log n) e Space(n)

- Estendiamo ora questa definizione….

Classe P

- *Algoritmo polinomiale (spazio o tempo): esistono due costanti c, $n_0 > 0$ t.c. il numero di passi elementari (celle di memoria utilizzate) è al più n^c per ogni input di dimensione n e per ogni $n > n_0$*

- La classe P è la classe dei problemi risolvibili in tempo polinomiale nella dimensione *n* dell'istanza di ingresso:

$$P = \bigcup_{c=0}^{\infty} \text{Time}(n^c)$$

Classe PSpace

- La classe PSpace è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathrm{PSpace} = \bigcup_{c=0}^{\infty} \mathrm{Space}(n^{c})$$

Classe ExpTime

- La classe ExpTime è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso:

$$\mathrm{ExpTime} = \bigcup_{c=0}^{\infty} \mathrm{Time}(2^{n^{c}})$$

Relazioni

- Un algoritmo polinomiale può avere accesso al più a quante diverse locazioni di memoria (ordine di grandezza)?
  - Polinomiale!
  - Quindi P $\subseteq$ PSpace
- Inoltre risulta
  - PSpace $\subseteq$ ExpTime
  - *Informalmente: assumendo che le locazioni di memoria siano binarie, n^c diverse locazioni di memoria possono trovarsi in al più 2^{n^c} stati diversi*

Relazioni

- Non è noto (ad oggi) se le inclusioni siano proprie
- L'unico risultato di separazione dimostrato finora riguarda P e ExpTime
  - Esiste un problema che può essere risolto in tempo esponenziale, ma per cui tempo polinomiale non è sufficiente

Esempi

- Tutti i problemi visti finora sono in P
- Il problema delle Torri di Hanoi in quale classe di complessità si trova?
  – ExpTime
- Altri esempi interessanti:
  – *Generazione di sequenze*
  – *Generazione di permutazioni*

Genera le 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sottoinsiemi di un insieme di n elementi
- A[i]=1 sse l'i-esimo elemento è selezionato

Es. n = 3	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7

- Struttura ricorsiva della generazione

$00\underline{0}, 10\underline{0}, 01\underline{0}, 11\underline{0}$, 001, 101, 011, 111

$0\underline{0}0, 1\underline{0}0$, 010, 110

$\underline{0}00$

fissa il bit a 1 e ricorri come per il bit a 0

Genera le 2^n sequenze binarie

- Equivale a generare ricorsivamente tutti i sotto-insiemi di un insieme di n elementi
- A[i]=1 sse l'i-esimo elemento è selezionato

```
GeneraBinarie( A, b ):
  IF (b == 0) {
    Elabora( A );
  } ELSE {
    A[b-1] = 0;
    GeneraBinarie( A, b-1 );
    A[b-1] = 1;
    GeneraBinarie( A, b-1 );
  }
```

invocato con b=n

Genera le n! permutazioni di A

a b c d	a b d c	a d c b	d b c a
b a c d	b a d c	d a c b	b d c a
a c b d	a d b c	a c d b	d c b a
c a b d	d a b c	c a d b	c d b a
c b a d	d b a c	c d a b	c b d a
b c a d	b d a c	d c a b	b c d a
i = 3	i = 2	i = 1	i = 0

Per i = n-1, ..., 1, 0:

- scambia A[i] con A[n-1];
- i primi n-1 elementi di A sono ricorsivamente permutati **nella stessa maniera** (indipenden. da i);
- Scambia l'ultimo elemento A[n-1] con A[i] (per rimetterli a posto).

Genera le n! permutazioni di A

```
GeneraPermutazioni( A, p ):   ⟨pre: i primi p
  IF (p == 0) {
    Elabora( A );
  } ELSE {
    FOR (i = p-1; i >= 0; i = i-1) {
      Scambia( i, p-1 );
      GeneraPermutazioni( A, p-1 );
      Scambia( i, p-1 );
    }
  }
```

Invocata con p=n

Esempi

- Altro esempio interessante, utilizzato ampiamente nella teoria della complessità:
  - *Soddisfacibilità di formule booleane*

Definizioni

- Insieme V di variabili Booleane
  - Letterale: variabile o sua negazione
  - Clausola: disgiunzione (OR) di letterali
- Un’espressione Booleana su V si dice in forma normale congiuntiva (FNC) se è espressa come congiunzione di clausole (AND di OR)

Definizioni

- Una formula Booleana quantificata è una espressione in FNC preceduta da una sequenza di quantificatori universali ed esistenziali ($\forall$, $\exists$) che legano ***tutte*** le variabili

Esempi

$$V = \{x, y, z, w\}$$

$$FNC : (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$$

$$QFNC : \exists x\ \forall y\ \exists z\ \forall w : (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$$

SAT

- Data una espressione in *forma normale congiuntiva*, il problema della soddisfacibilità (SAT) richiede di verificare se esiste una assegnazione di valori di verità alle variabili che rende l’espressione vera
- Il problema delle formule Booleane quantificate richiede invece di verificare se una certa formula Booleana quantificata è vera

Esempi

- La formula

$$(x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$$

è soddisfatta dall'assegnazione

$$x = 1 \quad y = 1 \quad z = 0 \quad w = 1$$

Esempi

- La formula quantificata

$$\exists x \ \forall y \ \exists z \ \forall w : (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$$

non è vera, in quanto l'ultima clausola non può essere verificata da

$$y = 0$$

e la variabile *y* è preceduta da ∀

Algoritmo per formule quantificate

- Algoritmo ricorsivo
- Assegna valore 0 alla prima variabile e risolvi il resto. Poni in v_0 il risultato
- Assegna valore 1 alla prima variabile e risolvi il resto. Poni in v_1 il risultato
- Se il quantificatore della prima variabile è ∃, allora restituisci $v_0 \vee v_1$
- Se il quantificatore della prima variabile è ∀, allora restituisci $v_0 \wedge v_1$

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre $v_0 \vee v_1$
- Se n è il numero delle variabili, quante diverse configurazioni esaminano gli algoritmi?
  - 2^n
  - Entrambi i problemi sono in ExpTime
  - Sono in PSpace?

Sì → PSpace ⊆ ExpTime

Algoritmo per SAT

- Simile al precedente
  - Si immaginano tutte le variabili *quantificate esistenzialmente* e si restituisce sempre $v_0 \vee v_1$
- Se n è il numero delle variabili, quante diverse algoritr

Algoritmi polinomiali non noti!

  - 2^n
  - Entrambi i problemi sono in ExpTime
  - Sono in PSpace?

Sì → PSpace $\subseteq$ ExpTime

Certificato

- In un problema decisionale siamo interessati a verificare se una istanza del problema soddisfa una certa proprietà
- Spesso, in caso di risposta affermativa, oltre alla semplice risposta (binaria) si richiede di fornire anche un oggetto y, dipendente dall'istanza x e dal problema, che possa *certificare* il fatto che x soddisfa la proprietà, giustificando quindi la risposta
  - y viene detto *certificato*

Certificato

- Certificato per SAT?
  - Un'assegnazione di verità alle variabili che renda vera l'espressione
- Certificato per il problema delle formule Booleane quantificate?
  - Un pò più complicato, vero?

Certificato

- Certificato per il problema delle formule Booleane quantificate?
  - Non è sufficiente esibire un'assegnazione di valori di verità alle variabili
  - Nel caso peggiore (quantificatori $\forall$) quante ne servono?
    - Numero esponenziale
  - In questo caso è difficile esprimere anche un certificato!

La classe NP

- Queste osservazioni suggeriscono di utilizzare il costo della verifica di una soluzione per caratterizzare la complessità del problema stesso

- *Informalmente: NP è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale*

Esempi

- Verificare SAT
  - Come?
- Non si può fare altrettanto con il problema delle formule Booleane quantificate
  - Attualmente non noto se tale problema sia in NP
    - Si congettura di no!
- Ma cosa vuol dire NP?
  - P sta per polinomiale, ma N?
    - Non vuol dire NON…

Non determinismo

- Negli algoritmi visti finora ogni passo è determinato univocamente dallo stato della computazione
  - Algoritmi deterministici
- Un algoritmo non deterministico, oltre alle normali istruzioni, può eseguire istruzioni del tipo

  Indovina $z \in \{0,1\}$
- Il valore di *z* influenza la prosecuzione della computazione, indirizzandola nella "giusta" direzione.

Esempio

- Un algoritmo non deterministico per SAT come potrebbe funzionare?
  - Indovina i valori da assegnare alle variabili e poi verifica (deterministicamente) la bontà dell'assegnazione fatta
  - Computazione descritta da un albero, dove le ramificazioni corrispondono alle scelte non deterministiche (istruzioni *indovina*)
    - Quella determistica è descritta da una catena
  - Quindi per SAT, se la FNC è soddisfacibile, ***esiste*** almeno un cammino che porta a una foglia con valore 1

Esempio

- E un algoritmo non deterministico per il problema delle formule Booleane quantificate?
  - Per le variabili esistenziali è facile: procediamo come con SAT
    - *Indoviniamo* il valore di queste variabili
  - Problema per le variabili universali
    - La formula deve essere vera per TUTTI i possibili assegnamenti a queste variabili
    - Quindi il non determinismo non aiuta affatto

La classe NP

- Data una qualunque funzione *f(n)*, chiamiamo NTime(*f(n)*) l'insiemi dei problemi decisionali che possono essere risolti da un algoritmo *non deterministico* in tempo O*(f(n))*
- La classe NP è la classe dei problemi risolvibili in tempo *polinomiale non deterministico* nella dimensione n dell'istanza di ingresso:

$$\mathrm{NP} = \bigcup_{c=0}^{\infty} \mathrm{NTime}(n^c)$$

La classe NP: definizioni

- *Informalmente: NP è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale (deterministico)*
- La classe NP è la classe dei problemi risolvibili in tempo *polinomiale non deterministico* nella dimensione n dell'istanza di ingresso

La classe NP: definizioni

Relazione fra le due definizioni

Ogni algoritmo non deterministico può essere articolato in due fasi:

1. Non deterministica di costruzione del certificato
2. Deterministica di verifica del certificato

Gerarchia delle classi

- P è incluso in NP oppure no?
  - Ovviamente sì!
    - Un algoritmo deterministico è un caso particolare di un algoritmo non deterministico, in cui l'istruzione *indovina* non è mai usata
    - Visione diversa: ogni problema in P ammette un certificato verificabile in tempo polinomiale….come mai?
      - Eseguo l'algoritmo che risolve il problema per costruire il certificato!

Gerarchia delle classi

- NP è incluso in PSpace oppure no?
  - Ovviamente sì!
    - La fase deterministica di verifica può essere condotta in tempo polinomiale solo se il certificato ha dimensione polinomiale!

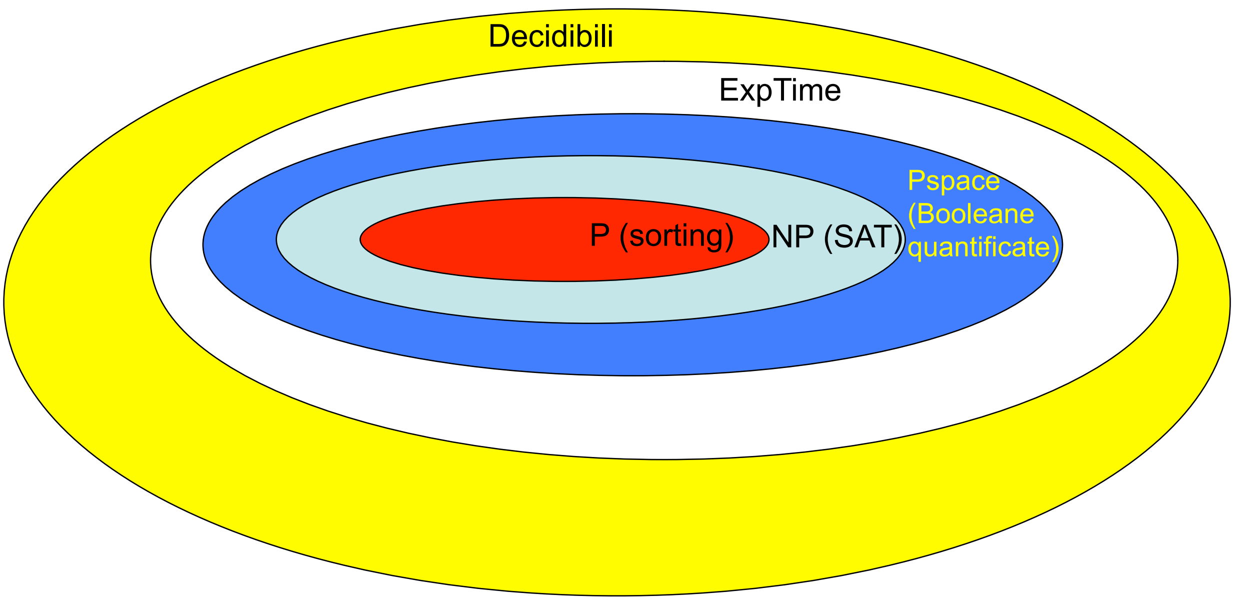
Gerarchia delle classi

- Quindi abbiamo

  $P \subseteq NP \subseteq PSpace \subseteq ExpTime$

- Si congettura inoltre che le inclusioni siano proprie
  - Nessuno è finora riuscito a dimostrarlo
  - I problemi che si ritiene appartengano a NP ma non a P si dicono *NP-completi*
  - Si ritiene che il problema delle formule Booleane quantificate (che appartiene a PSpace) non appartenga a NP

Gerarchia delle classi



Problemi NP-completi

- Caratterizzano i problemi più difficili all'interno della classe NP
  - Se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale, e P = NP
  - Quindi: o tutti i problemi NP-completi sono risolvibili deterministicamente in tempo polinomiale o nessuno di essi lo è

Riduzioni polinomiali

Dati due problemi decisionali

$$\Pi_1 \subseteq I_1 \times \{0,1\} \qquad \Pi_2 \subseteq I_2 \times \{0,1\}$$

diremo che Π_1 *si riduce in tempo polinomiale a* Π_2

$$\Pi_1 \leq_p \Pi_2$$

se esiste una *funzione* $f: I_1 \rightarrow I_2$ t.c.

- f è *calcolabile in tempo polinomiale*
- per ogni istanza x di Π_1 e ogni soluzione $s \in \{0,1\}$

$$(x, s) \in \Pi_1 \Leftrightarrow (f(x), s) \in \Pi_2$$

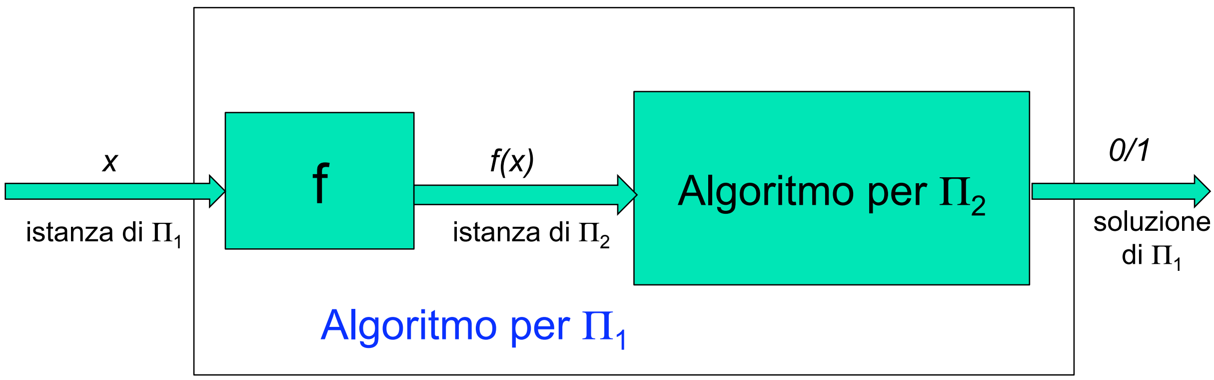
Riduzioni polinomiali

In altri termini, f ***trasforma*** un'istanza di Π_1 in un'istanza di Π_2 in modo tale che

- istanze positive di Π_1 risultino in istanze positive di Π_2
- istanze negative di Π_1 risultino in istanze negative di Π_2

Riduzioni polinomiali

Quindi, se esistesse un algoritmo per risolvere Π_2, potremmo utilizzarlo per risolvere Π_1:



Riduzioni polinomiali

$$\Pi_1 \leq_p \Pi_2 \quad \text{e} \quad \Pi_2 \in P \;\Rightarrow\; \Pi_1 \in P$$

Problemi NP ardui

Un problema decisionale Π si dice NP-arduo se

per ogni $\Pi' \in NP$, $\Pi' \leq_p \Pi$

Problemi NP completi

- Un problema decisionale Π si dice NP-completo se
  - $\Pi \in NP$
  - Π è NP-arduo
- I problemi NPC sono i più “difficili” in NP
- Se si scoprisse un algoritmo polinomiale per risolvere un problema NPC, allora P = NP.

Problemi NP completi

- Dimostrare che un problema è in NP può essere facile
  - Esibire un certificato polinomiale
- Non è altrettanto facile dimostrare che un problema Π è NP-arduo
  - Bisogna dimostrare che TUTTI i problemi in NP si riducono polinomialmente a Π!
  - In realtà la prima dimostrazione di NP-completezza aggira il problema

Teorema di Cook

SAT

- problema della soddisfacibilità di una espressione booleane in forma normale congiuntiva (FNC):
  - **FNC:** AND di **clausole**
    - **clausola:** OR di **letterali**
      - **letterali:** variabili booleane e loro negazioni

Teorema
SAT è NP completo

Teorema di Cook (idea)

- *Cook ha mostrato un algoritmo che, dati un qualunque problema Π ed una qualunque istanza x per Π, costruisce una espressione Booleana in forma normale congiuntiva che descrive il calcolo di un algoritmo per risolvere Π su x*
- *L'espressione è vera se e solo se l'algoritmo restituisce 1*

Dimostrazioni di NP-completezza

- Sfruttano la transitività delle riduzione polinomiale

Se $\Pi_1 \leq_p \Pi_2$ e $\Pi_2 \leq_p \Pi_3$, allora $\Pi_1 \leq_p \Pi_3$

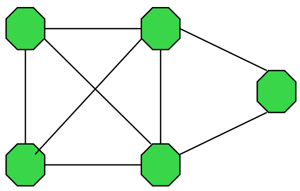
Problemi NP completi

- *Un problema decisionale Π è NP completo se*
  - $\Pi \in NP$
  - SAT $\leq_p \Pi$

  (o un qualsiasi altro problema NPC)

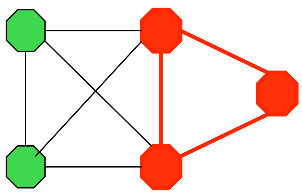
CLIQUE

- Dato un grafo G = (V,E) e un intero k > 0, stabilire se G contiene un sottografo completo di k nodi



CLIQUE

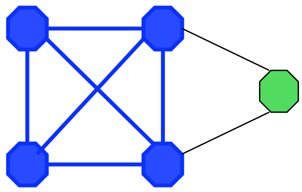
- Dato un grafo G = (V,E) e un intero k > 0, stabilire se G contiene un sottografo completo di k nodi



Clique di 3 nodi

CLIQUE

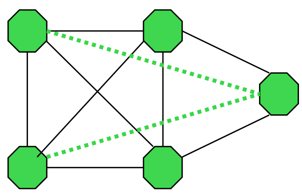
- Dato un grafo G = (V,E) e un intero k > 0, stabilire se G contiene un sottografo completo di k nodi



Clique di 4 nodi

CLIQUE

- Dato un grafo G = (V,E) e un intero k > 0, stabilire se G contiene un sottografo completo di k nodi



Non contiene
clique di 5 nodi

CLIQUE è NP completo

SAT $\leq_p$ CLIQUE

data una espressione booleana F in forma normale congiuntiva con k clausole

costruire in tempo polinomiale

un grafo G che contiene una **clique di k vertici se e solo se F è soddisfacibile**.

Riduzione: vertici

- Ad ogni letterale in ciascuna clausola di F corrisponde un vertice in G.

Esempio:

$F = (a \vee b) \wedge (!a \vee !b \vee c) \wedge !c$

$G = (V, E)$,

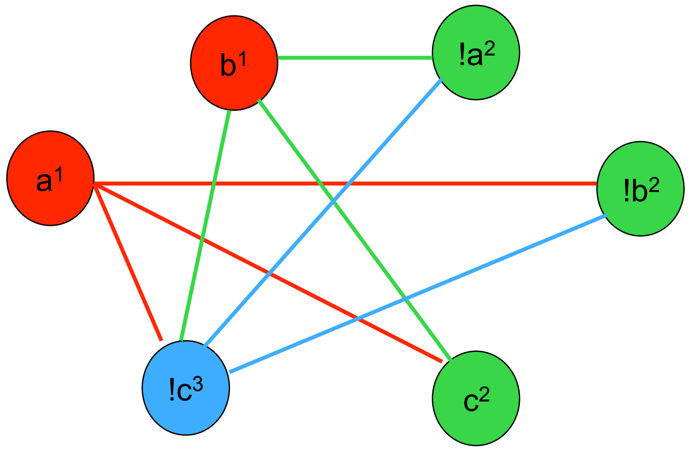
$V = \{ a^1, b^1, !a^2, !b^2, c^2, !c^3 \}$

Riduzione: archi

$$(x^i , y^j) \in E \Leftrightarrow i \neq j \text{ e } x \neq !y$$

Due letterali sono adiacenti in G se e solo se possono essere veri contemporaneamente.

F = (a v b) ∧ (!a v !b v c) ∧ !c



Clique in G

- *composta da k nodi*

  uno per ogni clausola di F

- *non può contenere due nodi della stessa clausola*

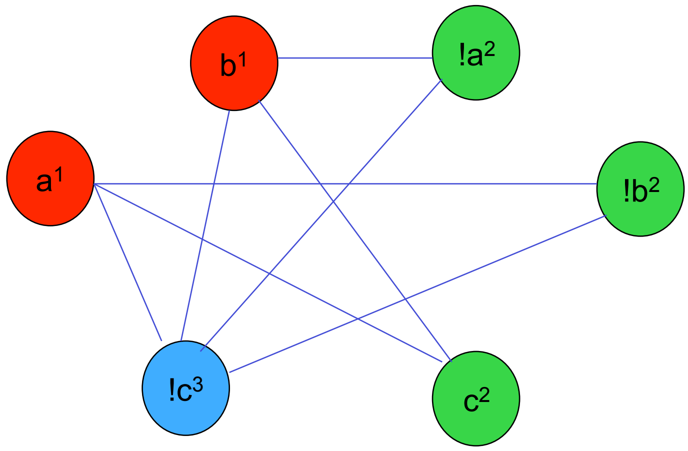
  perché non sono adiacenti.

Riduzione

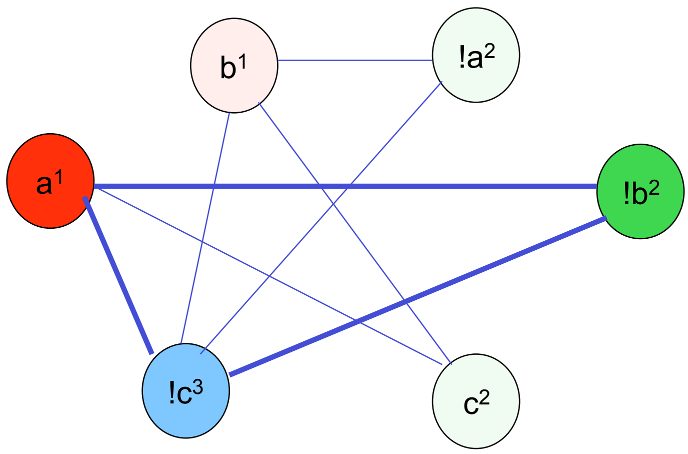
G contiene una clique ⇒ F è soddisfacibile

- si dà valore **1** (true) ai k letterali che corrispondono ai nodi della clique
- tutte la clausole corrispondenti diventano di valore **1** (true)
- F = **1** (true), soddisfacibile.

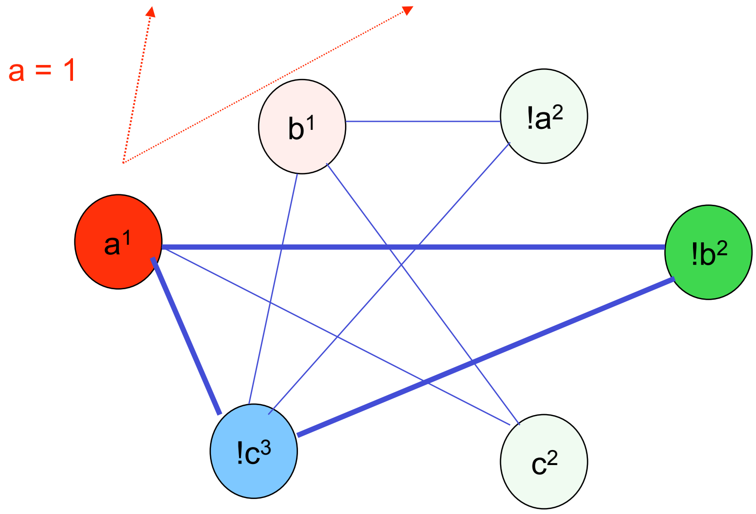
F = (a v b) ∧ (!a v !b v c) ∧ !c



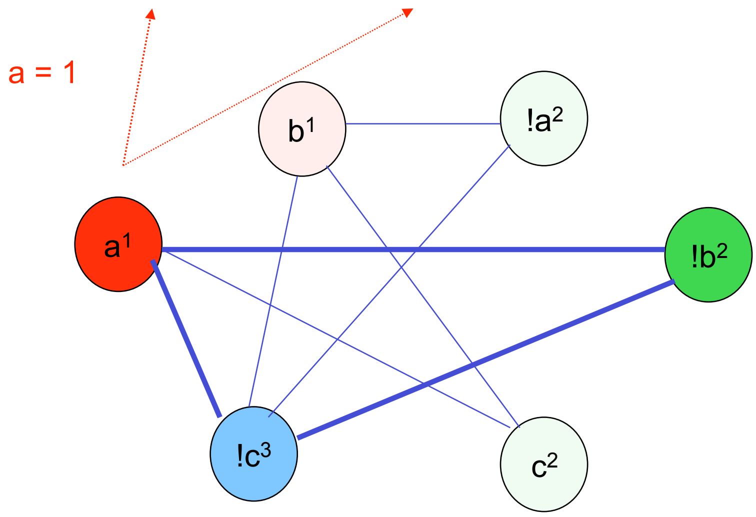
F = (a v b) ∧ (!a v !b v c) ∧ !c



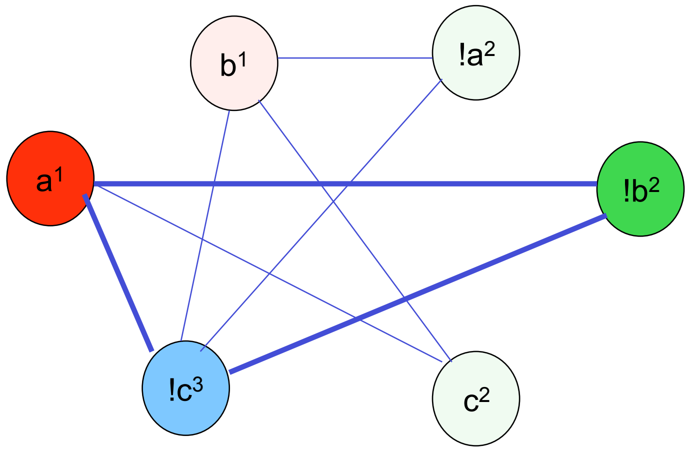
F = (a ∨ b) ∧ (!a ∨ !b ∨ c) ∧ !c



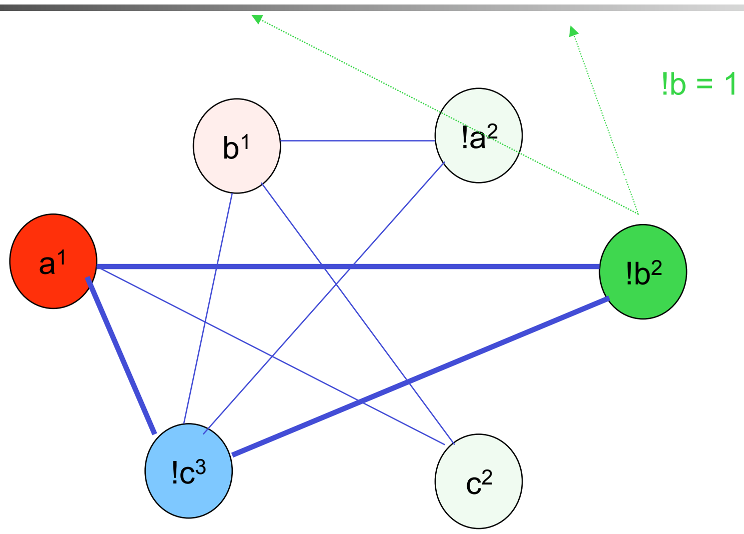
F = (1 ∨ b) ∧ (0 ∨ !b ∨ c) ∧ !c



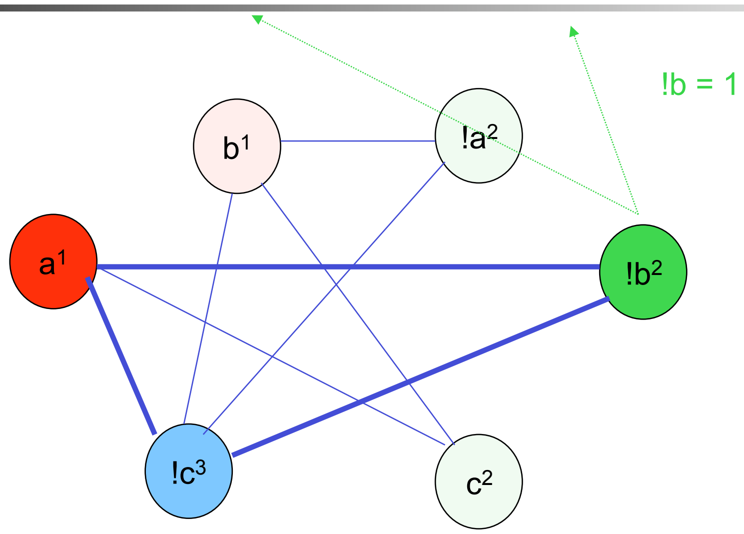
$F = (1 \lor b) \land (0 \lor !b \lor c) \land !c$



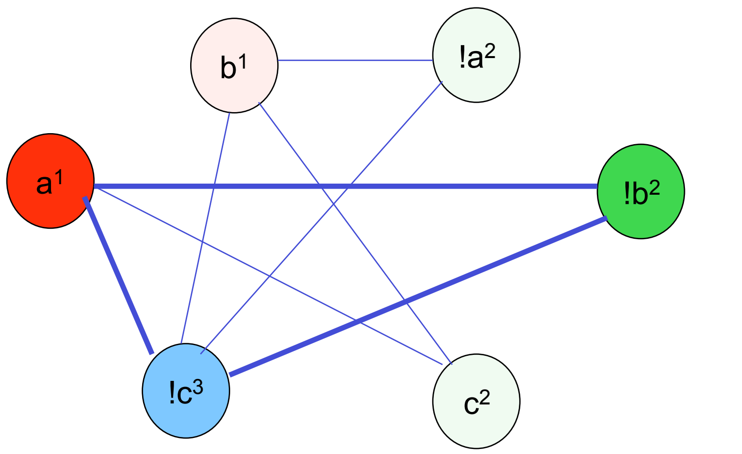
$F = (1 \lor \mathbf{b}) \land (0 \lor \mathbf{!b} \lor c) \land !c$



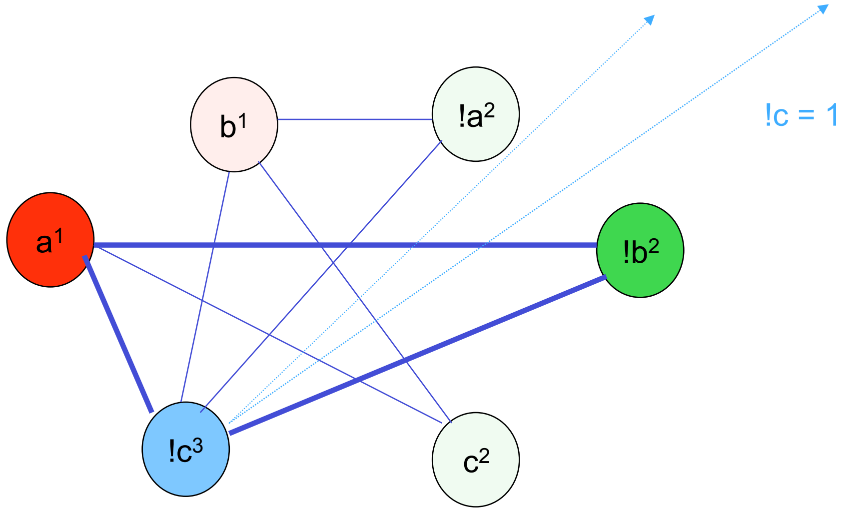
$F = (1 \vee 0) \wedge (0 \vee 1 \vee c) \wedge !c$



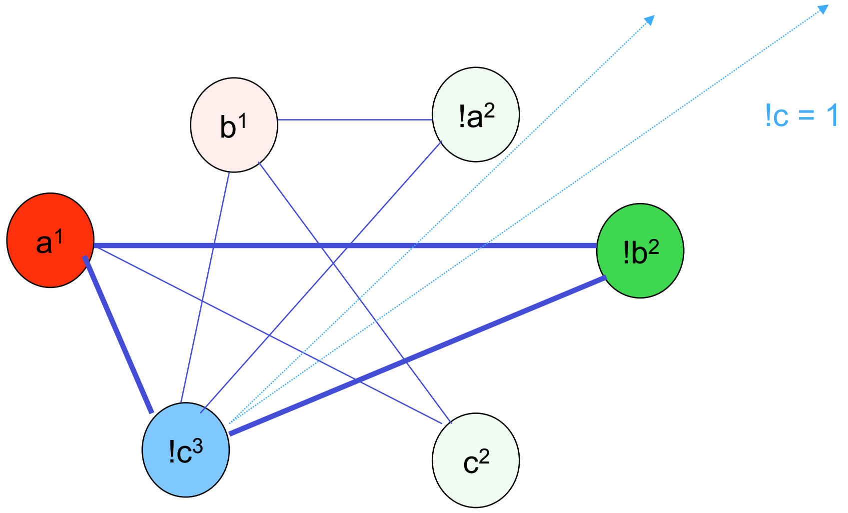
$F = (1 \vee 0) \wedge (0 \vee 1 \vee c) \wedge !c$



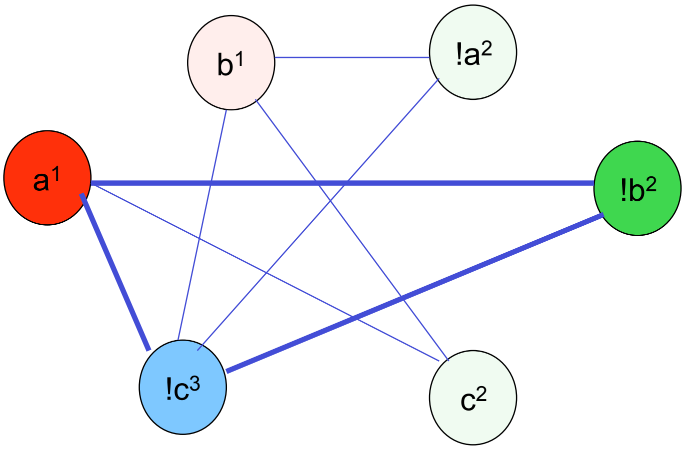
$F = (1 \vee 0) \wedge (0 \vee 1 \vee c) \wedge !c$



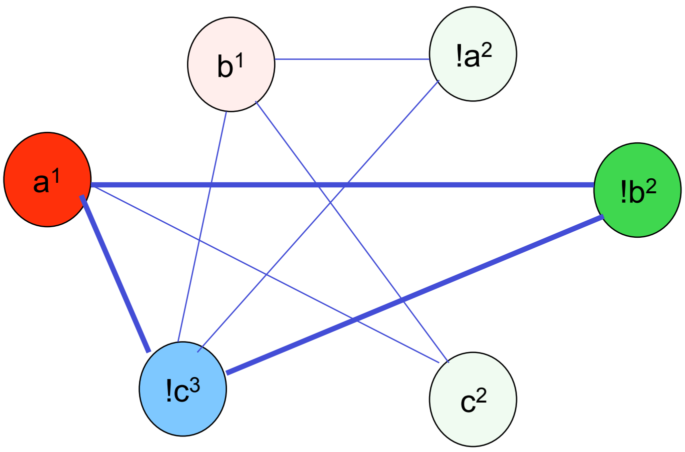
$F = (1 \vee 0) \wedge (0 \vee 1 \vee 0) \wedge 1$



$F = (1 \lor 0) \land (0 \lor 1 \lor 0) \land 1$



$F = (1) \land (1) \land 1 = 1$



Riduzione

F è soddisfacibile $\Rightarrow$ G contiene una clique

- esiste almeno un letterale vero per ogni clausola
- i corrispondenti vertici in G formano una clique.

Riduzione

- *La riduzione da* F *a* G = (V,E) *si esegue in tempo polinomiale:*
  - n = # variabili
  - k = # clausole
  - $|V| \leq n\ k$
  - l'esistenza di un arco si stabilisce in tempo costante
  - $|E| \leq O((n\ k)^2)$

Problemi NP equivalenti

- $SAT \leq_p CLIQUE$ $\Rightarrow$ CLIQUE è NP completo
- SAT è NP completo $\Rightarrow$ $CLIQUE \leq_p SAT$
- *SAT e CLIQUE sono **NP equivalenti***.
- *Tutti i problemi NP completi sono tra loro **NP equivalenti**.*

Altri famosi problemi NP-completi

- **Copertura di vertici**
  - Una copertura di vertici (*vertex cover*) di un grafo G=(V,E) è un insieme di vertici $C \subseteq V$ tale che per ogni $(u,v) \in E$, almeno uno tra u e v appartiene a C
  - Bisogna verificare se, dati G e un intero k, esiste una copertura di vertici di G di dimensione al più k
- **SUDOKU**

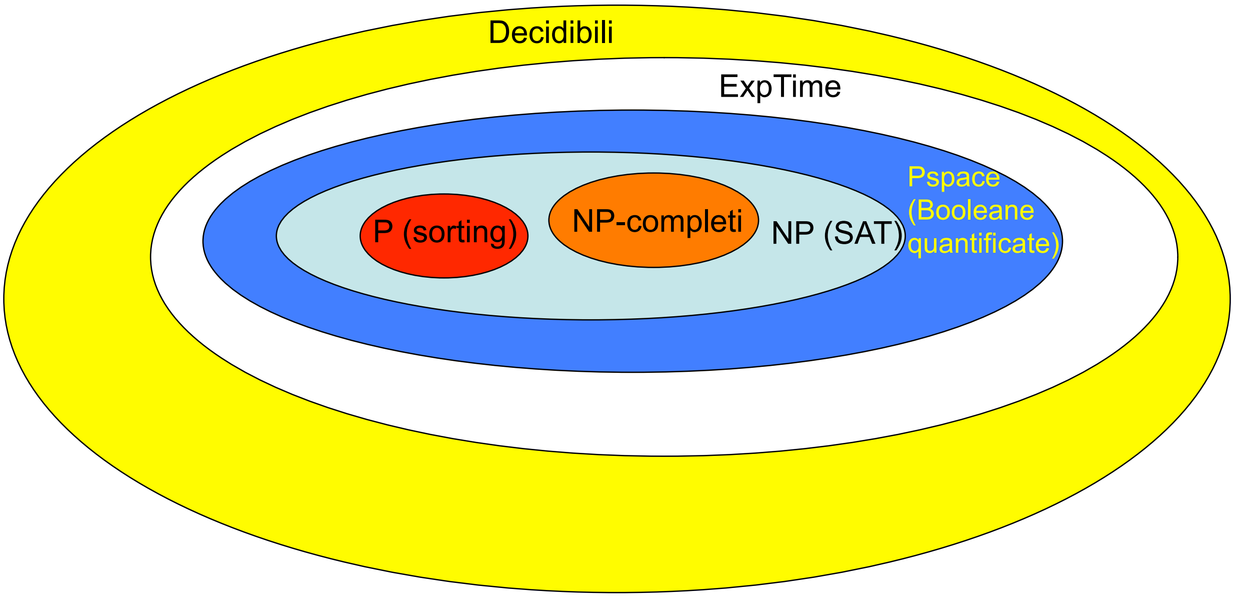
Altri famosi problemi NP-completi

- **Commesso viaggiatore**
  - Dati un grafo *completo* G con pesi w sugli archi ed un intero k, bisogna verificare se esiste un ciclo di peso al più k che attraversa ogni vertice una ed una sola volta
- **Colorazione**
  - Dati un grafo G ed un intero k, bisogna verificare se è possibile colorare i vertici di G con al più k colori tali che due vertici adiacenti non siano dello stesso colore

Altri famosi problemi NP-completi

- **Somme di sottoinsiemi**
  - Dati un insieme S di numeri naturali ed un intero t, bisogna verificare se esiste un sottoinsieme di S i cui elementi sommano esattamente a t
- **Zaino**
  - Dati un intero *k*, uno zaino di capacità c, e *n* oggetti di dimensioni $s_1, \ldots, s_n$ cui sono associati profitti $p_1, \ldots, p_n$, bisogna verificare se esiste un sottoinsieme degli oggetti di dimensione **al più** *c* che garantisca profitto **almeno** *k*

Gerarchia delle classi aggiornata



Algoritmi di ottimizzazione

- Abbiamo visto che un algoritmo di ottimizzazione può essere trasformato in un algoritmo di decisione che non è più difficile da risolvere del problema stesso
  - MSC: date due stringhe ed un intero k, esiste una SC di almeno k caratteri?
- Cosa fare nel caso si renda necessario risolvere un problema di ottimizzazione la cui versione decisionale sia NP-completa?

Algoritmi di approssimazione

- A volte, avere una soluzione esatta non è strettamente necessario
- Una soluzione che non si discosti troppo da quella ottima potrebbe comunque risultare utile
  - Ed è magari più semplice da calcolare
- La teoria dell'approssimazione formalizza questa semplice idea

Algoritmi di approssimazione

- In generale esistono numerose soluzioni ammissibili, ma non tutte sono ottime
  - Problema della colorazione di un grafo
  - Soluzione approssimata: colorare ogni vertice con un colore diverso
  - Soluzione ottima solo se il grafo è completo

Algoritmi di approssimazione

- Sia Π un problema di ottimizzazione
- Un algoritmo di approssimazione per Π ha fattore di approssimazione r(n) se il costo **C** della soluzione prodotta dall'algoritmo su una qualunque istanza di dimensione n differisce di un fattore moltiplicativo al più **r(n)** dal costo **C*** di una soluzione ottima.

Algoritmi di approssimazione

- *Minimizzazione*: $C \leq r(n)\ C^*$
- *Massimizzazione*: $C \geq C^*/r(n)$

- $r(n) \geq 1$
- Se $r(n) = 1$, C è chiaramente una soluzione ottima
- C sarà tanto peggiore quanto più si discosta da C* (tanto più r(n) si discosta da 1)

FINE