

UNIVERSITÀ DEGLI STUDI DI VERONA

Dispensa del corso di Complessità

Sebastiano Fregnan

2019/20

Indice

1	Problemi	5
1.1	Risolvibile VS. Verificabile	5
1.2	Verifica difficile	6
1.3	Definizioni introduttive	7
1.4	Trattabilità di un problema	8
2	Problemi P, NP e Riduzioni	11
2.1	Problema k -COL	11
2.2	Riduzione k -COL $\preceq (k+1)$ -COL	11
2.3	Problema SAT	13
2.4	Riduzione 3-COL \preceq SAT	13
2.5	Problema k -SAT	13
2.6	Schema attuale delle riduzioni	14
2.7	Riduzione SAT \preceq 3-SAT	14
2.8	Schema attuale delle riduzioni	14
2.9	Problema NAE- k -SAT	15
2.10	Riduzione 3-SAT \preceq 3-COL	15
2.10.1	Riduzione 3-SAT \preceq NAE-4-SAT	15
2.10.2	Riduzione NAE-4-SAT \preceq NAE-3-SAT	16
2.10.3	Riduzione NAE-3-SAT \preceq 3-COL	16
2.11	Schema attuale delle riduzioni	18
3	Problemi NP-COMPLETI, co-NP e co-NP-COMPLETI	19
3.1	Problemi NPC	19
3.2	Problema CIRCUIT-SAT	19
3.3	Esistenza dei problemi NPC	20
3.4	Riduzione CIRCUIT-SAT \preceq SAT	21
3.5	Dimostrazioni di NP-COMPLETEZZA	21
3.6	Problemi co-NP	22
3.7	Dimostrazione $P = co-P$	23
3.8	Dimostrazione $NP \stackrel{?}{=} co-NP$	23
3.9	“Perché” si crede $NP \neq co-NP$	23
3.10	Stato attuale delle classi	24
3.11	Problema SMALLFACTOR	24
3.12	Dimostrazione SMALLFACTOR $\in co-NP$	24
3.13	“Perché” si crede $P \subset NP \cap co-NP$	24
3.14	Problemi co-NPC	25
3.15	Problema TAU	25
3.16	Stato attuale delle classi	25
3.17	Problema D-HAMCYCLE	26
3.18	Dimostrazione D-HAMCYCLE $\in NPC$	26
3.19	Problema D-HAMPATH	28
3.20	Dimostrazione D-HAMPATH $\in NPC$	28
3.21	Problema HAMCYCLE	28
3.22	Riduzione D-HAMCYCLE \preceq HAMCYCLE	28
3.23	Dimostrazione HAMPATH $\in NPC$	29
3.24	Problema INDSET	29
3.25	Dimostrazione INDSET $\in NPC$	29
3.26	Problema VERTCOV	30
3.27	Dimostrazione VERTCOV $\in NPC$	31
3.28	Problema CLIQUE	31
3.29	Dimostrazione CLIQUE $\in NPC$	32
3.30	Schema attuale delle riduzioni	32

4	Determinismo e Macchine di Turing	33
4.1	Fondamenta delle classi P ed NP	33
4.2	Problema 2-SAT	34
4.3	Dimostrazione 2-SAT \in P	35
4.4	Problema REACH	36
5	Problemi debolmente NPC	37
5.1	Problema MAX-CUT	37
5.2	Dimostrazione MAX-CUT \in NPC	37
5.3	Problema MAX- k -SAT	38
5.4	Dimostrazione MAX-2-SAT \in NPC	38
5.5	Schema attuale delle riduzioni	39
5.6	Problema SUBSETSUM	39
5.7	Dimostrazione SUBSETSUM \in NPC	39
6	Prove di separabilità	43
6.1	Dimostrazione P \neq EXP	43
6.1.1	Teorema della gerarchia temporale	43
6.1.2	Conclusioni	45
6.2	Problemi “puramente” NP	45
7	Decisione vs. Ricerca	47
7.1	Self-reducibility	47
7.1.1	SAT è self-reducible	47
7.1.2	CLIQUE è self-reducible	48
7.2	Primo teorema di self-reducibility	49
7.3	GRAPHISOMORPHISM è self-reducible	50
7.4	Secondo teorema di self-reducibility	51
7.5	Terzo teorema di self-reducibility	52
8	Algoritmi di approssimazione e Inapprossimabilità	53
8.1	Problemi di ottimizzazione	53
8.2	Approssimazione	53
8.3	Classi di approssimazione	55
8.3.1	Problema KNAPSACK	56
8.3.2	Riduzione SUBSETSUM \preceq KNAPSACK ^{DEC}	58
8.3.3	Classi APX , PTAS e FPTAS	58
8.3.4	Tutti i problemi sono approssimabili?	59
8.3.5	Risultati di inapprossimabilità	61
8.3.6	Raffinare le approssimazioni	62
8.3.7	Ulteriori risultati di inapprossimabilità	63
8.4	L'approssimabilità è “transitiva”?	64
8.4.1	Problema MAX- k -XORSAT	64
8.4.2	Approssimabilità di MAX- k -XORSAT	65
8.4.3	Transività verso MAX- k -SAT	66
8.4.4	Transività verso INDSET	66
8.4.5	Transività verso VERTCOV	67
8.5	Conclusioni	67
9	Complessità spaziale	69
9.1	Gerarchia di complessità spaziale	69
9.2	Dimostrazione PSPACE = NPSPACE	71
9.3	PSPACE completezza	72
9.4	Esistenza dei problemi PSPACE -HARD	73

1 | Problemi

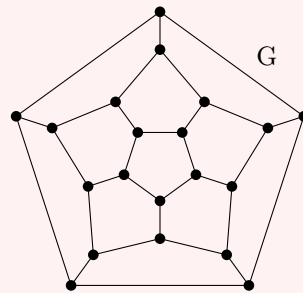
Le domande di fondo sono: Come crescono le *risorse* necessarie per *risolvere* un *problema* al crescere della sua dimensione? Come *misuro* la dimensione di un problema? Perché alcuni problemi sono *facili* e altri sono più *difficili*? Dove risiede la difficoltà di un problema? Come possiamo aggirarla? Dobbiamo necessariamente formalizzare alcuni termini intuitivi.

1.1 Risolvibile VS. Verificabile

Diamo tre esempi e per ognuno mostriamo due problemi.

Percorsi nei multigrafi

Abbiamo un grafo:



Problema 1 : Esiste un percorso che tocca ogni arco una sola volta?

Questo è il problema dei 7 ponti di Königsberg e, per cominciare, si potrebbe trovare un cammino tramite bruteforce, ma l'algoritmo è laborioso e inefficiente per casi più grandi (per n archi, 3 per ogni vertice, abbiamo circa 2^n possibilità). Anziché cercare di *trovare* il cammino, cerchiamo invece di *verificare* se esiste ne almeno uno, senza però fornirlo. Eulero propone nel 1736 una soluzione *semplice*:

Teorema di Eulero

Dato un multigrafo (dove dunque un vertice può avere più archi uscenti) unidirezionale, per *verificare* se esso possiede o meno un *tour Euleriano* (percorso che appunto passa per ogni arco una volta sola) basta controllare che ogni vertice abbia grado pari o che altrimenti solamente due siano dispari. Nel primo caso il percorso parte da un nodo e torna allo stesso, nel secondo caso il percorso parte da uno dei due nodi dispari e arriva nell'altro.

Scriviamo dunque l'algoritmo

Algorithm 1 Esistenza di un cammino Euleriano

```
1: function ISEULGRAPH(G)
2:   odd_vertex_num := 0
3:   for all vertex v of V(G) do
4:     if deg(v) is odd then odd_vertex_num++
5:   if odd-vertex-num is neither 0 nor 2 then return false
6:   return true
```

La *complessità* (costo) dell'algoritmo è *lineare*, infatti abbiamo $O(|E|+|V|)$, il problema scala linearmente con la dimensione del problema. Abbiamo quindi un algoritmo che, dato un grafo, ci permette di dire se esso è Euleriano o meno e, data una soluzione per il problema (quindi dato un cammino Euleriano nel grafo) possiamo verificare banalmente se esso è valido o meno.

Problema 2 : Esiste un percorso che tocca ogni vertice una sola volta?

Questo è il problema del puzzle di Hamilton, che si impone di trovare tour Hamiltoniano (che tocca appunto tutti i vertici una sola volta). Anche qui si può usare la tecnica di bruteforce, provando tutte le possibilità di scelta degli archi (2^n) oppure tutte le permutazioni dei vertici possibili ($n! > 2^n$, ancora peggio ...). Il problema è che meglio di così non sappiamo fare, per ora l'algoritmo migliore ha complessità $O(1.657^n)$. Ma qual è la migliore complessità ottenibile per decidere se un grafo è Hamiltoniano? Ad oggi non lo sappiamo, forse esponenziale? Pensiamo invece alla complessità di verificare se un grafo è (o non è) Hamiltoniano: dato un tour Hamiltoniano, basta controllare che sia continuo passo passo e controllare che tutti i vertici vengano toccati.

Raggruppamenti vincolati

Dato un insieme di N numeri tali che la loro somma è $\sum_{i=0}^N n_i = S$, partizionare tali numeri in due gruppi tali che la somma dei numeri che li compongono sia $S/2$.

Problema 1 : risolvere per $N = 38$.

Si può anche qui provare con il bruteforce, valutando tutte le possibilità: si ha $\binom{N}{N/2} = 3.5 \times 10^{10} > 2^N$, quindi anche qui abbiamo una complessità esponenziale (vedremo che però si può risolvere efficientemente con programmazione dinamica).

Problema 2 : risolvere per $N = 38000$.

Qui ovviamente i numeri sono estremamente più alti. Quanto costa invece verificare se esiste una soluzione? Banale, basta verificare se un gruppo ha $N/2$ numeri e se la loro somma fa $S/2$.

Numeri primi

Dato un numero N

Problema 1 : dire se N è primo.

Abbiamo un algoritmo per ora con complessità $O((\log N)^{6+\epsilon})$, mentre la verifica utilizza il medesimo algoritmo, controllando se esiste almeno un divisore diverso da 1 e N stesso.

Problema 2 : scomporre N in fattori primi.

Qui ad oggi non abbiamo ancora nulla se non provare ogni possibilità, ma la verifica è banale, infatti data una scomposizione basta moltiplicare i fattori e controllare che dia N . Facciamoci un'altra domanda:

Problema 3 : Quanto costa verificare che N ha fattori minori di q ?

Vedremo che esiste un algoritmo semplice che, dati N e q , ci permette di verificare (o meno!) tale quesito.

Tracciamo una tabella per i problemi visti finora e notiamo che quanto un problema è di facile risoluzione allora è anche di facile verifica (banalmente, basta trovare la soluzione e controllare che sia la stessa soluzione fornita). Per i problemi difficili da risolvere lasciamo ancora un punto interrogativo.

Problema	Risolvere	Verificare
G è Euleriano?	facile	facile
G è Hamiltoniano?	difficile?	facile
Partizioni a somme uguali	difficile?	facile
N è primo?	facile	facile
N ha fattori piccoli ($< q$)?	difficile?	facile

Vedremo che sarà facile verificare se un grafo è (ma non se non è) Hamiltoniano, mentre sarà facile verificare se N ha (o non ha!) fattori piccoli di q .

1.2 Verifica difficile

Dalla tabella sopra sembra che verificare un problema sia sempre facile, vediamo ora però che non è affatto così. Data una partita a scacchi con la dicitura “matto in N mosse”, quanto costa verificare che il bianco possa effettivamente fare scacco matto in N mosse? Sembra che per ora l'unico modo sia mostrare che esiste una mossa bianca w_1 tale che per ogni mossa nera b_1 esiste una mossa w_2 tale che per ogni mossa b_2 esiste ... tale

che esiste una mossa bianca w_N che fa scacco matto. Ovviamente questo procedimento è esponenzialmente costoso, e quindi anche il verificare qualcosa può essere oneroso.

1.3 Definizioni introduttive

Problema

Un problema computazionale \mathbb{A} è una relazione $\mathbb{A} \subseteq \mathcal{I}(\mathbb{A}) \times S(\mathbb{A})$ che lega un insieme *infinito* di istanze $\mathcal{I}(\mathbb{A})$ a tutte le possibili soluzioni $S(\mathbb{A})$. Ad ogni istanza $x \in \mathcal{I}(\mathbb{A})$ è associato un insieme di soluzioni $sol(x)$ per quell'istanza e quindi le soluzioni totali del problema sono

$$S(\mathbb{A}) = \bigcup_x sol(x).$$

Esempio

Il problema “trovare un cammino Hamiltoniano” sarà dunque espresso come

$$\text{HAMPATH} = \{(G, P_G) \mid G \text{ è un grafo, } P_G \text{ è un cammino Ham. in } G\}$$

dove istanze e soluzioni sono

$$\begin{aligned} \mathcal{I}(\text{HAMPATH}) &= \{\text{tutti i grafi } G\} \\ \forall G, sol(G) &= \{P \mid P \text{ è un cammino Ham. in } G\}. \end{aligned}$$

Risolvere un problema computazionale significa definire un algoritmo A tale che

$$\forall x \in \mathcal{I}(\mathbb{A}), A(x) = y \in sol(x).$$

Quello che vogliamo è valutare la difficoltà di \mathbb{A} in termini delle risorse necessarie ad un algoritmo che risolve \mathbb{A} , cioè tempo e spazio. Diciamo che \mathbb{A} è più difficile di \mathbb{B} se il primo necessita di più risorse rispetto al secondo.

Osservazione: non tutte le domande sono considerabili come problemi computazionali, infatti “sotto l'ipotesi di una partita perfetta, il bianco ha sempre una strategia vincente?” non è un problema computazionale, perché il gioco degli scacchi ha numero *finito*, seppur enorme, di possibili partite.

Efficienza di un algoritmo

Misuriamo le prestazioni di un algoritmo in base all'ordine di crescita $T(n)$ delle risorse necessarie nel caso peggiore (e nel caso medio) di un certo problema, in base alla *dimensione* n dell'istanza.

Complessità di un problema

Per definire quanto è difficile un problema diamo le nozioni di

Upper bound (*quanto bene sappiamo risolvere il problema*) cioè il massimo numero di risorse utilizzate da almeno *un* algoritmo che risolve quel problema. Rappresenta un “massimo noto” della complessità di quel problema, cioè: sicuramente possiamo fare peggio, forse possiamo fare meglio, ma per ora riusciamo a fare così;

Lower bound (*quanto è intrinsecamente difficile il problema*) cioè il minimo numero di risorse necessario per *ogni* algoritmo che risolve quel problema. Rappresenta un “minimo noto” della complessità di quel problema, quindi dobbiamo dimostrare che *ogni* algoritmo è almeno così.

Per poter affermare che un problema \mathbb{A} è più difficile di \mathbb{B} dobbiamo quindi dimostrare che esiste un algoritmo $A_{\mathbb{B}}$ per \mathbb{B} di complessità $T_{\mathbb{B}} = O(f(n))$ tale che per ogni algoritmo $A_{\mathbb{A}}$ per \mathbb{A} di complessità $T_{\mathbb{A}}$ si ha $T_{\mathbb{A}} = \Omega(f(n))$, per ogni algoritmo.

Esempio

Il problema SEARCH è più facile del problema SORT, T_{SEARCH} ha upper bound $O(n)$ (per trovare basta scorrere) mentre T_{SORT} ha lower bound $\Omega(n \log n)$, e ovviamente $O(n) < \Omega(n \log n)$.

1.4 Trattabilità di un problema

Gli algoritmi che abbiamo visto fino ad ora avevano tutti due tipi di crescita differenti:

crescita polinomiale quando le risorse richiedono n^k con k costante, quindi al variare di n si aggiunge una costante moltiplicativa;

crescita esponenziale quando le risorse richiedono c^n con $c > 1$ costante, quindi al variare di n si aumenta l'esponente.

Al fine pratico, la differenza tra polinomiale ed esponenziale è conducibile alla differenza tra finito e infinito, quindi tra risolvibile in tempo utile e invece irrisolvibile. Definiamo quindi la classificazione:

P classe di problemi risolvibili in tempo polinomiale $O(n^c)$ per qualche costante c

Riscriviamo la tabella già vista in utilizzando la nuova definizione data

Problema	Risolvere	Verificare	Classe
G è Euleriano?	polytime	polytime	P
G è Hamiltoniano?	difficile?	polytime	?
Partizioni a somme uguali	difficile?	polytime	P
N è primo?	polytime	polytime	P
N ha fattori piccoli ($< q$)?	difficile?	polytime	?

Tutti i problemi con soluzione e verifica in tempo polinomiale sono quindi in **P**, mentre il problema delle partizioni è anch'esso in **P** poiché come abbiamo detto prima vedremo una soluzione che ne renderà la soluzione facile.

Problema decisionale

Un problema dove la risposta è sì/no (y/n) è detto decisionale; singolarmente, ci saranno quindi “istanze sì” e “istanze no”. \mathbb{A} è decisionale se $\mathbb{A} : \mathcal{I}(\mathbb{A}) \mapsto \{\text{yes}, \text{no}\}$.

Problema di ricerca

Un problema dove, diversamente da un problema decisionale, la risposta è una certa soluzione per un'istanza x è $y \in \text{sol}(x)$ è detto di ricerca.

Osservazione: per ogni problema di ricerca di una certa difficoltà esiste un problema di decisione della stessa difficoltà.

Con queste definizioni di problemi, ignoriamo i problemi di ricerca e raffiniamo **P** solo sui decisionali:

P classe di *problemi decisionali* risolvibili in tempo polinomiale $O(n^c)$ nella taglia n dell'istanza.

Definiamo per controparte la classe:

EXP classe di problemi decisionali risolvibili in tempo esponenziale $O(2^{\text{poly}(n)})$ nella taglia n dell'istanza.

notiamo che $\mathbf{P} \subseteq \mathbf{EXP}$ poiché per ogni funzione polinomiale $p(n)$ e ogni funzione esponenziale a^{bn} con $a > 1$ abbiamo che $p(n) = O(a^{bn})$.

Come abbiamo visto nella tabella, $\text{EULPATH} \in \mathbf{P}$, ma $\text{HAMPATH} \in \mathbf{P}$? Non lo sappiamo ancora, non abbiamo ancora un algoritmo, ma sappiamo che almeno la verifica è polinomiale e sappiamo che per ogni istanza per cui la risposta è sì esiste una *prova semplice*, cioè un *certificato* che può essere verificato in tempo polinomiale.

Definiamo quindi un'ulteriore classe:

NP classe di problemi decisionali dove ogni *istanza sì* ha una *prova semplice*.

Immediatamente notiamo che $\mathbf{P} \subseteq \mathbf{NP}$ (poiché se per un problema è possibile trovare una soluzione in polytime allora per la prova semplice basta risolvere il problema) e notiamo che $\text{HAMPATH} \in \mathbf{NP}$.

Aggiungiamo un altro strato di formalismo e ridefiniamo \mathbf{P} come

$$\mathbf{P} = \{ \mathbb{A} \mid \exists \text{ algoritmo } A_{\mathbb{A}}(\cdot) \text{ polytime in } |x| \text{ t.c.} \\ \forall x \in \mathcal{I}(\mathbb{A}), \mathbb{A}(x) = \text{yes} \iff A_{\mathbb{A}}(x) = \text{yes} \}$$

Chiamiamo $A_{\mathbb{A}}$ il **risolutore** (polinomiale) di \mathbb{A} , cioè di complessità $T_A(|x|) = O(|x|^c)$. Riscriviamo \mathbf{NP} come

$$\mathbf{NP} = \{ \mathbb{A} \mid \exists c \text{ costante, } \exists \text{ algoritmo } B(\cdot, \cdot) \text{ polytime in } |x|, |y| \text{ t.c.} \\ \forall x \in \mathcal{I}(\mathbb{A}), \exists y \in \{0, 1\}^{|x|^c} \text{ t.c. } \mathbb{A}(x) = \text{yes} \iff B(x, y) = \text{yes} \}$$

Chiamiamo B il **verificatore** o **certificatore** (polinomiale) per il problema \mathbb{A} e $y \in \{0, 1\}^{|x|^c}$ il **certificato** da prodotto da B , rappresentato da una stringa binaria (sarà poi chiaro il motivo) di lunghezza polinomiale nella taglia dell'istanza appunto.

Dato che la taglia del certificato y è $\text{poly}(|x|)$ e il verificatore B è polytime nel proprio input (x, y) per definizione, la verifica di $B(x, y)$ sarà $\text{poly}(|x|)$. Infatti, $|y| = O(|x|^d)$ e l'istanza di B ha taglia $N = |x| + |y| = |x| + |x|^d$, quindi il tempo di esecuzione di B è $T_B(N) = O(N^c) = O((|x| + |x|^d)^c) = O(|x|^{d \cdot c})$, che è $\text{poly}(|x|)$.

Esempio

Applichiamo la definizione di \mathbf{NP} ad HAMPATH . Sappiamo che l'istanza è un grafo $G = (V, E) \in \mathcal{I}(\text{HAMPATH})$ di taglia $|G| = |V| + |E| = n + m$, mentre il certificato è un cammino $P = \{v_1, \dots, v_n\}$ di taglia $|P| = n = O((n + m)^1)$, dove v_i sono i vertici del grafo. Notiamo che come da prima condizione per \mathbf{NP} , la taglia $|P|$ del certificato è polinomiale nella taglia $|G|$ dell'istanza, per $c = 1$. Il verificatore $B(G, P)$ ha taglia dell'input $N = |G| + |P| = 2n + m = O(n^2)$ e

1. verifica che P abbia n vertici [tempo $O(n)$], ma potrebbero essere tutti uguali...;
2. verifica che P abbia tutti i vertici del grafo G [tempo $O(n)$], ma forse non esistono quegli archi...;
3. verifica che $\forall i = 1 \dots n - 1, (v_i, v_{i+1}) \in E$ [tempo $O(n \times m)$].

Con queste verifiche è immediato che il verificatore B accetta solo istanze P che sono cammini Hamiltoniani su G e quindi accetta se e solo se G è un'istanza **yes** per HAMPATH . Ricordando che $E \subseteq V \times V$ e quindi $|E| \leq |V|^2$, notiamo che come da seconda condizione per \mathbf{NP} , il verificatore B ha complessità polytime $O(n^3)$ nella taglia dell'istanza $N = O(n^2)$ per $c = 3/2$, e quindi $\text{HAMPATH} \in \mathbf{NP}$.

Dato che in problemi \mathbf{NP} non hanno un algoritmo polinomiale, proviamo a vedere se almeno hanno un algoritmo esponenziale, ovvero se $\mathbf{NP} \subseteq \mathbf{EXP}$. Vogliamo quindi scoprire se $\forall \mathbb{A} \in \mathbf{NP} \implies \mathbb{A} \in \mathbf{EXP}$: preso un $\mathbb{A} \in \mathbf{NP}$, di cui ricordiamo la definizione

$$\mathbb{A} \in \mathbf{NP} \implies \exists \text{ una costante } c \text{ e algoritmo polinomiale } B(\cdot, \cdot) \text{ t.c.} \\ \forall x \in \mathcal{I}(\mathbb{A}), \mathbb{A}(x) = \text{yes} \iff \exists y \in \{0, 1\}^{|x|^c} \text{ t.c. } B(x, y) = \text{yes.}$$

possiamo scrivere un algoritmo \mathbb{A} -SOLVER che, per un certo $x \in \mathcal{I}(\mathbb{A})$

Algorithm 2 \mathbb{A} -SOLVER (algoritmo risolutore per \mathbb{A})

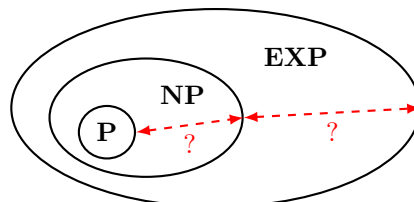
Require: una istanza x

- 1: **for all** istanza y in $\{0, 1\}^{|x|^c}$ **do**
 - 2: **if** $B(x, y) = \text{yes}$ **then return** true
 - 3: **return** false
-

L'algoritmo \mathbb{A} -SOLVER ha $2^{|x|^c}$ possibili y e per ognuna si esegue il verificatore B , quindi la sua complessità è $O(2^{|x|^c} \times |x|^c) = O(2^{|x|^c})$, cioè esponenziale nella taglia dell'istanza x .

Quindi per ora possiamo dire $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. Vedremo che $\mathbf{P} \subsetneq \mathbf{EXP}$, e quindi $\mathbf{P} \subset \mathbf{EXP}$, cioè \mathbf{NP} avrà spazio tra \mathbf{P} ed \mathbf{EXP} , ma non sappiamo ancora quanto sia largo \mathbf{NP} e soprattutto se $\mathbf{P} = \mathbf{NP}$, ma è sentore comune che $\mathbf{P} \neq \mathbf{NP}$ e quindi che $\mathbf{P} \subset \mathbf{NP}$.

$$\mathbf{P} \stackrel{?}{\subset} \mathbf{NP} \stackrel{?}{\subset} \mathbf{EXP}$$



2 | Problemi P, NP e Riduzioni

2.1 Problema k -COL

Supponiamo di avere un insieme di eventi che vogliamo allocare in k slot di tempo, sapendo che per ogni coppia di eventi esistono partecipanti interessati ad entrambi ma che potranno quindi partecipare ad uno solo dei due. Usando un grafo G , ogni evento è un vertice $v \in V$ e ogni arco identifica che esiste un partecipante interessato ad entrambi i vertici, simboleggiando quindi i conflitti.

k -COLORAZIONE

INPUT un grafo $G = (V, E)$

OUTPUT $\text{yes} \iff$ esiste una colorazione $c(\cdot)$ di G con k colori tale che per ogni arco $(u, v) \in E$ il colore di u è diverso da quello di v

Studiamo il problema al crescere di k :

- i grafi 1-COL sono solo quelli senza edge, basta controllare $E = \emptyset$, quindi 1-COL $\in \mathbf{P}$;
- i grafi 2-COL sono solo quelli bipartiti (quindi divisibili in due insiemi di vertici tali che gli archi vanno solo da un insieme all'altro), basta controllare che non ci sia un ciclo dispari, fattibile con la BFS su G controllando che vertici alla stessa profondità non abbiano archi che li collegano, quindi 2-COL $\in \mathbf{P}$;
- i grafi 3-COL non sappiamo se sono fattibili in \mathbf{P} , e anzi crediamo di no. Abbiamo però 3-COL $\in \mathbf{NP}$, infatti il certificato è un assegnamento ad ogni vertice di un colore $c(v_i) \in \{1, 2, 3\}$ ed ha taglia $|y| = n = O(|G|^1)$ polinomiale nella taglia $|G|$ per $c = 1$. Il verificatore esegue i seguenti step:
 1. controlla che tutti i vertici ci sia un colore [in $O(n)$];
 2. controlla che ci siano esattamente k colori [in $O(n)$];
 3. controlla che ogni vertice v_i abbia i suoi adiacenti $v_j \in \text{Adj}(v_i)$ di un colore diverso dal suo [in $O(m)$];

e quindi B ha complessità polytime $O(n + m) = O(|G|)$ polinomiale nella taglia $|G|$. Abbiamo il certificato di taglia polinomiale nella taglia dell'istanza e il verificatore polytime nella taglia della sua istanza, quindi è dimostrata l'appartenenza a \mathbf{NP} .

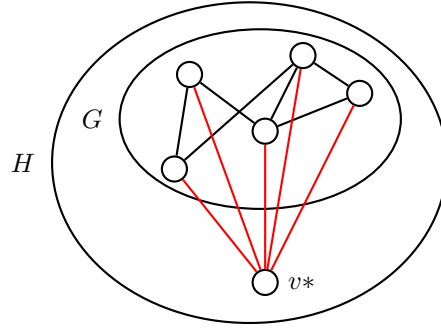
Tutti i grafi k -COL con $k > 2$ sono solo in \mathbf{NP} , e in generale quindi k -COL $\in \mathbf{NP}$.

2.2 Riduzione k -COL $\preceq (k+1)$ -COL

Ora vogliamo vedere se è più difficile k -COL piuttosto che $(k+1)$ -COL, poiché se $(k+1)$ -COL fosse facile allora potremmo stringere la difficoltà per k -COL a facile, con una **riduzione** da k -COL a $(k+1)$ -COL.

Supponiamo che $(k+1)$ -COL sia facile e che quindi esista un risolutore polinomiale B per $(k+1)$ -COL sull'istanza H grafo tale che $B(H) = \text{yes} \iff H$ è $(k+1)$ -COL, quindi esiste c tale che B è polytime $T_B(H) = O(|H|^c)$ nella taglia dell'istanza. Costruiamo ora un risolutore polinomiale A per k -COL usando B come subroutine: per adattarsi alle istanze di B , inizialmente A trasforma le sue istanze G in istanze per B , creando un grafo $H = (V(H), E(H))$ con $V(H) = V(G) \cup \{v^*\}$ e $E(H) = E(G) \cup \{(v^*, v) \mid v \in V(G)\}$, dove v^* è un nuovo vertice collegato a tutti gli altri vertici, e poi ritorna $B(H)$. Vogliamo quindi dimostrare che

$$B(H) = \text{yes} \iff H \text{ è } (k+1)\text{-COL} \iff G \text{ è } k\text{-COL} \iff A(G) = \text{yes}.$$



Dimostrazione. Trattiamo la doppia implicazione al centro

- \Leftarrow Se G è k -COL allora posso impostare v^* di un colore diverso dai precedenti (il $k+1$ -esimo colore). In questo modo tutti gli archi di H hanno vertici di colori diversi;
- \Rightarrow Se H è $(k+1)$ -COL ogni colore assegnato ai vertici di G è diverso dal colore di v^* per costruzione, e l'insieme dei colori di G è quindi al massimo k . Dato che gli archi di G sono anche archi di H , essi saranno già colorati propriamente, rendendo G k -COL.

Studiamo la complessità di A . Il risolutore A deve:

1. creare il grafo H , aggiungendo un arco per ogni vertice [tempo $O(|V(G)|) = O(n)$]. Vediamo che la taglia di H è $|H| = |V(G)| + |\{v^*\}| + |E(G)| + |\{(v, v^*) \mid v \in V(G)\}| = n + 1 + m + n = O(n^2)$;
2. applica $B(H)$, che è polytime nella taglia di H e anche nella taglia di G [tempo $O(|H|^c) = O((n^2)^c) = O(n^{2c}) = O(|G|^c)$];

quindi A è polinomiale in $|G|$ e risolve k -COL in polytime. \square

Quindi abbiamo provato che se $(k+1)$ -COL è facile allora k -COL è facile ed equivalentemente, se k -COL è difficile allora $(k+1)$ -COL è difficile e cioè che possiamo **ridurre** il problema k -COL al problema $(k+1)$ -COL. Per farlo, data un'istanza G , la mappiamo ad un'istanza H e se G è **yes** allora anche H è **yes**, dunque k -COL è difficile come $(k+1)$ -COL.

Riduzione alla Karp

Il problema \mathbb{A} è riducibile (alla Karp) al problema \mathbb{B} , usando la notazione $\mathbb{A} \preceq \mathbb{B}$ letta “ \mathbb{A} si riduce a \mathbb{B} ”, se esiste una funzione $f : \mathcal{I}(\mathbb{A}) \mapsto \mathcal{I}(\mathbb{B})$ computabile da un *algoritmo polytime* nella taglia dell'istanza di \mathbb{A} tale che per ogni $x \in \mathcal{I}(\mathbb{A})$, $\mathbb{A}(x) = \text{yes} \iff \mathbb{B}(f(x)) = \text{yes}$. In pratica, per risolvere un'istanza di \mathbb{A} prima la trasformiamo in modo *semplice* in una istanza di \mathbb{B} e poi risolviamo \mathbb{B} .

Osservazione: Se \mathbb{A} e \mathbb{B} sono problemi di decisione tali che $\mathbb{A} \preceq \mathbb{B}$ allora

1. $\mathbb{B} \in \mathbf{P} \implies \mathbb{A} \in \mathbf{P}$
2. $\mathbb{A} \notin \mathbf{P} \implies \mathbb{B} \notin \mathbf{P}$

Dimostrazione. Dimostriamo solo il primo enunciato, il secondo è il contropositivo. Se $\mathbb{B} \in \mathbf{P}$ allora esiste un algoritmo polytime B per \mathbb{B} tale che $\forall y \in \mathcal{I}(\mathbb{B})$, $B(y) = \text{yes} \iff \mathbb{B}(y) = \text{yes}$. Se $\mathbb{A} \preceq \mathbb{B}$ allora esiste un algoritmo polytime R_f che calcola f tale che $\forall x \in \mathcal{I}(\mathbb{A})$, $\mathbb{A}(x) = \text{yes} \iff \mathbb{B}(f(x)) = \text{yes}$. Definiamo un algoritmo A per \mathbb{A} tale che $\forall x \in \mathcal{I}(\mathbb{A})$, $A(x) = B(R_f(x))$. Da tale algoritmo si evince che

$$\begin{aligned} A(x) = \text{yes} &\iff B(R_f(x)) = \text{yes} \\ &\iff B(f(x)) = \text{yes} \\ &\iff \mathbb{B}(f(x)) = \text{yes} \\ &\iff \mathbb{A}(x) = \text{yes} \end{aligned}$$

e quindi A risolve \mathbb{A} ed è polytime perché è composizione di algoritmi polytime. \square

Osservazione: Vale la proprietà transitiva: se $\mathbb{A} \preceq \mathbb{B}$ e $\mathbb{B} \preceq \mathbb{C}$ allora $\mathbb{A} \preceq \mathbb{C}$.

2.3 Problema SAT

Satisfiability

INPUT formule ϕ di clausole in CNF (Conjunctive Normal Form)

OUTPUT **yes** \iff esiste un assegnamento a alle variabili tale che $\phi(a) = \mathbf{T}$, quindi almeno un letterale per ogni clausola deve essere \mathbf{T}

2.4 Riduzione 3-COL \preceq SAT

Trasformiamo 3-COL nel problema SAT (di soddisfacibilità logica). Fissiamo $k = 3$ e diciamo che G è 3-COL se esiste un assegnamento di colori tale che

1. ogni vertice v ha assegnato uno ed un solo colore;
2. per ogni arco (u, v) , i vertici u e v non hanno entrambi colore i .

Per ogni vertice v del grafo G usiamo k variabili, nel nostro caso x_1^v, x_2^v, x_3^v , dove ogni x_i^v è vera se e solo se al vertice v viene assegnato il colore i (quindi solo una sarà \mathbf{T}). Scriviamo ora due formula CNF che codificano le proprietà sopra elencate:

1. $\forall v \in V, \zeta(v) = (x_1^v \vee x_2^v \vee x_3^v) \wedge (\bar{x}_1^v \vee \bar{x}_2^v) \wedge (\bar{x}_1^v \vee \bar{x}_3^v) \wedge (\bar{x}_2^v \vee \bar{x}_3^v)$
2. $\forall u, v \in V, \psi(u, v) = (\bar{x}_1^u \vee \bar{x}_1^v) \wedge (\bar{x}_2^u \vee \bar{x}_2^v) \wedge (\bar{x}_3^u \vee \bar{x}_3^v)$

Quindi G è 3-COL se e solo se

$$\phi(x) = \bigwedge_{v \in V} \zeta(v) \wedge \bigwedge_{(u,v) \in E} \psi(u, v)$$

è soddisfacibile, cioè se esiste un assegnamento di valori alle variabili x tale che $\phi(x) = \mathbf{T}$.

Dato che G ha $|V| = n$ ed $|E| = m$, la formula ϕ ha $N = 3n$ variabili (3 per ogni vertice) e $M = 4n + 3m$ clausole (4 per ogni vertice e 3 per ogni arco), quindi la taglia di ϕ è polinomiale nella taglia di G e trasformare G in ϕ è ottenibile in polytime nella taglia di G (basta scorrere ogni vertice e ogni arco [tempo $O(n + m)$]) e quindi $k\text{-COL} \preceq \text{SAT}$.

Riduzione $k\text{-COL} \preceq \text{SAT}$ viene lasciata come esercizio.

2.5 Problema $k\text{-SAT}$

k-Satisfiability

INPUT formule ϕ di clausole in $k\text{-CNF}$ (ogni clausola ha esattamente k letterali)

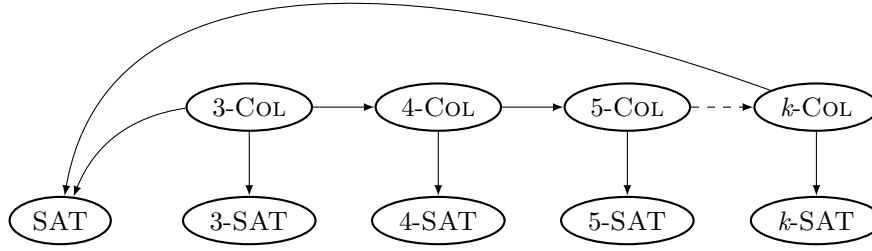
OUTPUT **yes** \iff esiste un assegnamento a alle variabili tale che $\phi(a) = \mathbf{T}$, quindi almeno un letterale per ogni clausola deve essere \mathbf{T}

Riduzione 3-COL \preceq 3-SAT viene lasciata questa riduzione come esercizio.

Riduzione $k\text{-COL} \preceq k\text{-SAT}$ viene lasciata questa riduzione come esercizio.

2.6 Schema attuale delle riduzioni

Ogni freccia $\textcircled{\mathbb{A}} \longrightarrow \textcircled{\mathbb{B}}$ indica la relazione $\mathbb{A} \preceq \mathbb{B}$.



Quindi se uno qualsiasi tra i problemi elencati è \mathbf{P} allora $3\text{-COL} \in \mathbf{P}$, mentre al contrario se $3\text{-COL} \notin \mathbf{P}$ allora anche tutti gli altri non sono \mathbf{P} .

2.7 Riduzione $\text{SAT} \preceq 3\text{-SAT}$

Se limito le formule ad una struttura ben definita allora la complessità “non cambia”. Mostriamo che data una CNF ϕ possiamo ottenere una 3-CNF ψ in polytime tale che ϕ è soddisfacibile $\iff \psi$ è soddisfacibile.

Dobbiamo quindi trasformare ogni clausola di ϕ in un insieme di clausole con esattamente 3 letterali e identifichiamo due casi:

- se una clausola ha meno di 3 letterali, ne duplico qualcuno, fattibile in polytime

$$\text{(es.) } C^{(i)} = (x_j \vee \bar{x}_\ell) \text{ diventa } D^{(i)} = (x_j \vee x_j \vee \bar{x}_\ell);$$

- se una clausola ha più di 3 letterali la spezzo collegando i letterali da nuove variabili

$$\text{(es.) } C^{(i)} = (\bar{x}_1 \vee x_2 \vee x_3 \vee \bar{x}_4 \vee x_5) \text{ diventa } D^{(i)} = (\bar{x}_1 \vee x_2 \vee z_1^i) \wedge (\bar{z}_1^i \vee x_3 \vee z_2^i) \wedge (\bar{z}_2^i \vee \bar{x}_4 \vee x_5)$$

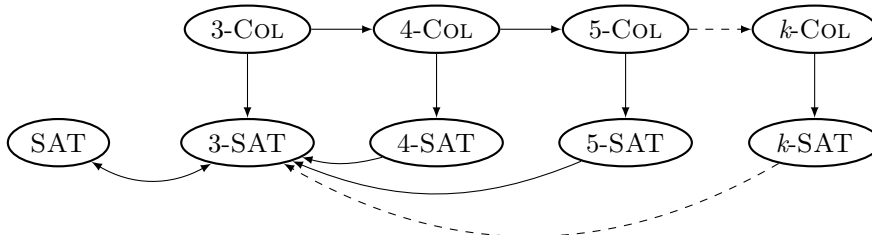
Per ogni clausola $C^{(i)}$ di k_i letterali quindi uso $k_i - 2$ clausole $D_j^{(i)}$ di 3 letterali, e così facendo posso trasformare in polytime una qualsiasi formula k -CNF in una 3-CNF, dimostrando che $\text{SAT} \preceq 3\text{-SAT}$.

Riduzione $k\text{-SAT} \preceq 3\text{-SAT}$ seguendo il procedimento visto sopra, basta trasformare tutte le clausole in k -CNF in clausole 3-CNF, fattibile in polytime.

Riduzione $3\text{-SAT} \preceq \text{SAT}$ semplicemente usando la funzione identità, una formula 3-CNF è già in CNF, fattibile in tempo costante, quindi in polytime.

2.8 Schema attuale delle riduzioni

Notiamo che $\text{SAT} \equiv 3\text{-SAT}$ e inoltre che le riduzioni dirette da $k\text{-COL}$ a SAT sono state cancellate poiché implicite per transitività.



Come vediamo, se $3\text{-SAT} \in \mathbf{P}$ allora $\text{SAT}, k\text{-SAT}, k\text{-COL} \in \mathbf{P}$, mentre al contrario se uno qualsiasi dei problemi elencati non è \mathbf{P} allora $3\text{-SAT} \notin \mathbf{P}$. Decidiamo di porre l'attenzione su 3-SAT anziché su SAT semplicemente perché è più strutturato, anche se sono equivalenti.

2.9 Problema NAE- k -SAT

Not-All-Equal k -Satisfiability

INPUT formule ϕ di clausole in k -CNF

OUTPUT **yes** \iff esiste un assegnamento a alle variabili tale che $\phi(a) = \text{T}$ e per ogni clausola esiste almeno un letterale T e almeno un letterale F

Osservazione: Per un qualsiasi assegnamento $a = (a_1 \dots a_n, z)$ che NAE-soddisfa ϕ anche il suo negato \bar{a} NAE-soddisfa ϕ , poiché se in a c'è $z = \text{T}$ e almeno un altro letterale a F per ogni clausola di ϕ , di conseguenza in \bar{a} ci sarà $z = \text{F}$ e almeno un altro letterale a T.

Esempio

Data la formula

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

per l'assegnamento $a = (a_1, a_2, a_3) = (\text{T}, \text{T}, \text{T})$ ogni clausola ha un letterale vero e uno falso, quindi a NAE-soddisfa ϕ . Per l'osservazione vista prima, anche $\bar{a} = (\text{F}, \text{F}, \text{F})$ NAE-soddisfa ϕ . Al contrario invece, $a' = (\text{T}, \text{F}, \text{F})$ soddisfa ϕ (ogni clausola ha un letterale T) ma non NAE-soddisfa ϕ .

2.10 Riduzione 3-SAT \preceq 3-COL

Vogliamo ridurre 3-SAT \preceq 3-COL e per farlo prima riduciamo 3-SAT \preceq NAE-4-SAT, poi NAE-4-SAT \preceq NAE-3-SAT e infine NAE-3-SAT \preceq 3-COL.

2.10.1 Riduzione 3-SAT \preceq NAE-4-SAT

Sia $\phi(x_1 \dots x_n) \in \mathcal{I}(3\text{-SAT})$ una formula 3-CNF $\phi(x_1 \dots x_n) = C^{(1)} \wedge \dots \wedge C^{(m)}$ dove ogni clausola è $C^{(i)} = \ell_1^{(i)} \vee \ell_2^{(i)} \vee \ell_3^{(i)}$ e ogni letterale $\ell_j^{(i)} \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$.

Desideriamo trasformare ϕ in una formula ψ 4-CNF, quindi ϕ sarà soddisfacibile se e solo se ψ sarà NAE-soddisfacibile. Definiamo la formula 4-CNF

$$\psi(x_1 \dots x_n, z) = D^{(1)} \wedge \dots \wedge D^{(m)} \text{ di clausole } D^{(i)} = C^{(i)} \vee z = (\ell_1^{(i)} \vee \ell_2^{(i)} \vee \ell_3^{(i)} \vee z).$$

costruibile in polytime nella taglia di ϕ (basta scorrere ϕ aggiungendo z a ogni clausola).

Dimostrazione. Vediamo ora la doppia implicazione:

\Rightarrow se ϕ è soddisfacibile allora ψ è NAE-soddisfacibile, cioè esiste $a = a_1 \dots a_n$ tale che $\phi(a_1 \dots a_n) = \text{T}$, quindi per ogni clausola $C^{(i)}$, a rende almeno un letterale T. Consideriamo ora l'assegnamento $b \in \{\text{T}, \text{F}\}^{(n+1)}$ dove $b = (a_1 \dots a_n, \text{F})$, quindi un assegnamento identico ad a ma aumentato di un valore F per z : dato che in ogni $D^{(i)}$ c'è una clausola $C^{(i)}$ e letterale z , ci sarà sicuramente un letterale T dato da a_i e sicuramente un letterale F, proprio z . Abbiamo quindi che b NAE-soddisfa ψ ;

\Leftarrow se ψ è NAE-soddisfacibile allora ϕ è soddisfacibile, ci troviamo in due casi:

- esiste un assegnamento b' che pone $z = \text{F}$ e uno almeno un letterale per ogni clausola necessariamente a T, dato che ogni clausola $D^{(i)} = (\ell_1^{(i)} \vee \ell_2^{(i)} \vee \ell_3^{(i)} \vee z)$ deve contenerne almeno un T. Quindi, almeno uno tra $\ell_1^{(i)}, \ell_2^{(i)}, \ell_3^{(i)}$ sarà T e, costruendo una clausola $C^{(i)} = (\ell_1^{(i)} \vee \ell_2^{(i)} \vee \ell_3^{(i)})$, definiamo $\phi(x_1 \dots x_n) = C^{(1)} \wedge \dots \wedge C^{(m)}$, formula 3-CNF sempre soddisfatta per questi casi di b' ;
- esiste un assegnamento b'' che pone $z = \text{T}$ e almeno un letterale per ogni clausola necessariamente a F. In questo caso non possiamo essere certi che per ogni clausola ci sia sempre un altro letterale T, necessario per 3-SAT. Per l'osservazione vista sopra però, basta prendere \bar{b}'' per ricondurci a b' . \square

2.10.2 Riduzione NAE-4-SAT \preceq NAE-3-SAT

Sia $\phi(x_1 \dots x_n) \in \mathcal{I}(\text{NAE-4-SAT})$ una formula 4-CNF $\phi(x_1 \dots x_n) = C^{(1)} \wedge \dots \wedge C^{(m)}$ dove ogni clausola è $C^{(i)} = \ell_1^{(i)} \vee \ell_2^{(i)} \vee \ell_3^{(i)} \vee \ell_4^{(i)}$.

Desideriamo trasformare ϕ in una formula ψ 3-CNF, quindi ϕ sarà NAE-soddisfacibile se e solo se ψ sarà NAE-soddisfacibile. Definiamo la formula 3-CNF

$$\begin{aligned} \psi(x_1 \dots x_n, z) &= D_1^{(1)} \wedge D_2^{(1)} \wedge \dots \wedge D_1^{(m)} \wedge D_2^{(m)} \\ \text{dove } D_1^{(i)} \wedge D_2^{(i)} &= (\ell_1^{(i)} \vee \ell_2^{(i)} \vee z_i) \wedge (\bar{z}_i \vee \ell_3^{(i)} \vee \ell_4^{(i)}). \end{aligned}$$

costruibile in polytime nella taglia di ϕ (basta scorrere ϕ spezzando con z_i ogni clausola).

Dimostrazione. Vediamo ora la doppia implicazione:

\Rightarrow se ϕ è NAE-soddisfacibile allora ψ è NAE-soddisfacibile, cioè esiste un assegnamento a che per ogni clausola $C^{(i)}$ rende almeno un letterale T e almeno un letterale F. Visto che non sappiamo come vengono ripartiti i due letterali tra $D_1^{(i)}$ e $D_2^{(i)}$, dobbiamo procedere per casi:

- se $D_1^{(i)}$ contiene il letterale T e $D_2^{(i)}$ contiene il letterale F, allora basta porre $z_i = \text{F}$, così da rendere $D_1^{(i)}$ e $D_2^{(i)}$ NAE-soddisfatte;
- se al contrario $D_1^{(i)}$ contiene il letterale F e $D_2^{(i)}$ contiene il letterale T, allora basta porre $z_i = \text{T}$, così da rendere $D_1^{(i)}$ e $D_2^{(i)}$ NAE-soddisfatte;
- se $D_1^{(i)}$ contiene sia il letterale T che quello F, allora controllo $D_2^{(i)}$:
 - ★ se $\ell_3^{(i)}$ e $\ell_4^{(i)}$ hanno lo stesso valore T, basta imporre $\bar{z}_i = \text{F}$;
 - ★ se $\ell_3^{(i)}$ e $\ell_4^{(i)}$ hanno lo stesso valore F, basta imporre $\bar{z}_i = \text{T}$;
 - ★ se $\ell_3^{(i)}$ e $\ell_4^{(i)}$ hanno valore uno T e l'altro F, allora z_i ha valore qualsiasi;
- se al contrario $D_2^{(i)}$ contiene sia il letterale T che quello F, è reciproco al precedente.

\Leftarrow se ψ è NAE-soddisfacibile allora ϕ è NAE-soddisfacibile, quindi esiste un assegnamento a tale che ogni clausola 3-CNF di ψ ha almeno un letterale T e un letterale F. Ovviamente, basta aggiungere una variabile z ad ogni clausola impostata a un valore qualsiasi per ottenere clausole 4-CNF. \square

2.10.3 Riduzione NAE-3-SAT \preceq 3-COL

Dobbiamo trovare una mappatura da una formula ϕ 3-CNF a un grafo G 3-COL. Per farlo, generiamo un vertice v al quale colleghiamo tutte le coppie di variabili x_i, \bar{x}_i collegate anche tra loro (per convenienza questi sottografi vengono chiamati *triangoli variabile*); poi, per ogni clausola 3-CNF, costruiamo un sottografo a triangolo aggiungendo un vertice per ogni variabile, (per convenienza questi sottografi vengono chiamati *triangoli clausola*), e per ognuno di questi vertici aggiungiamo un arco al vertice del negato della sua variabile, presente tra i *triangoli variabile*. In questo modo, dato N il numero di letterali e M il numero di clausole, otteniamo un grafo G con $|V| = 1 + 2N + 3M$ e $|E| = 3N + 3M + 3M$, quindi costruibile in polytime sulla taglia di ϕ . Dobbiamo ora dimostrare la doppia implicazione della mappatura:

Dimostrazione. ϕ è NAE-soddisfacibile se e solo se G è 3-COL.

\Rightarrow se ϕ è NAE-soddisfacibile allora G è 3-COL, quindi esiste un assegnamento a che NAE-soddisfa ϕ . In base ai valori di a , descriviamo la seguente colorazione: al vertice v viene dato il colore **3** mentre ad ogni vertice x_i e \bar{x}_i riceve il colore **1** se sono T o **2** se sono F, quindi

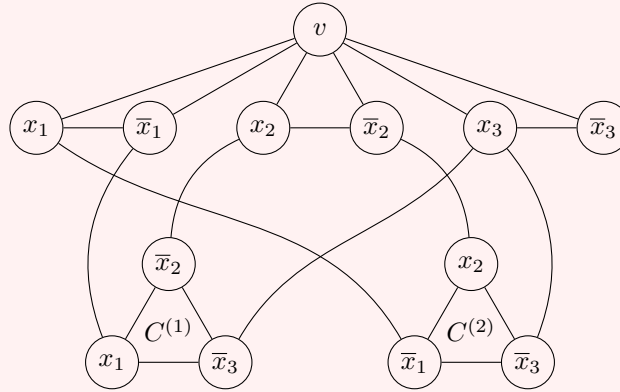
$$\begin{aligned} c(v) &= \mathbf{3} \\ c(V(\ell)) &= \begin{cases} \mathbf{1} & \text{se } \ell = \text{T} \\ \mathbf{2} & \text{se } \ell = \text{F} \end{cases} \end{aligned}$$

e così facendo ogni vertice verrà colorato. Notiamo che ogni arco che collega un *triangolo variabile* ad un *triangolo clausola* sarà propriamente colorato, poiché ognuno di questi archi va da un vertice di una variabile al vertice del suo negato. Dato che però in ogni clausola 3-CNF due letterali hanno lo stesso valore T o F, nei corrispettivi vertici dei *triangoli clausola* del grafo ci saranno due vertici adiacenti con lo stesso colore: il problema è risolto colorando uno qualsiasi dei due vertici usando il colore **3**;

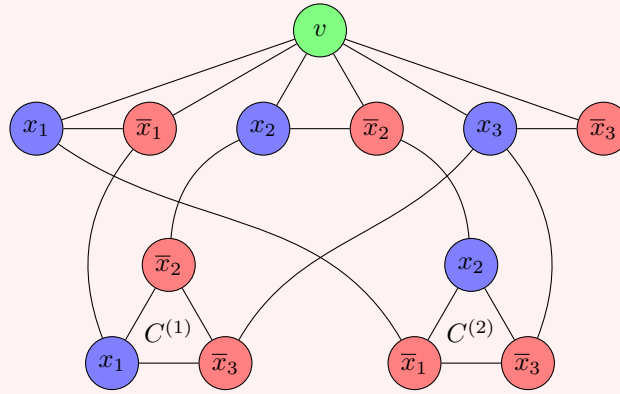
\Leftarrow se G è 3-COL allora ϕ è NAE-soddisfacibile, quindi esiste una 3-COL propria dei vertici del grafo. Supponendo $c(v) = \mathbf{1}$, mappiamo $\mathbf{2} \rightarrow \text{T}$ e $\mathbf{3} \rightarrow \text{F}$ e data la particolare costruzione del grafo, non ci saranno conflitti negli archi: in questo modo, andiamo semplicemente a prendere per ogni nodo la sua variabile e il suo colore, per poi creare l'assegnamento a , come descritto sopra dal mapping. \square

Esempio

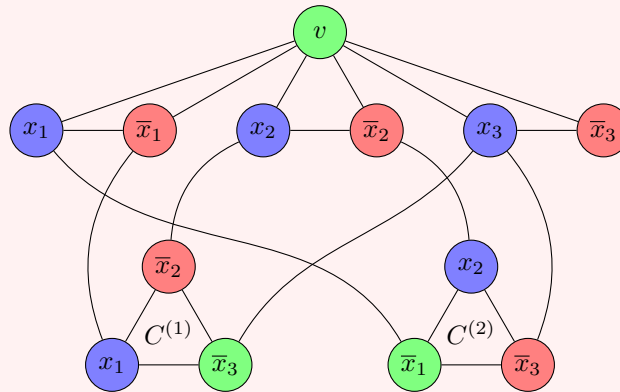
Data la formula $\phi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$, costruiamo il grafo G :



Ora, l'assegnamento $a = (T, T, T)$ NAE-soddisfa ϕ , e quindi esiste una 3-COL per G . Seguiamo il procedimento descritto sopra:

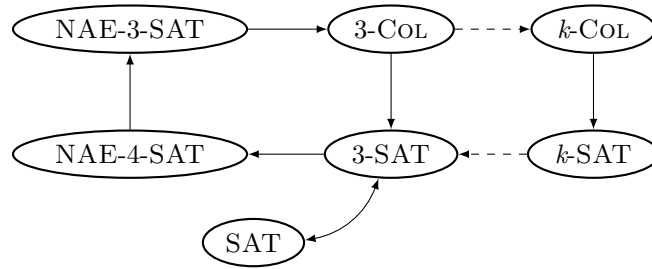


e notiamo che tutti gli archi sono propriamente colorati tranne quelli dei triangoli clausola, in particolare (\bar{x}_2, \bar{x}_3) e (\bar{x}_1, \bar{x}_3) . Scegliamo ora arbitrariamente per ognuno di questi archi uno dei due vertici e coloriamolo usando il colore di v ; ora G è 3-COL.



2.11 Schema attuale delle riduzioni

Notiamo che $\text{SAT} \equiv 3\text{-SAT} \equiv 3\text{-COL}$. Le riduzioni dirette intermedie $k\text{-COL} \preceq (k+1)\text{-COL}$ e $k\text{-SAT} \preceq (k+1)\text{-SAT}$ sono state lasciate implicite.



Ora la relazione tra questi problemi è più forte, infatti sono tutti equivalenti tra loro e basta che uno sia **P** affinché tutti gli altri siano **P**, o viceversa **NP**. Per ora sappiamo che $3\text{-COL} \in \mathbf{NP}$, quindi tutti i problemi qui sopra sono **NP**, ma nulla vieta che possa essere **P**. Quindi raggruppiamo questi problemi nell'insieme

$$S = \{k\text{-COL} \mid k \geq 3\} \cup \{k\text{-SAT} \mid k \geq 3\} \cup \{\text{SAT}\}$$

e per ora inseriamoli nella categoria **NP**. Cercheremo da ora in poi di cercare problemi **P** da aggiungere all'insieme S , così da dimostrare che tutti i problemi in S sono in realtà **P** (però falliremo).

3 | Problemi NP-COMPLETI, co-NP e co-NP-COMPLETI

3.1 Problemi NPC

Definiamo la classe dei problemi **NP-COMPLETI** come

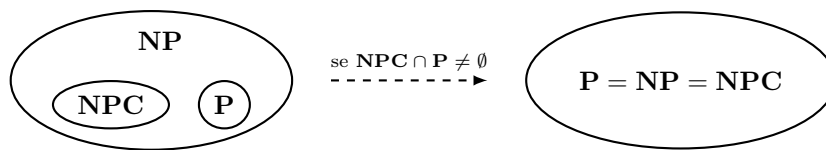
$$\mathbf{NPC} = \{A \mid A \in \mathbf{NP} \wedge \forall B \in \mathbf{NP}, B \preceq A\}$$

ovvero l'insieme dei problemi **NP** ai quali tutti i problemi **NP** si riducono. In pratica, tutti i problemi **NPC** rappresentano il nucleo più difficile dei problemi **NP**, poiché ogni **NP** si mappa a un **NPC**.

Osservazione: Se $\mathbf{NPC} \cap \mathbf{P} \neq \emptyset$, cioè esiste un problema **NPC** in **P**, allora $\mathbf{NP} = \mathbf{P}$.

Dimostrazione. Supponiamo che esista $A \in \mathbf{NPC} \cap \mathbf{P}$ (cioè A è in **NPC** e anche in **P**), allora per definizione $\forall B \in \mathbf{NP}, B \preceq A$ e quindi $B \in \mathbf{P}$. Questo significa che $\mathbf{NP} \subseteq \mathbf{P}$, e dato che $\mathbf{P} \subseteq \mathbf{NP}$, otteniamo $\mathbf{P} = \mathbf{NP}$. \square

Per quanto visto finora, **NP** contiene gli insiemi disgiunti **P** ed **NPC** e se anche solo un problema **NPC** risultasse essere **P**, allora si collapserebbe a $\mathbf{P} \equiv \mathbf{NP} \equiv \mathbf{NPC}$.



3.2 Problema CIRCUIT-SAT

Il problema CIRCUIT-SAT è un problema SAT in termini di *circuiti booleani*.

CIRCUIT-SAT

INPUT un circuito booleano C

OUTPUT **yes** \iff esiste un assegnamento x per C tale che $C(x) = 1 \equiv \mathbf{T}$

Circuito booleano

Un **circuito booleano** (con n input e 1 output) è un grafo aciclico diretto (albero) con n sorgenti (nodi con soli archi uscenti) ed 1 pozzo (nodo con solo archi entranti); ogni nodo non sorgente ha 1 o 2 archi entranti e 1 arco uscente ed è etichettato con un operatore booleano **AND** \wedge , **OR** \vee o **NOT** \neg .

Dato un assegnamento $x = (x_1 \dots x_n)$ ai nodi sorgente, la funzione $val(x)$ associa

- ad ogni nodo sorgente il suo valore assegnato;
- ad ogni nodo non sorgente il risultato del suo operatore applicato ai valori dei nodi a lui collegati.

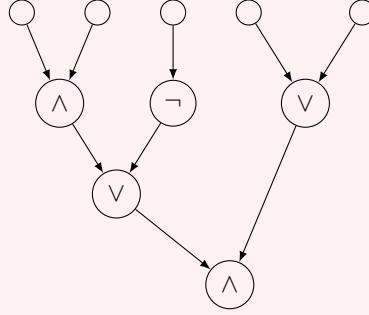
La funzione $C(x)$ calcolata da C associa a un assegnamento $x \in \{0, 1\}^n \equiv \{\mathbf{F}, \mathbf{T}\}^n$ il valore $val(t) \in \{0, 1\} \equiv \{\mathbf{F}, \mathbf{T}\}$ del nodo pozzo t .

La taglia del circuito C è il numero di nodi non sorgenti (dei “nodi porta logica”).

Osservazione: Dato un qualsiasi problema $\mathbb{A} \in \mathbf{P}$, esiste una famiglia di circuiti $\{C_n\}_{n \in \mathbb{N}}$ in cui ogni circuito C_n è costruibile in polytime $O(n^d)$ e ha taglia polinomiale $|C_n| = O(n^d)$ per una certa costante d . Ogni circuito della famiglia è tale che $\forall x \in \mathcal{I}(\mathbb{A})$ tale che $|x| = n$ si ha $\mathbb{A}(x) = \text{yes} \iff C_n(x) = 1$. In pratica per ogni taglia degli input di \mathbb{A} possiamo costruire un circuito che simula l'esecuzione di \mathbb{A} su tutti gli input di quella taglia. Possiamo fare un'analogia sul come gli algoritmi software vengono tradotti in comandi hardware sulla circuiteria del processore.

Esempio

In circuito C mappa la formula $\phi = ((x_1 \wedge x_2) \vee \bar{x}_3) \wedge (x_4 \vee x_5)$.



3.3 Esistenza dei problemi NPC

Per adesso non abbiamo ancora visto un problema **NPC**, quindi non sappiamo se effettivamente questi esistano o meno. Abbiamo introdotto un problema nuovo, per in quale esiste un teorema che lo dimostra **NPC**.

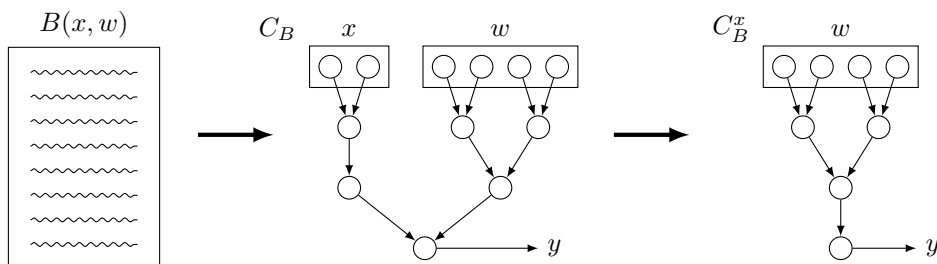
Teorema di Cook-Levin

Il problema CIRCUIT-SAT è **NPC**.

Dimostrazione. Utilizziamo la definizione di **NPC**: dobbiamo dimostrare che

- $\text{CIRCUIT-SAT} \in \mathbf{NP}$, e quindi fornire
 - un certificato, che in questo caso è banalmente un input x tale che $C(x) = 1$;
 - un verificatore, il quale calcola livello per livello a partire dai vertici sorgente il valore dei nodi, fino al pozzo t ; se $\text{val}(t) = 1$ ritornerà **T**, al contrario **F**. Questo è fattibile, scorrendo ogni vertice ed ogni nodo, in polytime $O(|V(C)| + |E(C)|) = O(|V(C)|) = O(|C|)$;
- $\forall \mathbb{A} \in \mathbf{NP}$, $\mathbb{A} \preceq \text{CIRCUIT-SAT}$, e quindi dobbiamo fornire un algoritmo polinomiale in $x \in \mathcal{I}(\mathbb{A})$ che trasforma x in un circuito C tale che $\mathbb{A}(x) = \text{yes} \iff \exists w \mid C(w) = 1$.

Prendiamo $\mathbb{A} \in \mathbf{NP}$, per il quale quindi esiste un verificatore $B(\cdot, \cdot)$ e una costante c tale che $\forall x \in \mathcal{I}(\mathbb{A})$, $\mathbb{A}(x) = \text{yes} \iff \exists w \in \{0, 1\}^{|x|^c}$ t.c. $B(x, w) = \text{yes}$. L'idea è di trasformare B in un circuito usando il lemma visto sopra: data $|x|$, poiché B è polytime in $|x|$ per definizione, esiste un circuito C_B , costruibile in polytime di $|x|$ e di taglia $\text{poly}(|x|)$, tale che $C_B(x, w) = B(x, w)$. Ora fisso l'input x in $C_B(x, w)$, creando un nuovo circuito $C_B^x(w)$ tale che $\forall w$, $C_B^x(w) = C_B(x, w) = B(x, w)$ e quindi $C_B^x(w) = 1 \iff B(x, w) = \text{yes} \iff \mathbb{A}(x) = \text{yes}$.



In questo modo dato un qualsiasi problema \mathbb{A} , ogni input $x \in \mathcal{I}(\mathbb{A})$ si mappa ad un circuito $C_B^x \in \mathcal{I}(\text{CIRCUIT-SAT})$ tale che $C_B^x(w) = 1 \iff B(x, w) = 1$. \square

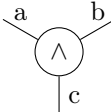
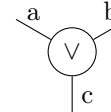

Osservazione: Nella dimostrazione abbiamo ottenuto un risultato ulteriore: dato un qualsiasi problema \mathbb{A} , per trovare il certificato di una certa istanza x basterà trovare l'assegnamento w per il circuito C_B^x mappato ad x tale che $C_B^x(w) = 1$.

Esempio

Dato un grafo G per HAMPATH, per trovare il cammino Hamiltoniano w in G basta trasformare G in un circuito C_B^x e trovare il w tale che $C_B^x(w) = 1$.

3.4 Riduzione CIRCUIT-SAT \preceq SAT

Dobbiamo trovare una mappatura per $C \in \mathcal{I}(\text{CIRCUIT-SAT})$ a una formula $\phi \in \mathcal{I}(\text{SAT})$ tale che $\exists x \mid C(x) = 1 \iff \exists y \mid \phi(y) = \text{T}$. Per costruire la formula ϕ , aggiungiamo un arco uscente al nodo pozzo, che rappresenta il valore finale del circuito C , e associamo ad ogni nodo del circuito una CNF usando il seguente schema:

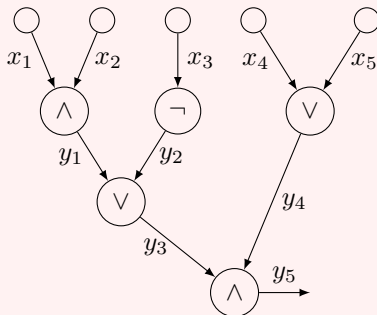
		
$a \wedge b = c$	$a \vee b = c$	$\bar{a} = c$
$\equiv ((\bar{a} \wedge \bar{b}) \vee c) \wedge ((a \wedge b) \vee \bar{c})$	$\equiv ((\bar{a} \vee \bar{b}) \vee c) \wedge ((a \vee b) \vee \bar{c})$	$\equiv ((\bar{a}) \vee c) \wedge ((\bar{a}) \vee \bar{c})$
$\equiv (\bar{a} \vee \bar{b} \vee c) \wedge (a \vee \bar{c}) \wedge (b \vee \bar{c})$	$\equiv ((\bar{a} \wedge \bar{b}) \vee c) \wedge (a \vee b \vee c)$	$\equiv (a \vee c) \wedge (\bar{a} \vee \bar{c})$
	$\equiv (\bar{a} \vee c) \wedge (\bar{b} \vee c) \wedge (a \vee b \vee c)$	

Ora mettiamo in *AND* tutte le formule così ottenute e il valore finale del pozzo: in questo modo abbiamo ottenuto la formula ϕ tale che $\phi(x, y) = \text{T} \iff C(x) = 1$, dove x sono le variabili dei nodi sorgente mentre y sono tutte le variabili prodotte dagli altri nodi.

Siccome $\text{CIRCUIT-SAT} \in \mathbf{NPC}$ si ha che $\forall \mathbb{A} \in \mathbf{NP}$, $\mathbb{A} \preceq \text{CIRCUIT-SAT}$, e dato che $\text{SAT} \in \mathbf{NP}$ e $\text{CIRCUIT-SAT} \preceq \text{SAT}$, abbiamo che $\forall \mathbb{A} \in \mathbf{NP}$, $\mathbb{A} \preceq \text{CIRCUIT-SAT} \preceq \text{SAT}$: abbiamo dimostrato che $\text{SAT} \in \mathbf{NPC}$ e che in particolare tutti i problemi in S sono **NPC**.

Esempio

Usiamo il circuito visto prima e assegniamo i valori agli archi. La formula risultante quindi, seguendo il procedimento descritto sopra, sarà:



$$\begin{aligned}
 \phi(x, y) = & [x_1 \wedge x_2 = y_1] \\
 & \wedge [\bar{x}_3 = y_2] \\
 & \wedge [x_4 \vee x_5 = y_4] \\
 & \wedge [y_1 \vee y_2 = y_3] \\
 & \wedge [y_3 \wedge y_4 = y_5] \\
 & \wedge y_5
 \end{aligned}$$

3.5 Dimostrazioni di NP-COMPLETEZZA

Per dimostrare che un problema $\mathbb{B} \in \mathbf{NPC}$ dobbiamo applicare la definizione di **NPC**, che richiede $\mathbb{B} \in \mathbf{NP}$ e che \mathbb{B} sia **NP-HARD**. Per la seconda condizione possiamo invece trovare $\exists \mathbb{A} \in \mathbf{NPC}$ tale che $\mathbb{A} \preceq \mathbb{B}$, poiché dato che $\forall \mathbb{K} \in \mathbf{NP}$, $\mathbb{K} \preceq \mathbb{A}$ allora se $\mathbb{A} \preceq \mathbb{B}$ per transitività si ha $\forall \mathbb{K}$, $\mathbb{K} \preceq \mathbb{A} \preceq \mathbb{B}$, che è la condizione di **NP-HARNESS** per \mathbb{B} .

Osservazione: Dato un problema \mathbb{B} tale che $\mathbb{A} \preceq \mathbb{B}$, $\mathbb{B} \preceq \mathbb{C}$ con $\mathbb{A}, \mathbb{C} \in \mathbf{NPC}$, vogliamo dimostrare che $\mathbb{B} \in \mathbf{NPC}$. Per farlo, seguiamo la definizione di \mathbf{NPC} , quindi è necessario

- $\mathbb{B} \in \mathbf{NP}$, e vale perché $\mathbb{B} \preceq \mathbb{C}$ e $\mathbb{C} \in \mathbf{NP}$;
- $\forall \mathbb{K} \in \mathbf{NP}$, $\mathbb{K} \preceq \mathbb{B}$ oppure $\exists \mathbb{K} \in \mathbf{NPC}$, $\mathbb{K} \preceq \mathbb{B}$, che vale poiché $\mathbb{A} \preceq \mathbb{B}$ e $\mathbb{A} \in \mathbf{NPC}$.

Vediamo un esempio del metodo per riduzione nella seguente osservazione:

Osservazione: Se $\mathbf{NPC} \cap \mathbf{P} \neq \emptyset$ allora ogni problema \mathbf{NP} non banale (cioè che ha istanza **yes** e istanza **no**) è \mathbf{NPC} .

Dimostrazione. Se $\mathbb{A} \in \mathbf{NPC} \cap \mathbf{P}$ allora per la definizione di \mathbf{P} esiste un algoritmo polinomiale A tale che $\forall x \in \mathcal{I}(\mathbb{A})$, $A(x) = \mathbb{A}(x)$. Prendiamo un qualsiasi problema $\mathbb{B} \in \mathbf{NP}$ non banale (cioè tale che per $x_1, x_2 \in \mathcal{I}(\mathbb{B})$, $\mathbb{B}(x_1) = \mathbf{yes}$, $\mathbb{B}(x_2) = \mathbf{no}$) e dimostriamo

- $\mathbb{B} \in \mathbf{NP}$, per definizione di \mathbb{B} ;
- $\forall \mathbb{K} \in \mathbf{NP}$, $\mathbb{K} \preceq \mathbb{B}$ oppure $\exists \mathbb{K} \in \mathbf{NPC}$, $\mathbb{K} \preceq \mathbb{B}$, quindi riduciamo $\mathbb{A} \preceq \mathbb{B}$. La funzione di riduzione $R : \mathcal{I}(\mathbb{A}) \rightarrow \mathcal{I}(\mathbb{B})$ mappa secondo la seguente definizione

$$\forall y \in \mathcal{I}(\mathbb{A}), R(y) = \begin{cases} x_1 & \text{se } A(y) = \mathbf{yes} \\ x_2 & \text{se } A(y) = \mathbf{no} \end{cases}$$

ed è polytime poiché il risolutore A è polytime per definizione. \square

Esempio

Se $\mathbf{P} \cap \mathbf{NPC} \neq \emptyset$ allora il problema non banale BIT-1, che prende in input un bit b e ritorna **yes** se e solo se $b = 1$, è in \mathbf{NPC} . La dimostrazione è immediata da quella vista sopra, dove x_1 e x_2 saranno gli unici valori binari di BIT-1.

3.6 Problemi co-NP

Osserviamo che la definizione di \mathbf{NP} è asimmetrica rispetto alle istanze **yes** e **no**, infatti è definito per problemi che hanno certificatori che rispondono **yes** per istanze **yes**; per le istanze **no** invece dobbiamo procedere per bruteforce. Per ogni problema di decisione \mathbb{A} è però possibile creare il problema opposto $\overline{\mathbb{A}}$, ovvero

$$\overline{\mathbb{A}}(x) = \mathbf{yes} \iff \mathbb{A}(x) = \mathbf{no}.$$

Data una classe \mathcal{A} di problemi decisionali, definiamo la classe dei problemi opposti

$$\text{co-}\mathcal{A} = \{\mathbb{A} \mid \overline{\mathbb{A}} \in \mathcal{A}\}$$

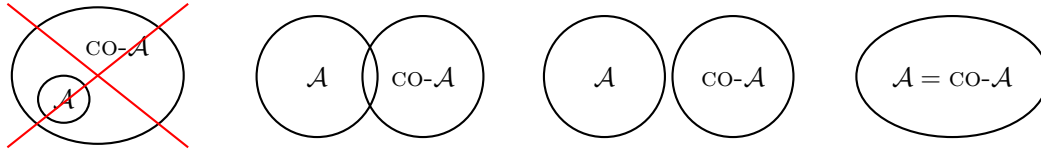
Esempio

Se scopriremo che $\text{HAMPATH} \in \mathbf{P}$ (“è vero che G è Hamiltoniano?”) implicherebbe che $\overline{\text{HAMPATH}} \in \text{co-}\mathbf{P}$ (“è vero che G non è Hamiltoniano?”).

Quello che ci chiediamo è: se giro la domanda, cosa succede al problema? Cioè, domandare l’esistenza è più o meno difficile rispetto a domandare la non esistenza?

Osservazione: Se \mathcal{A} classe di problemi decisionali è tale che $\mathcal{A} \subseteq \text{co-}\mathcal{A}$ allora $\mathcal{A} = \text{co-}\mathcal{A}$. Infatti, se prendiamo un problema $\mathbb{A} \in \text{co-}\mathcal{A}$ sappiamo che $\overline{\mathbb{A}} \in \mathcal{A} \subseteq \text{co-}\mathcal{A}$ per ipotesi e quindi $\overline{\overline{\mathbb{A}}} \in \text{co-}\mathcal{A}$, da cui $\mathbb{A} \in \mathcal{A}$. Per definizione $\mathbb{A} \in \text{co-}\mathcal{A}$, quindi $\text{co-}\mathcal{A} \subseteq \mathcal{A}$, il che implica $\mathcal{A} = \text{co-}\mathcal{A}$.

Da questo ne deriva che non è possibile avere singolarmente $\mathcal{A} \subseteq \text{co-}\mathcal{A}$ oppure $\text{co-}\mathcal{A} \subseteq \mathcal{A}$, e infatti al massimo \mathcal{A} e $\text{co-}\mathcal{A}$ possono essere insiemi disgiunti, equivalenti o intersecati.



3.7 Dimostrazione $P = CO-P$

Prendiamo $A \in P$ e il suo complemento $\bar{A} \in CO-P$. Sappiamo che per A esiste un algoritmo A tale che $A(x) = \mathbb{A}(x)$, e per \bar{A} definisco l'algoritmo B che semplicemente computa $A(x)$ e nega il suo risultato. B risolve in polytime il problema decisionale \bar{A} e quindi $\bar{A} \in P$. Ovviamente ora, come visto sopra, scriviamo $\bar{A} \in P \implies A \in CO-P$, ma dato che $A \in P$, si ha che $CO-P = P$.

Da questo ne deriva $NP \cap CO-NP \neq \emptyset$, infatti dato che $P \subset NP$ allora $CO-P \subseteq CO-NP$, e siccome $P = CO-P$ si avrà $P \subset NP \cap CO-NP$. In particolare, non si sa se P copre per intero l'intersezione oppure se è un sottoinsieme stretto; inoltre, non sappiamo ancora se $NP = CO-NP$, quindi abbiamo quattro scenari:



Osservazione: $A \preceq B \iff \bar{A} \preceq \bar{B}$. Infatti, se $A \preceq B$ allora esiste una riduzione polytime R tale che $A(x) = \text{yes} \iff B(R(x)) = \text{yes}$, e quindi negando $A(x) = \text{no} \iff B(x) = \text{no}$ ottengo $\bar{A} = \text{yes} \iff \bar{B} = \text{yes}$ e posso quindi ridurre $\bar{A} \preceq \bar{B}$ utilizzando sempre R .

3.8 Dimostrazione $NP \stackrel{?}{=} CO-NP$

Osservazione: $NP = CO-NP \iff \exists A \in NPC \cap CO-NP$.

Dimostrazione. Vediamo ambi i lati della implicazione

\Rightarrow se $NP = CO-NP$ allora $SAT \in CO-NP$ (e già è NPC);

\Leftarrow supponiamo che esista $A \in NPC \cap CO-NP$ e prendiamo un problema $B \in CO-NP$. Abbiamo che $B \in NP \implies B \preceq A \implies B \preceq \bar{A}$, poiché $A \in NPC$. Dato che $A \in CO-NP$, si ha che $\bar{A} \in NP$ e quindi $B \in NP$ per la riduzione precedente. Dato che $CO-NP \subseteq NP$ si ha che $CO-NP = NP$. \square

A questo punto resta ancora da verificare che $NPC \preceq P$, ma crediamo di no.

3.9 “Perché” si crede $NP \neq CO-NP$

La logica dietro ad NP è intrinsecamente semplice, infatti riscrivendo la definizione di NP in forma logica come

$$NP = \{A(x) = \exists w \in \{0,1\}^{\text{poly}(|x|)} \text{ t.c. } B(x,w) \in P\}$$

quindi si può dire che un problema è NP se esiste un certificato per ogni sua istanza yes . NP è quindi un problema esistenziale. Al contrario abbiamo

$$CO-NP = \{\bar{A}(x) \mid A(x) \in NP\} = \{\forall w \in \{0,1\}^{\text{poly}(|x|)} \text{ t.c. } \bar{B}(x,w) \in CO-P = P\}$$

dove compare il quantificatore \forall .

Se $NP = CO-NP$ allora il predicato $\forall w \in \{0,1\}^{\text{poly}(|x|)} \text{ t.c. } \bar{B}(x,w)$ che descrive $CO-NP$ deve poter descrivere anche NP , cioè si deve soddisfare

$$\forall w \in \{0,1\}^{\text{poly}(|x|)} \text{ t.c. } \bar{B}(x,w) = \exists w' \in \{0,1\}^{\text{poly}(|x|)} \text{ t.c. } B'(x,w')$$

quindi devo essere in grado di trasformare un predicato universale \forall in un predicato esistenziale \exists , che è difficile da credere dal punto di vista logico. In soldoni, per dire che un grafo non ha un cammino Hamiltoniano devo mostrare che ogni possibile cammino non è Hamiltoniano.

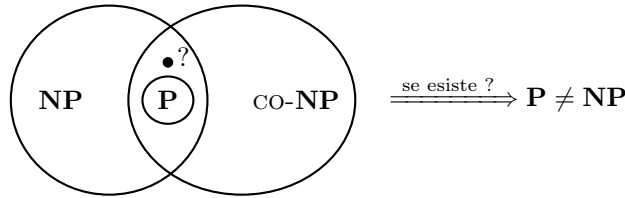
I problemi in **NP** sono problemi per i quali è facile verificare la risposta **yes**, mentre i problemi in **co-NP** sono problemi per i quali è facile verificare la risposta **no**, per questo crediamo $\mathbf{NP} \neq \mathbf{co-NP}$. Nonostante le difficoltà, nel caso in cui $\mathbf{NP} = \mathbf{co-NP}$, mancherebbe ancora da dimostrare se $\mathbf{P} = \mathbf{NP}$, che è il nucleo del nostro dilemma: “verificare è facile come risolvere?”.

Osservazione: Dato che $\mathbf{P} = \mathbf{co-P}$, se $\mathbf{NP} \neq \mathbf{co-NP} \implies \mathbf{P} \neq \mathbf{NP}$, quindi dimostrare $\mathbf{NP} \neq \mathbf{co-NP}$ è un altro approccio alla dimostrazione che $\mathbf{P} \neq \mathbf{NP}$.

3.10 Stato attuale delle classi

I problemi in **P** sono simmetrici rispetto a **yes** e **no**, mentre i problemi in **NP** sono definiti solo se hanno certificato **yes**, poiché rispondono alla domanda esistenziale “esiste un certificato?”. la difficoltà del rendere simmetrica la definizione di **NP** risiede nel fatto che non sempre è possibile trasformare un’espressione che impatta su un numero esponenziale di certificati in una che impatta su un solo certificato.

Cerchiamo un altro approccio per avvicinarci a $\mathbf{P} \neq \mathbf{NP}$: per dimostrare che **P** non può espandersi ad **NP** basta trovare un problema in $\mathbf{NP} \cap \mathbf{co-NP}$ ma non in **P**.



3.11 Problema SMALLFACTOR

SMALLFACTOR

INPUT un intero N e un intero k

OUTPUT $\text{yes} \iff N$ ha un fattore primo $\leq k$.

$\text{SMALLFACTOR} \in \mathbf{NP}$, infatti basta dare un fattore primo q (certificato) di N , e il verificatore farà: ① check $q \leq k$, ② check N/q non ha resto e ③ check q sia primo.

3.12 Dimostrazione $\text{SMALLFACTOR} \in \mathbf{co-NP}$

Diamo un verificatore che dimostri che un intero N non ha un fattore $\leq k$. Il certificato è la scomposizione in fattori primi di N , ovvero l’insieme $q_1 \dots q_t$ di fattori e $\ell_1 \dots \ell_t$ di esponenti, che è $\text{poly}(|N|)$. Il verificatore farà:

1. check $q_1 \dots q_t$ siano primi [fattibile in polytime];
2. check $q_1^{\ell_1} \times \dots \times q_t^{\ell_t} = N$;
3. check $\min_i q_i > k$.

La taglia dell’istanza per SMALLFACTOR è il numero di bit per scrivere N e k , quindi $|N| = \log N$ e siccome $k \leq N$ possiamo trascurare $|k|$, quindi le istanze avranno taglia $|N|$. Resta da mostrare che il certificato sia $\text{poly}(|N|) = \text{poly}(\log N) = O((\log N)^c)$: ogni fattore è minore di N , quindi $q_i \leq N \implies |q_i| \leq \log N$, mentre ogni esponente è al massimo $\log N$, infatti $q_i^{\ell_i} \leq N \implies \ell_i \log q_i \leq \log N$; il certificato è quindi $\text{poly}(|N|)$ e in conclusione $\text{SMALLFACTOR} \in \mathbf{co-NP}$. \square

3.13 “Perché” si crede $\mathbf{P} \subset \mathbf{NP} \cap \mathbf{co-NP}$

Con dimostrazione appena vista abbiamo che $\text{SMALLFACTOR} \in \mathbf{NP} \cap \mathbf{co-NP}$ e per adesso non siamo ancora stati in grado di trovare una soluzione polinomiale per questo problema, quindi per ora $\text{SMALLFACTOR} \notin \mathbf{P}$: vediamo che si pongono le condizioni per dimostrare che esiste un problema in $(\mathbf{NP} \cap \mathbf{co-NP}) \setminus \mathbf{P}$, e quindi per quanto visto nel capitolo precedente basterebbe dimostrare $\text{SMALLFACTOR} \notin \mathbf{P}$ per avere $\mathbf{P} \neq \mathbf{NP}$.

3.14 Problemi co-NPC

Per la relazione $\mathbb{A} \preceq \mathbb{B} \iff \overline{\mathbb{A}} \preceq \overline{\mathbb{B}}$, nella classe **co-NP** possiamo identificare dei problemi analoghi a quelli **NPC**, ovvero i problemi **co-NP-COMPLETI**.

Per definizione di **NPC**, se $\mathbb{A} \in \mathbf{NPC}$ allora $\mathbb{A} \in \mathbf{NP}$ e $\forall \mathbb{B} \in \mathbf{NP}, \mathbb{B} \preceq \mathbb{A}$; diciamo che un problema è **co-NPC** se semplicemente prendendo l'opposto della definizione di **NPC**:

$$\mathbf{co-NPC} = \{\overline{\mathbb{A}} \mid \overline{\mathbb{A}} \in \mathbf{co-NP} \wedge \forall \overline{\mathbb{B}} \in \mathbf{co-NP}, \overline{\mathbb{B}} \preceq \overline{\mathbb{A}}\}.$$

Dato un qualsiasi problema $\mathbb{A} \in \mathbf{NPC}$, abbiamo quindi che $\overline{\mathbb{A}} \in \mathbf{co-NPC}$.

Osservazione: $\mathbf{NP} = \mathbf{co-NP} \iff \exists \mathbb{A} \in \mathbf{co-NPC} \cap \mathbf{NP}$, che è un corollario dell'osservazione di qualche capitolo precedente riguardo $\mathbf{NPC} \cap \mathbf{co-NP}$.

3.15 Problema TAU

Tautologia

INPUT una formula ϕ

OUTPUT **yes** $\iff \phi$ è un tautologia, ovvero una formula che per ogni assegnamento a delle variabili è sempre T.

Dimostrazione. Per provare $\mathbf{TAU} \in \mathbf{co-NPC}$, come facciamo per **NPC**, proviamo che

- $\mathbf{TAU} \in \mathbf{co-NP}$, fattibile in polytime provando che $\overline{\mathbf{TAU}} \in \mathbf{NP}$: devo dare
 - un certificato a che rende falsa ϕ ;
 - un verificatore che semplicemente valuta $\phi(a)$;
- $\forall \mathbb{B} \in \mathbf{co-NP}, \mathbb{B} \preceq \mathbf{TAU}$ fattibile anche riducendo $\mathbf{SAT} \preceq \overline{\mathbf{TAU}}$, sapendo $\mathbf{SAT} \in \mathbf{NPC}$. Data una formula $\psi \in \mathcal{I}(\mathbf{SAT})$ la riduzione è semplicemente la negazione della formula, quindi $\phi = \overline{\psi}$. La formula ψ è soddisfacibile se e solo se esiste un assegnamento a che rende ψ vera, e che quindi rende per definizione ϕ falsa. $\mathbf{SAT}(\psi) = \mathbf{yes}$ se e solo se ϕ non è una tautologia, cioè se $\overline{\mathbf{TAU}}(\phi) = \mathbf{yes}$. \square

3.16 Stato attuale delle classi

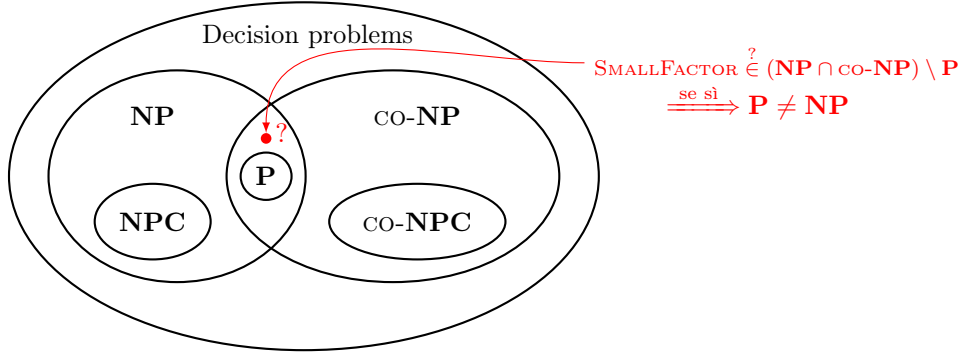
Per i problemi **P** possiamo risolvere in modo facile istanze **yes** e **no**, per i problemi **NP** possiamo solo certificare in modo facile istanze **yes**, ma non per tutti è anche facile certificare istanze **no**, per questo esistono i problemi **co-NP**, per i quali è possibile certificare in modo facile istanze **no**. In **NP** alcuni problemi appaiono “non più semplici” degli altri, e questi sono chiamati problemi **NPC**; lo stesso avviene per i problemi **co-NP**, che quindi avranno un nucleo di problemi “non più semplici” degli altri detti **co-NPC**.

Per dimostrare $\mathbf{P} \neq \mathbf{NP}$ possiamo usare più metodi:

- dimostrare che $\exists \mathbb{A} \in (\mathbf{NP} \cap \mathbf{co-NP}) \setminus \mathbf{P}$ (come per **SMALLFACTOR**); oppure
- dimostrare che $\mathbf{NP} \neq \mathbf{co-NP}$ (così da obbligare **P** a stare nell'intersezione), dimostrando che $\mathbf{NPC} \cap \mathbf{co-NP} = \emptyset$ o viceversa $\mathbf{co-NPC} \cap \mathbf{NP} = \emptyset$.

Al contrario, per dimostrare $\mathbf{P} = \mathbf{NP}$ possiamo

- dimostrare che $\exists \mathbb{A} \in \mathbf{P} \cap \mathbf{NPC}$ o viceversa $\exists \mathbb{A} \in \mathbf{P} \cap \mathbf{co-NPC}$; oppure
- dimostrare che $\mathbf{NP} = \mathbf{co-NP}$, dimostrando che $\exists \mathbb{A} \in \mathbf{NPC} \cap \mathbf{co-NP}$ o viceversa $\exists \mathbb{A} \in \mathbf{co-NPC} \cap \mathbf{NP}$.



Inoltre, introduciamo brevemente una nozione usata finora usata ma mai approfondita nel perché: la **hardness**. Ogni volta che abbiamo dato la definizione di *completezza* di una classe \mathcal{A} di problemi, affinché un problema \mathbb{A} fosse completo in \mathcal{A} , cioè $\mathbb{A} \in \mathcal{AC}$, c'erano due clausole da soddisfare: ① $\mathbb{A} \in \mathcal{A}$, ② $\forall \mathbb{B} \in \mathcal{A}, \mathbb{B} \preceq \mathbb{A}$. La seconda clausola è la condizione di **\mathcal{A} -hardness**, che rende un problema di una certa classe \mathcal{A} più difficile rispetto agli altri problemi in quella classe, poiché ogni altro problema in quella classe è a lui riducibile.

3.17 Problema D-HAMCYCLE

D-HAMCYCLE

INPUT un grafo diretto $G = (V, E)$

OUTPUT $\text{yes} \iff G$ ha un ciclo diretto che tocca ogni vertice una sola volta

3.18 Dimostrazione D-HAMCYCLE \in NPC

Come al solito, dobbiamo provare che D-HAMCYCLE \in NP (fatto nel primo capitolo) e che ogni problema NP si riduce a D-HAMCYCLE, oppure alternativamente (e in modo più semplice) che vale 3-SAT \preceq D-HAMCYCLE, perché 3-SAT \in NPC.

Ci soffermiamo solo sul secondo punto: dobbiamo dimostrare che esiste un mapping da una formula $\phi(x_1 \dots x_n)$ 3-CNF a un grafo diretto $G = (V, E)$ e che ϕ è soddisfacibile se e solo se G ha un ciclo Hamiltoniano. Utilizziamo come esempio operativo la formula

$$\phi(x_1, x_2, x_3) = C1 \wedge C2 \wedge C3 = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

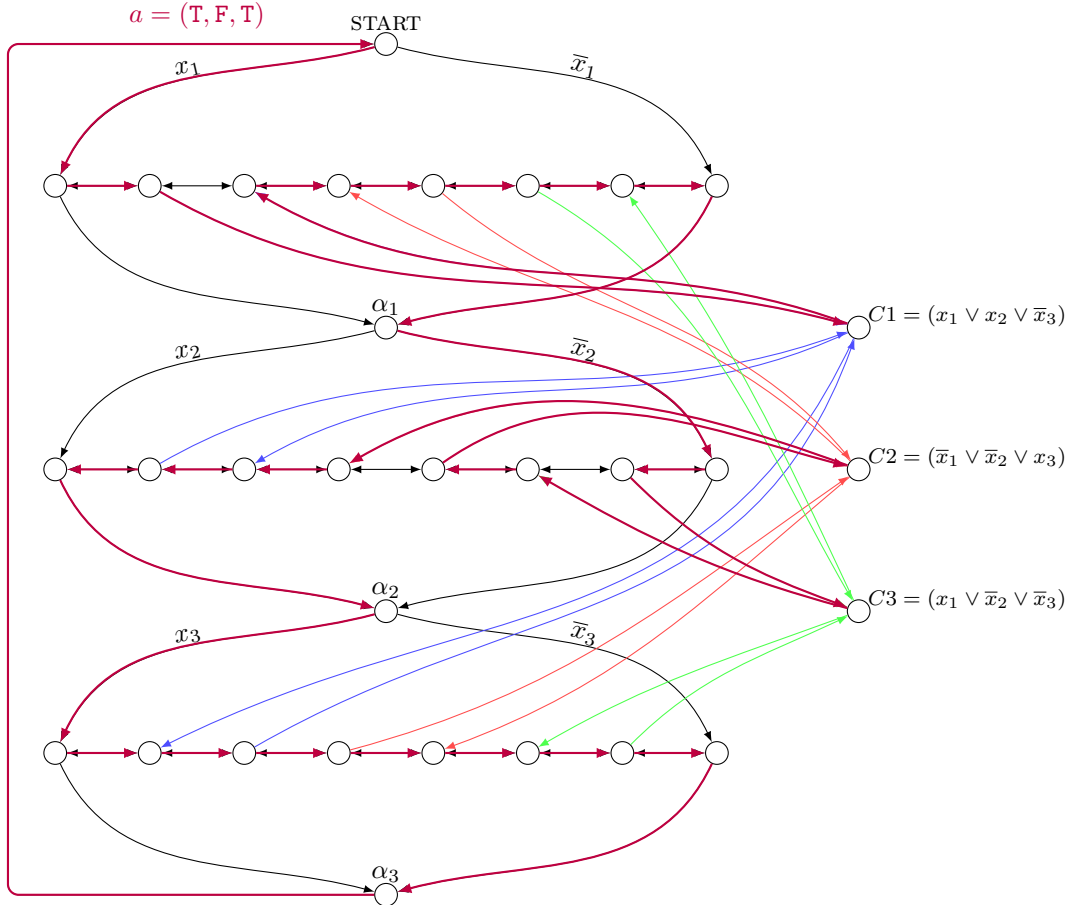
e costruiamo un grafo G per cui ogni assegnamento a tale che $\phi(a) = \text{T}$ permetta di ottenere un ciclo Hamiltoniano in G . Procediamo seguendo i passi:

1. creiamo un nodo iniziale START;
2. creiamo un “segmento” di $2a_i + 2$ nodi affiancati e collegati per ogni variabile x_i , dove a_i è il numero di occorrenze di x_i e \bar{x}_i in ϕ , e un nodo “collo” α_i sotto ogni segmento;
3. colleghiamo il nodo iniziale ai due nodi agli estremi del primo segmento, e poi a partire da questo segmento colleghiamo i nodi agli estremi al nodo “collo” sotto di esso e da questo facciamo partire due nuovi archi per gli estremi del prossimo segmento, andando in pratica a delineare una forma ovale attorno ad ogni segmento;
4. colleghiamo l'ultimo nodo collo al nodo iniziale. In questo modo, a partire dal nodo iniziale avremo due modi di percorrere ogni segmento per avere un cammino Hamiltoniano. Nel nostro esempio avremo quindi 8 cammini, uno per ogni assegnamento a possibile dalle combinazioni delle 3 variabili x_i , che poi si chiudono sul nodo iniziale diventando cicli;
5. aggiungiamo un nodo per ogni clausola C_j della formula ϕ e in base ai letterali contenuti nella clausola aggiungiamo degli archi da e verso i vari segmenti: ogni segmento infatti è stato costruito per avere una coppia di nodi affiancati per ogni clausola, che vengono collegati nel seguente modo:
 - se la clausola contiene una variabile x_i allora nel i -esimo segmento viene collegato il nodo di sinistra al nodo della clausola e quest'ultimo con il nodo di destra;

- se la clausola contiene una variabile \bar{x}_i allora nel i -esimo segmento viene collegato il nodo *di destra* al nodo della clausola e quest'ultimo con il nodo *di sinistra*;
- se la clausola contiene più copie di una variabile x_i o \bar{x}_i , gli archi sono disposti nuovamente come descritto sopra per sulle coppie di nodi successive.

In questo modo, quando ci troviamo in α_{i-1} (o in START), partendo dall'arco x_i si attraversa “da sinistra a destra” il segmento di x_i e si potrà entrare in tutte le clausole che contengono x_i , mentre partendo dall'arco \bar{x}_i si attraversa “da destra a sinistra” il segmento di x_i e si potrà entrare in tutte le clausole che contengono \bar{x}_i . Nel nostro esempio, l'assegnamento $a = (T, F, T)$ rende vera ϕ e si mappa al percorso **viola**, andando a sinistra poiché x_1 è posto T, passando per C_1 *poiché è possibile* e così via.

$$\phi(x_1, x_2, x_3) = C1 \wedge C2 \wedge C3 = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$



Affinché il ciclo sia Hamiltoniano si deve passare per ogni nodo una ed una sola volta: dal punto di vista dei valori di verità, passare per un nodo C_j significa mostrare che c'è almeno una variabile x_i posto a T, che è necessario affinché ϕ sia soddisfatta. Il cammino è costruito arbitrariamente in modo tale che si passi per i nodi clausola *appena possibile*: come vediamo infatti, si poteva toccare C_2 anche attraversando il segmento di x_3 , ma seguendo quanto appena detto lo abbiamo toccato attraversando invece il segmento di x_2 , cioè appena ne abbiamo avuto la possibilità. Con questo abbiamo dimostrato che esiste una mappatura, ma ci manca ancora da verificare che sia polinomiale nella taglia della formula ϕ .

Il grafo G ha taglia $|V| + |E|$; cominciando dai nodi, G contiene: ① un nodo iniziale, ② un segmento di $2 \cdot \text{NUMOCC}(x_i) + 2$ nodi più un aggiuntivo nodo sotto di esso, per ogni variabile x_i , ③ un nodo per ogni clausola C_j (con m numero di clausole), e in totale quindi

$$\begin{aligned} |V| &= 1 + \sum_{i=1}^n ((2 \cdot \text{NUMOCC}(x_i) + 2) + 1) + m \\ &= 1 + ((2 \cdot 3m + 2n) + n) + m = 7m + 3n + 1 = O(|\phi|^c) \end{aligned}$$

quindi $|V| = \text{poly}(|\phi|)$ perché $|\phi|$ è polinomiale nel numero di variabili nelle clausole che la compongono (nel calcolo è stato usato $\sum_{i=1}^n 2 \cdot \text{NUMOCC}(x_i) = 2 \cdot 3m$ poiché la somma di tutte le occorrenze delle variabili x_i e \bar{x}_i in una 3-CNF è 3 volte il numero delle clausole); infine gli archi E sono al massimo $|V|^2$ perché il grafo è diretto, quindi $|E| = \text{poly}(|\phi|)$, e in conclusione G è costruibile in polytime di ϕ . \square

3.19 Problema D-HAMPATH

D-HAMPATH

INPUT un grafo diretto $G = (V, E)$

OUTPUT $\text{yes} \iff G$ ha un cammino che tocca ogni vertice una sola volta

3.20 Dimostrazione D-HAMPATH \in NPC

Seguiamo passo passo la dimostrazione vista sopra ma non colleghiamo l'ultimo nodo "collo" al nodo iniziale: in questo modo non si avrà un ciclo ma semplicemente un cammino. \square

3.21 Problema HAMCYCLE

HAMCYCLE

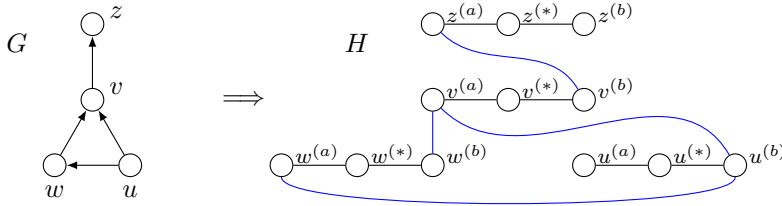
INPUT un grafo diretto $G = (V, E)$

OUTPUT $\text{yes} \iff G$ ha un ciclo che tocca ogni vertice una sola volta

3.22 Riduzione D-HAMCYCLE \preceq HAMCYCLE

Dato un grafo diretto $G \in \mathcal{I}(\text{D-HAMCYCLE})$ con $G = (V(G), E(G))$, vogliamo trasformarlo in un altro grafo $H \in \mathcal{I}(\text{HAMCYCLE})$ con $H = (V(H), E(H))$ tale che G ha un ciclo Hamiltoniano diretto se e solo se H ha un ciclo Hamiltoniano. Costruiamo H in questo modo:

1. per ogni vertice $v \in V(G)$, in $V(H)$ inseriamo un "segmento" 3 vertici $v^{(a)}, v^{(*)}, v^{(b)}$ collegati tra loro in questo ordine;
2. per ogni arco diretto $\overrightarrow{(s, t)} \in E(G)$ aggiungiamo un arco $(s^{(b)}, t^{(a)}) \in E(H)$.



Dimostriamo ora la doppia implicazione:

\Rightarrow Se G ha un ciclo Hamiltoniano diretto e se, senza perdita di generalità, assumiamo che il ciclo sia $C = v_1, v_2, v_3 \dots v_n, v_1$, allora in H avremo il ciclo formato dai nodi

$$C' = v_1^{(a)}, v_1^{(*)}, v_1^{(b)}, v_2^{(a)}, v_2^{(*)}, v_2^{(b)} \dots v_n^{(a)}, v_n^{(*)}, v_n^{(b)}, v_1^{(a)};$$

\Leftarrow se H ha ciclo Hamiltoniano, facciamo le seguenti osservazioni per esso:

- ogni nodo $v^{(*)}$ deve necessariamente essere tra $v^{(a)}$ e $v^{(b)}$, ovvero abbiamo i casi $\dots v^{(a)}, v^{(*)}, v^{(b)} \dots$ oppure $\dots v^{(b)}, v^{(*)}, v^{(a)} \dots$;
- ogni vertice $v_i^{(b)}$ precede sempre immediatamente $v_{i+1}^{(a)}$, oppure ogni vertice $v_i^{(a)}$ precede sempre immediatamente $v_{i+1}^{(b)}$

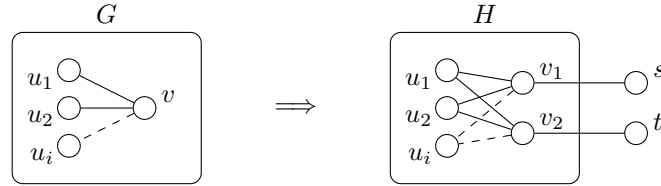
Per queste osservazioni, gruppi di nodi del tipo $v_i^{(a)}, v_i^{(*)}, v_i^{(b)}$ possono essere raggruppati in un unico nodo v_i collegato direttamente al nodo $v_{i+1}^{(b)}$ successivo, oppure gruppi di nodi del tipo $v_i^{(b)}, v_i^{(*)}, v_i^{(a)}$ possono essere raggruppati in un unico nodo v_i collegato direttamente al nodo $v_{i+1}^{(a)}$ successivo.

Il grafo ha $|V(H)| = 3|V(G)|$ ed è quindi costruibile in polytime di $|G|$.

Inoltre, nel caso in cui un nodo abbia un autoanello possiamo semplicemente ignorare quel arco, poiché comunque non lo utilizzeremo mai (toccheremmo due volte lo stesso nodo). \square

3.23 Dimostrazione HAMPATH \in NPC

Sappiamo che HAMPATH \in NP dal primo capitolo e dobbiamo ora dimostrare che HAMPATH è NP-HARD, quindi dimostriamo HAMCYCLE \preceq HAMPATH, poiché HAMCYCLE \in NPC. Dato un grafo $G \in \mathcal{I}(\text{HAMCYCLE})$ vogliamo mapparlo ad un grafo $H \in \mathcal{I}(\text{HAMPATH})$ e per farlo trasformiamo un qualsiasi nodo $v \in V(G)$ in una coppia di nodi $v_1, v_2 \in V(H)$, collegati entrambi agli stessi nodi a cui è collegato v , ma non tra loro. Inoltre colleghiamo il nodo v_1 ad un nuovo nodo s ed il nodo v_2 ad un nuovo nodo t .



Dimostriamo che G ha un ciclo Hamiltoniano se e solo se H ha un cammino Hamiltoniano dove s e t sono gli estremi:

- \Rightarrow se G ha un ciclo Hamiltoniano allora esso può essere spezzato proprio su v e prolungato per arrivare a s e t e creare quindi un cammino Hamiltoniano in H , che parte da s e arriva a t (o viceversa);
- \Leftarrow se H ha un cammino Hamiltoniano allora deve necessariamente avere due nodi s e t come estremi, dopo s c'è v_1 mentre prima di t c'è v_2 , quindi posso eliminare s e t e unire v_1 e v_2 per ottenere un ciclo Hamiltoniano.

Infine, la riduzione è ovviamente polytime in $|G|$, poiché semplicemente spezza un nodo e ne aggiunge due. \square

3.24 Problema INDSET

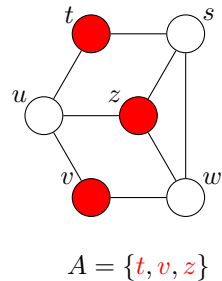
INDEPENDENTSET

INPUT un grafo $G = (V, E)$ non diretto e un parametro intero k

OUTPUT **yes** \iff esiste un sottoinsieme di vertici $A \subseteq V$ di almeno $|A| \geq k$ elementi tale che $\forall u, v \in A, (u, v) \notin E$ (cioè A è un insieme indipendente in V).

Il problema è applicabile ad esempio per una festa, nella quale $v_1 \dots v_n$ sono conoscenti e gli archi (v_i, v_j) indicano che v_i e v_j non vanno d'accordo e per la quale è richiesto di trovare un'insieme di almeno k conoscenti tale che nessuna coppia sia nemica.

Ad esempio, nel grafo a destra l'insieme di vertici $A = \{t, v, z\}$ è un independent set di taglia $k = 3$, infatti i vertici al suo interno non sono collegati tra loro, e quindi possiamo dire che A è una soluzione per l'istanza $(G, 3) \in \text{INDSET}$; è bene far notare che A non è l'unico independent set per G , quindi non è l'unica soluzione per l'istanza $(G, 3)$.



3.25 Dimostrazione INDSET \in NPC

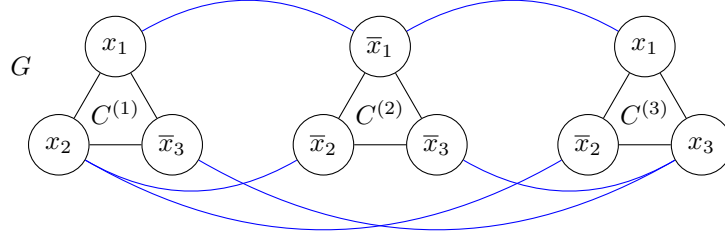
Come al solito dobbiamo dimostrare che INDSET \in NP, quindi

- il certificato è l'indipendent set A di taglia almeno k ;
- il verificatore ① controlla $|A| \geq k$ e ② per ogni coppia $u, v \in A$ controlla che $(u, v) \notin E$; l'algoritmo deve scorrere sui nodi u, v e controllare ogni edge in E , quindi impiega tempo $O(|A|^2 \cdot |E|)$, ed è dunque polytime in $(|V|, |E|)$ poiché $|A| \leq |V|$;

e che INDSET è NP-HARD, scegliendo la riduzione $3\text{-SAT} \preceq \text{INDSET}$.

Data una formula $\phi(x_1 \dots x_n) = C^{(1)} \wedge \dots \wedge C^{(m)}$ 3-CNF costruiamo il grafo G seguendo il procedimento: per ogni clausola $C^{(i)} = (\ell_1^i \vee \ell_2^i \vee \ell_3^i)$ aggiungiamo un *triangolo clausola* (come per la dimostrazione $3\text{-SAT} \preceq 3\text{-COL}$) e colleghiamo con un arco i vertici di due triangoli diversi se sono etichettati con letterali opposti. Ad esempio,

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



Ora vogliamo trovare un independent set di taglia imposta a $k = m$, cioè pari al numero di clausole, e dobbiamo dimostrare come al solito la doppia implicazione: ϕ è soddisfacibile se e solo se in G esiste un insieme indipendente A di taglia $k = m$.

\Rightarrow se ϕ è soddisfacibile allora esiste un assegnamento a tale che in ogni clausola esiste uno e un solo letterale posto a T; identifichiamo tali letterali $\ell_{j_1}^1 \dots \ell_{j_m}^m$ usando la notazione $\ell_{j_i}^i$, dove i è come al solito l'indice della clausola $C^{(i)}$ e j_i è l'indice del letterale che tra i 3 in $C^{(i)}$ è posto a T.

Consideriamo i vertici corrispondenti a questi letterali nel grafo G e raggruppiamoli nell'insieme $A = \{\ell_{j_i}^i \mid i = 1 \dots m\}$, che notiamo essere un insieme indipendente in G . La dimostrazione è per assurdo: se A non fosse indipendente allora esisterebbero due vertici $\ell_{j_s}^s, \ell_{j_t}^t$ entrambi con letterali a T per costruzione di A , che formano un arco $(\ell_{j_s}^s, \ell_{j_t}^t)$ tra due diversi triangoli clausola in G ; dato che per costruzione del grafo questo sarebbe possibile se e solo se $\ell_{j_s}^s = \bar{\ell}_{j_t}^t$ (gli archi tra triangoli clausola collegano vertici di letterali opposti) ed entrambi sono T, incontriamo una contraddizione, poiché quindi non potrebbero essere collegati. Dato che ogni clausola ha un letterale T e ci sono m clausole, A avrà esattamente $k = m$ elementi, come volevamo.

Ad esempio, l'assegnamento $a = (T, F, T)$ pone a T le variabili x_1, \bar{x}_2, x_3 , che quindi raccogliamo nell'insieme $A = \{x_1, \bar{x}_2, x_3\}$, che è evidentemente un independent set.

\Leftarrow se G ha un independent set A di taglia $|A| = m$, facciamo due osservazioni:

- se $A = \{v_1 \dots v_m\}$ allora ogni $v_i, v_j \in A$ sono in triangoli clausola diversi;
- $v_i, v_j \in A$ corrispondono a letterali ℓ_a^i, ℓ_b^j non opposti ($\ell_a^i \neq \bar{\ell}_b^j$).

Chiamiamo α l'assegnamento di verità *parziale* per ϕ ottenuto mettendo a T i letterali corrispondenti ai vertici in A . Poiché questi letterali non sono in conflitto (per costruzione di A), questo assegnamento parziale è proprio e soddisfa ogni clausola (poiché A contiene un vertice per triangolo clausola). Se ad ogni variabile è stato assegnato un valore il procedimento è concluso, mentre se ad alcune variabili manca ancora il valore gliene viene assegnato uno arbitrariamente, poiché in ogni caso sarà ininfluente (le variabili necessarie per la soddisfacibilità sono in A).

Ad esempio, l'assegnamento parziale $\alpha = (T, F, \cdot)$ soddisfa tutti i triangoli, ma manca ancora un valore per x_3 , quindi ne assegno uno in modo arbitrario, per esempio F.

La costruzione di G è polinomiale, infatti il numero di vertici $|V| = 3m = \text{poly}(|\phi|)$, poiché la dimensione della formula è $|\phi| = 3m$. \square

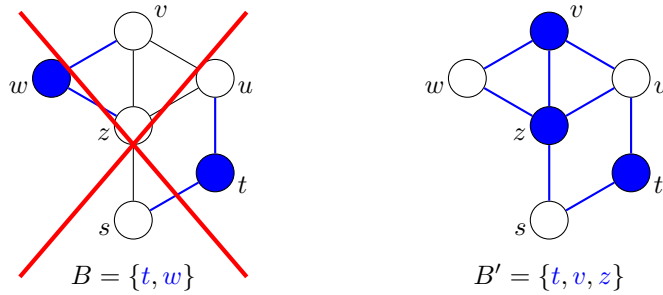
3.26 Problema VERTCOV

VERTEXCOVER

INPUT un grafo $G = (V, E)$ non diretto e un parametro intero k

OUTPUT **yes** \iff esiste un sottoinsieme $B \subseteq V$ di taglia $|B| \leq k$ vertici tale che $\forall (u, v) \in E, \{u, v\} \cap B \neq \emptyset$, ovvero almeno un vertice per ogni arco è in B , quindi i vertici in B *coprono* tutti gli archi di G .

Un esempio classico è quello della disposizione di sensori: vogliamo installare un numero massimo di k telecamere per supervisionare il piano di un edificio e controllare ogni corridoio. Nel grafo qui sotto ad esempio, l'insieme $B = \{t, w\}$ non è un vertex cover, infatti è evidente che rimangono ancora 4 archi da coprire, mentre invece $B' = \{t, v, z\}$ è un vertex cover, di taglia $k = 3$, ed è quindi una soluzione per l'istanza $(G, 3) \in \text{VERTCOV}$.

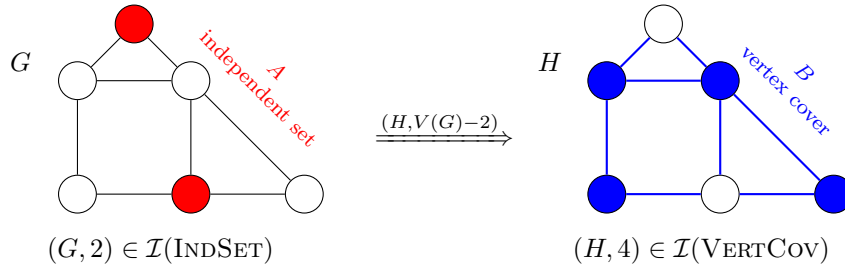


3.27 Dimostrazione $\text{VERTCOV} \in \text{NPC}$

Come al solito dobbiamo dimostrare che $\text{VERTCOV} \in \text{NP}$, quindi

- il certificato è un insieme B di vertici di taglia $|B| \leq k$;
- il verificatore ① controlla $|B| \leq k$ e ② per ogni arco $\{u, v\} \in E$ controlla che $\{u, v\} \cap B \neq \emptyset$, quindi fattibile in $O(|E| \cdot |B|)$.

e che VERTCOV è **NP**-hard, riducendo $\text{INDSET} \preceq \text{VERTCOV}$ poiché $\text{INDSET} \in \text{NPC}$. Dato $(G, k') \in \mathcal{I}(\text{INDSET})$ costruiamo $(H, k) \in \mathcal{I}(\text{VERTCOV})$ tale che esiste $A \subseteq V(G)$ con $|A| \geq k'$ independent set in G se e solo se esiste $B \subseteq V(H)$ con $|B| \leq k$ vertex cover per H . Definiamo $H = G$ e $k = |V(G)| - k'$, quindi la costruzione è ovviamente polinomiale, e in particolare definiamo $B = V(G) \setminus A$.



Ora dobbiamo dimostrare la solita doppia implicazione:

\Rightarrow se A è un independent set in G allora $B = V(G) \setminus A$ è un vertex cover per H , ovvero ogni arco in E ha uno dei vertici in B . La dimostrazione è per assurdo: se non fosse vero allora esisterebbe un arco che non tocca alcun vertice in B , ma tale arco non può essere tra due vertici di A , quindi deve toccare un vertice in $V(G) \setminus A = B$, assurdo;

\Leftarrow se B è un vertex cover per H allora $A = V(G) \setminus B$ è un independent set in G , ovvero ogni vertice in A non ha archi con gli altri in A . La dimostrazione è nuovamente per assurdo: se non fosse vero allora esisterebbe un arco tra due vertici non in B , ma questo non è possibile perché abbiamo assunto B vertex cover, assurdo.

In pratica, VERTCOV è il complemento di INDSET e le loro soluzioni sono altrettanto complementari: dato un grafo G , risolvendo VERTCOV risolviamo anche INDSET , e viceversa. \square

3.28 Problema CLIQUE

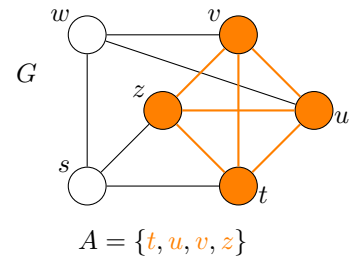
CLIQUE

INPUT un grafo $G = (V, E)$ e un parametro k

OUTPUT **yes** \iff esiste un sottoinsieme $A \subseteq V$ con $|A| = k$ tale che $\forall u, v \in A, (u, v) \in E$, in pratica se esiste un sottografo completamente connesso in G con k vertici

Un esempio pratico torna ad essere quello della festa, dove però questa volta vogliamo trovare il gruppo con la massima rete di amicizie, ovvero il sottoinsieme completamente connesso della taglia k .

Notiamo immediatamente l'analogia con INDSET: dato un grafo G , per INDSET lo vediamo come un “grafo di inimicizie”, che collega nodi in conflitto tra loro, mentre per CLIQUE lo vediamo come un “grafo di amicizie”, dove tutti i vertici sono legati tra loro se non sono in conflitto, quindi il semplice “cambio di prospettiva” ci permette di passare da un problema all'altro. Più formalmente, data un'istanza di INDSET, se prendiamo il complemento degli archi di G , cioè togliamo gli archi quando ci sono e li aggiungiamo dove mancano, possiamo trasformarla in un'istanza di CLIQUE; sfrutteremo proprio questa osservazione per dimostrare $\text{CLIQUE} \in \text{NPC}$

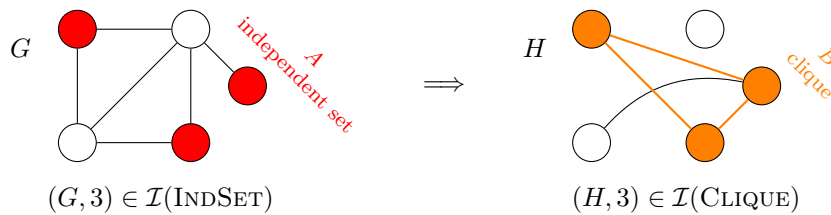


3.29 Dimostrazione $\text{CLIQUE} \in \text{NPC}$

Dobbiamo dimostrare che $\text{CLIQUE} \in \text{NP}$, quindi

- il certificato è una clique di taglia k ;
- il verificatore ① controlla $|A| = k$, ② controlla che $\forall u, v \in A, (u, v) \in E$;

e che CLIQUE è NP-HARD , riducendo $\text{INDSET} \preceq \text{CLIQUE}$ date le considerazioni viste sopra e poiché $\text{INDSET} \in \text{NPC}$. Definiamo una mappatura da $(G = (V, E), k)$ a $(H = (V, \bar{E}), k)$, tenendo identici E e k e definendo il complemento $\bar{E} = \{(u, v) \mid u, v \in V\} \setminus E$, cioè l'insieme degli archi non presenti in E .

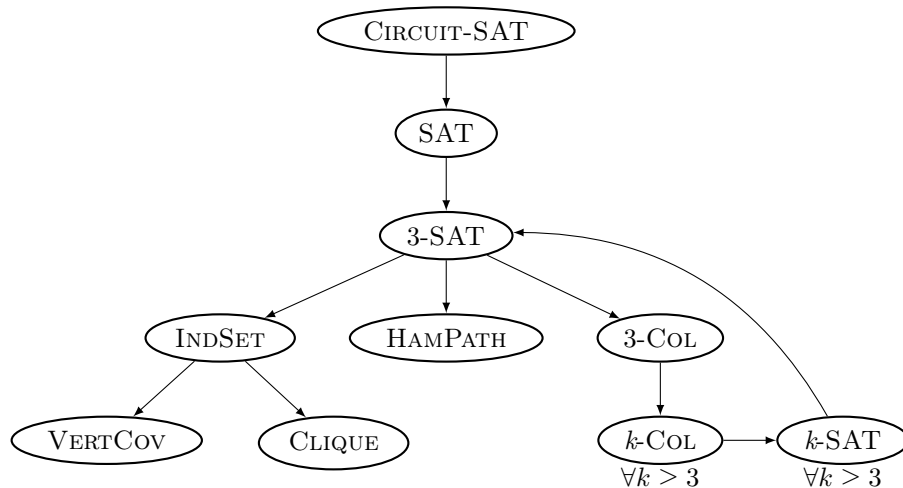


Dobbiamo ora dimostrare la doppia implicazione, ed è possibile farlo in una sola riga:

$$\begin{aligned} A \subseteq G \text{ independent set} &\iff \forall u, v \in A, (u, v) \notin E(G) = E \\ &\iff \forall u, v \in A, (u, v) \in \bar{E} = E(H) \iff A \subseteq H \text{ clique.} \quad \square \end{aligned}$$

3.30 Schema attuale delle riduzioni

Tutti i problemi visti finora sono NPC e derivabili da CIRCUIT-SAT , che sembra essere il “padre” di tutti i problemi NPC . Notiamo che i problemi $k\text{-SAT}$ e $k\text{-COL}$ per $k \leq 2$ sono P e che EULPATH rimane comunque P , quindi la domanda che ci poniamo ora è: cosa davvero differenzia le classi P ed NP a livello di calcolo? Perché non riusciamo ad assimilarle in una sola grande classe di problemi?



Notiamo che mancano alcuni problemi visti, come $\text{NAE-}k\text{-SAT}$, TAU e D-HAMCYCLE , semplicemente per non intasare troppo lo schema.

4 | Determinismo e Macchine di Turing

Abbiamo lasciato in sospeso finora le profonde motivazioni alla base della distinzione tra le classi di problemi incontrate, limitandoci a darne le definizioni formali e dei nomi non troppo precisi (**P** per *polinomiale* e **NP** per *non polinomiale*). Ora andiamo a ripulire le ambiguità e definire le fondamenta di queste classi.

4.1 Fondamenta delle classi P ed NP

Per definire le classi **P**, **NP** e tutte le loro derivate abbiamo bisogno di due “meta-classi”, che descrivono una differenza importante dal punto di vista computazionale.

DETERMINISTIC TIME

La classe di complessità $\text{TIME}(f(n))$ contiene tutti i problemi che sono risolvibili con un certo *algoritmo deterministico*, ed è quindi definita come

$$\text{TIME}(f(n)) = \{A \mid A \text{ ammette un algoritmo deterministico di complessità temporale } O(f(n))\}$$

dove $f(n)$ descrive il tempo necessario all'algoritmo $A(x)$ per un certo problema $A(x)$ quando $|x| = n$.

DETERMINISTIC POLYNOMIAL TIME

Ridefiniamo la classe di complessità **P** come

$$\mathbf{P} = \bigcup_{c>0} \text{TIME}(n^c)$$

quindi contenente problemi per i quali esiste un algoritmo *deterministico polinomiale* che determina se un'istanza è **yes** o **no**.

Un *algoritmo nondeterministico* invece estende le funzionalità del determinismo includendo un'istruzione **goto both**, che divide la traccia di esecuzione corrente del programma in due tracce ℓ_1 ed ℓ_2 ; tali tracce vengono eseguite in parallelo indipendentemente, come se le avessimo assegnate a due cloni della macchina precedente.

Esempio

Per il problema SAT, l'algoritmo nondeterministico SAT-SOLVER prende in input una formula $\phi(x_1 \dots x_n)$ in CNF e crea 2^n tracce di esecuzione, una per ogni possibile assegnazione delle variabili x_i . Queste tracce lavoreranno indipendenti tra loro, in modo da scoprire usando n diramazioni l'assegnamento che pone $\phi(x_1 \dots x_n) = \text{T}$; l'algoritmo nondeterministico impiega dunque tempo lineare nel numero di variabili in ϕ , quindi è polytime nel suo input.

Algorithm 3 SAT-SOLVER

Require: una formula ϕ

```
1: for i = 1, n do
2:   goto both 4, 6
3:    $x_i = \text{T}$ 
4:   goto 7
5:    $x_i = \text{F}$ 
6: if  $\phi(x_1 \dots x_n) = \text{T}$  then
7:   return yes
```

La complessità temporale di un algoritmo è la lunghezza massima tra quelle delle sue tracce di computazione; se l'algoritmo ha output possibile **yes/no** diciamo che il suo output è **yes** (per una data istanza x) se *esiste* una traccia che termina con output **yes**; se invece per *ogni possibile* traccia l'output è **no** allora l'output dell'algoritmo è **no** (se singolarmente una traccia ritorna **no** non succede nulla e si aspettano tutte le altre tracce).

NONDETERMINISTIC TIME

La classe di complessità $\text{NTIME}(f(n))$ contiene tutti i problemi che sono risolvibili con un certo

algoritmo nondeterministico, ed è quindi definita come

$$\mathbf{NTIME}(f(n)) = \{A \mid A \text{ ammette un algoritmo } \textit{nondeterministico} \text{ di complessità temporale } O(f(n))\}$$

dove $f(n)$ descrive il tempo necessario all'algoritmo $A(x)$ per un certo problema $A(x)$ quando $|x| = n$.

NONDETERMINISTIC POLINOMIAL TIME

Ridefiniamo la classe di complessità **NP** come

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

quindi contenente problemi per i quali esiste un algoritmo *nondeterministico polinomiale* che determina se un'istanza è **yes** o **no**.

Dimostriamo ora che queste nuove definizioni più formali racchiudono quelle date in precedenza: notiamo immediatamente che è preservato $\mathbf{P} \subseteq \mathbf{NP}$ poiché ogni algoritmo deterministico si può vedere come una traccia di un algoritmo nondeterministico.

Ora dobbiamo far vedere che un algoritmo nondeterministico è verificabile in modo deterministico, quindi dobbiamo dimostrare che le due definizioni sono equivalenti:

$$\begin{aligned} \mathbf{NTIME}(f(n)) &= \{A \mid \exists \text{ algoritmo nondeterministico con complessità di tempo } O(f(n))\} \\ &= \{A \mid \exists \text{ algoritmo deterministico } B(x, w) \text{ con complessità } O(f(|x|)) \\ &\quad \text{t.c. } A(x) = \text{yes} \iff \exists w \text{ t.c. } B(x, w) = \text{yes}\} \end{aligned}$$

\Rightarrow dato un algoritmo nondeterministico A per A , scriviamo il verificatore deterministico esattamente come A e ad ogni istruzione **goto both** usiamo il certificato w per decidere quale traccia seguire in base ai bit di w (che è definito come stringa binaria!);

\Leftarrow dato un verificatore deterministico B per A , dato che B termina in $O(f(|x|))$ allora si ha necessariamente che $|w| = O(f(|x|))$, poiché altrimenti B non potrebbe decidere quale traccia seguire ad ogni diramazione (non ci sarebbero abbastanza bit per prendere tutte le decisioni necessarie). Detto questo, l'algoritmo nondeterministico A è costruito in modo da creare nondeterministicamente ogni possibile w della lunghezza necessaria, diramando l'esecuzione ad ogni bit, e alla fine di ogni traccia applica il certificato w appena creato e x in B , che verifica deterministicamente se x è istanza **yes** o **no**; visto che la prima parte di A è nondeterministica, tutto A è nondeterministico. In pratica, seguiamo tutte le strade e restituiamo la prima positiva al verificatore.

Per concludere, ridefiniamo anche **EXP** come

EXPONENTIAL TIME

La classe di complessità **EXP** è definita come

$$\mathbf{EXP} = \bigcup_{c>0} \mathbf{TIME}((2^n)^c)$$

e contiene quindi problemi per i quali esiste un algoritmo *deterministico esponenziale* che determina se un'istanza è **yes** o **no**.

4.2 Problema 2-SAT

2-Satisfiability

INPUT una formula ϕ 2-CNF di m clausole

OUTPUT **yes** $\iff \phi$ è soddisfacibile

Intuitivamente questo problema è **P**, infatti dato che ogni clausola ha sempre 2 letterali, una volta scelto un assegnamento basta propagarlo nella formula; vediamo più in dettaglio.

4.3 Dimostrazione 2-SAT $\in P$

Dato che ogni clausola è nella forma $C = \ell_1 \vee \ell_2$ ed ha quindi solo 2 letterali, per poter soddisfare ϕ in ogni clausola almeno uno dei due deve essere T, cosa che possiamo formalizzare con i vincoli $\bar{\ell}_2 \implies \ell_1$ e $\bar{\ell}_1 \implies \ell_2$. Per la dimostrazione creo un grafo diretto con due nodi per ogni variabile, uno per x_i e l'altro per \bar{x}_i , e per ogni clausola aggiungo degli archi $\bar{x}_2 \rightarrow x_1$ e $\bar{x}_1 \rightarrow x_2$ per rappresentare i vincoli di assegnamento: i cammini nel grafo specificano quindi tutte le implicazioni conseguenti ad un primo assegnamento arbitrario, in pratica quando viene deciso il valore di un letterale, gli archi codificano le *dipendenze* dei valori che altri letterali devono avere.

Se in un cammino si ottiene $x \rightsquigarrow \bar{x}$ oppure $\bar{x} \rightsquigarrow x$ significa che assegnando un certo valore al letterale x sarebbe necessario assegnare lo stesso valore anche a \bar{x} , cosa ovviamente impossibile: questo cammino porta ad una contraddizione logica del letterale x e quindi non può rappresentare un assegnamento valido per ϕ .

Vogliamo quindi dimostrare che ϕ è soddisfacibile se e solo se esiste un cammino privo di contraddizioni $x \rightsquigarrow \bar{x}$ oppure $\bar{x} \rightsquigarrow x$:

\Leftarrow se esiste $x \rightsquigarrow \bar{x}$ oppure $\bar{x} \rightsquigarrow x$ allora ϕ non è soddisfacibile (visto prima);

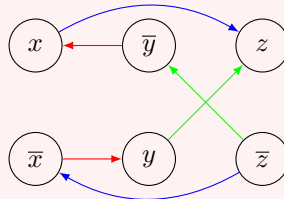
\Rightarrow se $\forall x, x \not\rightsquigarrow \bar{x}$ e $\bar{x} \not\rightsquigarrow x$ allora ϕ è soddisfacibile: supponiamo che $\forall x$ almeno uno dei due cammini non esista, in particolare non esista $x \rightsquigarrow \bar{x}$, e poniamo $x = T$, propagando il valore di verità su ogni cammino che parte da x (ponendo cioè a T tutti i letterali su ogni cammino). Dimostriamo ora l'assenza delle seguenti inconsistenze:

- creazione di contraddizioni su x : verificato per ipotesi, abbiamo supposto $x \not\rightsquigarrow \bar{x}$;
- presenza di entrambi i cammini $x \rightsquigarrow y$ e $x \rightsquigarrow \bar{y}$: notiamo che il grafo è simmetrico, ovvero se esiste $s \rightarrow t$ allora esiste anche $\bar{s} \rightarrow \bar{t}$ (proprio perché gli archi rappresentano i vincoli); se per assurdo questa inconsistenza dovesse capitare, dato che abbiamo $x \rightsquigarrow y$ di conseguenza avremmo $\bar{y} \rightsquigarrow \bar{x}$, quindi $x \rightsquigarrow \bar{y} \rightsquigarrow \bar{x}$, che è in contraddizione con l'ipotesi, assurdo.

In questo modo abbiamo mappato una formula 2-CNF ad un grafo diretto: questo ci verrà molto utile ora per la costruzione di un algoritmo polinomiale.

Esempio

Per la formula $\phi = (x \vee y) \wedge (\bar{y} \vee z) \wedge (\bar{x} \vee z)$ costruiamo il grafo



Notiamo che partendo dal nodo \bar{z} e seguendo il cammino inferiore (o anche quello superiore), otteniamo $\bar{z} \rightsquigarrow z$, che è una contraddizione logica; se invece partiamo dal nodo \bar{x} il cammino $\bar{x} \rightarrow y \rightarrow z$ non ha nessuna contraddizione e codifica l'assegnamento $a(x, y, z) = (F, T, T)$, che soddisfa infatti ϕ . Notiamo inoltre che l'assegnamento di x può essere completamente arbitrario, infatti partendo dal nodo x otteniamo l'assegnamento parziale $a(x, y, z) = (T, \cdot, T)$, il quale soddisfa ϕ se $y = T$.

Per risolvere 2-SAT posso scrivere l'algoritmo 2-SAT-SOLVER che

1. costruisce il grafo G come descritto sopra (fattibile in $\text{poly}(\phi)$);
2. ritorna no se esiste un nodo x con cammini $x \rightsquigarrow \bar{x}$ oppure $\bar{x} \rightsquigarrow x$;
3. finché esiste una variabile x che non ha un valore di verità
 - se $x \not\rightsquigarrow \bar{x}$ assegno $x = T$ e propago da x tutti i possibili cammini mettendo a T i vertici raggiungibili, ritornando infine l'assegnamento;
 - se $\bar{x} \not\rightsquigarrow x$ assegno $\bar{x} = T$ e propago da \bar{x} tutti i possibili cammini mettendo a T i vertici raggiungibili, ritornando infine l'assegnamento;

Per trovare il cammino $x \rightsquigarrow \bar{x}$ oppure $\bar{x} \rightsquigarrow x$ posso usare una BFS, che ha costo $O(|V| + |E|)$, e dato che devo farlo per ogni x , impiego in totale $O(|V| \cdot (|V| + |E|)) = O(|V|^2) = \text{poly}(|G|)$; la propagazione è fattibile utilizzando di nuovo una BFS, quindi l'algoritmo è polinomiale. \square

4.4 Problema REACH

Reachability

INPUT un grafo diretto $G = (V, E)$ con $s, t \in V(G)$

OUTPUT **yes** \iff esiste il cammino $s \rightsquigarrow t$

REACH è chiaramente **P**, si risolve con una BFS (non lo dimostriamo), ma ci interessa altro in questo momento: abbiamo visto che 2-SAT si risolve in pratica mediante un numero $\text{poly}(\phi)$ di chiamate a REACH, infatti l'algoritmo 2-SAT-SOLVER è riscrivibile come

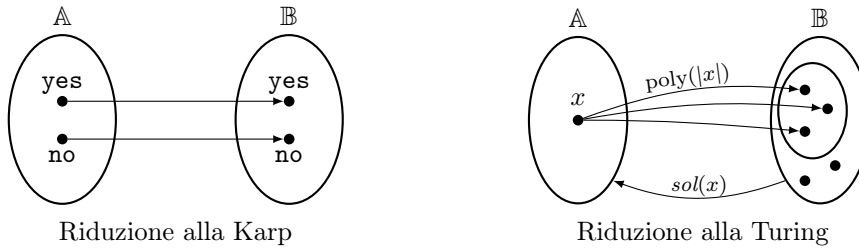
1. costruisci G
2. $\forall x$, se $\text{REACH}(G, x, \bar{x}) = \text{yes}$ e $\text{REACH}(G, \bar{x}, x) = \text{yes}$ allora ritorna **no**.

quindi possiamo risolvere 2-SAT utilizzando REACH.

Riduzione alla Turing

Il problema \mathbb{A} è riducibile (alla Turing o alla Cook) al problema \mathbb{B} , usando la notazione $\mathbb{A} \preceq_T \mathbb{B}$, se esiste un algoritmo polinomiale che risolve istanze x di \mathbb{A} utilizzando al più un numero $\text{poly}(|x|)$ di chiamate a istanze di \mathbb{B} .

Questa riduzione è più forte di quella alla Karp ed infatti ne è un'estensione, infatti la riduzione alla Karp mappa istanze **yes** e **no** tra \mathbb{A} e \mathbb{B} tramite un algoritmo polinomiale, mentre la riduzione alla Turing mappa istanze di \mathbb{A} con istanze di \mathbb{B} che risolvono porzioni di \mathbb{A} , per poi usare tali soluzioni per costruire la soluzione per \mathbb{A} .



Notiamo che se $\mathbb{B} \in \mathbf{P}$ e $\mathbb{A} \preceq_T \mathbb{B}$ allora $\mathbb{A} \in \mathbf{P}$, poiché può appunto essere risolto con un numero polinomiale di chiamate a un algoritmo polinomiale.

5 | Problemi debolmente NPC

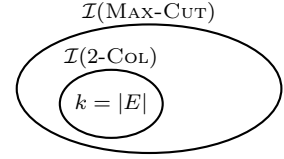
5.1 Problema MAX-CUT

MAX-CUT

INPUT un grafo non diretto $G = (V, E)$ e un parametro intero k

OUTPUT **yes** \iff esiste una bicolorazione di G che bicolore almeno k archi, in pratica se posso spezzare almeno k archi poiché connettono nodi di colori diversi

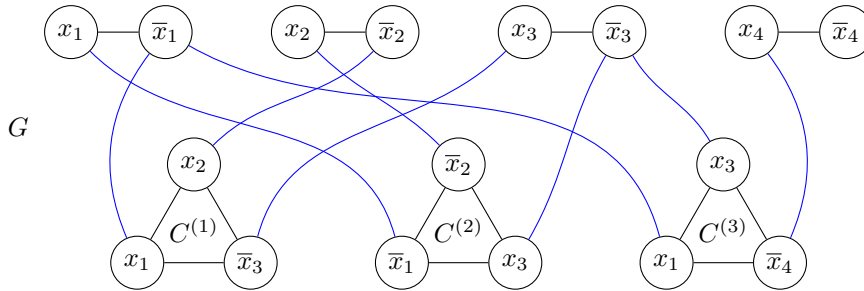
Notiamo che se $k = |E|$ il problema equivale a verificare $G \in 2\text{-COL}$, cioè che G sia un grafo bipartito, che come abbiamo visto all'inizio è un problema **P**, quindi alcune istanze di MAX-CUT sono più semplici di altre; dimostriamo che in generale $\text{MAX-CUT} \in \text{NPC}$.



5.2 Dimostrazione MAX-CUT \in NPC

Innanzitutto $\text{MAX-CUT} \in \text{NP}$ poiché il certificato è la bicolorazione e il verificatore conta gli archi bicolati, quindi ha complessità $O(|E|) = \text{poly}(|G|)$. Ora vogliamo ridurre $\text{NAE-3-SAT} \preceq \text{MAX-CUT}$ poiché $\text{NAE-3-SAT} \in \text{NPC}$, quindi cerchiamo una mappatura da una formula ϕ 3-CNF a una istanza (G, k) : supponendo $\phi(x_1 \dots x_n) = C^1 \wedge \dots \wedge C^m$, creiamo un grafo con un *segmento variabile* per ogni variabile (quindi ogni una coppia di variabili opposte x_i e \bar{x}_i è collegata da un arco) e un *triangolo clausola* per ogni clausola, in cui ogni vertice si collega al suo opposto tra i *segmenti variabile* (la costruzione è identica cioè a quella per $\text{NAE-3-SAT} \preceq 3\text{-COL}$, ma non abbiamo il vertice radice). Ad esempio,

$$\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \bar{x}_4)$$



Ora impongo che in G vengano bicolati il massimo possibile di archi, quindi imposto $k = 5m + n$: questo perché così posso spezzare ① due archi in ogni triangolo clausola, cioè $2m$ archi (ricordiamo che la formula è NAE-3-CNF , quindi ci saranno sempre due variabili con lo stesso valore e l'arco tra loro non potrà essere spezzato), ② ogni arco nei segmenti variabile, cioè n archi, e ③ ogni arco tra i segmenti variabile e i triangoli clausola, che sono $3m$.

Dimostrazione. ϕ è NAE-soddisfacibile $\iff G$ ha k archi bicolati

\Rightarrow esiste un assegnamento a che rende in ogni clausola un letterale **T** e uno **F**, e con questo creo la colorazione $\mathbf{T} \rightarrow \text{NERO}$ e $\mathbf{F} \rightarrow \text{BIANCO}$: per ogni triangolo avrò quindi sempre 2 archi bicolati, ogni arco tra triangoli e segmenti sarà bicoloreto e ogni segmento sarà bicoloreto, quindi avrò $3m + n + 2m = k$ archi bicolati;

\Leftarrow esiste una bicolorazione $\psi : V(G) \rightarrow \{\text{NERO}, \text{BIANCO}\}$ tale almeno k archi di G sono bicolati, cioè $\forall (u, v) \in A \subseteq E(G), |A| \geq k, \{\psi(u), \psi(v)\} = \{\text{NERO}, \text{BIANCO}\}$. Per costruzione del grafo, ψ può colorare ≤ 2 archi per ogni triangolo e $\leq 3m + n$ archi tra tutti gli altri $3m + n$ archi, quindi in totale ψ può bicoloreto $\leq 2m + 3m + n = 5m + n = k$ archi; data l'assunzione che ψ possa bicoloreto $\geq k$ archi, ψ bicoloreto esattamente k archi, ovvero esattamente 2 per triangolo e tutti gli altri.

Definiamo l'assegnamento $a_i = T \iff \psi(x_i) = \mathbf{NERO}$, fattibile scorrendo sui segmenti variabile del grafo, e dato che in ogni triangolo c'è almeno un nodo per entrambi i colori, ogni clausola ha almeno un letterale T e uno F , quindi ϕ è NAE-soddisfatta. \square

Ricollegandoci a quanto detto prima, alcune istanze di MAX-CUT sono mappabili a istanze 2-COL, mentre altre sono mappabili a NAE-3-SAT, come abbiamo appena dimostrato.

5.3 Problema MAX- k -SAT

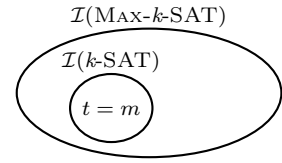
MAX- k -SAT

INPUT una formula ϕ k -CNF e un parametro intero t

OUTPUT **yes** \iff esiste un assegnamento che soddisfa almeno t clausole

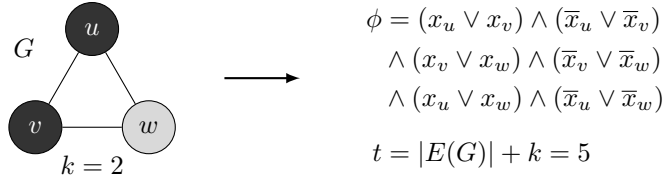
Cominciamo notando che per $t = m$ il problema equivale a k -SAT, quindi MAX- k -SAT contiene le istanze di k -SAT e di conseguenza k -SAT \preceq MAX- k -SAT se $k > 3$. Dato che $\forall k$, MAX- k -SAT $\in \mathbf{NP}$, dove il certificato è l'assegnamento e il verificatore conta le clausole soddisfatte, in conclusione abbiamo MAX- k -SAT $\in \mathbf{NPC}$ per $k > 3$.

Come abbiamo visto, di solito i problemi a 2 variabili sono \mathbf{P} , quindi ci chiediamo se anche MAX-2-SAT $\in \mathbf{P}$: questa volta no, infatti certamente vale 2-SAT \preceq MAX-2-SAT, ma non ci dice niente (2-SAT $\in \mathbf{P}$ e un problema \mathbf{P} è sempre riducibile a un problema \mathbf{NP}). Questa volta non riusciamo a spostare MAX-2-SAT in \mathbf{P} , ma al contrario abbiamo MAX-2-SAT $\in \mathbf{NPC}$, dimostrabile con MAX-CUT \preceq MAX-2-SAT.



5.4 Dimostrazione MAX-2-SAT $\in \mathbf{NPC}$

Come già detto sopra MAX-2-SAT $\in \mathbf{NP}$, ora utilizziamo MAX-CUT \preceq MAX-2-SAT: trovo una mappatura da un grafo G con k archi bicolorati a una formula ϕ 2-CNF. Dato che in ogni arco bicolorato (u, v) i vertici u, v non possono avere lo stesso colore, costruisco ϕ aggiungendo $x_u \neq x_v$ per ogni arco (u, v) , che è equivalente ad aggiungere $(x_u \vee x_v) \wedge (\bar{x}_u \vee \bar{x}_v)$ e fattibile in tempo polinomiale scorrendo sugli archi, e definisco il parametro $t = |E(G)| + k$. Ad esempio



Dimostrazione. G ha k archi bicolorati $\iff \phi$ ha almeno t clausole soddisfatte

\Rightarrow se esiste una colorazione $\psi : V(G) \rightarrow \{\mathbf{NERO}, \mathbf{BIANCO}\}$ tale che almeno k archi di G sono bicolorati, definiamo l'assegnamento a dove per x_v assegniamo

$$a_v = \begin{cases} T & \text{se } \psi(v) = \mathbf{NERO} \\ F & \text{se } \psi(v) = \mathbf{BIANCO} \end{cases}$$

e notiamo che per ogni arco (u, v) ,

- se (u, v) non è bicolorato da ψ allora i suoi nodi hanno colori uguali $\psi(u) = \psi(v)$, quindi $a_u = a_v$ e solo una delle clausole in $(x_u \vee x_v) \wedge (\bar{x}_u \vee \bar{x}_v)$ viene soddisfatta;
- se (u, v) è bicolorato da ψ allora i suoi nodi hanno colori diversi $\psi(u) \neq \psi(v)$, quindi $a_u \neq a_v$ ed entrambe le clausole in $(x_u \vee x_v) \wedge (\bar{x}_u \vee \bar{x}_v)$ sono soddisfatte.

Dato che ci sono $|E(G)|$ archi di cui solo k sono bicolorati, avremo $|E(G)|$ clausole soddisfatte, una per ogni arco, più altre k soddisfatte dai soli archi bicolorati, quindi in totale $|E(G)| + k = t$.

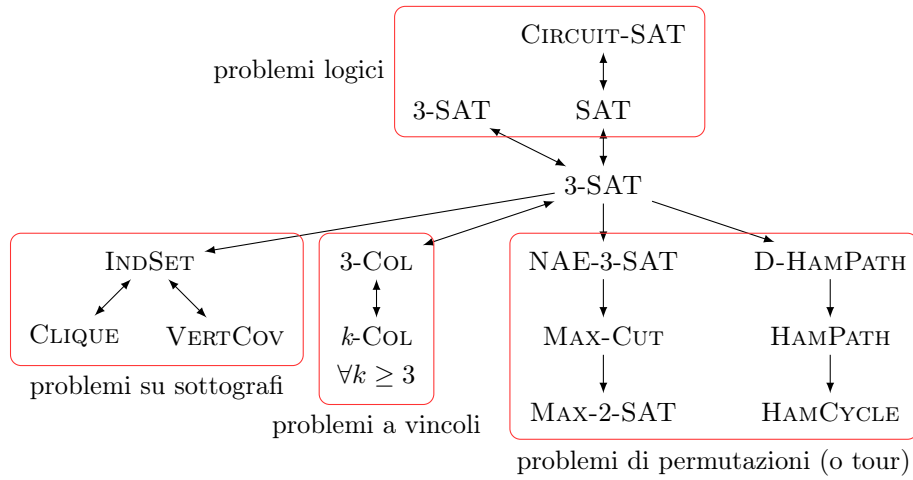
\Leftarrow se esiste un assegnamento che soddisfa almeno $t = |E| + k$ clausole di ϕ allora dato che ϕ contiene $|E(G)|$ clausole, per il principio della piccionaia ci sono almeno t coppie di clausole $(x_u \vee x_v) \wedge (\bar{x}_u \vee \bar{x}_v)$ in cui entrambe le clausole sono soddisfatte e perciò le loro due variabili sono assegnate a valori diversi. Se ora imposto la funzione di colorazione

$$\psi(v) = \begin{cases} \text{BIANCO} & \text{se } a_v = \text{T} \\ \text{NERO} & \text{se } a_v = \text{F} \end{cases}$$

solo le t coppie di clausole completamente soddisfatte danno vertici con colori diversi, e quindi ottengo un grafo con t archi bicolorati. \square

5.5 Schema attuale delle riduzioni

Come possiamo vedere, problemi di natura molto diversa sono in realtà tutti della stessa famiglia. Notiamo che alcune riduzioni sono ora doppie, ma non andremo a dimostrarle.



Abbiamo visto che per alcuni problemi **P** risolverne la versione di ottimizzazione è un problema **NPC**, in particolare abbiamo visto che 2-SAT $\in \mathbf{P}$ (tutte le clausole sono soddisfatte) e MAX-2-SAT $\in \mathbf{NPC}$ (almeno t clausole sono soddisfatte), quindi le sottigliezze che stanno nelle definizioni dei problemi possono cambiare drasticamente la dimensione del problema. Un altro esempio potrebbe essere MIN-CUT $\in \mathbf{P}$ (al più k archi bicolorati) e MAX-CUT $\in \mathbf{NPC}$ (almeno k archi bicolorati); studiamo ora il problema SUBSETSUM, che diventa **NPC** per istanze di grandi dimensioni.

5.6 Problema SUBSETSUM

SUBSETSUM

INPUT un insieme di interi $A = \{a_1 \dots a_r\}$ e un intero S

OUTPUT **yes** \iff esiste un sottoinsieme di A la cui somma degli elementi è S

Ad esempio, per l'istanza $(A = \{13, 5, 7, 9, 32, 128, 5\}, S = 57)$ abbiamo $a_1 + a_2 + a_3 + a_5 = 13 + 5 + 7 + 32 = 57 = S$, quindi è un'istanza **yes**.

5.7 Dimostrazione SUBSETSUM $\in \mathbf{NPC}$

Dimostriamo innanzitutto SUBSETSUM $\in \mathbf{NP}$: il certificato è un sottoinsieme $X \subseteq A$ tale che $\sum_{a \in X} a = S$ e il verificatore semplicemente somma e controlla che il valore sia S , quindi è polytime. Passiamo alla dimostrazione che SUBSETSUM è **NP-HARD**, ottenuta riducendo 3-SAT \preceq SUBSETSUM: la mappatura va da una formula ϕ 3-CNF (con n variabili e m clausole) a un insieme di numeri A e un valore S . In particolare, costruiamo $2n + 2m$ numeri a_k ed il numero S nel seguente modo:

- a_k ed S sono composti da esattamente $m + n$ cifre (per esempio 175, 089, 001);

- ogni a_k è composto con le cifre $\{0, 1\}$, mentre S è composto con le cifre $\{0, 1, 3\}$;
- il sistema numerico di ogni numero a_k e di S è posizionale e dato dagli indici $C_m \dots C_1$ e $x_n \dots x_1$ in quest'ordine, quindi parleremo di "posizione C_j " e di "posizione x_i ";
- la funzione $a(\cdot)$ costruisce i numeri a_k in base alle variabili x_i e \bar{x}_i e alle clausole C_j :
 - ★ i numeri $a(x_i)$ pongono a 1 la posizione x_i , e C_j solo se clausola C_j contiene x_i ;
 - ★ i numeri $a(\bar{x}_i)$ pongono a 1 la posizione x_i , e C_j solo se clausola C_j contiene \bar{x}_i ;
 - ★ i numeri $a(C_j)$ e $a(C'_j)$ pongono a 1 la posizione C_j ;
 - ★ per tutti i numeri, qualsiasi altra posizione è posta a 0.

Utilizziamo la seguente mappatura per identificare i numeri così ottenuti

$$\begin{aligned}
 a_{2i-1} = a(x_i) &= 00 \dots \underset{\text{se } x_i \in C_j}{0001000} \dots \underset{x_i}{00100} \dots 00 \\
 a_{2i} = a(\bar{x}_i) &= 00 \dots \underset{\text{se } \bar{x}_i \in C_j}{0001000} \dots \underset{x_i}{00100} \dots 00 \\
 a_{2n+2j-1} = a(C_j) &= 00 \dots \underset{C_j}{0002000} \dots 00000 \dots 00 \\
 a_{2n+2j} = a(C'_j) &= 00 \dots \underset{C_j}{0002000} \dots 00000 \dots 00
 \end{aligned}$$

quindi avremo ad esempio i numeri $a_1 = a(x_1)$, $a_2 = a(\bar{x}_1)$, $a_3 = a(x_2)$, $a_{2n} = a(\bar{x}_n)$, $a_{2n+1} = a(C_1)$, $a_{2n+2} = a(C'_1)$, $a_{2n+3} = a(C_2)$, $a_{2n+2m} = a(C'_m)$ e così via.

- il numero S pone a 3 tutte le posizioni C_j e a 1 tutte le posizioni x_i .

Dimostrazione. ϕ 3-CNF è soddisfatta \iff la somma dei numeri $a_k \in X \subseteq A$ da S

\Rightarrow esiste un assegnamento $\alpha = \alpha_1 \dots \alpha_n$ che pone a T almeno un letterale in ogni clausola di ϕ , e di conseguenza prendo i numeri

- ★ $a(x_i)$ se $x_i = T$, cioè se $\alpha_i = T$
- ★ $a(\bar{x}_i)$ se $\bar{x}_i = T$, cioè se $\alpha_i = F$

così da avere uno e un solo 1 in ogni posizione x_i . In pratica la posizione x_i riceve 1 da $a(x_i)$ oppure da $a(\bar{x}_i)$ poiché questi sono mutualmente esclusivi in base ad α_i .

A questo punto, in base ai numeri $a(x_i)$ e $a(\bar{x}_i)$ che ho appena preso ho 3 casi:

- ★ la posizione C_j ha ricevuto tre 1, il che significa che tutti e tre i letterali nella clausola C_j hanno valore T, quindi la somma dei numeri presi fa 3 in posizione C_j e non devo sommare né $a(C_j)$ né $a(C'_j)$;
- ★ la posizione C_j ha ricevuto due 1, il che significa che solo due letterali nella clausola C_j hanno valore T, quindi la somma dei numeri presi fa 2 in posizione C_j e devo ulteriormente sommare o $a(C_j)$ o $a(C'_j)$, arbitrariamente;
- ★ la posizione C_j ha ricevuto un solo 1, il che significa che un solo letterale nella clausola C_j ha valore T, quindi la somma dei numeri presi fa 1 in posizione C_j e devo ulteriormente sommare sia $a(C_j)$ sia $a(C'_j)$.

Per costruzione, la posizione C_j non può ricevere più di tre 1, in quanto necessiterebbe che C_j avesse più di tre letterali T, e non può nemmeno ricevere meno di un 1, in quanto necessiterebbe che C_j non avesse alcun letterale T.

Ricapitolando, abbiamo costruito i numeri in modo tale che quelli mappati dall'assegnamento α sommati insieme pongano ad 1 tutte le posizioni x_i e pongano le posizioni C_j modo tale da poter essere aggiustate dai numeri $a(C_j)$: così facendo, la somma di tutti i numeri avrà sempre 3 nelle posizioni C_j e 1 nelle posizioni x_i e quindi sarà esattamente la S che abbiamo costruito prima.

\Leftarrow esiste un $X \subseteq A$ e un S tale che $\sum_{a \in X} a = S$, e possiamo fare alcune osservazioni:

- ★ per ogni x_i , X può contenere un solo numero tra $a(x_i)$ e $a(\bar{x}_i)$ poiché se li contenesse entrambi la posizione x_i di S dovrebbe essere 2, ma è 1. Definiamo quindi l'assegnamento come $a_i = T \iff a(x_i) \in X$ o al contrario $a_i = F \iff a(\bar{x}_i) \in X$
- ★ la somma dei numeri $a(C_j)$ e $a(C'_j)$ da al massimo 2, quindi dato che S ha 3 in posizione C_j , almeno un numero $a(x_i)$ o $a(\bar{x}_i)$ deve contenere il rimanente 1 in posizione C_j , il che significa che x_i o \bar{x}_i sono T e contenuti nella clausola C_j , che è quindi soddisfatta. \square

Mostriamo infine che la costruzione è polinomiale: la taglia dell'istanza di SUBSETSUM è $|\{a_1 \dots a_r\}, S| = 2n + 2m + 1$ numeri di $n + m$ cifre, quindi $O((n + m)(2n + 2m + 1))$, che è $\text{poly}(|\phi|)$. Notiamo che però i numeri nell'istanza di SUBSETSUM sono appunto di $n + m$ cifre, quindi sono in quantità esponenziale in $n + m$, in particolare $S = 33 \dots 3311 \dots 11 \simeq 3^{n+m}$.

In pratica stiamo dicendo che le istanze di SAT si mappano solo a istanze di SUBSETSUM esponenzialmente grandi rispetto a loro: in generale, diciamo che un problema numerico (come lo è SUBSETSUM) è *debolmente NPC* se è NPC solo per numeri molto grandi.

In conclusione, $\text{SUBSETSUM} \in \mathbf{NPC}$ solo per numeri molto grandi, quindi se tutti gli r numeri in A ed S sono esprimibili con $O((\log r)^c) = \text{poly}(r)$ cifre (o meglio, bit), allora $\text{SUBSETSUM} \in \mathbf{P}$, risolvibile con programmazione dinamica (è il problema dello zaino).

Esempio

Data la formula

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

applichiamo quanto descritto dalla mappatura e otteniamo

C_2	C_1	x_3	x_2	x_1	
0	1	0	0	1	$a(x_1)$
1	0	0	0	1	$a(\bar{x}_1)$
0	1	0	1	0	$a(x_2)$
1	0	0	1	0	$a(\bar{x}_2)$
1	0	1	0	0	$a(x_3)$
0	1	1	0	0	$a(\bar{x}_3)$
0	1	0	0	0	$a(C_1)$
0	1	0	0	0	$a(C'_1)$
1	0	0	0	0	$a(C_2)$
1	0	0	0	0	$a(C'_2)$
3	3	1	1	1	S

$(A, S) \in \mathcal{I}(\text{SUBSETSUM})$

$A = \{01001, 10001, 01010, 10010, 10100, 01100, 01000, 01000, 10000, 10000\}$

$S = 33311$

La formula ϕ è soddisfatta dall'assegnamento $\alpha(x_1, x_2, x_3) = (\mathbf{T}, \mathbf{F}, \mathbf{T})$, che si mappa ai numeri $a(x_1) = 01001$, $a(\bar{x}_2) = 10010$ e $a(x_3) = 10100$, che sommati insieme danno 21111, quindi aggiungiamo anche i numeri $a(C_1) = 01000$, $a(C'_1) = 01000$ e $a(C_2) = 10000$, così da ottenere $S = 33111$.

6 | Prove di separabilità

Abbiamo sempre detto che $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$, senza mai avere la certezza di una effettiva *separazione* tra le classi. Dimostriamo che non tutti i problemi esponenziali possono essere resi polinomiali, ovvero che $\mathbf{P} \neq \mathbf{EXP}$.

6.1 Dimostrazione $\mathbf{P} \neq \mathbf{EXP}$

Notiamo che $\mathbf{TIME}(f(n))$ è inclusiva, ovvero dato $g(n) < f(n)$, $\mathbf{TIME}(g(n)) \subsetneq \mathbf{TIME}(f(n))$: in pratica, se per \mathbb{A} esiste solamente un algoritmo $O(f(n))$, nessun algoritmo $O(g(n))$ potrà risolvere \mathbb{A} , perché ora ha meno tempo.

o piccolo

La notazione asintotica $o(f(n))$ è una versione più potente della $O(f(n))$, tale che $g(n) = o(f(n))$ se $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$, quindi $\exists n_0 \mid n > n_0, g(n) \ll f(n)$.

6.1.1 Teorema della gerarchia temporale

Teorema della gerarchia temporale

Data una funzione $f(n)$ **time constructible**, cioè per la quale esiste un algoritmo che la calcola in $O(f(n))$, per ogni funzione $g(n) = o(f(n))$ vale la relazione

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

dove $s(n)$ è il tempo necessario per simulare n passi di un algoritmo contando inoltre quanti passi sono stati fatti.

Questa funzione dipende modello di computazione che stiamo usando, infatti su una macchina RAM l'overhead computazionale del conteggio è una costante, quindi $s(n) = O(n)$, mentre su una macchina di Turing abbiamo $s(n) = O(n \log n)$; nel nostro caso usiamo linguaggi di programmazione ad alto livello, quindi siamo su macchine RAM e l'overhead è costante.

Dimostrazione. L'idea è che se abbiamo la promessa che un algoritmo termina in un certo tempo possiamo simularlo in quel tempo, ma non in meno, poiché altrimenti dovremmo indovinare alcuni passi dell'algoritmo. A questo proposito, definiamo un problema che vuole predire se un algoritmo termina entro un certo tempo, e per farlo dimostriamo che non possibile se non aspettando il termine dell'algoritmo stesso.

PREDICT- f

INPUT un programma π e un input x

OUTPUT $\pi(x)$ se $\pi(x)$ termina in al più $f(|x|)$ passi, no altrimenti

L'algoritmo che implementa **PREDICT- f** semplicemente esegue i passi di π e intanto li conta: se arriva all'output ed è ancora entro $f(|x|)$ passi ritorna quel output, se altrimenti ne fa di più ritorna **no**; in pratica diamo un tempo massimo a π , dopo il quale terminiamo subito. **PREDICT- f** è quindi risolvibile in $O(s(f(n)))$, e in particolare **PREDICT- f** $\in \mathbf{TIME}(s(f(n)))$ con $n = |x|$; dato che nel nostro caso $s(n) = O(n)$, abbiamo **PREDICT- f** $\in \mathbf{TIME}(f(n))$.

Ad esempio, per **PREDICT- n^2** $\in \mathbf{TIME}(s(n^2))$ [= $\mathbf{TIME}(n^2)$], se **PREDICT- n^2** (π , 0001) termina in $|0001|^2 = 16$ passi allora ritorna $\pi(0001)$, altrimenti ritorna **no**.

Mostriamo ora che **PREDICT- f** non può essere risolto in meno di $f(n)$ passi, cioè che **PREDICT- f** $\notin \mathbf{TIME}(g(n))$ dove $g(n) = o(f(n))$, e per farlo introduciamo un altro problema, il quale è definito in maniera contraddittoria.

CATCH-22- f^1 **INPUT** un programma π **OUTPUT** $\overline{\pi(\pi)}$ se $\pi(\pi)$ termina in al più $f(|\pi|)$ passi, no altrimenti

L'algoritmo che implementa CATCH-22- f semplicemente simula π per $f(|\pi|)$ passi e ritorna $\overline{\pi(\pi)}$ se termina entro $f(|\pi|)$ e no altrimenti. CATCH-22- f è quindi risolvibile in $O(s(f(n)))$, e in particolare CATCH-22- $f \in \mathbf{TIME}(s(f(n)))$. Notiamo inoltre che la definizione è in funzione di PREDICT- f , infatti CATCH-22- $f(\pi) = \overline{\text{PREDICT-}f(\pi, \pi)}$.

Visto che CATCH-22- f è un caso particolare di PREDICT- f , varrà la stessa considerazione vista sopra, quindi mostriamo che CATCH-22- f non può essere risolto in meno di $f(n)$ passi, cioè CATCH-22- $f \notin \mathbf{TIME}(g(n))$ dove $g(n) = o(f(n))$.

Dimostriamo per assurdo: supponiamo CATCH-22- $f \in \mathbf{TIME}(g(n))$ dove $g(n) = o(f(n))$, ovvero che esista un programma π_{22} che implementa l'algoritmo per CATCH-22- f e che termina sempre in $g(n)$ passi. In questo caso, $\forall \pi \in \mathcal{I}(\text{CATCH-22-}f)$, $\pi_{22}(\pi) = \text{CATCH-22-}f(\pi)$ termina in $g(|\pi|)$ passi e in particolare $\pi_{22}(\pi_{22}) = \text{CATCH-22-}f(\pi_{22})$ termina in $g(|\pi_{22}|)$ passi, e dato che $g(|\pi_{22}|) < f(|\pi_{22}|)$, per definizione CATCH-22- $f(\pi_{22})$ ritorna $\overline{\pi_{22}(\pi_{22})}$, ma a questo punto $\pi_{22}(\pi_{22}) = \text{CATCH-22-}f(\pi_{22}) = \overline{\pi_{22}(\pi_{22})}$, assurdo.

In conclusione, fissato $f(n)$, un qualsiasi programma che garantisce di poter risolvere CATCH-22- f in meno di $f(n)$ passi non dà l'output giusto quando eseguito con il proprio codice come input, quindi la garanzia è impossibile da avere. Notiamo che la tecnica utilizzata in questa dimostrazione è l'*argomento diagonale di Cantor* e che il problema CATCH-22- f è analogo ad HALTINGPROBLEM, solo che anziché andare in loop imposta un timer.

Non siamo ancora arrivati però alla dimostrazione del teorema, infatti pur avendo una contraddizione su CATCH-22- f , possiamo definire un "programma toppa"

$$\pi'_{22}(x) = \begin{cases} \overline{\pi_{22}(x)} & \text{se } x = \pi_{22} \\ \pi_{22}(x) & \text{altrimenti} \end{cases}$$

che risolve CATCH-22- f e aggiusta il comportamento del singolo caso $\pi_{22}(\pi_{22})$; vogliamo quindi trovare un problema che fallisce su un'infinità di input.

CATCH-23- f^2 **INPUT** un programma π e un input 1^ℓ stringa di ℓ 1**OUTPUT** $\overline{\pi(\pi, 1^\ell)}$ se $\pi(\pi, 1^\ell)$ termina in $f(|\pi| + \ell)$ passi, no altrimenti

L'algoritmo che implementa CATCH-23- f , di nuovo, simula π per $f(|\pi| + \ell)$ passi e ritorna $\overline{\pi(\pi, 1^\ell)}$ se termina entro $f(|\pi| + \ell)$ e no altrimenti. CATCH-23- f è quindi risolvibile in $O(s(f(n)))$, e in particolare CATCH-23- $f \in \mathbf{TIME}(s(f(n)))$. Nuovamente notiamo che la definizione è in funzione di PREDICT- f , infatti CATCH-23- $f(\pi, 1^\ell) = \overline{\text{PREDICT-}f(\pi, (\pi, 1^\ell))}$.

Nuovamente, visto che CATCH-23- f è un caso particolare di PREDICT- f , varrà la stessa considerazione vista sopra, quindi mostriamo che CATCH-23- f non può essere risolto in meno di $f(n)$ passi, cioè CATCH-23- $f \notin \mathbf{TIME}(g(n))$ dove $g(n) = o(f(n))$.

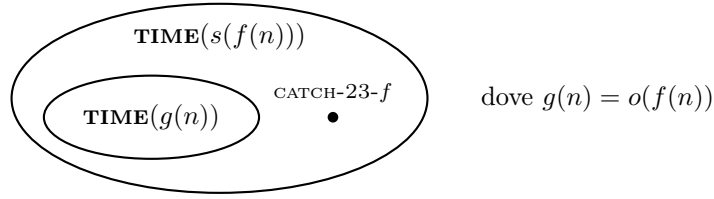
Dimostriamo per assurdo: supponiamo CATCH-23- $f \in \mathbf{TIME}(g(n))$ dove $g(n) = o(f(n))$, ovvero che esista un programma π_{23} che implementa l'algoritmo per CATCH-23- f e che termina sempre in $g(n)$ passi. In questo caso, $\forall \pi \in \mathcal{I}(\text{CATCH-23-}f)$, $\pi_{23}(\pi, 1^\ell) = \text{CATCH-23-}f(\pi, 1^\ell)$ termina in $g(|\pi| + \ell)$ passi e in particolare $\pi_{23}(\pi_{23}, 1^\ell) = \text{CATCH-23-}f(\pi_{23}, 1^\ell)$ termina in $g(|\pi_{23}| + \ell)$ passi, e dato che $g(|\pi_{23}| + \ell) < f(|\pi_{23}| + \ell)$, per definizione CATCH-22- $f(\pi_{23}, 1^\ell)$ ritorna $\overline{\pi_{23}(\pi_{23}, 1^\ell)}$, ma a questo punto $\pi_{23}(\pi_{23}, 1^\ell) = \text{CATCH-23-}f(\pi_{23}, 1^\ell) = \overline{\pi_{23}(\pi_{23}, 1^\ell)}$, assurdo.

Questa assurdità è più forte di quella per CATCH-22- f , poiché dato che $g(n) = o(f(n))$, per qualsiasi $n \rightarrow \infty$ avremo $g(n) < f(n)$, e quindi siccome $n = |\pi| + \ell$, basta avere valori ℓ infinitamente grandi affinché π_{23} fallisca

¹"Catch-22" è un libro che ruota attorno ad un comma autocontraddittorio del regolamento militare, che permette di astenersi da una missione ad alto rischio solo se si richiede una visita per dimostrare di essere pazzi, e quindi potenzialmente rovinare l'esito della missione; tale richiesta può pervenire solo da una persona ritenuta sana, poiché solo un sano di mente vedrebbe il rischio della missione, quindi fare richiesta implica il non essere pazzi e l'impossibilità di evitare la missione, mentre non fare richiesta implica l'essere pazzi ed avere i requisiti per poter lasciare la missione, ma non poterla lasciare perché non si è fatta richiesta.

²È un gioco di parole su "Catch-22", come fosse ad un livello successivo

a risolvere CATCH-23- f . Siccome π_{23} fallisce per infiniti input 1^ℓ , non possiamo definire infiniti “programmi toppa” per aggiustare π_{23} in ogni input su cui fallisce, quindi questa dimostrazione è sufficiente a dimostrare che PREDICT- f non è risolvibile in meno di $f(n)$ passi.



□

6.1.2 Conclusioni

Dimostrato questo teorema, abbiamo i seguenti risultati:

- per $f(n) = n^2$, $g(n) = n$, ovviamente $n = o(n^2)$, quindi $\text{TIME}(n) \subsetneq \text{TIME}(n^2)$;
- per $f(n) = n \log n$, $g(n) = n$, ovviamente $n = o(n \log n)$, quindi $\text{TIME}(n) \subsetneq \text{TIME}(n \log n)$;
- per $f(n) = 2^n$, $g(n) = n^k$ con k costante, ovviamente $n^k = o(2^n)$, quindi $\forall k$, $\text{TIME}(n^k) \subsetneq \text{TIME}(2^n)$;

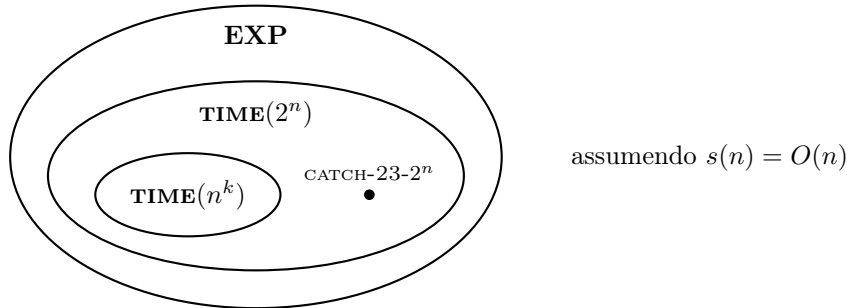
e dall'ultimo punto abbiamo

$$\mathbf{P} = \bigcup_{k>0} \text{TIME}(n^k) \subsetneq \bigcup_{c>0} \text{TIME}((2^n)^c) = \mathbf{EXP}$$

ovvero $\mathbf{P} \subsetneq \mathbf{EXP}$, quindi in conclusione $\mathbf{P} \neq \mathbf{EXP}$.

□

Per tornare ai problemi usati nella dimostrazione, CATCH-23- $2^n \in \text{TIME}(2^n)$, quindi non può essere risolto in tempo $O(n^k)$ perché $n^k = o(2^n)$ e dunque CATCH-23- $2^n \notin \mathbf{P}$.



Sfortunatamente questo teorema (e quindi l'argomento diagonale, che è potentissimo) non è abbastanza forte per dimostrare anche $\mathbf{P} \neq \mathbf{NP}$, poiché abbiamo ancora l'ostacolo del diverso modello di computazione (determinismo VS. nondeterminismo), che in \mathbf{P} vs \mathbf{EXP} non c'era (usano entrambi modelli deterministici). Quello che abbiamo dimostrato però è che in $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$ sappiamo $\mathbf{P} \neq \mathbf{EXP}$, quindi uno dei due \subseteq è in realtà un \subsetneq , ma non sappiamo ancora quale.

6.2 Problemi “puramente” NP

Abbiamo visto che \mathbf{NP} contiene \mathbf{P} ed \mathbf{NPC} e che molti problemi \mathbf{NP} si riducono in \mathbf{NPC} , quindi vogliamo vedere se esistono problemi “puramente” \mathbf{NP} , non \mathbf{P} e nemmeno \mathbf{NPC} (e in particolare che $\text{SAT} \not\leq \mathbf{A}$).

Teorema di Ladner

Se $\mathbf{P} \neq \mathbf{NP}$ allora $\mathbf{P} \cup \mathbf{NPC} \neq \mathbf{NP}$, cioè esistono problemi \mathbf{NP} non \mathbf{P} e non \mathbf{NPC} .

Dimostrazione. Sotto l'ipotesi che $\mathbf{P} \neq \mathbf{NP}$, ovvero che esista $\mathbf{A} \in \mathbf{NP}$ tale che $\mathbf{A} \notin \mathbf{P}$ e \mathbf{A} non è \mathbf{NP} -HARD, impostiamo tale \mathbf{A} come

$$\mathbf{A}(x) = \begin{cases} \text{SAT}(x) & \text{se } f(|x|) \text{ è pari} \\ \text{no} & \text{se } f(|x|) \text{ è dispari} \end{cases}$$

utilizzando la funzione definita *ciclicamente*

$$f(0) = 0$$

$$f(n) = \begin{cases} \text{se } f(n-1) = 2k \text{ allora se tra le prime } n \text{ stringhe binarie} \\ \quad \begin{cases} \exists y \text{ tale che } \pi_k(y) \neq \mathbb{A}(y) \text{ allora } f(n) = f(n-1) + 1 \\ \nexists y \text{ tale che } \pi_k(y) \neq \mathbb{A}(y) \text{ allora } f(n) = f(n-1) \end{cases} \\ \text{se } f(n-1) = 2k+1 \text{ allora se tra le prime } n \text{ stringhe binarie} \\ \quad \begin{cases} \exists y \text{ tale che } \text{SAT}(y) \neq \mathbb{A}(\pi_k(y)) \text{ allora } f(n) = f(n-1) + 1 \\ \nexists y \text{ tale che } \text{SAT}(y) \neq \mathbb{A}(\pi_k(y)) \text{ allora } f(n) = f(n-1) \end{cases} \end{cases}$$

dove π_k è il k -esimo programma in linguaggio C (in ordine lessicografico) che termina in tempo polinomiale (è solo un formalismo per enumerare gli algoritmi polinomiali).

Innanzitutto mostriamo che dato x , $f(|x|) = \text{poly}(|x|)$. L'input x ha taglia $|x| = n$, quindi per trovare il programma corrispondente devo scorrere al massimo $O(n)$; ora scorro tutte le n stringhe y , di nuovo $O(n)$, fino a che non ne trovo una che soddisfa $\text{SAT}(y)$: queste stringhe hanno dimensione $|y| = \log n$ bit e quindi dato che $\text{SAT}(y) = O(2^{|y|})$ abbiamo che il controllo di soddisfacibilità è $O(n)$, e quindi complessivamente $f(n) = \text{poly}(n)$.

Ora, poiché $f(n)$ è calcolabile in polytime, \mathbb{A} è verificabile in polytime, quindi $\mathbb{A} \in \mathbf{NP}$, dove il certificato è x e il verificatore calcola $f(|x|)$ e rifiuta se è dispari e verifica il certificato per SAT se è pari.

Ora manca da dimostrare $\mathbb{A} \notin \mathbf{P}$ e che \mathbb{A} non è **NP-HARD**. $\mathbb{A} \notin \mathbf{P}$ se nessun algoritmo polinomiale π_k è tale che $\forall y, \mathbb{A}(y) = \pi_k(y)$. Supponendo per assurdo che esista tale π_k , la funzione f una volta arrivata al numero $2k$ non cambierà più valore: in termini più formali, dato un n_0 tale che $f(n_0) = 2k$, raggiunto il momento di valutare $f(n_0 + 1)$ entreremo nel ramo superiore della definizione di $f(n)$, nel quale, dato che abbiamo definito $\pi_k(y) = \mathbb{A}(y)$, non può esistere un y tale che $\pi_k(y) \neq \mathbb{A}(y)$, quindi avremo il risultato $f(n) = f(n-1)$, e in particolare nel nostro caso $f(n_0 + 1) = f(n_0)$; in generale

$$f(n_0) = 2k \implies f(n_0 + 1) = 2k \implies f(n_0 + 2) = 2k \implies \dots \forall n \geq n_0, f(n) = 2k$$

cioè raggiunto un certo valore n_0 avremo sempre $f(n) = 2k$, quindi per definizione di \mathbb{A} dopo n_0 avremo sempre $\text{SAT}(x)$, e in conclusione

$$\begin{aligned} \mathbb{A} \in \mathbf{P} &\implies \exists k \mid \pi_k(x) = \mathbb{A}(x) \\ &\implies \exists n_0 \mid \forall n > n_0, f(n) = 2k \\ &\implies \forall |x| > n_0, \mathbb{A}(x) = \text{SAT}(x) \\ &\implies \pi_k(x) = \text{SAT}(x) \\ &\implies \text{SAT} \in \mathbf{P} \implies \mathbf{P} = \mathbf{NP}, \text{ assurdo, per ipotesi } \mathbf{P} \neq \mathbf{NP} \end{aligned}$$

Ora manca di dimostrare solo che \mathbb{A} non è **NP-HARD**. Sempre per assurdo, supponiamo che $\mathbb{A} \in \mathbf{NPC}$, in particolare che $\text{SAT} \preceq \mathbb{A}$, e quindi esiste un algoritmo polinomiale π_k che mappa SAT ad \mathbb{A} , per definizione tale che $\forall y, \text{SAT}(y) = \mathbb{A}(\pi_k(y))$.

Nuovamente, la funzione f una volta arrivata al numero $2k + 1$ non cambierà più valore: in termini più formali, dato un n_0 tale che $f(n_0) = 2k + 1$, raggiunto il momento di valutare $f(n_0 + 1)$ entreremo nel ramo inferiore della definizione di $f(n)$, nel quale, dato che abbiamo $\text{SAT}(y) = \mathbb{A}(\pi_k(y))$ per ogni stringa y , avremo sempre il risultato $f(n) = f(n-1)$, e in particolare nel nostro caso $f(n_0 + 1) = f(n_0)$; in generale quindi

$$f(n_0) = 2k + 1 \implies f(n_0 + 1) = 2k + 1 \implies \dots \forall n \geq n_0, f(n) = 2k + 1$$

cioè raggiunto un certo valore n_0 avremo sempre $f(n) = 2k$, quindi per definizione di \mathbb{A} dopo n_0 avremo sempre **no**, possiamo scrivere un algoritmo polinomiale che cabla i risultati fino a n_0 e poi risponde sempre **no**, quindi $\mathbb{A} \in \mathbf{P}$, e in conclusione

$$\mathbb{A} \in \mathbf{P} \wedge \text{SAT} \preceq \mathbb{A} \implies \text{SAT} \in \mathbf{P} \implies \mathbf{P} = \mathbf{NP}, \text{ assurdo, per ipotesi } \mathbf{P} \neq \mathbf{NP}$$

Essenzialmente f è costruita in modo ciclico per eliminare tutti gli algoritmi polinomiali che risolvono \mathbb{A} o che mappano la riduzione verso \mathbb{A} , quindi \mathbb{A} è “*semplicemente*” **NP**. \square

Con questo, abbiamo dimostrato che in **NP** esistono anche altri gradi di difficoltà oltre a **P** e **NPC**, che in particolare chiameremo **NP-INTERMEDIATE**.

7 | Decisione Vs. Ricerca

All'inizio del corso abbiamo visto che un problema di ottimizzazione mappa istanze di input a più soluzioni valide, tra le quali ce n'è una ottima, che è l'obiettivo che vogliamo raggiungere. Per capire se un problema di ottimizzazione è difficile ne abbiamo studiato la versione di decisione, nella quale ci limitiamo a dire se una soluzione esiste o meno; da questo studio sono emerse le varie classi **P**, **NP**, **NPC**, **EXP**, ecc. e abbiamo studiato in varie forme la dicotomia **P** vs. **NP**.

Abbiamo capito che se il problema di decisione è difficile allora anche il problema di ottimizzazione sarà difficile, ma possiamo dire il contrario? Quello che ci chiediamo ora è: se un problema di decisione è facile, cosa possiamo riguardo sua versione di ottimizzazione?

7.1 Self-reducibility

In generale, se A^{DEC} è un problema di decisione in **NP** e B è un verificatore per A^{DEC} , quindi tale che $A^{\text{DEC}}(x) = \text{yes} \iff \exists w \mid B(x, w) = \text{yes}$, allora per ogni $x \in \mathcal{I}(A^{\text{DEC}})$ tale che se $A^{\text{DEC}}(x) = \text{yes}$, il problema di ricerca A^{SEARCH} trova quel w tale che $B(x, w) = \text{yes}$.

Notiamo che se riesco a trovare la soluzione per un problema ovviamente posso dire se c'è o meno una soluzione per quel problema, quindi un qualsiasi problema di decisione si riduce alla Turing alla sua versione di ricerca, in formule $\forall A, A^{\text{DEC}} \preceq_T A^{\text{SEARCH}}$, e quindi se $A^{\text{SEARCH}} \in \mathbf{P}$ allora $A^{\text{DEC}} \in \mathbf{P}$. Questa era quasi una banalità, ma domandiamoci ora se vale il contrario, ovvero se sapere che un problema ha una soluzione ci permette anche di ottenere tale soluzione, in formule

$$A^{\text{SEARCH}} \stackrel{?}{\preceq}_T A^{\text{DEC}}$$

Innanzitutto definiamo uno strumento molto potente

Oracolo

Dato un problema A , un **oracolo** \mathcal{MB}_A è un algoritmo *ideale* che risolve in tempo polinomiale (e di solito addirittura costante) il problema di decisione A^{DEC} .

e descriviamo formalmente la questione posta sopra

Self-reducibility

Un problema A è **self-reducible** se $A^{\text{SEARCH}} \preceq_T A^{\text{DEC}}$, cioè se la versione di ricerca di A non è più difficile della versione di decisione di A .

Solitamente si suppone l'esistenza di un oracolo \mathcal{MB}_A , il quale viene chiamato dalla funzione di riduzione un numero polinomiale di volte per trovare la soluzione di A^{SEARCH} . Questa supposizione molto forte viene a verificarsi nel momento in cui A^{DEC} fosse **P**, in quanto quindi esisterebbe un algoritmo polinomiale per l'oracolo \mathcal{MB}_A e di conseguenza anche A^{SEARCH} sarebbe **P**.

Riassumendo, se A è self-reducible allora quando $A^{\text{DEC}} \in \mathbf{P}$, anche $A^{\text{SEARCH}} \in \mathbf{P}$.

Per cominciare, rinfreschiamo un paio di problemi e dimostriamone la self-reducibility.

7.1.1 SAT è self-reducible

SAT

INPUT formula ϕ CNF

OUTPUT

DECISIONE: $\text{yes} \iff \phi$ è soddisfacibile;

RICERCA: un assegnamento a tale che $\phi(a) = \text{T}$ se esiste per ϕ , altrimenti no

Dato il problema SAT, supponiamo di avere un oracolo $\mathcal{MB}_{\text{SAT}}$, che può dire in tempo costante se una certa formula ϕ è soddisfacibile o no: con questo, $\text{SAT}^{\text{SEARCH}}$ si risolve in tempo polinomiale facendo al più un numero di chiamate polinomiali nella taglia $|\phi|$ all'oracolo $\mathcal{MB}_{\text{SAT}}$, e quindi possiamo costruire l'assegnamento a (se esiste) che soddisfa ϕ .

Dimostrazione. Supponiamo di avere una formula $\phi(x_1 \dots x_n)$ CNF e un oracolo $\mathcal{MB}_{\text{SAT}}$. Scrivo un risolutore $\text{SAT}^{\text{DEC}}\text{-SOLVER}$ che ritorna **no** se l'oracolo dice che ϕ non è soddisfacibile, altrimenti fissa una ad una ogni variabile x_i ad un certo valore a_i , ottenendo ogni volta una **restrizione di ϕ con $\{x_i \leftarrow a_i\}$** , e controlla ogni volta che la formula rimanga soddisfacibile con gli assegnamenti usati finora, altrimenti usa \bar{a}_i . Scriviamo l'algoritmo

Algorithm 4 $\text{SAT}^{\text{DEC}}\text{-SOLVER}$

Require: una formula ϕ

```

1: if  $\mathcal{MB}_{\text{SAT}}(\phi) = \text{no}$  then return no
2: for  $i \leftarrow 1$  to  $n$  do
3:   if  $\mathcal{MB}_{\text{SAT}}(\phi_{x_1 \leftarrow a_1 \dots x_{i-1} \leftarrow a_{i-1}}) = \text{yes}$  then
4:      $a_i \leftarrow \text{T}$ 
5:   else
6:      $a_i \leftarrow \text{F}$ 
7: return  $a = (a_1 \dots a_n)$ 

```

Esempio

Per $\phi(x_1 \dots x_4) = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_4)$ abbiamo $\mathcal{MB}_{\text{SAT}}(\phi) = \text{yes}$, quindi fissiamo $x_1 = \text{T}$ e otteniamo la restrizione $\phi_{x_1 \leftarrow \text{T}} = (\bar{x}_2 \vee x_3) \wedge (\bar{x}_4)$, per la quale $\mathcal{MB}_{\text{SAT}}(\phi_{x_1 \leftarrow \text{T}}) = \text{T}$, quindi fissiamo $x_2 \leftarrow \text{T}$, e avanti così, fino ad ottenere $a(x_1 \dots x_4) = (\text{T}, \text{T}, \text{T}, \text{F})$

In pratica, con questa ricerca non devo mai tornare indietro sulle decisioni già prese, quindi è *backtrack free*, e dato che uso tempo costante per ogni variabile, l'algoritmo è polinomiale in $|\phi|$: in conclusione $\text{SAT}^{\text{SEARCH}} \preceq_T \text{SAT}^{\text{DEC}}$ e quindi SAT è self-reducible. \square

7.1.2 CLIQUE è self-reducible

CLIQUE

INPUT grafo G e un parametro $k \in \mathbb{N}$

OUTPUT

DECISIONE: $\text{yes} \iff$ esiste in G una clique di taglia $\geq k$;
RICERCA: una clique di taglia $\geq k$ se esiste in G , altrimenti **no**;

Dimostrazione. Supponiamo che $\text{CLIQUE}^{\text{DEC}}$ abbia un oracolo $\mathcal{MB}_{\text{CLIQUE}}(G, k) = \text{yes} \iff G$ ha una clique di taglia k ; vogliamo un algoritmo che da (G, k) trova, se esiste, una clique di G di taglia k : possiamo procedere scorrendo uno alla volta tutti i vertici, eliminando quelli che non fanno parte di una clique di taglia k . Scriviamo l'algoritmo

Algorithm 5 $\text{CLIQUE}^{\text{SEARCH}}\text{-SOLVER}$

Require: un grafo G e un intero k

```

1: if  $\mathcal{MB}_{\text{CLIQUE}}(G, k) = \text{no}$  then return no
2: for all vertice  $v \in V(G)$  do
3:   if  $\mathcal{MB}_{\text{CLIQUE}}(G - v, k) = \text{yes}$  then
4:     return  $G \leftarrow G - v$ 
5: return  $k$  vertici nel grafo rimanente

```

Dato che faccio $O(n)$ chiamate a $\mathcal{MB}_{\text{CLIQUE}}$, quindi $\text{CLIQUE}^{\text{SEARCH}} \preceq_T \text{CLIQUE}^{\text{DEC}}$. \square

Quindi per ora SAT e CLIQUE sono self-reducible, ma questi sono prima di tutto **NP**, quindi ci chiediamo: ogni problema **NP** è self-reducible? Mostriamo che

- ogni problema **NPC** è self-reducible;
- se $(\mathbf{NP} \cap \mathbf{co-NP}) \setminus \mathbf{P} \neq \emptyset$ allora esiste un problema in **NP** non self-reducible;
- se $\mathbf{P} \neq \mathbf{NP}$ allora esiste un problema in **NP** non self-reducible.

e cioè che i problemi di decisione più difficili lo sono anche per la ricerca, poiché anche se sono self-reducible, non possiamo (ad oggi) avere un oracolo per loro.

7.2 Primo teorema di self-reducibility

Teorema

Se \mathbb{X} è **NPC** allora \mathbb{X} è self-reducible (supponendo che esista $\mathcal{MB}_{\mathbb{X}}$).

Dimostrazione. Dato un problema $\mathbb{X} \in \mathbf{NPC}$, abbiamo che

- $\mathbb{X} \in \mathbf{NP}$, quindi ① $\mathbb{X} \preceq \text{SAT}$ e ② esiste un verificatore $B(x, w)$ per \mathbb{X} tale che $\mathbb{X}(x) = \text{yes} \iff \exists w \mid B(x, w) = \text{yes}$;

Nella dimostrazione del teorema di Cook-Levin abbiamo visto che per un qualsiasi problema $\mathbb{X} \in \mathbf{NP}$ il verificatore $B(x, w)$ è mappabile ad una famiglia di circuiti booleani $C_B(x, w)$, e che in particolare fissato un certo x abbiamo un circuito C_B^x tale che $C_B^x(w) = \mathbf{T} \iff \exists w \mid B(x, w) = \text{yes}$. Inoltre, $\text{CIRCUIT-SAT} \preceq \text{SAT}$, quindi possiamo mappare il circuito C_B^x ad una formula $\phi_{C_B^x}$ tale che $C_B^x(w) = \mathbf{T} \iff \exists w, y \mid \phi_{C_B^x}(w, y) = \mathbf{T}$, dove y è l'assegnamento da dare a tutte le porte intermedie del circuito (fare riferimento alla dimostrazione di $\text{CIRCUIT-SAT} \preceq \text{SAT}$ per ulteriori chiarimenti). Questo ci dice che se riusciamo a trovare l'assegnamento (w, y) che soddisfa la formula $\phi_{C_B^x}$ abbiamo l'input w che verifica il circuito C_B^x , che è anche la soluzione per il problema di ricerca $\mathbb{X}^{\text{SEARCH}}$: in pratica, se so risolvere $\text{SAT}^{\text{SEARCH}}$ per $\phi_{C_B^x}$ ottengo la soluzione w , che è esattamente la soluzione per l'istanza x di $\mathbb{X}^{\text{SEARCH}}$.

Formalmente, per un qualsiasi problema $\mathbb{X} \in \mathbf{NPC}$, dato un certo $x \in \mathcal{I}(\mathbb{X})$, siccome

$$\exists w \mid B(x, w) = \text{yes} \iff \exists w \mid C_B^x(w) = \mathbf{T} \iff \exists w, y \mid \phi_{C_B^x}(w, y) = \mathbf{T},$$

abbiamo che $w = \text{SAT}^{\text{SEARCH}}(\phi_{C_B^x})$, e quindi la soluzione w per $\mathbb{X}^{\text{SEARCH}}$ è ottenuta risolvendo $\text{SAT}^{\text{SEARCH}}$.

- \mathbb{X} è **NP-HARD**, quindi ③ $\text{SAT} \preceq \mathbb{X}$ (perché SAT è anche **NP**).

Supponiamo che esista un oracolo $\mathcal{MB}_{\mathbb{X}}$, quindi possiamo scrivere un algoritmo per $\mathbb{X}^{\text{SEARCH}}$ che facendo al più un numero polinomiale di chiamate a $\mathcal{MB}_{\mathbb{X}}$ costruisce w tale che $B(x, w) = \text{yes}$, se $\mathbb{X}(x) = \text{yes}$.

Dato che abbiamo un oracolo $\mathcal{MB}_{\mathbb{X}}$ e che $\text{SAT} \preceq \mathbb{X}$, possiamo scrivere un algoritmo polinomiale (fa al più un numero polinomiale di chiamate a $\mathcal{MB}_{\mathbb{X}}$) che implementa l'oracolo $\mathcal{MB}_{\text{SAT}}$, e quindi possiamo risolvere SAT^{DEC} in tempo polinomiale.

Algorithm 6 Implementazione di $\mathcal{MB}_{\text{SAT}}$

Require: un oracolo $\mathcal{MB}_{\mathbb{X}}$ per $\mathbb{X} \in \mathbf{NPC}$

```

1: function  $\mathcal{MB}_{\text{SAT}}(\phi)$ 
2:   // dato che  $\text{SAT} \preceq \mathbb{X}$  tramite una funzione  $f$ 
3:   // trasforma  $\phi \in \mathcal{I}(\text{SAT})$  in  $x_\phi \in \mathcal{I}(\mathbb{X})$ 
4:    $x_\phi \leftarrow f(\phi)$ 
5:   return  $\mathcal{MB}_{\mathbb{X}}(x_\phi)$ 
```

Formalmente abbiamo $\mathcal{MB}_{\text{SAT}} \preceq_T \mathcal{MB}_{\mathbb{X}}$, e quindi

$$\text{SAT}(\phi) = \text{yes} \iff \mathcal{MB}_{\text{SAT}}(\phi) = \text{yes} \iff \mathcal{MB}_{\mathbb{X}}(x_\phi) = \text{yes} \iff \mathbb{X}(x_\phi) = \text{yes}.$$

il che ci permette di avere un oracolo $\mathcal{MB}_{\text{SAT}}$.

In conclusione quindi, possiamo risolvere $\mathbb{X}^{\text{SEARCH}}$ risolvendo $\text{SAT}^{\text{SEARCH}}$ e usando $\mathcal{MB}_{\mathbb{X}}$ abbiamo costruito $\mathcal{MB}_{\text{SAT}}$: dato che SAT è self-reducible (quindi $\text{SAT}^{\text{SEARCH}} \preceq_T \text{SAT}^{\text{DEC}}$ se abbiamo un oracolo $\mathcal{MB}_{\text{SAT}}$) e dato che appunto abbiamo l'oracolo $\mathcal{MB}_{\text{SAT}}$, possiamo usare l'algoritmo $\text{SAT}^{\text{SEARCH}}\text{-SOLVER}$ per risolvere $\text{SAT}^{\text{SEARCH}}$, e di conseguenza $\mathbb{X}^{\text{SEARCH}}$. \square

Notiamo che per la dimostrazione abbiamo dovuto usare entrambi i requisiti di **NPC**: se \mathbb{X} non fosse **NP-HARD** non potrei ottenere l'oracolo $\mathcal{MB}_{\text{SAT}}$, quindi non potrei fare la mappatura, se invece \mathbb{X} non fosse **NP** non potrei mapparlo a SAT , che è self-reducible; per questi motivi, per i problemi **NPC** sicuramente vale il teorema, ma non è ancora sufficiente per dimostrare la self-reducibility anche per i soli problemi **NP**.

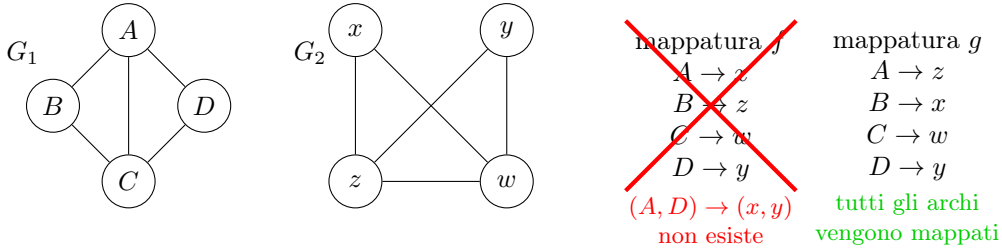
Esempio

Data un'istanza $(G, k) \in \mathcal{I}(\text{CLIQUE})$ posso mapparla tramite il teorema di Cook-Levin ad un circuito $C_B^{(G,k)}$ e conseguentemente ad una istanza $\phi_{C_B^{(G,k)}}$ o semplicemente $\phi \in \mathcal{I}(\text{SAT})$, dove $\phi(v_1 \dots v_n, y_1 \dots y_t)$ è costruita usando ogni vertice $v_i \in V(G)$ ed ogni variabile *spuria* y (quelle interne) ottenuta dal circuito. Per trovare l'assegnamento per ϕ uso $\text{SAT}^{\text{SEARCH}}\text{-SOLVER}$, che usa l'oracolo $\mathcal{MB}_{\text{SAT}}$, il quale è ottenibile da $\mathcal{MB}_{\text{CLIQUE}}$ dato che $\text{SAT} \preceq \text{CLIQUE}$. Trovato l'assegnamento per ϕ torno indietro e ottengo la clique che risolve $\text{CLIQUE}^{\text{SEARCH}}$, quindi CLIQUE è self-reducible

7.3 GRAPHISOMORPHISM è self-reducible**GRAPHISOMORPHISM**

INPUT un grafo $G_1 = (V_1, E_1)$ e un grafo $G_2 = (V_2, E_2)$

OUTPUT **yes** $\iff G_1$ è isomorfo a G_2 , cioè se esiste una mappatura $f : V_1 \rightarrow V_2$ biiettiva tale che $(u, v) \in E_1 \iff (f(u), f(v)) \in E_2$, cioè esiste una mappatura 1:1 di un grafo ad un altro.

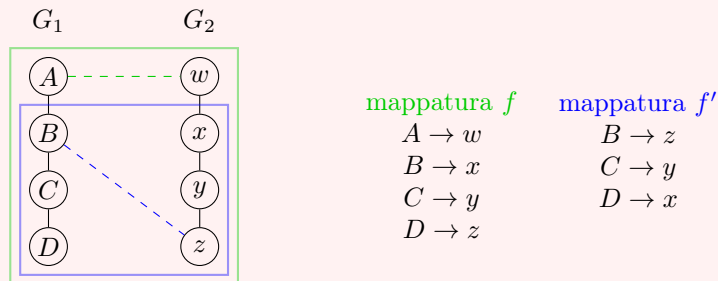


Notiamo che non necessariamente esiste un solo isomorfismo tra due grafi: nell'esempio sopra infatti, un'altra mappatura poteva avere $A \rightarrow w$ e $C \rightarrow z$. Ad oggi non si sa ancora se $\text{GRAPHISOMORPHISM} \in \mathbf{NPC}$, probabilmente no, non sappiamo nemmeno a che classe appartiene, ma intanto dimostriamo che è self-reducible.

Questa volta dobbiamo usare un approccio differente, infatti non possiamo rimuovere un vertice alla volta e chiedere all'oracolo se stiamo seguendo la strada giusta, in quanto in due grafi è possibile che ci siano più *sotto-isomorfismi* possibili, e rimuovere un nodo potrebbe farci passare da uno all'altro, mantenendo il mapping 1:1 per ogni vertice, ma creando un'inconsistenza globale.

Esempio

Notiamo che per i grafi G_1 e G_2 esiste una mappatura f , e che quindi sono isomorfi. Adottassimo l'approccio di rimozione, una volta tolti i vertici A e w nei due sotto grafi l'algoritmo potrebbe seguire una nuova mappatura f' , che li rende comunque isomorfi a detta dell'oracolo, ma in modo inconsistente rispetto alla rimozione precedente.



Adottiamo quindi un approccio di espansione, in cui alla i -esima coppia di vertici u e v che controlliamo aggiungiamo una *stella* S di n^2i vertici ad ognuno, dove $n = |V(G)| = |V(H)|$. In questo modo, dato che abbiamo aggiunto molti più vertici di quelli presenti prima e che solo u e v ne hanno esattamente così tanti, abbiamo la certezza che quei due vertici sono mappati tra loro (anche se i grafi fossero completamente connessi avremmo comunque tantissimi vertici aggiuntivi) e che ad ogni futura coppia non si utilizzino sotto-isomorfismi inconsistenti con quelli precedenti, proprio perché adesso abbiamo le stelle che identificano i vertici già controllati. L'algoritmo è il seguente:

Algorithm 7 GRAPHISOMORPHISM^{SEARCH}-SOLVER**Require:** un grafo G e un grafo H

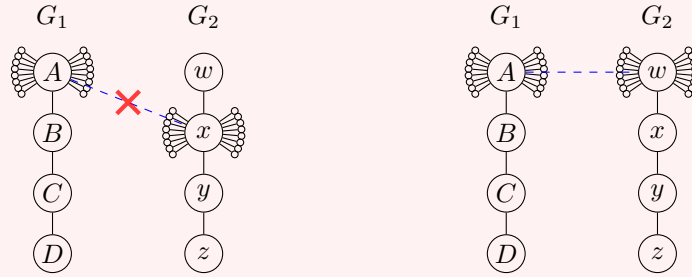
```

1: if  $\mathcal{MB}_{\text{GI}}(G, H) = \text{no}$  then return no
2:  $R_G \leftarrow \emptyset, \quad H_G \leftarrow \emptyset$ 
3: for all vertice  $u \in V(G) \setminus R_G$  do
4:   for all vertice  $v \in V(H) \setminus R_H$  do
5:     if  $\mathcal{MB}_{\text{GI}}(G \oplus S_u(n^2i), H \oplus S_v(n^2i)) = \text{yes}$  then
6:       imposta  $f(v) = u$ 
7:       // aggiorno i grafi
8:        $G \leftarrow G \oplus S_u(n^2i), \quad H \leftarrow H \oplus S_v(n^2i)$ 
9:       // vertici già controllati
10:       $R_G \leftarrow R_G \cup \{u\}, \quad R_H \leftarrow R_H \cup \{v\}$ 
11:     else
12:       return no
13: return la mappatura  $f$ 

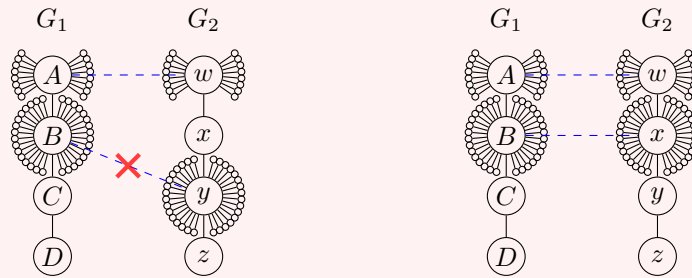
```

Esempio

Per gli stessi grafi G_1 e G_2 di prima, adottiamo ora l'algoritmo: per A mostriamo come viene gestito l'accoppiamento sbagliato con x e per contrapposizione quello corretto con w ; successivamente mostriamo la stessa cosa anche per B , prima accoppiandolo erroneamente con y e poi correttamente con x .



Notiamo che appunto nel primo caso A ha grado 17, mentre x ha grado 18, quindi non potrà esistere un isomorfismo che accoppia A ad x ; nel secondo caso invece sia A che w hanno grado 17, quindi i grafi vengono aggiornati e si prosegue controllando B usando una stella di 32 vertici. I passi per C e D non vengono mostrati.



Per ogni coppia di vertici faccio una chiamata all'oracolo \mathcal{MB}_{GI} , quindi $O(n^2)$ chiamate, e aggiungo un numero polinomiale di vertici per formare le stelle, al massimo $O(n \cdot n^2)$ vertici poiché l'ultima coppia è per forza l' n -esima, quindi la complessità dell'algoritmo è $O(n^3) = \text{poly}(|(G, H)|)$ e GRAPHISOMORPHISM è self-reducible.

7.4 Secondo teorema di self-reducibility**Teorema**

Se $\mathbf{P} \neq \mathbf{NP} \cap \text{co-NP}$ allora $\exists \mathbf{A} \in \mathbf{NP}$ che non è self-reducible (supponendo che esista $\mathcal{MB}_{\mathbb{B}}$)

Dimostrazione. Supponiamo che esista un problema $\mathbb{B} \in (\mathbf{NP} \cap \mathbf{co-NP}) \setminus \mathbf{P}$ e con questo costruiamo un problema $\mathbb{A} \in \mathbf{P} \subseteq \mathbf{NP}$ (come nel teorema) non self-reducible. Per una certa costante c , dato che $\mathbb{B} \in \mathbf{NP}$ abbiamo un verificatore $V^{\text{yes}}(x, w)$ tale che

$$\exists w \mid w \in \{0, 1\}^{|x|^c} \wedge V^{\text{yes}}(x, w) = \mathbf{T} \iff \mathbb{B}(x) = \text{yes}$$

e dato che $\mathbb{B} \in \mathbf{co-NP}$ abbiamo anche un verificatore $V^{\text{no}}(x, w)$ tale che

$$\exists w \mid w \in \{0, 1\}^{|x|^c} \wedge V^{\text{no}}(x, w) = \mathbf{T} \iff \mathbb{B}(x) = \text{no}.$$

Ora definiamo \mathbb{A} in modo che

$$\mathbb{A}(x) = \text{yes} \iff \exists w \mid V^{\text{yes}}(x, w) \vee V^{\text{no}}(x, w) = \mathbf{T}$$

in pratica, per ogni istanza x abbiamo $\mathbb{A}(x) = \text{yes}$ e quindi ovviamente $\mathbb{A} \in \mathbf{P}$ poiché basta scrivere un algoritmo che ritorna sempre **yes**, che è tempo costante; inoltre, dato che \mathbb{A} è anche \mathbf{NP} , esiste un verificatore per \mathbb{A} , definibile come $V_{\mathbb{A}}(x, w) = V^{\text{yes}}(x, w) \vee V^{\text{no}}(x, w)$.

Se per assurdo \mathbb{A} fosse self-reducible allora per definizione esisterebbe un algoritmo T che in tempo polinomiale calcola w per x , in formule $T(x) = w \mid V_{\mathbb{A}}(x, w) = \mathbf{T}$, quindi dato che $V_{\mathbb{A}}(x, w) = V^{\text{yes}}(x, w) \vee V^{\text{no}}(x, w)$ almeno uno dei due verificatori deve essere \mathbf{T} : con l'algoritmo polinomiale T potrei costruire un algoritmo *polinomiale* per \mathbb{B}^{DEC}

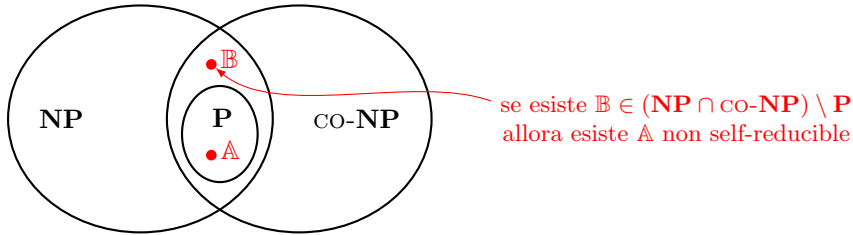
Algorithm 8 \mathbb{B}^{DEC} -SOLVER

Require: una qualsiasi istanza $x \in \mathcal{I}(\mathbb{B})$

- 1: $w \leftarrow T(x)$
 - 2: **if** $V^{\text{yes}}(x, w) = \mathbf{T}$ **then**
 - 3: **return** **yes** // $\mathbb{B}(x) = \text{yes}$
 - 4: **else**
 - 5: **return** **no** // $\mathbb{B}(x) = \text{no}$
-

quindi $\mathbb{B} \in \mathbf{P}$, assurdo per ipotesi, quindi \mathbb{A} non può essere self-reducible. □

Abbiamo già dimostrato in precedenza che probabilmente $\text{SMALLFACTOR} \in (\mathbf{NP} \cap \mathbf{co-NP}) \setminus \mathbf{P}$, quindi probabilmente esiste almeno un problema $\mathbb{A} \in \mathbf{NP}$ che non è self-reducible. Al contrario, FACTOR (il problema di fattorizzare un numero), che purtroppo crediamo essere in $\mathbf{NP} \setminus \mathbf{P}$, non crediamo sia self-reducible, infatti $\text{FACTOR}^{\text{DEC}}$ (il problema di dire se un numero è fattorizzabile o meno, cioè se è primo) è in \mathbf{P} , ma non crediamo che $\text{FACTOR}^{\text{SEARCH}}$ (dare i fattori di un numero) si possa ridurre a $\text{FACTOR}^{\text{DEC}}$.



7.5 Terzo teorema di self-reducibility

Usando delle ipotesi più deboli, se $\mathbf{P} \neq \mathbf{NP}$ il problema di $\text{COLORABILITÀ PROPRIA DI UN GRAFO PLANARE}$ è \mathbf{NP} ma non self-reducible.

8 | Algoritmi di approssimazione e Inapprossimabilità

Tutti i problemi visti fino ad ora erano di decisione (dire se un'istanza ha soluzione o meno) o di ricerca (dare la soluzione di una istanza), ma spesso è necessario trovare la *migliore* soluzione rispetto alle altre: introduciamo i problemi di **ottimizzazione**, nei quali troviamo vincoli più restrittivi sullo spazio di ricerca, che richiedono i valori *massimi* o *minimi*.

8.1 Problemi di ottimizzazione

Cominciamo innanzitutto prendendo come esempio il problema VERTCOV ed esponendo una sua possibile definizione come problema di ricerca, ponendo attenzione sulle sottili differenze con le altre definizioni.

VERTCOV	
INPUT	un grafo G e un parametro k
OUTPUT	<p>DECISIONE: $\text{yes} \iff$ esiste un insieme A di vertici che coprono ogni arco di G tale che $A \leq k$;</p> <p>RICERCA: un insieme A di vertici che coprono ogni arco di G tale che $A \leq k$;</p> <p>OTTIMIZZAZIONE: il più piccolo insieme A di vertici che coprono ogni arco di G.</p>

Notiamo che ovviamente $\text{VERTCOV}^{\text{DEC}} \preceq_T \text{VERTCOV}^{\text{SEARCH}} \preceq_T \text{VERTCOV}^{\text{OPT}}$, inoltre sfruttando il fatto che VERTCOV è self-reducible (non lo dimostriamo), supponendo che esista l'oracolo $\mathcal{MB}_{\text{VERTCOV}}$, abbiamo che $\text{VERTCOV}^{\text{OPT}} \preceq_T \text{VERTCOV}^{\text{SEARCH}} \preceq_T \text{VERTCOV}^{\text{DEC}}$.

I problemi di ottimizzazione sono i più difficili ma anche quelli più comuni nei casi reali, quindi dobbiamo trovare un modo per risolverli il meglio possibile in tempo utile: introduciamo il concetto di **algoritmo di approssimazione**, che ci permette di risolvere il problema ritornando una soluzione “quasi” ottimale.

8.2 Approssimazione

Dato un problema \mathbb{A} , usiamo come al solito la notazione $A(x)$ per indicare la soluzione ritornata sull'istanza x dall'algoritmo A per \mathbb{A} , e introduciamo la notazione

Soluzione ottimale $\text{OPT}(x)$

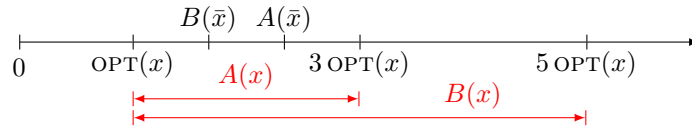
Dato un problema \mathbb{A} , indichiamo con $\text{OPT}(x)$ la soluzione ottimale dell'istanza x .

Impostiamo inoltre una funzione $\text{val}(w)$ per mappare ogni soluzione w ad un valore v , che indica la qualità della soluzione. Da ora in poi, senza perdita di generalità, abuseremo della notazione $A(x)$ e $\text{OPT}(x)$ per indicare rispettivamente $\text{val}(A(x))$ e $\text{val}(\text{OPT}(x))$, poiché siamo più interessati alla qualità della soluzioni piuttosto che delle soluzioni in se.

Per valutare la *bontà* dell'algoritmo A studiamo la **garanzia di approssimazione r** :

- dato il problema di *minimizzazione* \mathbb{A} chiamiamo l'algoritmo A una r -approssimazione di \mathbb{A} se per ogni istanza x abbiamo $\frac{A(x)}{\text{OPT}(x)} \leq r$;
- dato il problema di *massimizzazione* \mathbb{A} chiamiamo l'algoritmo A una r -approssimazione di \mathbb{A} se per ogni istanza x abbiamo $\frac{\text{OPT}(x)}{A(x)} \leq r$

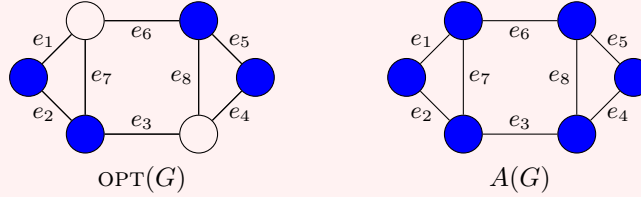
Questi algoritmi saranno approssimazioni *per ogni* istanza di \mathbb{A} , quindi se A 3-approssima \mathbb{A} mentre B 5-approssima \mathbb{A} possiamo avere un certo \bar{x} per cui $B(\bar{x}) < A(\bar{x})$, poiché appunto una r -approssimazione significa che un algoritmo *più di così* non può fare, ma *di meno* è ovviamente possibile (ad esempio B potrebbe essere ottimale proprio su quella specifica istanza \bar{x}). Notiamo che quindi se A 3-approssima \mathbb{A} , sicuramente anche 5-approssima.



Idealmente vorremmo un algoritmo che 1-approssima \mathbb{A} , ma se $\mathbf{P} \neq \mathbf{NP}$ (come crediamo) non possiamo avere l'oracolo $\mathcal{MB}_{\mathbb{A}}$ e quindi la riduzione $\mathbb{A}^{\text{OPT}} \preceq_T \mathbb{A}^{\text{SEARCH}} \preceq_T \mathbb{A}^{\text{DEC}}$ non vale, il che rende \mathbb{A}^{OPT} un problema difficile e non risolvibile in tempo utile. In questo caso, cerchiamo di avere algoritmi di approssimazione con valori di r il più possibile vicini a 1, così da avere algoritmi che ottimizzano nel miglior tempo possibile.

Esempio

Diamo un algoritmo A per VERTCOV che seleziona un arco non coperto, aggiunge entrambi i suoi vertici nella soluzione e ripete finché tutti gli archi non sono coperti.



Usiamo come valore il numero di vertici della soluzione e notiamo che $\text{OPT}(G) = 4$ mentre $A(G) = 6$ (considerando di aver scorso in ordine gli archi), quindi l'algoritmo A è una 1.5-approssimazione per l'istanza G di VERTCOV. In generale, per ogni istanza di VERTCOV, A sicuramente darà una soluzione valida (e per alcune istanze anche quella ottimale), quindi mostriamo che l'algoritmo A garantisce 2-approssimazione, cioè che per ogni istanza G abbiamo $\frac{A(G)}{\text{OPT}(G)} \leq 2$.

Dato che l'algoritmo non può mai superare l'ottimalità, supponiamo l'esistenza di un valore lower bound $LB \leq \text{OPT}(G)$ e dato che vogliamo dimostrare una 2-approssimazione, impostiamo $A(G) \leq 2LB \leq 2\text{OPT}(G)$. Consideriamo gli archi che A sceglie da coprire: l'algoritmo funziona per *matching* degli archi, aggiungendo i vertici solo a coppie, e dato che nessun arco collegato a quei vertici verrà mai preso in considerazione (sono già coperti), tutti gli archi scelti sono disgiunti tra loro. Sia M l'insieme di archi presi in considerazione da A : sicuramente $A(G) \leq 2M$ poiché per ogni arco $(u, v) \in M$, u e v fanno parte della soluzione, mentre $\text{OPT}(G) \geq |M|$ poiché dato che gli archi sono disgiunti la soluzione ottimale deve considerare almeno un vertice per ogni arco considerato da A . Da questo ne deriva che $\frac{A(G)}{\text{OPT}(G)} \leq \frac{2M}{M} = 2$, quindi A è una 2-approssimazione per \mathbb{A} , dove $|M| = LB$.

Quello che crediamo è che $\forall \varepsilon > 0$ non esista un algoritmo di $(2 - \varepsilon)$ -approssimazione per VERTCOV, il che significa che per quanto banale possa sembrare l'algoritmo A sopra mostrato, non è dimostrabile che esista un algoritmo che è *sempre* meglio di A .

Mostriamo ora una 1-approssimazione ideale, introducendo un nuovo problema:

MAKESPAN (schedulizzazione)

INPUT n job ognuno con tempo di esecuzione J_i da distribuire su m macchine

OUTPUT assegnamento dei job alle macchine che distribuisca il meglio possibile i job tra le macchine, ovvero tale che $\max_{\ell} L(\ell)$ sia il minimo possibile, dove $L(\ell) = \sum_{\text{job per } \ell} J_i$ è il carico della macchina ℓ .

In pratica voglio completare l'esecuzione di ogni job nel minor tempo possibile: questo purtroppo è un problema **NPC**, poiché il problema di decisione associato è anch'esso **NPC**. La soluzione ideale sarebbe dividere in maniera perfettamente equa i job tra le macchine, ma non è sempre possibile.

Esempio

Dati $J_i = \{5, 7, 2, 14, 9, 11, 7\}$ e $m = 3$, possiamo assegnare $J_1, J_6 \in M_1$, $J_4 \in M_2$ e $J_2, J_3, J_5, J_7 \in M_3$. In questo modo $L(M_1) = 16$, $L(M_2) = 14$ ed $L(M_3) = 25$, quindi la soluzione avrà un makespan di $\max_{\ell} L(\ell) = 25$. Se invece assegniamo $J_1, J_4 \in M_1$, $J_2, J_6 \in M_2$ e $J_3, J_5, J_7 \in M_3$ abbiamo $L(M_1) = 19$, $L(M_2) = 18$ ed $L(M_3) = 18$, quindi un maxspan di $\max_{\ell} L(\ell) = 19$, che è la soluzione ottimale.

Soffermiamoci per curiosità su un caso particolare, con n job e 2 macchine: questo è il problema che abbiamo visto all'inizio del corso

PARTITION**INPUT** interi $a_1 \dots a_n$ **OUTPUT** un insieme di indici $I \subseteq \{1 \dots n\}$ tale che $\sum_{j \in I} a_j = \frac{a_1 + \dots + a_n}{2}$, cioè l'insieme di numeri che sommati tra loro danno la metà della somma di tutti i numeri

Notiamo che $\text{SUBSETSUM} \preceq \text{PARTITION} \preceq \text{MAKESPAN}^{\text{DEC}}$, quindi se $\mathbf{P} \neq \mathbf{NP}$ non esiste un algoritmo polinomiale per MAKESPAN (che è il motivo per cui $\text{MAKESPAN} \in \mathbf{NPC}$), e quindi cerchiamo un algoritmo che lo approssimi il meglio possibile.

Data un'istanza $x = (m, J_1 \dots J_n)$, definiamo il carico totale come $\text{TOT} = \sum_{i=1}^n J_i$, di conseguenza sicuramente $\text{OPT}(x) \geq \frac{\text{TOT}}{m}$, e quindi usiamo $\frac{\text{TOT}}{m}$ (il carico medio ideale per ogni macchina) come lower bound. Dividiamo i job in due sottoinsiemi rispetto ad una frazione del carico totale: scelta una certa soglia T costante (quindi indipendente dal numero n di job), chiamiamo i *job pesanti* $P = \{J_i \mid J_i \geq \frac{\text{TOT}}{T}\}$ e i *job leggeri* $L = \{J_i \mid J_i < \frac{\text{TOT}}{T}\}$. Scriviamo l'algoritmo in due fasi:

- ① risolvi in maniera ottima il problema sul sottoinsieme P ;
- ② assegna iterativamente i job in L alla macchina con minor carico (in maniera greedy).

Ora valutiamo il tempo di esecuzione (che vogliamo essere polinomiale) ...

- nella fase ① abbiamo $|P| \leq T$ (se per assurdo $|P| > T$ avremmo T job tali che $J_i \geq \frac{\text{TOT}}{T}$ la cui somma è maggiore del totale di *tutti* gli n job, impossibile ovviamente), quindi costa $O(m^T)$ (per ognuno degli T job bisogna assegnare una macchina tra le m disponibili) e dato che T è costante, questa fase è polytime;
- nella fase ② semplicemente scorro sulle macchine ed assegno ogni job in L , quindi costa $O(n \cdot m)$, che è di nuovo polytime;

...e il valore della soluzione prodotta (che vogliamo rimanga piccolo), indicizzando con M la macchina di massimo carico alla fine dell'algoritmo:

- se alla fine della fase ① abbiamo già $L = \emptyset$ non ci sono altri job da assegnare e la soluzione è già ottima;
- se M non ha job leggeri allora la soluzione è di nuovo ottima poiché tutti job piccoli sono stati distribuiti nelle altre macchine;
- se invece M ha assegnato un job $j \in L$, siccome i job leggeri vengono assegnati alla macchina con il minor carico (per definizione), allora tolto j da M tutte le altre macchine avranno carico almeno uguale a M , cioè $\forall \ell \neq M, L(M) - j \leq L(\ell)$, quindi

$$\begin{aligned} \text{TOT} &= \overbrace{L(M) + \sum_{\ell \neq M} L(\ell)}^{\text{carico effettivo}} \geq \overbrace{(m-1)(L(M) - j) + L(M)}^{\text{carico stimato}} \\ &\geq \frac{\text{TOT}}{m} \geq L(M) - \frac{m-1}{m} j && \text{manipolazione e tratto } m \\ &\geq L(M) - \frac{m-1}{m} \frac{\text{TOT}}{T} && \text{per def. } j < \frac{\text{TOT}}{T} \\ L(M) &\leq \left(1 + \frac{m-1}{T}\right) \frac{\text{TOT}}{m} && \text{esplicitiamo } L(M) \\ &\leq \left(1 + \frac{m-1}{T}\right) \text{OPT}(x) && \text{per def. } \text{OPT}(x) \geq \frac{\text{TOT}}{m} \end{aligned}$$

Soluzione dell'algoritmo è quindi al più $1 + \frac{m-1}{T}$ volte l'ottimo, cioè garantisce $(1 + \frac{m-1}{T})$ -approssimazione e, maggiore scelgo T , migliore sarà l'approssimazione, quindi impostando $\varepsilon \geq \frac{m-1}{T} > 0$ abbiamo la soglia $T \geq \frac{m-1}{\varepsilon}$, che tende alla 1-approssimazione. Ad esempio, supponendo di volere 1.5-approssimazione, impostiamo $\varepsilon = 0.5$ e quindi prendiamo $T \geq 2(m-1)$. Questa scelta ha più senso se vista dall'altro lato: le macchine a mia disposizione lavorano al meglio con una certa soglia T , di conseguenza avrò al più una certa $(1 + \varepsilon)$ -approssimazione.

Come detto prima, il tempo di esecuzione dell'intero algoritmo è $O(m^T)$, ovvero $O(m^{\frac{m-1}{\varepsilon}})$, quindi in generale cresce esponenzialmente con $\frac{1}{\varepsilon}$, ma quando m e ε sono costanti l'algoritmo è polytime.

In conclusione, per MAKESPAN abbiamo un algoritmo che $(1 + \varepsilon)$ -approssima per un ε scelto a piacere, e quindi è molto più potente di quello proposto per VERTCOV .

8.3 Classi di approssimazione

Abbiamo visto che VERTCOV è 2-approssimabile, ovvero che esiste un algoritmo A per VERTCOV tale che $r_A \leq 2$, mentre MAKESPAN è $(1 + \varepsilon)$ -approssimabile $\forall \varepsilon > 0$ dall'algoritmo A_ε , che ha complessità $O(\text{poly}(m, n)2^{1/\varepsilon})$, quando le macchine m sono in numero costante. Dividiamo ora questi problemi in classi di complessità rispetto

alla potenza dell'approssimazione che hanno, e mostriamo che si può fare di meglio rispetto a **MAKESPAN** e che invece a volte non si può proprio approssimare.

8.3.1 Problema **KNAPSACK**

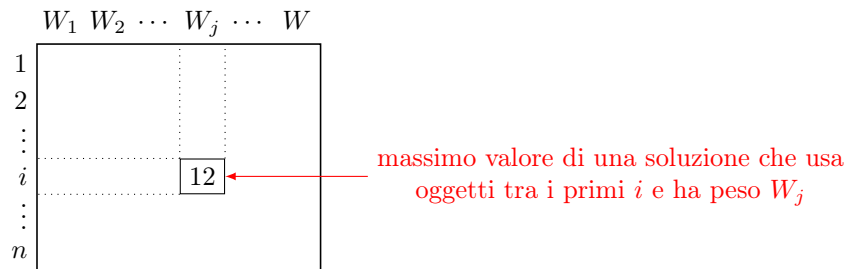
KNAPSACK

INPUT n oggetti, ognuno di valore v_i e peso w_i , e la capacità massima w dello zaino

OUTPUT un insieme di indici $O \subseteq \{1 \dots n\}$ tale che $\sum_{i \in I} w_i \leq w$ e $\sum_{i \in O} v_i \geq \sum_{i \in O'} v_i$, ovvero l'insieme di oggetti che ha valore massimo nella capacità dello zaino.

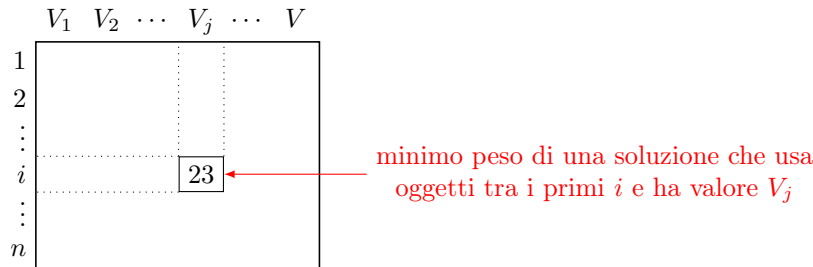
Dato che **KNAPSACK** è di massimizzazione, è **NP-HARD**, ed è dimostrabile che $\text{SUBSETSUM} \preceq \text{KNAPSACK}^{\text{DEC}}$ (mappando $v_i = w_i$, lo vedremo dopo) e che quindi $\text{SUBSETSUM} \preceq_T \text{KNAPSACK}$.

Possiamo risolvere **KNAPSACK** in programmazione dinamica e classicamente si usa una matrice in cui le righe sono gli indici i degli oggetti e le colonne sono pesi W_j tutti inferiori alla capacità massima W ; ogni cella (i, j) indica il valore della migliore soluzione utilizzando un sottoinsieme dei primi i oggetti e di peso totale al massimo W_j .



Nota questa matrice, l'algoritmo semplicemente scorre su ogni cella e trova la più alta, impiegando tempo $O(nW)$, ma se W è esponenziale rispetto a n allora l'algoritmo è esponenziale, e non c'è molto che possiamo fare per sistemare la situazione.

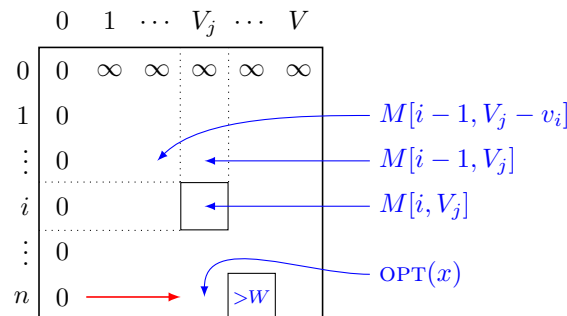
Usiamo quindi un altro algoritmo in programmazione dinamica, sempre con una matrice, ma usando per le colonne valori V_j tutti inferiori al valore totale possibile $V = \sum_{i=1}^n v_i$; ogni cella (i, j) indica il peso minimo della soluzione che utilizza un sottoinsieme dei primi i oggetti e ha valore V_j .



Chiamiamo $M[i, V_j]$ la cella e ne diamo una definizione più formale:

$$M[i, V_j] = \begin{cases} 0 & \text{se } V_j \leq 0 \\ +\infty & \text{se } V_j > 0, i = 0 \\ \min\{M[i-1, V_j], w_i + M[i-1, V_j - v_i]\} & \text{altrimenti} \end{cases}$$

in questo modo possiamo computare ogni cella in base alle righe e alle colonne precedenti, scegliendo di prendere l'oggetto i solo se aggiungendo il suo peso w_i non superiamo la migliore soluzione finora (quella in cui i è escluso)



e una volta riempita l'ultima riga, conosco il minimo peso di ogni possibile configurazione che mi permette di usare tutti gli oggetti. A partire da sinistra, scorro tutta l'ultima riga finché non trovo una cella di peso maggiore rispetto a W : la cella precedente sarà quella della soluzione ottima, quindi $\text{OPT}(x) = M[n, V_j]$ dove $V_j = \max_{V_j} (M[n, V_j] \leq W)$. L'algoritmo deve costruire tutte le celle della matrice, quindi ha costo $O(nV)$ come prima, ma se usiamo il valore massimo $v_{\max} = \max_i v_i$ tra quelli di tutti gli oggetti notiamo che $V \leq nv_{\max}$, e quindi l'algoritmo è $O(n^2 v_{\max})$. In conclusione abbiamo un algoritmo ottimo per KNAPSACK, e se v_{\max} è polinomiale in n allora l'algoritmo è polytime, quindi KNAPSACK è un problema **NP**-HARD debole, poiché appunto la sua difficoltà dipende dalla magnitudo dei valori.

Sfruttiamo questo punto debole del problema per forzare tutte le istanze ad essere polinomiali: se riscriviamo ogni valore come $v_i = b\nu_i$ con ν_i polinomiale in n abbiamo una famiglia di soluzioni ottime al variare dei valori di un fattore comune $b > 0$, quindi data una qualsiasi istanza basta ridurre i valori e risolvere questa nuova istanza.

Data l'istanza $I = (v_1 \dots v_n, w_1 \dots w_n, w)$ e assumiamo che tutti gli oggetti abbiano peso $w_i \leq W$ (tanto sono comunque eliminabili, non saranno in nessuna soluzione e quindi non servono); definiamo le istanze

$$\hat{I} = (\hat{v}_1 \dots \hat{v}_n, w_1 \dots w_n, w) \text{ dove } \hat{v}_i = \left\lceil \frac{v_i}{\mu} \right\rceil$$

$$\tilde{I} = (\tilde{v}_1 \dots \tilde{v}_n, w_1 \dots w_n, w) \text{ dove } \tilde{v}_i = \left\lceil \frac{v_i}{\mu} \right\rceil \mu$$

e dato che \hat{I} può essere resa polinomiale in n impostando μ in modo opportuno, possiamo risolvere \hat{I} in polytime. Notiamo che ogni soluzione ottima per \hat{I} è ottima anche per \tilde{I} poiché cambiano solo del fattore di scala μ come detto sopra, quindi ora manca solo una relazione con I : notiamo che

$$\textcircled{1} \quad v_i \leq \tilde{v}_i \leq \left(\frac{v_i}{\mu} + 1 \right) \mu = v_i + \mu,$$

e se prendiamo un'insieme di oggetti con valori $\tilde{v}_{i_1} \dots \tilde{v}_{i_k}$ avremo che

$$\tilde{v}_{i_1} + \dots + \tilde{v}_{i_k} \leq v_{i_1} + \dots + v_{i_k} + n\mu$$

quindi la mutazione dei valori da parte di \tilde{I} può aumentare il valore totale di I al massimo di $n\mu$. Dato che sicuramente $\text{OPT}(I) \geq v_{\max}$ abbiamo $n\mu \leq \varepsilon v_{\max} \leq \varepsilon \text{OPT}(I)$, quindi

$$\textcircled{2} \quad \mu = \frac{\varepsilon v_{\max}}{n}$$

e con questo abbiamo $\hat{v}_i = \left\lceil \frac{v_i n}{\varepsilon v_{\max}} \right\rceil = O\left(\frac{n}{\varepsilon}\right)$ data una $\varepsilon > 0$ costante, quindi \hat{I} si può risolvere in polytime $O\left(\frac{n^3}{\varepsilon}\right)$ dall'algoritmo e per quanto detto prima la soluzione per \hat{I} sarà la stessa anche per \tilde{I} , quindi anche \tilde{I} è risolvibile in polytime; chiamiamo $O_{\tilde{I}} \subseteq \{1 \dots n\}$ la soluzione per \tilde{I} . Ora vogliamo mostrare che il valore $\text{VAL}_{DP} = \sum_{i \in O_{\tilde{I}}} v_i$ ottenuto risolvendo \tilde{I} non è molto più piccola del nostro obiettivo $\text{OPT}(I) = \sum_{i \in O_I} v_i$, e che quindi possiamo usare $O_{\tilde{I}}$ come approssimazione per I : chiamiamo O_I la soluzione ottima di I , quindi

$$\begin{aligned} \text{OPT}(I) &= \sum_{i \in O_I} v_i && \text{def. di } \text{OPT}(I) \\ &\leq \sum_{i \in O_I} \tilde{v}_i \leq \sum_{i \in O_{\tilde{I}}} \tilde{v}_i \leq \sum_{i \in O_{\tilde{I}}} (v_i + \mu) && \text{sfruttando } \textcircled{1} \\ &\leq \mu |O_{\tilde{I}}| + \sum_{i \in O_{\tilde{I}}} v_i = \text{VAL}_{DP} + \mu |O_{\tilde{I}}| && \text{def. di } \text{VAL}_{DP} \\ &\leq \text{VAL}_{DP} + n\mu = \text{VAL}_{DP} + \varepsilon v_{\max} && \text{sfruttando } \textcircled{2} \\ &\leq \text{VAL}_{DP} + \varepsilon \text{OPT}(I) && \text{poiché } \text{OPT}(I) \geq v_{\max} \end{aligned}$$

quindi in conclusione $\text{OPT}(I) \leq \text{VAL}_{DP} + \varepsilon \text{OPT}(I)$ e di conseguenza $\text{VAL}_{DP} \geq (1 - \varepsilon) \text{OPT}(I)$; calcoliamo il valore r

$$\frac{\text{OPT}(I)}{\text{VAL}_{DP}} \leq \frac{1}{1 - \varepsilon} \leq (1 + \varepsilon') \quad \text{per un certo } \varepsilon' \geq \frac{\varepsilon}{1 - \varepsilon} \text{ e } \varepsilon' > 0$$

quindi questo algoritmo in programmazione dinamica $(1 + \varepsilon')$ -approssima KNAPSACK.

Riassumendo, per ogni fissato $\varepsilon > 0$ possiamo ottenere una $(1 + \varepsilon)$ -approssimazione per KNAPSACK in tempo $O\left(\frac{n^3}{\varepsilon}\right)$: data l'istanza I , impostiamo $\mu = \frac{\varepsilon v_{\max}}{n}$, creiamo \hat{I} mappando $\hat{v}_i = \left\lceil \frac{v_i}{\mu} \right\rceil$, e infine risolviamo \hat{I} in

polytime $O(\frac{n^3}{\varepsilon})$. L'algoritmo è polinomiale sia in n che in ε^{-1} , e quindi “migliore” sia rispetto a **MAKESPAN** che ovviamente rispetto a **VERTCOV**.

Tutto questo è possibile poiché **KNAPSACK** è debolmente **NP-HARD**, quindi se è polinomiale per numeri piccoli possiamo scalarli a numeri grandi per risolvere polinomialmente ogni istanza, con l'algoritmo *pseudo-polytime* che abbiamo visto. Al contrario invece, come vedremo, un problema fortemente **NP-HARD** non può avere un algoritmo pseudo-polytime e quindi non può esistere un algoritmo di $(1 + \varepsilon)$ -approssimazione polinomiale sia in $|x|$ che in ε^{-1} come per **KNAPSACK**.

8.3.2 Riduzione **SUBSETSUM** \preceq **KNAPSACK**^{DEC}

Dimostriamo **SUBSETSUM** \preceq **KNAPSACK**^{DEC}, dove per il secondo la risposta è **yes** se esiste una soluzione $O \subseteq \{1 \dots n\}$ tale che $\sum_{i \in O} w_i \leq W$ e $\sum_{i \in O} v_i \geq k$, mentre per il primo la soluzione per $(a_1 \dots a_n, S) \in \mathcal{I}(\text{SUBSETSUM})$ è un sottoinsieme $A \subseteq \{1 \dots n\}$ tale che $\sum_{i \in A} a_i = S$. La mappatura è semplice, basta fissare $v_i = a_i$, $w_i = a_i$ e $k = S$, $W = S$: così facendo, ora avrò $\sum_{i \in O} a_i \geq S$ (condizione sui valori) e $\sum_{i \in O} a_i \leq S$ (condizione sulla capienza), quindi sto chiedendo esattamente $\sum_{i \in O} a_i = S$, e quindi posso risolvere **SUBSETSUM** usando **KNAPSACK**.

8.3.3 Classi **APX**, **PTAS** e **FPTAS**

Per adesso abbiamo visto che **VERTCOV** è 2-approssimabile, e non possiamo fare di meglio, mentre per **MAKESPAN** e **KNAPSACK** abbiamo $(1 + \varepsilon)$ -approssimazione, ma con una importante differenza: **MAKESPAN** è polytime nella taglia n dell'input, mentre **KNAPSACK** è polytime sia in n che in ε^{-1} . Rispetto a queste tre tipologie di problemi, definiamo le seguenti classi di approssimazione (per semplicità usiamo problemi di minimizzazione).

Approximable

La classe di complessità **APX** è definita come

$$\mathbf{APX} = \{ \mathbb{A} \mid \exists \text{ un algoritmo polytime per } \mathbb{A} \text{ che } r\text{-approssima } \mathbb{A} \}$$

cioè contiene problemi approssimabili.

Notiamo che ovviamente **VERTCOV**, **MAKESPAN** e **KNAPSACK** appartengono ad **APX** in quanto sono tutti approssimabili.

Polynomial Time Approximation Scheme

La classe di complessità **PTAS** è definita come

$$\mathbf{PTAS} = \{ \mathbb{A} \mid \exists \text{ un algoritmo } O(\text{poly}(|x|)) \text{ per } \mathbb{A} \text{ che } (1 + \varepsilon)\text{-approssima } \mathbb{A} \}$$

cioè contiene problemi approssimabili in polytime con una certa bontà ε .

Questa classe contiene **MAKESPAN** e **KNAPSACK** in quanto entrambi possono essere approssimati con una bontà arbitraria ε ; chiamiamo **PTAS** l'algoritmo che permette ad \mathbb{A} di appartenere a **PTAS**.

Fully Polynomial Time Approximation Scheme

La classe di complessità **FPTAS** è definita come

$$\mathbf{FPTAS} = \{ \mathbb{A} \mid \exists \text{ un algoritmo } O(\text{poly}(|x|, \frac{1}{\varepsilon})) \text{ per } \mathbb{A} \text{ che } (1 + \varepsilon)\text{-approssima } \mathbb{A} \}$$

cioè contiene problemi approssimabili completamente polytime con bontà ε .

Per adesso abbiamo visto solo **KNAPSACK** come problema in questa classe; anche in questo caso chiameremo **FPTAS** l'algoritmo che permette ad \mathbb{A} di appartenere a **FPTAS**.

Dalle definizioni che abbiamo appena dato, possiamo ridefinire **P** come la classe che contiene problemi approssimati in modo ottimo, ovvero con $r = 1$, quindi

$$\mathbf{P} \subseteq \mathbf{FPTAS} \subseteq \mathbf{PTAS} \subseteq \mathbf{APX}$$

8.3.4 Tutti i problemi sono approssimabili?

TRAVELINGSALESMANPROBLEM (*simmetrico*)

INPUT grafo indiretto completo K_n (con $n = |V|$ vertici) e una funzione peso $w : E \rightarrow \mathbb{Z}^+$ sugli archi

OUTPUT permutazione $i_1 \dots i_n$ dei vertici tale che $w(v_{i_n}, v_{i_1}) + \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}})$ è minima, ovvero dare il ciclo Hamiltoniano di costo minimo

Dimostriamo che TSP è **NP-HARD** tramite $\text{HAMCYCLE} \preceq \text{TSP}$: dato $G = (V, E) \in \mathcal{I}(\text{HAMCYCLE})$, creo K_n completando G e do un peso maggiore a tutti gli archi che ho aggiunto, quindi il mapping è

$$n = |V| \quad w(u, v) = \begin{cases} 1 & \text{se } (u, v) \in E \\ 2 & \text{se } (u, v) \notin E \end{cases}$$

e al solito dimostriamo la doppia implicazione.

Dimostrazione. G è Hamiltoniano $\iff \exists$ una soluzione di TSP su (K_n, w) di valore $\leq n$:

- \Rightarrow se G è Hamiltoniano, allora esiste $i_1 \dots i_n$ tale che $v_{i_1} \dots v_{i_n}$ è un ciclo Hamiltoniano per G ; lo stesso ciclo esiste anche in K_n e passa per tutti gli archi di peso 1, quindi il costo di questo ciclo è n ;
- \Leftarrow se esiste un ciclo Hamiltoniano di peso n in K_n allora sicuramente sta usando solo archi di peso 1 (questo perché se usasse anche solo un arco di costo 2 per mantenere il costo del percorso pari a n dovrebbero esserci $n - 1$ archi, ma non sarebbe più un ciclo Hamiltoniano). Dato che vengono usati solo archi di costo 1, ovvero tutti quelli che appartengono anche a G , il ciclo Hamiltoniano è valido anche per G . \square

Quindi con questa dimostrazione sappiamo che trovare la soluzione di TSP è difficile, quindi ci sarà necessario ricorrere alle approssimazioni: qui sorge il problema, poiché infatti TSP non è approssimabile in tempo utile!

Teorema

Se $\mathbf{P} \neq \mathbf{NP}$, dato $t = O(2^n)$, ogni algoritmo polinomiale per TSP non può garantire approssimazione $\leq t$

Questo significa che dato ad esempio un grafo con $n = 20$ vertici, non riusciamo nemmeno a garantire una 2^{20} -approssimazione, e quindi il problema è praticamente inapprossimabile.

Dimostrazione. Se $\mathbf{P} \neq \mathbf{NP}$, supponiamo per assurdo un algoritmo polytime A che t -approssima TSP. Data $G = (V, E) \in \mathcal{I}(\text{HAMCYCLE})$ la mappiamo a $I_G = (K_n, w) \in \mathcal{I}(\text{TSP})$ impostando

$$n = |V| \quad w(e) = \begin{cases} 1 & \text{se } e \in E \\ n(t-1) + 2 & \text{se } e \notin E \end{cases}$$

Osserviamo che,

- come prima, se G è Hamiltoniano allora I_G ha soluzione con costo n , quindi $\text{OPT}(I_G) \leq n$ (in particolare $= n$) e siccome A è una t -approssimazione per TSP abbiamo $A(I_G) \leq t \text{OPT}(I_G) = tn$. Visto che le uniche soluzioni di costo $\leq tn$ sono i cicli Hamiltoniani per G , anche $A(I_G)$ sarà Hamiltoniano per K_n ;
- se invece G non è Hamiltoniano ogni soluzione per I_G usa almeno un arco non in E , quindi tutte le soluzioni hanno costo $\geq (n(t-1) + 2) + n - 1 = nt + 1$, e in particolare $\text{OPT}(I_G) \geq nt + 1$, quindi siccome A è una t -approssimazione per TSP abbiamo $A(I_G) \geq \text{OPT}(I_G) \geq nt + 1$.

In questo modo, se la soluzione ritornata da A ha costo $\leq nt$ allora è un ciclo Hamiltoniano in G , mentre al contrario se ha costo $\geq nt + 1$ allora non esiste un ciclo Hamiltoniano in G , e quindi dalla soluzione ritornata da A deduco con certezza se G è Hamiltoniano o meno: in questo modo, A è polytime per HAMCYCLE, quindi abbiamo un algoritmo polytime per il problema $\text{HAMCYCLE} \in \mathbf{NPC}$, e quindi $\mathbf{P} = \mathbf{NP}$, assurdo per ipotesi.

Come chiarimento finale, abbiamo bisogno che $t = O(2^n)$ per costruire in polytime l'istanza I_G , altrimenti la riduzione non sarebbe valida: il numero di bit che utilizziamo per $w(e)$ deve essere $\text{poly}(n)$, cioè $w(e) = O(2^{\text{poly}(n)})$, e quindi la riduzione è valida per ogni $t = O(2^n)$. \square

Con questo abbiamo dimostrato che se $\mathbf{P} \neq \mathbf{NP}$ allora $\text{TSP} \in \mathbf{NP-HARD}$ e $\text{TSP} \notin \mathbf{APX}$, o almeno la sua versione *simmetrica*, in cui andare da u a v costa come andare da v a u . Se modelliamo il grafo con archi diretti, otteniamo la versione di TSP *asimmetrica*, molto più utile nei casi reali (pensiamo ad una città con strade a senso unico), ma continua a non essere approssimabile (non lo mostriamo). Studiamo invece la

versione di TSP *metrica*, in cui la funzione di peso $w(e)$ rispetti la disuguaglianza triangolare, ovvero è tale che $\forall u, v, z \ w(u, v) + w(v, z) \geq w(u, z)$.

TRAVELINGSALESMANPROBLEM (*metrico*)

INPUT come per TSP simmetrico, ma $w(e)$ deve soddisfare $\forall u, v, z \ w(u, v) + w(v, z) \geq w(u, z)$

OUTPUT come per TSP simmetrico

Intuitivamente TSP-M è **NP-HARD**: prima abbiamo dimostrato la hardness di TSP usando $w(e) \in \{1, 2\}$, che rispetta sempre la disuguaglianza triangolare, quindi in realtà prima abbiamo dimostrato $\text{HAMCYCLE} \preceq \text{TSP-M}$ (e poi ovviamente abbiamo $\text{TSP-M} \preceq \text{TSP}$).

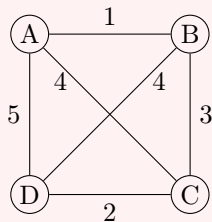
Dimostriamo che con questa condizione extra su $w(e)$ possiamo approssimare TSP-M: data un'istanza $(K_n, w) \in \mathcal{I}(\text{TSP-M})$, scriviamo un'algoritmo che

1. estrae un minimum spanning tree T di (K_n, w) , cioè un albero che tocca tutti i vertici in K_n usando i percorsi di costo minimo (fattibile in polytime): notiamo che $w(T) \leq \text{OPT}(K_n, w)$, poiché quando costruiamo l'albero tutti i cicli vengono spezzati, e quindi ogni possibile soluzione ottima sarà più costosa di almeno un arco rispetto all'albero T ;
2. duplica tutti gli archi di T ottenendo l'albero D tale che $w(D) = 2w(T)$;
3. cerco in D un ciclo Euleriano C^E , che sicuramente esiste poiché avendo duplicato gli archi abbiamo un numero pari di archi e vale il teorema di Eulero (per trovare il percorso possiamo scorrere l'albero come farebbe la BFS e tornare indietro da ogni bivio proprio perché abbiamo un altro arco percorribile);
4. trasformo C^E in un ciclo Hamiltoniano C^H di K_n usando delle "scorciatoie": comincio a percorrere C^E e, ogni volta che dovrei tornare indietro ad un vertice già toccato, passo invece ad un vertice adiacente al corrente in K_n che non ho ancora toccato, cosa sempre possibile poiché K_n è completamente connesso. In questo modo uso solo uno dei due archi duplicati e quindi evito di toccare due volte lo stesso vertice, condizione necessaria per un ciclo Hamiltoniano. Notiamo che C^H usa degli archi scorciatoia non presenti in C^E , i quali dato che $w(e)$ è metrica avranno tutti costi inferiori rispetto al percorso impiegato in C^E per spostarsi sul medesimo vertice, quindi vale $w(C^H) \leq w(C^E)$; l'algoritmo infine ritorna C^H .

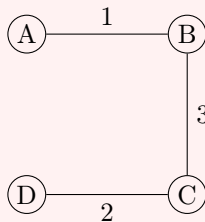
In conclusione abbiamo $A(K_n, w) = w(C^H) \leq w(C^E) = 2w(T) \leq 2\text{OPT}(K_n, w)$, quindi $\frac{A(K_n, w)}{\text{OPT}(K_n, w)} \leq 2$, e l'algoritmo garantisce 2-approssimazione per TSP-M (possiamo fare anche meglio, l'algoritmo di Cristofides è una $\frac{3}{2}$ -approssimazione per TSP metrico).

Esempio

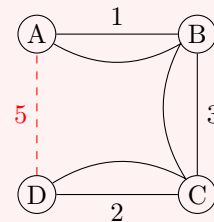
Dato il grafo completamente connesso K_n i cui archi sono già etichettati con la funzione di peso $w(e)$ metrica, applichiamo l'algoritmo: estraiamo il MST (ad esempio usando Kruskal, prendendo gli archi in ordine di costo minore senza creare un ciclo), duplichiamo gli archi e troviamo un ciclo Euleriano C^E .



grafo K_n



MST T di K_n



MST D

Trasformiamo il ciclo $C^E = ABCDCBA$ in C^H scorrendo dall'inizio: passiamo dai vertici $ABCD$ tranquillamente, ora dovremmo tornare in C ma lo abbiamo già esplorato, quindi ci spostiamo su un vertice che non abbiamo ancora esplorato adiacente a D nel grafo K_n ; dato che li abbiamo già esplorati tutti, semplicemente chiudiamo il ciclo tornando ad A , quindi otteniamo $C^H = ABCDA$; notiamo che proprio perché l'arco AD è una scorciatoia e $w(e)$ è metrica, vale la disuguaglianza $w(AD) \leq w(AB) + w(BC) + w(CD)$. Il costo del percorso ottimo è $w(T) = 6$, mentre il costo del ciclo è $w(C^H) = 11$, quindi abbiamo una $\frac{11}{6} \leq 2$ -approssimazione per questa istanza di TSP-M.

8.3.5 Risultati di inapprossimabilità

Come possiamo dire che per un certo problema \mathbb{A} non esistono algoritmi di r -approssimazione? Possiamo utilizzare il *teorema del gap*, che suppone $\mathbf{P} \neq \mathbf{NP}$ per dimostrare che non esiste un'approssimazione per \mathbb{A}^{OPT} .

Teorema del gap

Supposto $\mathbf{P} \neq \mathbf{NP}$, dato un problema \mathbb{A}^{OPT} , un problema $\mathbb{B}^{\text{DEC}} \in \mathbf{NPC}$ e una funzione di riduzione polinomiale $f : \mathcal{I}(\mathbb{B}) \rightarrow \mathcal{I}(\mathbb{A})$ tale che $\mathbb{B} \preceq_f \mathbb{A}$, il **teorema del gap** afferma che, se

$$\mathbb{B}(x) = \text{yes} \implies \text{OPT}_{\mathbb{A}}(f(x)) \leq a \text{ costante}$$

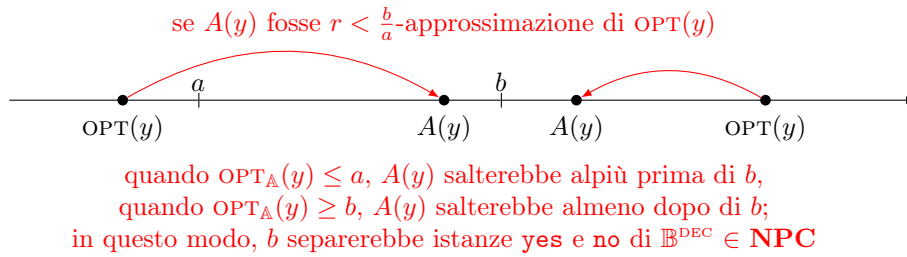
$$\mathbb{B}(x) = \text{no} \implies \text{OPT}_{\mathbb{A}}(f(x)) \geq b \text{ costante}$$

allora *sicuramente* per qualsiasi $r < \frac{b}{a}$ non esiste alcun algoritmo $A(y)$ di r -approssimazione per \mathbb{A} .

Riformuliamo: se tutte le istanze di \mathbb{B} fossero mappabili da un lato o dall'altro del gap tra a e b , avendo un algoritmo A che r -approssima \mathbb{A} (dove $r < \frac{b}{a}$) potremmo risolvere polinomialmente il problema decisionale \mathbb{B} . Supponendo infatti l'esistenza di A polytime, data $x \in \mathcal{I}(\mathbb{B})$, la mappo a $y = f(x) \in \mathcal{I}(\mathbb{A})$ usando una f costruita appositamente per poter separare tra a e b , e

- se $\text{OPT}_{\mathbb{A}}(y) \leq a$ allora $A(y) \leq r \text{OPT}_{\mathbb{A}}(y) < \frac{b}{a}a = b$, e quindi con $A(y) < b$ distinguiamo istanze x **yes**;
- se $\text{OPT}_{\mathbb{A}}(y) \geq b$ allora $r \text{OPT}_{\mathbb{A}}(y) \geq A(y) > \text{OPT}_{\mathbb{A}}(y) \geq b$, e quindi con $A(y) > b$ distinguiamo istanze x **no**;

In base al valore ottenuto da $A(y)$ posso quindi distinguere tutte le istanze x , ovvero risolvere polinomialmente $\mathbb{B} \in \mathbf{NPC}$, che è impossibile nell'ipotesi $\mathbf{P} \neq \mathbf{NP}$: l'algoritmo A non può esistere.



Osservazione: diciamo che *sicuramente* per $r < \frac{b}{a}$ non esistono algoritmi di r -approssimazione poiché invece per $r \geq \frac{b}{a}$ non sappiamo nulla, potrebbero esistere oppure no.

Osservazione: notiamo che per $a = b$ stiamo verificando che un problema è **NP-HARD**

Esempio

Nella dimostrazione di inapprossimabilità di TSP abbiamo usato HAMCYCLE come problema **NPC** e abbiamo mappato l'ottimo usando la funzione $w(e)$: le istanze **yes** a pesi $\text{OPT}(I_G) \leq n$, le istanze **no** a pesi $\text{OPT}(I_G) \geq nt + 1$. In questo modo si crea un vuoto (un gap appunto) di soluzioni $\text{OPT}(I_G)$ tra n e $nt + 1$, quindi cercare di approssimare con $r < \frac{nt+1}{n} = t + \frac{1}{n}$ e in particolare con $r = t$ è impossibile.

Vediamo un ulteriore esempio su un nuovo problema

MINCOL

INPUT un grafo G

OUTPUT una colorazione propria di G con il minimo possibile di colori

Usiamo 3-COL come problema B e l'identità f come riduzione. Se x è **yes** per \mathbb{B} (cioè è 3 colorabile) allora $\text{OPT}(f(x)) \leq 3 = a$, mentre se invece x è **no** per \mathbb{B} (cioè non è 3 colorabile) allora $\text{OPT}(f(x)) \geq 4 = b$, quindi il teorema del gap ci dice che non esiste un algoritmo di approssimazione per MINCOL con garanzia $r < \frac{4}{3}$.

Infatti, supponiamo per assurdo che un algoritmo A fosse una $\frac{4}{3}$ -approssimazione per MINCOL: dato un grafo G , se fosse 3 colorabile in modo ottimo, cioè $\text{OPT}(G) = 3$, allora $A(G)$ per definizione darebbe una soluzione $A(G) < \frac{4}{3} \text{OPT}(G) = 4$, quindi darebbe una 3 colorazione, mentre se G non fosse 3 colorabile allora

$A(G) \geq \text{OPT}(G) \geq 4$, quindi darebbe una colorazione maggiore di 3. Questo significa che A sarebbe in grado di distinguere polinomialmente istanze **yes** e istanze **no** di 3-COL $\in \mathbf{NPC}$ semplicemente impostando come soglia 3.5 (o comunque un valore tra 3 e 4), che è assurdo se $\mathbf{P} \neq \mathbf{NP}$. In conclusione abbiamo $\text{MINCOL} \notin \mathbf{PTAS}$ e quindi non possiamo nemmeno dire (e non lo faremo) se è approssimabile per $r \geq \frac{4}{3}$.

8.3.6 Raffinare le approssimazioni

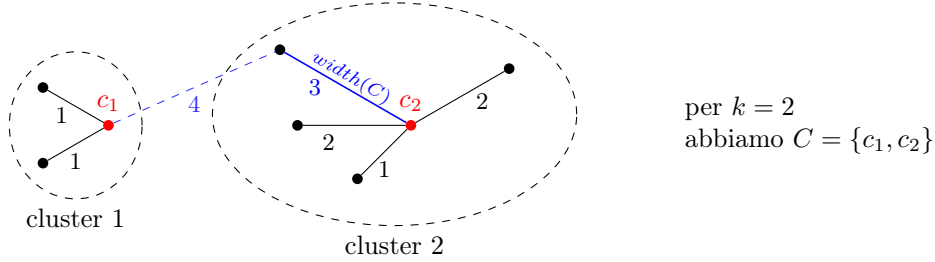
Per ora abbiamo dimostrato che alcuni problemi si possono dimostrare e altri no, adesso cerchiamo di dimostrare che, per un certo problema approssimato con una certa r non esistono approssimazioni migliori. Cominciamo con il problema k -CENTER *metrico*, che è essenzialmente un problema di clustering.

k -CENTER (metrico)

INPUT un insieme di punti $P = \{x_1 \dots x_n\}$, una funzione $d(x, y)$ e un parametro k . La funzione d è una **metrica** tra punti, per cui valgono le proprietà: ① identità $d(x, y) = 0 \iff x = y$, ② simmetria $d(x, y) = d(y, x)$ e ③ disuguaglianza triangolare $d(x, y) + d(y, z) \geq d(x, z)$

OUTPUT un sottinsieme di punti $C \subseteq P$ detti *centri* dei cluster tale che $|C| = k$ e che l'ampiezza massima $\text{width}(C)$ dei cluster sia la minima, in pratica $\text{width}(C) = \max_x d(x, C)$ e $d(x, C) = \min_{c \in C} d(x, c)$.

k -CENTER cerca dei punti *centro* per ognuno dei k cluster desiderati, e ogni punto appartiene ad un certo cluster se la sua distanza dal suo centro è minore rispetto a quella da tutti gli altri centri: in questo modo sto creando k gruppi di oggetti quanto più vicini possibili tra loro.



Osservazione: per questa trattazione del problema i centri devono essere scelti tra i punti che ci vengono dati, in altre trattazioni possono anche essere punti ideali non forniti.

Scriviamo un algoritmo A^G con approccio greedy, che man mano sceglie come centro il punto più distante dai centri scelti finora; in questo modo $\text{width}(C)$ scenderà sempre di più:

Algorithm 9 $A^G = k$ -CENTER-SOLVER

```

1:  $C = \emptyset$ 
2: for  $i = 1, k$  do
3:    $c_i = \arg \max_x d(x, C)$ 
4:    $C \leftarrow C \cup \{c_i\}$ 
5: return  $C$ 

```

L'algoritmo è per costruzione polytime, ma dimostriamo che è una 2-approssimazione. Sia C^* una soluzione ottima e $d^* = \text{width}(C^*)$, mentre C^A la soluzione di A^G : ora abbiamo due casi

- se ogni centro c identificato da A^G è contenuto in un solo cluster di centro c^* tra quelli di C^* allora

$$\forall x, d(x, C^A) \leq \underbrace{d(x, c) \leq d(x, c^*) + d(c^*, c)}_{\text{perché } d(x, y) \text{ è metrica}} \leq 2d^*,$$

quindi $\text{width}(C^A) = \max_x d(x, C^A) \leq 2d^*$, e abbiamo una 2-approssimazione;

- se invece due centri c_1 e c_2 identificati da A^G sono contenuti in un solo cluster di centro c^* tra quelli di C^* (e per il teorema della piccionaia esiste un cluster di C^* che non contiene centri di A^G), allora siccome i centri di A^G vengono scelti in base alla massima ampiezza in quel momento abbiamo

$$\text{width}(C^A) \leq d(c_1, c_2) \leq d(c_1, c^*) + d(c_2, c^*) \leq 2d^*$$

quindi nuovamente $\text{width}(C^A) = \max_x d(x, C^A) \leq 2d^*$, e abbiamo una 2-approssimazione.

In conclusione $\frac{A^G(I)}{\text{OPT } I} \leq 2$, quindi A^G è una 2-approssimazione per il problema k -CENTER.

Possiamo fare di meglio? No: usiamo la tecnica del gap e dimostriamo che possiamo ridurre il problema

DOMINATINGSET

INPUT un grafo $G = (V, E)$ e un parametro k

OUTPUT **yes** \iff esiste $A \subseteq V$ tale che $|A| = k$ e $\forall v \in V \setminus A$ esiste $u \in A$ | $(u, v) \in E$; in pratica, ogni vertice non scelto deve essere collegato ai vertici scelti (è una versione di VERTCOV sui vertici).

che è **NPC** (non lo dimostriamo), al trovare un clustering di ampiezza 1 in un'istanza di k -CENTER in cui tutti i clustering possibili hanno ampiezza 1 o 2. Dimostriamo cioè che $\text{DOM} \preceq k\text{-CENTER}^{\text{DEC}}$, forzando $\text{width}(C) \leq 1$ per i cluster C . Data un'istanza $(G, k) \in \mathcal{I}(\text{DOM})$ costruisco l'istanza $(P, d, k) \in \mathcal{I}(k\text{-CENTER})$ con $P = V$ e

$$d(u, v) = \begin{cases} 1 & \text{se } (u, v) \in E \\ 2 & \text{se } (u, v) \notin E \end{cases}$$

così da ottenere funzione $d(u, v)$ metrica (in particolare, con metrica $1/2$). A questo punto per risolvere $k\text{-CENTER}^{\text{DEC}}$ devo trovare un clustering C di k centri tale che $\text{width}(C) \leq 1$: dimostriamo la doppia implicazione

\Rightarrow se esiste A dominating set di taglia k allora imposto $C = A$ e, siccome A è dominante, $\forall x \in P$ esiste $y \in C$ tale che $x, y \in E \iff d(x, y) = 1$, e quindi abbiamo

$$d(x, C) = \begin{cases} 0 & \text{se } x \in A \\ 1 & \text{se } x \notin A \end{cases}$$

il che ci porta ad avere al massimo ampiezza 1, quindi $\text{width}(C) \leq 1$

\Leftarrow se esiste C con $\text{width}(C) \leq 1$ allora $\forall x \notin C$, $d(x, C) = 1$ e quindi $\exists c \in C$ tale che $d(x, c) = 1$, che rispetto alla definizione data prima significa che per ogni x esiste un c tale che $(x, c) \in E$: C è quindi un dominating set di taglia k per G e possiamo impostare $A = C$.

Abbiamo quindi dimostrato che una qualsiasi istanza di DOM si mappa ad una istanza di $k\text{-CENTER}^{\text{DEC}}$ e che quindi capire se un'istanza di $k\text{-CENTER}$ ha ampiezza al massimo 1 è un problema difficile: impostiamo $a = 1$ e $b = 2$, e per il teorema del gap il problema $k\text{-CENTER}$ non può avere r -approssimazione con $r < 2$, quindi l'algoritmo greedy A^G che già abbiamo descritto all'inizio (ha $r = 2$) fornisce la migliore approssimazione possibile per $k\text{-CENTER}$ metrico (quando **P** \neq **NP**).

Osservazione: se avessimo usato una metrica $2/3$ per la mappatura di DOM a $k\text{-CENTER}$ alla fine avremmo ottenuto che $k\text{-CENTER}$ non può avere r -approssimazione con $r < \frac{3}{2}$, che è un risultato utile ma non completo, infatti non ci dice nulla per $r \geq \frac{3}{2}$: infatti, come abbiamo visto, anche per $\frac{3}{2} \leq r < 2$ non possiamo avere r -approssimazione, quindi il risultato $r < 2$ visto sopra è l'upper bound più alto (non esistono metriche più piccole di $1/2$) che potevamo dare sullo studio dell'inapprossimabilità di $k\text{-CENTER}$.

In conclusione, abbiamo dimostrato che $k\text{-CENTER}$ è k -approssimabile solo per $k \geq 2$, quindi è approssimabile ma non polinomialmente nella taglia delle sue istanze, cioè $k\text{-CENTER} \in \mathbf{APX} \setminus \mathbf{PTAS}$.

8.3.7 Ulteriori risultati di inapprossimabilità

Per ora abbiamo visto come capire se un problema è approssimabile e con quale rapporto r , ora cerchiamo di filtrare ancora meglio i problemi usando delle varianti del teorema del gap per le classi **PTAS** ed **FPTAS**.

Teorema del gap per PTAS di minimizzazione

Dato un problema di *minimizzazione* \mathbb{A} con soluzioni solo di valori interi (quindi $\forall x \in \mathcal{I}(\mathbb{A}), \forall y \in \text{sol}(x), \text{val}(y) \in \mathbb{N}$) per il quale decidere se esiste una soluzione di valore $\leq k$ è un problema **NP-HARD** abbiamo che $\forall r < \frac{k+1}{k} = 1 + \frac{1}{k}$ non esiste un algoritmo di r -approssimazione.

Questo accade perché distinguere risultati fino a k ($\leq k$) e dopo k ($\geq k+1$) è un problema difficile appunto, e se fosse fattibile da un algoritmo di approssimazione potremmo risolvere un problema decisionale **NP-HARD** in tempo polinomiale, che è impossibile nell'ipotesi **P** \neq **NP**.

Questo risultato è quanto appena visto per $k\text{-CENTER}$ metrico, nel quale abbiamo impostato infatti distanze intere; abbiamo inoltre visto usando il teorema del gap con $\mathbb{B} = 3\text{-COL}$ che $\text{MINCOL} \notin \mathbf{PTAS}$, ed è esattamente

lo stesso risultato che otteniamo usando questo nuovo teorema con $k = 3$. **INDSET** e **CLIQUE** invece sono problemi di massimizzazione, quindi varrà l'analogo opposto:

Teorema del gap per PTAS di massimizzazione

Dato un problema di *massimizzazione* \mathbb{A} con soluzioni solo di valori interi per il quale decidere se esiste una soluzione di valore $\geq k + 1$ è un problema **NP-HARD** abbiamo che $\forall r < \frac{k+1}{k} = 1 + \frac{1}{k}$ non esiste un algoritmo di r -approssimazione.

In entrambi i casi stiamo dando una tecnica per poter dire se un \mathbb{A} può avere o meno un **PTAS**: se proviamo però ad utilizzarla per **VERTCOV** abbiamo bisogno di un valore k , attorno al quale far ruotare l'intera tecnica; potremmo trovare un problema decisionale che dice se le istanze di **VERTCOV** hanno al massimo soluzioni di $\leq k$ vertici, esattamente come abbiamo fatto per **MINCOL** con **3-COL**, ma seguiamo un'ulteriore nuova tecnica.

Teorema del gap per FPTAS

Dato un problema di ottimizzazione \mathbb{A} con *soluzioni a valori piccoli*, ovvero soluzioni solo di valori polinomiali nella taglia dell'istanza (quindi $\forall x \in \mathcal{I}(\mathbb{A}), \forall y \in \text{sol}(x), \text{val}(y) \leq p(|x|)$), non esiste un algoritmo **FPTAS** per \mathbb{A} .

Dimostrazione. L'idea è di utilizzare il gap tra $p(|x|) - 1$ e $p(|x|)$ per sfruttare il teorema del gap. Dato A un algoritmo **FPTAS** per \mathbb{A} , ovvero una $(1+\varepsilon)$ -approssimazione di $\text{OPT}(x)$, prendo $\varepsilon = \frac{1}{p(|x|)}$ così da permettere ad A di finire in $\text{poly}(|x|, \frac{1}{\varepsilon}) = \text{poly}(|x|, p(|x|)) = \text{poly}(|x|)$; questo servirà per dimostrare l'assurdo alla fine. Dato che \mathbb{A} è un problema **NP-HARD** (poiché è di ottimizzazione) esiste almeno un'istanza x su cui A non ritorna la soluzione ottima: per condizione del teorema, tale istanza x ha $\forall y \in \text{sol}(x), \text{val}(y) \in \{1 \dots p(|x|)\}$ e dato che per definizione $r = \frac{A(x, \frac{1}{\varepsilon})}{\text{OPT}(x)}$, il minor rapporto per r è ottenibile da valori delle soluzioni quanto più grandi e vicini possibili, ovvero per $\text{OPT}(x) = p(|x|) - 1$ e $A(x) = p(|x|)$. In questo modo otteniamo A deve avere il valore

$$r = \frac{A(x, \frac{1}{\varepsilon})}{\text{OPT}(x)} = \frac{p(|x|)}{p(|x|) - 1} = \frac{p(|x|) - 1 + 1}{p(|x|) - 1} = 1 + \frac{1}{p(|x|) - 1} > 1 + \varepsilon$$

che è assurdo per ipotesi, poiché avevamo $r < 1 + \varepsilon$, quindi A non può esistere e $\mathbb{A} \notin \text{FPTAS}$. \square

Tornando a **VERTCOV**, le sue soluzioni sono sempre insiemi di vertici in numero al massimo pari al numero di vertici totali, ovvero ha soluzioni a valori piccoli (polinomiali nella taglia dell'istanza), e di conseguenza non può avere un algoritmo **FPTAS**, quindi **VERTCOV** $\notin \text{FPTAS}$.

8.4 L'approssimabilità è “transitiva”?

Abbiamo visto che possiamo dimostrazioni di approssimabilità più o meno raffinate per molti problemi, mentre per altri è impossibile dare un'approssimazione entro un certo limite o addirittura in generale. Ora ci chiediamo: se $\mathbb{B} \preceq \mathbb{A}$ e ho una certa r -approssimazione per \mathbb{A} , anche \mathbb{B} è r -approssimabile? Se no, ci andiamo vicini o non si può dire proprio nulla? Studiamo il caso di **MAX- k -SAT**.

8.4.1 Problema MAX- k -XORSAT

Introduciamo un nuovo problema

MAX- k -XORSAT

INPUT una formula ϕ XOR- k -CNF (quindi con disgiunzioni esclusive \oplus) di m clausole e un parametro ℓ

OUTPUT **yes** \iff esiste un assegnamento w che soddisfa almeno $0 < \ell \leq m$ clausole

che assomiglia molto a **MAX- k -SAT** ed è come lui **NPC** anche per $k = 2$; dimostriamolo riducendo da **MAX-CUT**.

Dimostrazione. Dato un grafo $G = (V, E)$ vogliamo mapparlo ad una istanza $\phi_G = \bigwedge_j (\ell_{j_1} \oplus \ell_{j_2})$. Per **MAX-CUT** dobbiamo assegnare a ogni vertice un colore $\{\text{BIANCO}, \text{NERO}\}$ in modo da tagliare almeno ℓ archi (quelli formati da vertici colorati diversamente): mappiamo la bicolorazione ai valori $\{\text{T}, \text{F}\}$ per **MAX- k -XORSAT**, e costruiamo l'istanza $\phi_G = \bigwedge_{(u,v) \in E} (x_u \oplus x_v)$; in questo modo almeno ℓ clausole verranno soddisfatte poiché sono mappate ad altrettanti ℓ archi tagliabili. \square

8.4.2 Approssimabilità di MAX-k-XORSAT

Notiamo che, anche se MAX-k-XORSAT \in NPC, quando $\ell = m$ abbiamo MAX-k-XORSAT \in P, quindi dovremmo evitare questo caso durante la dimostrazione di inapprossimabilità di MAX-k-XORSAT. Definiamo ora un problema NPC associato a MAX-k-XORSAT da utilizzare per il teorema del gap:

GAP-k-XORSAT

INPUT una formula ϕ XOR-3-CNF di m clausole tale che, dati due parametri $0 < b < a \leq 1$, o abbiamo $\leq b \cdot m$ clausole soddisfacibili oppure abbiamo $\geq a \cdot m$ clausole soddisfacibili.

OUTPUT **yes** \iff esiste un assegnamento w che soddisfa $\geq a \cdot m$ clausole, ovviamente invece ritorna **no** quando gli unici assegnamenti possibili soddisfano $\leq b \cdot m$ clausole

Concentriamoci su $k = 3$ e notiamo che per $a = 1$ oppure $b \leq \frac{1}{2}$ abbiamo GAP-3-XORSAT \in P, infatti

- con $a = 1$, $\forall b$ abbiamo che GAP-3-XORSAT ammette formule ϕ che hanno almeno m oppure al massimo $b \cdot m$ clausole soddisfacibili e ritorna **yes** solo per istanze che soddisfano almeno m clausole: questo è risolvibile polinomialmente trasformando il problema in un sistema lineare di m equazioni a variabili $X_i \in \{0, 1\}$ mappando ogni clausola ad una equazione mediante $x_i \rightarrow X_i$ e $\bar{x}_i \rightarrow (1 - X_i)$;

Esempio

Dato GAP-3-XORSAT con $a = 1$, la formula $\phi = (\bar{x}_1 \oplus x_2 \oplus x_3) \wedge (\bar{x}_1 \oplus x_2 \oplus \bar{x}_3) \wedge (x_1 \oplus x_2 \oplus x_4)$ viene mappata al sistema sotto, risolvibile in tempo polinomiale usando algoritmi di algebra lineare.

$$\begin{cases} (1 - X_1) + X_2 + X_3 = 1 \\ (1 - X_1) + X_2 + (1 - X_3) = 1 \\ X_1 + X_2 + X_4 = 1 \end{cases}$$

- con $b \leq \frac{1}{2}$, $\forall a$ abbiamo che GAP-3-XORSAT ammette formule ϕ che hanno almeno $a \cdot m$ oppure al massimo $\frac{1}{2} \cdot m$ clausole soddisfacibili e ritorna **yes** solo per istanze che soddisfano almeno $a \cdot m$ clausole: dato che abbiamo clausole con 3 letterali avremo che $\frac{4}{8}$ degli assegnamenti possibili alle variabili soddisfarranno le singole clausole, quindi a partire dalla prima applico gli assegnamenti in modo *greedy* e non più del 50% delle clausole verrà soddisfatto. Dato che scegliere un valore per ogni variabile che incontro nello scorrimento della formula è lineare, anche in questo caso risolviamo polinomialmente.

Esempio

Dato GAP-3-XORSAT con $b = \frac{1}{2}$ e la formula vista sopra, cerco di soddisfare la prima clausola, assegnando arbitrariamente $x_1 = T, x_2 = T, x_3 = F$, in questo modo automaticamente la seconda clausola non verrà soddisfatta. Dato che già abbiamo $\frac{1}{3} = 33\%$ delle clausole soddisfatte, la terza non può essere soddisfatta, altrimenti andremmo al 66%, quindi semplicemente assegnamo $x_4 = T$. Se invece anche la terza clausola fosse già soddisfatta, dovremmo cambiare i valori iniziali scelti.

Ovviamente per usare GAP-k-XORSAT come problema decisionale per il teorema del gap abbiamo bisogno che di GAP-k-XORSAT \in NPC, quindi ci chiediamo ora se esistono dei parametri a e b che lo permettono.

Teorema (basato sul teorema PCP)

Per $k \geq 3$, per $a < 1$ oppure $b > \frac{1}{2}$ abbiamo GAP-k-XORSAT \in NPC

Finalmente abbiamo tutti gli ingredienti per dimostrare l'inapprossimabilità di MAX-3-XORSAT:

Dimostrazione. Impostati $b > \frac{1}{2}$ e $a < 1$, essi formano i limiti inferiore e superiore per il teorema del gap: se esistesse un algoritmo di r -approssimazione per MAX-3-XORSAT con $r < \frac{a}{b}$ (abbiamo invertito le lettere nella definizione di GAP-3-XORSAT) allora potremmo dire in tempo polinomiale se una formula ϕ in XOR-3-CNF soddisfa o meno ℓ clausole e quindi separare le formule che soddisfano $\leq a \cdot m$ clausole da quelle che soddisfano $\geq b \cdot m$ clausole, cioè potremmo risolvere in tempo polinomiale GAP-3-XORSAT, che è impossibile. In conclusione, se $P \neq NP$, MAX-3-XORSAT non può avere r -approssimazione con $r < 1/\frac{1}{2} = 2$. \square

8.4.3 Transività verso MAX- k -SAT

Ora arriviamo alla dimostrazione di “transività” dell’approssimabilità per MAX-3-XORSAT: per adesso abbiamo visto che GAP-3-XORSAT è **NPC** per ogni $a < 1$ e $b > \frac{1}{2}$ e che quindi non può esistere un algoritmo di r -approssimazione per MAX-3-XORSAT per ogni $r < \frac{a}{b}$ (ricordiamo che i parametri sono definiti nel problema al contrario rispetto al teorema del gap); ora vogliamo mostrare che da questo possiamo ottenere che $\exists a', b'$ tale che non esiste un algoritmo di r' -approssimazione per MAX-3-SAT per ogni $r' < \frac{a'}{b'}$.

Dimostrazione. Data una formula ϕ XOR-3-CNF composta di clausole $C = \ell_1 \oplus \ell_2 \oplus \ell_3$, la mappo ad una formula ψ 3-CNF nella forma $D_1 \wedge D_2 \wedge D_3 \wedge D_4 = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_1 \vee \bar{\ell}_2 \vee \bar{\ell}_3) \wedge (\bar{\ell}_1 \vee \ell_2 \vee \bar{\ell}_3) \wedge (\bar{\ell}_1 \vee \bar{\ell}_2 \vee \ell_3)$. Notiamo che

1. se C è soddisfatta allora D_1, D_2, D_3 e D_4 sono soddisfatte;
2. ogni assegnamento di ψ soddisfa almeno 3 clausole tra D_1, D_2, D_3 e D_4 anche se C non è soddisfatta;

quindi

- se ϕ ha $\geq am$ clausole soddisfatte, allora ψ avrà sicuramente $\geq 4am$ clausole soddisfatte (segue da ①, in realtà ce ne sono ancora, quelle date da ②, ma questo limite ci fa comodo); oppure
- se ϕ ha $\leq bm$ clausole soddisfatte, allora ψ avrà $\leq 4bm$ clausole soddisfatte (segue da ①) più altre $3(1-b)m$ clausole (segue da ②, dato che le clausole sicuramente non soddisfatte di ϕ sono le restanti $(1-b)$, per ognuna di esse avremo 3 delle relative clausole di ψ soddisfatte).

Per $a < 1$ oppure $b > \frac{1}{2}$ abbiamo che ψ deve soddisfare $\geq 4m$ oppure $\leq \frac{7}{2}m$ clausole, quindi c’è un gap tra le istanze soddisfacibili, e abbiamo che $4m/\frac{7}{2}m = \frac{8}{7}$ è un limite inferiore all’approssimazione di MAX-3-SAT, quindi, se $\mathbf{P} \neq \mathbf{NP}$, non esiste un algoritmo di r -approssimazione per MAX-3-SAT per ogni $r < \frac{8}{7}$. \square

Abbiamo dimostrato che per questi problemi il gap si muove “attraverso” la riduzione, mappando i limiti del gap iniziale a dei nuovi limiti.

Ma è il meglio che possiamo fare? Per questa riduzione, sì, dimostriamolo.

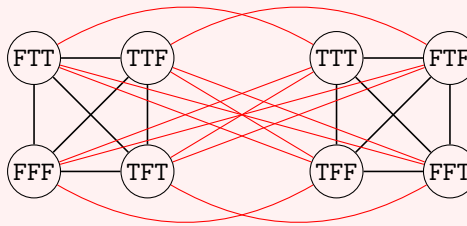
Dimostrazione. Per MAX-3-SAT si può trovare un algoritmo A^G che soddisfa *sempre* $\frac{7}{8}m$ clausole: data una formula ϕ 3-CNF di clausole $C = \ell_1 \vee \ell_2 \vee \ell_3$, notiamo che 7 tra i possibili 8 assegnamenti soddisfano C (in particolare, solo FFF non soddisfa C), quindi scegliendo valori a caso avremo soddisfacibilità di C in $\frac{7}{8}$ delle volte. Per ogni clausola C il valore atteso è quindi $\mathbb{E}(C) = \frac{7}{8}$, il che significa che per tutte le clausole avremo $\mathbb{E}(C_1) + \dots + \mathbb{E}(C_m) = \frac{7}{8}m$: su queste considerazioni, utilizziamo un algoritmo greedy A^G , il quale ordina le variabili seguendo la loro frequenza e man mano assegna loro i valori in modo da soddisfare il maggior numero di clausole possibili (questo è fattibile polinomialmente, si contano le clausole soddisfatte con $x_i = T$ e quelle con $x_i = F$ e si sceglie di conseguenza). Possiamo dimostrare (non lo facciamo) che A^G garantisce la soddisfacibilità almeno $\frac{7}{8}m$ clausole, quindi $\text{OPT}(\phi) \leq m$ (essendo ottimale al massimo le soddisfa tutte) mentre $A^G(\phi) \geq \frac{7}{8}m$, e in conclusione l’algoritmo A^G è una r -approssimazione per MAX-3-SAT dove $r = m/\frac{7}{8}m = \frac{8}{7}$, che è la massima garanzia che possiamo dare, quindi non avremo nessun algoritmo per $r < \frac{8}{7}$. \square

8.4.4 Transività verso INDSET

Mostriamo che possiamo anche ridurre MAX-3-XORSAT \preceq INDSET per garantire anche qui il gap. Per ogni clausola C_j in $\phi \in \mathcal{I}(\text{MAX-3-XORSAT})$ creiamo una clique di 4 vertici rappresentanti i valori di verità degli assegnamenti che soddisfano C_j e aggiungiamo archi tra vertici di clique differenti se i loro assegnamenti sono incompatibili, cioè se per la stessa variabile hanno due valori differenti.

Esempio

Data $\phi = (\bar{x}_1 \oplus x_2 \oplus x_3) \wedge (x_1 \oplus x_2 \oplus x_3)$ con $a = 1$ oppure $b \leq \frac{1}{2}$ la mappiamo al grafo G_ϕ



In questo modo abbiamo mappato tutti gli assegnamenti parziali incompatibili per MAX-3-XORSAT a vertici incompatibili (cioè connessi tra loro) per INDSET; siamo quindi passati da una formula ϕ con m clausole a un grafo G con $4m$ vertici. Dimostriamo la doppia implicazione

- \Rightarrow se esiste un assegnamento che soddisfa x clausole allora per ognuna di esse avremo un assegnamento parziale mappato ad un vertice in una clique di G ; tale vertice esclude gli altri vertici nella stessa clique (sono incompatibili) e quindi per x clique in G ci sarà uno ed un solo vertice scelto per l'indipendent set;
- \Leftarrow se esiste un independent set di x vertici allora essi apparterranno ognuno ad una clique differente, e dato che sono vertici indipendenti tra loro si mapperanno ad assegnamenti parziali compatibili tra loro, quindi x clausole verranno soddisfatte in ϕ .

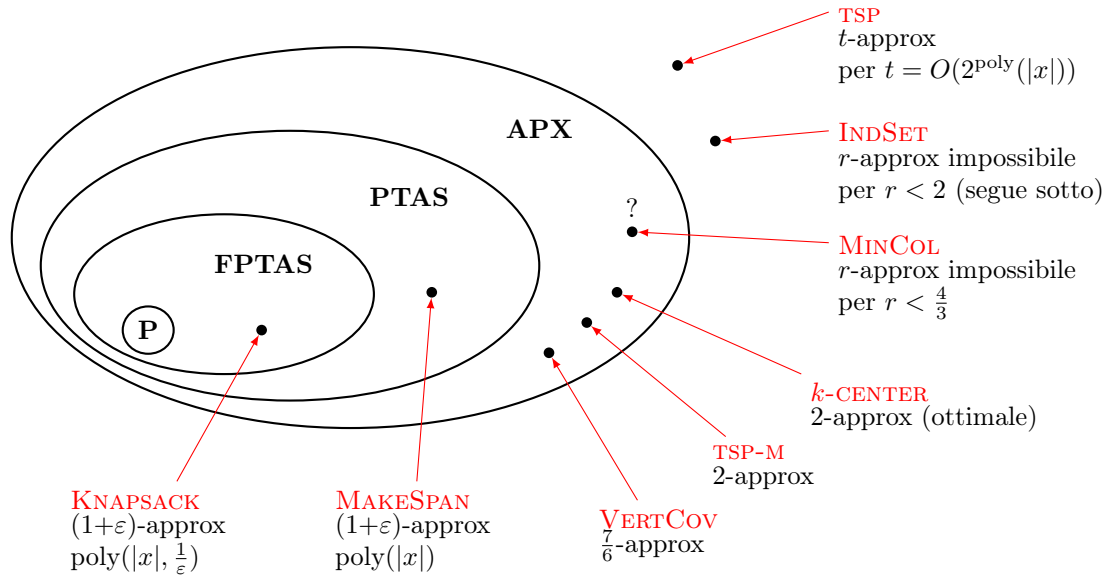
Dato che in MAX-3-XORSAT il gap va da m a $\frac{1}{2}m$ poiché altrimenti potrei risolvere GAP-3-XORSAT in tempo polinomiale, siccome abbiamo dimostrato esistere un mapping 1:1 tra MAX-3-XORSAT e INDSET, anche per quest'ultimo il gap sarà da m a $\frac{1}{2}m$, e quindi non esiste un algoritmo di r -approssimazione per INDSET dove $r < 2$.

8.4.5 Transività verso VERTCOV

Abbiamo visto che dato il grafo G , la soluzione per INDSET per quel grafo è il complemento della soluzione per VERTCOV per quel grafo, quindi dato che per G abbiamo $4m$ vertici, il gap sarà un mapping con il complemento dei vertici e andrà da $4m - m$ a $4m - \frac{1}{2}m$, ovvero da $3m$ a $\frac{7}{2}m$.

MAX-3-XORSAT	INDSET	VERTCOV
m clausole	G con $ V = 4m$ e k	G con $ V = 4m$ e $q = k - 2$
$a = 1m$	$a = 1m$	$a = 3m$
$b = \frac{1}{2}m$	$b = \frac{1}{2}m$	$b = \frac{7}{2}m$

In conclusione, non esiste un algoritmo di r -approssimazione per VERTCOV dove $r < \frac{7}{2}m/3m = \frac{7}{6}$, quindi $\text{VERTCOV} \in \text{APX} \setminus \text{PTAS}$, e abbiamo quindi raffinato l'approssimazione precedente per VERTCOV, la quale usava $r = 2$ e ci diceva solo che $\text{VERTCOV} \notin \text{FPTAS}$.



Come nota finale, abbiamo detto che INDSET non è approssimabile per $r < 2$, ma è dimostrabile (non lo vediamo) che addirittura non è approssimabile per r costanti, infatti INDSET rientra nella classe computazionale $f(n)$ -APX, ovvero approssimabile con garanzia polinomiale nella taglia dell'input.

8.5 Conclusioni

Dato un problema difficile, in alternativa ad un algoritmo esponenziale possiamo sviluppare un algoritmo che *da la garanzia* di approssimare per ogni istanza la soluzione ottima; per dimostrare approssimazioni possiamo ① impostare un lower bound per l'ottimo e arrivare al rapporto che definisce r , oppure ② usare la tecnica del gap sfruttando un problema NPC associato.

Un altro approccio è proporre *euristiche* di risoluzione, le quali sfruttando evidenza empirica dal problema, danno risultati efficientemente sui casi reali, ma non necessariamente portano sempre a soluzioni ottime.

Un ulteriore approccio è usare delle *tecniche parametrizzate*, le quali trovano sempre la soluzione ottima polinomialmente nella taglia dell'istanza, ma esponenzialmente rispetto ad un parametro (scelto) del problema.

Esempio

Per VERTCOV sappiamo esistere un algoritmo che 2-approssima, ma mostriamo ora che per grafi piccoli possiamo dare una soluzione ottima in tempo polinomiale, ovvero utilizziamo una tecnica parametrizzata sulla dimensione del grafo. Dato un grafo $G = (V, E)$, supponiamo che il miglior vertex cover abbiamo taglia k , ovvero $\text{OPT}(G) = k$.

Come primo approccio potremmo provare tutti i sottoinsiemi $A \subseteq V$ tali che $|A| = k$: in questo modo avremmo $\binom{n}{k} \approx n^k$ possibili insiemi da controllare, che diventa grande al crescere di k .

Un secondo approccio usa invece un albero: a partire da un vertice creo un albero decisionale dei possibili rami da percorrere, e ogni volta che decido quale ramo seguire sto coprendo un arco e passando al prossimo vertice; quando non ci sono più vertici da poter seguire in un certo sottoalbero, torno indietro (faccio backtracking) e comincio a seguire un altro ramo (tutti gli archi precedentemente coperti rimangono tali, sto semplicemente cambiando “zona” di esplorazione del grafo G). Dato che abbiamo sicuramente $|E| \leq nk$ (il caso limite è un grafo stella), per ogni nodo avrò $O(nk)$ rami da percorrere, e dato che l'albero ha $\leq 2^k$ nodi l'algoritmo costa $O(nk2^k)$, che significa che per grafi piccoli avremo k piccole, e quindi una soluzione polinomiale (in particolare lineare). Utilizzando questo algoritmo quindi riusciamo a porre $\text{VERTCOV} \in \text{FIXEDPARAMETERIZEDTRACTABLE}$, ovvero la classe dei problemi che risultano essere semplici fissando un parametro del problema (nel nostro caso, fissiamo piccolo il numero di vertici).

9 | Complessità spaziale

Nei capitoli precedenti ci siamo concentrati solamente sui tempi di esecuzione degli algoritmi, ma sui calcolatori reali dobbiamo anche tener conto dello spazio in memoria che essi vanno ad utilizzare.

Cominciamo definendo la controparte spaziale delle classi primitive di complessità temporale

SPACE($f(n)$)

La classe dei problemi per i quali esiste un algoritmo *deterministico* che risolve istanze di taglia n usando input *read-only* (altrimenti potremmo usarlo come memoria aggiuntiva) e $O(f(n))$ bit di memoria.

PSPACE

La classe dei problemi risolti *deterministicamente* usando memoria polinomiale nella taglia n dell'input

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

Andiamo subito a studiare un caso semplice e un suo possibile algoritmo

PALINDROME

INPUT una stringa x

OUTPUT $\text{yes} \iff x$ è palindroma

Algorithm 10 PALINDROME-SOLVER

Require: una stringa $x = x[1] \dots x[n]$

```
1: for  $i = 1 \dots n$  do  
2:   if  $x[i] \neq x[n - i + 1]$  then return no  
3: return yes
```

Dato che PALINDROME-SOLVER usa come memoria solo una variabile, l'indice di testa della stringa (quello di coda è computato da $n - i + 1$), essa occupa $O(\log n)$ bit, il che significa che $\text{PALINDROME} \in \mathbf{SPACE}(\log n)$.

Un esempio più interessante è quello di 3-SAT, vediamo un algoritmo risolutore

Algorithm 11 3-SAT-SOLVER

Require: una formula $\phi(x_1 \dots x_n)$ 3-CNF

```
1: for  $i = 0 \dots 2^n - 1$  do  
2:   for  $j = 1 \dots n$  do  
3:      $a_j \leftarrow$  il  $j$ -esimo bit di  $i$   
4:   if  $\phi(a_1 \dots a_n) = \text{T}$  then return yes  
5: return no
```

L'algoritmo esegue una ricerca esaustiva (ogni intero costruibile con n bit diventa un assegnamento per ϕ) e riutilizza le stesse n variabili binarie a_j , quindi usiamo $O(n)$ memoria e 3-SAT $\in \mathbf{PSPACE}$. Abbiamo quindi un risolutore per 3-SAT che utilizza tempo esponenziale (è un problema **NPC** dopotutto) ma spazio polinomiale, il che evidenzia un vantaggio essenziale del dominio spaziale:

lo spazio si può riutilizzare, il tempo no

9.1 Gerarchia di complessità spaziale

Come abbiamo visto per 3-SAT, siamo riusciti a risolvere un problema **NPC** con memoria polinomiale, il che significa che problemi onerosi nel tempo sono in realtà molto efficienti nello spazio.

Teorema

$$\mathbf{SPACE}(f(n)) \subseteq \mathbf{TIME}(2^{O(f(n))})$$

Dimostrazione. Ogni algoritmo che usa m bit di memoria può essere al massimo in 2^m stati differenti (le configurazioni dei bit in memoria), e se un'istanza è *risolta* da un algoritmo deterministico allora tale algoritmo non passa mai attraverso la stessa configurazione due volte (altrimenti sarebbe in un loop e non si potrebbe risolvere l'istanza). Dato un problema in $\mathbf{SPACE}(f(n))$, esiste un algoritmo che lo risolve usando $O(f(n))$ bit di memoria, e quindi ha al massimo $O(2^{O(f(n))})$ possibili configurazioni; per non entrare in loop non deve eseguire più di $O(2^{O(f(n))})$ passi, e quindi impiega tempo $O(2^{O(f(n))})$, per cui $\mathbf{SPACE}(f(n)) \subseteq \mathbf{TIME}(2^{O(f(n))})$. \square

Teorema

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

Dimostrazione. In ogni passo un algoritmo può modificare al massimo un numero limitato di bit in memoria, quindi in $O(f(n))$ step possono essere modificati al massimo $O(f(n))$ bit. \square

Osservazione: dal primo teorema possiamo fare le seguenti considerazioni

$$\mathbf{SPACE}(\log n) \subseteq \mathbf{TIME}(2^{O(\log n)}) = \mathbf{TIME}(\text{poly}(n)) = \mathbf{P} \implies \mathbf{SPACE}(\log n) \subseteq \mathbf{P}$$

$$\mathbf{PSPACE} = \mathbf{SPACE}(\text{poly}(n)) \subseteq \mathbf{TIME}(2^{O(\text{poly}(n))}) = \mathbf{EXPTIME} \implies \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$$

mentre dal secondo possiamo fare un'altra importante considerazione

$$\mathbf{P} = \mathbf{TIME}(\text{poly}(n)) \subseteq \mathbf{SPACE}(\text{poly}(n)) = \mathbf{PSPACE} \implies \mathbf{P} \subseteq \mathbf{PSPACE}$$

Ma cosa possiamo dire riguardo \mathbf{NP} e \mathbf{PSPACE} ?

Teorema

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

Dimostrazione. Un problema A è in $\mathbf{NTIME}(f(n))$ se ha un verificatore B deterministico polinomiale nella taglia $O(f(n))$ dell'istanza. Possiamo scrivere un algoritmo che, data un'istanza x , crea uno a uno tutti gli $O(f(n))$ possibili certificati w e usa B per verificarli in tempo $O(f(n))$; dato che verifichiamo un certificato alla volta impiegheremo tempo esponenziale, ma utilizzando solamente $O(f(n))$ bit di memoria. \square

Osservazione: dal teorema possiamo facciamo la seguente considerazione

$$\mathbf{NP} = \mathbf{NTIME}(\text{poly}(n)) \subseteq \mathbf{SPACE}(\text{poly}(n)) = \mathbf{PSPACE} \implies \mathbf{NP} \subseteq \mathbf{PSPACE}$$

$\mathbf{NSPACE}(f(n))$

La classe dei problemi per i quali esiste un algoritmo *nondeterministico* che risolve istanze di taglia n usando input *read-only* (altrimenti potremmo usarlo come memoria aggiuntiva) e $O(f(n))$ bit di memoria.

$\mathbf{NPSPACE}$

La classe dei problemi risolti *nondeterministicamente* usando memoria polinomiale nella taglia n dell'input

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

Teorema

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

Osservazione: anche qui facciamo una considerazione

$$NPSPACE = NSPACE(poly(n)) \subseteq TIME(2^{O(poly(n))}) = EXPTIME \implies NPSPACE \subseteq EXPTIME$$

In conclusione, raccogliendo tutte le considerazioni fatte finora otteniamo

$$SPACE(\log n) \subseteq P \subseteq NP \subseteq PSPACE \subseteq NPSPACE \subseteq EXPTIME$$

e sappiamo che $P \neq EXP$ (lo abbiamo dimostrato in un capitolo precedente) e che $NPSPACE(\log n) \neq PSPACE$ (non lo dimostriamo), quindi alcune delle inclusioni sopra sono strette, ma non sappiamo dove. In realtà crediamo che siano tutte strette, a parte una, che dimostreremo ora essere un'uguaglianza.

9.2 Dimostrazione $PSPACE = NPSPACE$

Ritorniamo ad un problema a noi noto

REACHABILITY

INPUT un grafo diretto G e due vertici s e t

OUTPUT $yes \iff$ esiste un percorso in G da s a t

Dato che l'algoritmo è una semplice BFS (polinomiale) su G , siccome $P \subseteq PSPACE$ abbiamo $REACH \in PSPACE$. Mostriamo però che si può fare di più: usiamo un algoritmo che trova il percorso per dicotomia.

Algorithm 12 MIDDLESEARCH

Require: una matrice di adiacenza A , due vertici i e j , la lunghezza u del percorso

```

1: function MIDDLESEARCH( $i, j, u$ )
2:   if  $u = 1$  then
3:     if  $i = j$  or  $A[i, j] = 1$  then return yes
4:     else return no
5:   for  $k = 1 \dots n$  do
6:     if MIDDLESEARCH( $i, k, u/2$ ) = yes and MIDDLESEARCH( $k, j, u/2$ ) = yes then
7:       return yes
8:   return no
```

Con questo approccio creiamo un albero di chiamate ricorsive di profondità $\log n$, ognuna delle quali occupa $O(\log n)$ bit in memoria per salvare i propri valori i , k e j , quindi lo spazio massimo utilizzato sarà $O(\log^2 n)$ e possiamo dire che $REACH \in SPACE(\log^2 n)$, che è molto più efficiente da punto di vista spaziale (anche se la complessità temporale è aumentata).

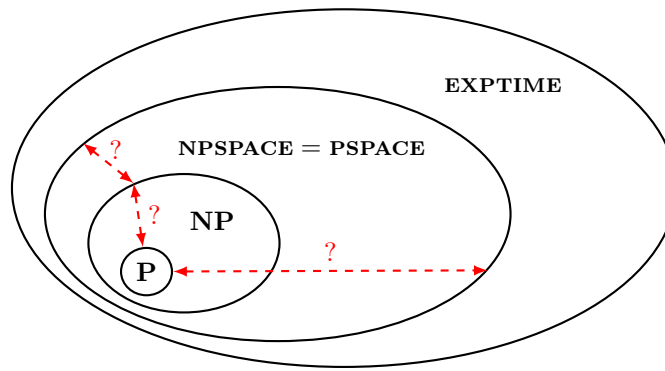
Da questo esempio deriva una considerazione ben più importante

Teorema di Savitch

$$NPSPACE(f(n)) \subseteq SPACE(f(n)^2)$$

cioè la differenza tra nondeterminismo e determinismo è *limitata* polinomialmente nel dominio spaziale.

Osservazione: dal teorema sopra otteniamo che $NPSPACE = PSPACE$.



9.3 PSPACE completezza

Nel dominio temporale abbiamo definito la classe dei problemi **NP**-COMPLETI, ovvero il nucleo dei problemi **NP** più difficili; vogliamo fare ora la stessa cosa anche per il dominio spaziale, mostrando che un problema deve necessariamente occupare almeno una certa quantità di spazio.

PSPACE completezza

Un problema A è **PSPACE**-COMPLETO se, come per il dominio temporale,

- $A \in \mathbf{PSPACE}$;
- A è **PSPACE**-HARD, ovvero $\forall B \in \mathbf{PSPACE}, B \preceq A$

Osservazione: per un qualsiasi problema $A \in \mathbf{PSPACE}$ -COMPLETO, abbiamo che $A \in \mathbf{P}$ se e solo se $\mathbf{PSPACE} = \mathbf{P}$ (la dimostrazione è semplice: il “se” è per definizione di **PSPACE**-COMPLETO, mentre per il “solo se” abbiamo $\forall B \in \mathbf{PSPACE} \implies B \preceq A$, quindi se $A \in \mathbf{P}$ anche $B \in \mathbf{P}$). Questo significa che se un qualsiasi problema in **PSPACE**-COMPLETO risulta essere anche **P**, allora l'intera gerarchia di classi computazionali (temporali e spaziali) collasserebbe a **P**.

Dimostriamo l'esistenza di un problema **PSPACE**-COMPLETO

Q-SAT

INPUT una formula $\phi(x_1 \dots x_n)$ CNF

OUTPUT $\text{yes} \iff$ la formula quantificata $\Phi = \exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n \phi(x_1 \dots x_n)$ è T

Questo problema rappresenta in pratica un gioco tra due persone e riprende il problema della partita a scacchi descritta nel primo capitolo: possiamo tradurre intuitivamente le assegnazioni come una concatenazione di mosse, dove P_1 sceglie x_1 , P_2 sceglie x_2 , P_1 sceglie x_3 e così via, e ϕ come lo stato iniziale della partita; raggiunta l'ultima mossa (la variabile x_n), la partita sarà terminata e verrà dichiarato il vincitore, ovvero P_1 se la formula risulta T, o altrimenti P_2 se la formula risulta F. Il problema Q-SAT chiede, tramite il formalismo della catena di quantificatori, se P_1 ha una strategia vincente a partire da ϕ (cioè se, dato lo stato iniziale ϕ della partita, esiste una prima mossa di P_1 per la quale per ogni prima mossa di P_2 esiste seconda mossa di P_1 ... per la quale P_1 vince dopo n mosse).

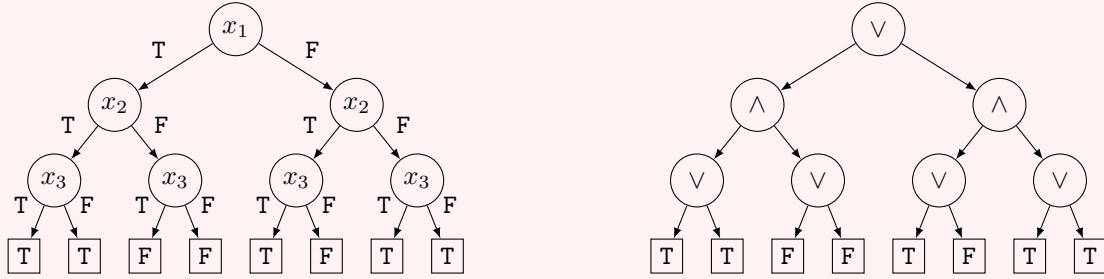
Possiamo vedere la risoluzione di una formula quantificata Φ come un albero decisionale binario in cui

- ogni ramo è un valore di verità T o F;
- ogni livello dell'albero contiene nodi che descrivono la stessa variabile;
- ogni nodo foglia è il risultato di ϕ utilizzando le assegnazioni scelte le percorso per arrivare a quella foglia

La soluzione si trova trasformando l'albero in un *albero AND-OR* a lui isomorfo, dove a partire dal nodo radice \vee , ogni livello alterna nodi \wedge a nodi \vee , codificando rispettivamente i quantificatori \forall (per il quale *tutti* sottoalberi devono essere T) ed \exists (per il quale *almeno* un sottoalbero deve valere T). In questo modo, semplicemente valutando ricorsivamente l'albero AND-OR possiamo dare la soluzione per Q-SAT.

Esempio

Data la formula $\phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$, utilizziamo l'albero di sinistra per rappresentare la formula quantificata $\Phi = \exists x_1 \forall x_2 \exists x_3 \phi(x_1, x_2, x_3)$ e l'albero di destra per rappresentare la soluzione di Q-SAT per Φ .



A partire dalla radice, valutiamo l'albero AND-OR assegnando valori alle variabili fino ad arrivare alle foglie, le quali dicono se con tali assegnamenti la formula ϕ è soddisfatta o meno.

Scriviamo quindi un algoritmo che, ad ogni scelta di variabile, in base al suo quantificatore, richiede che *tutti* i sottoalberi diano T o *almeno* un sottoalbero dia T:

Algorithm 13 Q-SAT-SOLVER

Require: una formula quantificata $\Phi = \exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n \phi(x_1 \dots x_n)$

Ensure: $\text{yes} \iff \Phi \text{ è T}$

- 1: **function** EVAL(ϕ)
- 2: **if** no quantifiers **then return** $\phi[\dots]$ // valutata usando gli assegnamenti finora
- 3: **if** $Q_1 = \exists$ **then return** EVAL($\Phi[x_1 \leftarrow \text{T}]$) OR EVAL($\Phi[x_1 \leftarrow \text{F}]$)
- 4: **if** $Q_1 = \forall$ **then return** EVAL($\Phi[x_1 \leftarrow \text{T}]$) AND EVAL($\Phi[x_1 \leftarrow \text{F}]$)

dove $\Phi[x_1 \leftarrow \text{T}] = Q_2 x_2 Q_3 x_3 \dots Q_n x_n \phi(\text{T}, x_2 \dots x_n)$. Questo algoritmo, essendo ricorsivo su un albero, è chiaramente esponenziale nel tempo, ma ci interessa lo spazio ora: nell'albero ci sono n livelli, uno per ogni assegnamento, e in ogni nodo devo mantenere in memoria gli assegnamenti delle variabili precedenti, quindi $O(n)$ bit (uno per ogni variabile, al massimo n), il che significa che l'algoritmo ha costo spaziale $O(n^2)$ e che Q-SAT \in PSPACE.

9.4 Esistenza dei problemi PSPACE-HARD

Teorema

Q-SAT è PSPACE-HARD

Dimostrazione. L'intuizione è la seguente: dato un problema $\mathbb{A} \in \text{PSPACE}$, l'algoritmo A che risolve $x \in \mathcal{I}(\mathbb{A})$ utilizza spazio $O(\text{poly}(n))$ con $n = |x|$, quindi A ha al massimo $O(2^{O(\text{poly}(n))})$ configurazioni di memoria possibili; l'algoritmo A non è altro che una codifica di tutti i passaggi da configurazione ad un'altra, e in pratica descrive un grafo i cui nodi sono le configurazioni e gli archi le computazioni volte da A . A partire dal nodo della configurazione iniziale, se l'istanza passata ad A è un'istanza **yes** allora potremo raggiungere la configurazione finale che codifica **yes**, e viceversa per il **no**: un qualsiasi problema $\mathbb{A} \in \text{PSPACE}$ è quindi equivalente al problema REACH, in cui il grafo è l'algoritmo A e i nodi iniziale e finale sono rispettivamente la configurazione di memoria iniziale e la configurazione che codifica **yes**.

Abbiamo visto che per risolvere il problema REACH possiamo usare MIDDLESEARCH e impiegare spazio $O(\log^2 n)$, l'interpretazione è la seguente: ogni volta che dimezziamo il percorso scegliendo il nodo al centro, stiamo codificando la scelta di una mossa da parte di P_1 , mentre P_2 sceglierà quale delle due porzioni del percorso riproporci, cercando man mano di farci fare la mossa sbagliata così da impedirci di raggiungere il nodo finale; se però P_1 riesce sempre a dare un nodo valido per il percorso, alla fine dell'algoritmo avremo raggiunto il nodo finale. Questo significa che il problema REACH è a sua volta equivalente a Q-SAT, e che quindi $\forall \mathbb{A} \in \text{PSPACE}, \mathbb{A} \preceq \text{Q-SAT}$, ovvero che Q-SAT è PSPACE-HARD. \square

In conclusione, Q-SAT è la versione spaziale di CIRCUIT-SAT.