Teoria della Computazione

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.

Capitolo 2

Prerequisiti di computazione

L'informatica è costruita su una logica matematica. Il punto di partenza è stato dettato da Turing (con la **macchina di Turing** (*TM*)) e questo pensiero si è poi sviluppato nel tempo. Turing, con la sua macchina logica, ha dimostrato che ci sono funzioni non calcolabili, verità logiche non dimostrabili.
Subito dopo la macchina di Turing nasce la teoria della **complessità computazionale**, col fine di classificare il problemi in base alla difficoltà delle soluzioni mediante macchine di calcolo. Tale *difficoltà* viene stimata rispetto a **spazio e tempo**. La teoria della **complessità computazionale** si riferisce a varie **classi di complessità** che classificano, in un primo approccio, *problemi decisionali* descritti da funzioni binarie che hanno in input una stringa sull'alfabeto $\{0, 1\}$ e restituiscono un bit (o 0 o 1). Questo perché le macchine di Turing ragionano in binario. Si ha quindi:

$$f : \{0, 1\}^* \to 0, 1$$

Esistono problemi che si è dimostrato non essere risolvibili in tempo efficiente.
Tra le classi abbiamo i **problemi NP** e **problemi P**. Inoltre i problemi NP sono a loro volta classificabili tra loro cercando i più difficili, ottenendo **problemi NP-hard** e **problemi NP-complete** (esistono varie dimostrazioni per la *NP-completezza*).

2.1 Tempo di calcolo di una TM

Definizione 1. *Sia $T : \mathbb{N} \to \mathbb{N}$ una funzione calcolabile da TM e $L\pi$ un linguaggio di decisione (dove π sta per "problema" e "di decisione" ci ricorda che il risultato sarà binario) allora una* ***TM deterministica*** *M accetta $L\pi$*

in tempo $T(n)$ se, $\forall x \in L\pi$, con $|x| = n$, M accetta x in $T(n)$ mosse o configurazioni

Definizione 2. *Un **problema di decisione** π riceve in input un'istanza x e l'output è:*

- *0 che vuole dire no*
- *1 che vuole dire yes*

Un linguaggio $L\pi$ restituisce 1 per tutti gli x che appartengono al linguaggio. Quindi $L\pi$ è l'insieme degli input di π su cui l'output è 1 (è l'analogo della funzione caratteristica *di un insieme, ovvero la funzione che risponde 1 sse un certo elemento appartiene all'insieme di riferimento).*
*La **funzione associata al problema** si chiama $f\pi$ ed è la funzione che dato un input restituisce 1 sse l'input appartiene al $L\pi$.*

Approfondiamo ora lo studio della **classe P**.

Definizione 3. *La classe dei linguaggi di decisione accettati in tempo $T(n) = cn^p$, $p \in \mathbb{N}$, $p \neq 0$ da una TM deterministica è detta **classe P**, quindi in un tempo polinomiale sulla dimensione dell'input n, è detta **classe P**. Quindi P è una classe di **problemi di decisione**.*
Potenzialmente p potrebbe anche non essere un intero in quanto si potrebbero avere tempi frazionari.

Definizione 4. *Si definisce che $L\pi$ è accettato da una TM in tempo $T(n)$ se $\exists\, T : \mathbb{N} \to \mathbb{N}$ calcolabile da TM e $\forall x \in L\pi$, con $|x| = n$, la TM accetta x e risponde 1 (yes) in al più $T(n)$ mosse di calcolo (dette anche configurazioni). Nel caso del modello della macchina RAM si ha la stessa situazione con però $T(n)$ **istruzioni RAM** e si dice che $L\pi$ è accettato dalla macchina RAM (si può dire che è anche deciso dell'algoritmo A della macchina RAM). In caso contrario la macchina RAM restituisce no, in quanto si parla di "decisione" oltre che di "accettazione" (a differenza della TM, dove però si può ottenere lo stesso discorso parlando di TM complementare M', che in $T(n)$ mi risponderà yes alla richiesta che un input non appartenga a $L\pi$, altrimenti bisogna fissare un limite di tempo per ottenere yes).*
È dimostrabile che se $L\pi$ è accettabile in tempo polinomiale allora nello stesso tempo è anche decidibile.
*La differenza tra accettazione e decisione sarà fondamentale nel **modello non deterministico**.*

Si ricordi che il **modello RAM (*Random Access Machine*)** è usato per studiare il tempo di calcolo di uno pseudocodice. È un modello teorico (una macchina teorica "simile" a quelle reali) dotato di istruzioni come *load, store, add, etc...* dove un codice (ipoteticamente in qualsiasi linguaggio incluso lo pseudocodice) viene tradotto in una sorta di linguaggio macchina (linguaggio RAM), dove n è un intero rappresentante il numero di istruzioni RAM necessarie per ottenere l'output (n è detto **tempo uniforme**). Sul linguaggio RAM si può studiare anche lo spazio calcolato come numero di bit necessari per la computazione (è detto **costo logaritmico**). In questo secondo punto il costo di un'istruzione, come ad esempio *load(n)*, è logaritmico rispetto all'operando n ($\log_2 n$), studia quindi la *dimensione* dell'input.

Consideriamo ora un modello basato su algoritmi.

Definizione 5. *Sia $L\pi$ un linguaggio di decisione e $t : \mathbb{N} \to \mathbb{N}$ una funzione calcolabile, allora un algoritmo A accetta $L\pi$ in tempo $T(n)$ sse $\forall x \in L\pi$, con $|x| = n$, A termina su input x dopo $T(|x|)$ passi di calcolo, ovvero istruzioni esguite, producendo 1 come output. Quindi P è la classe dei linguaggi di decisione accettati in tempo $T(n) = cn^p$, $p \in \mathbb{N}$ (con lo stesso discorso di sopra su p) da un algoritmo A in tempo polinomiale.*

Capitolo 3

Complessità computazionale

Si cerca di catalogare dal punto di vista computazionale i **problemi intrattabili**, ovvero problemi risolvibili ma non in modo **efficiente** (ovvero in tempo polinomiale). In alcuni casi si pensa che non esista una soluzione ma non si hanno dimostrazioni in merito mentre in altri casi è addirittura dimostrato. Abbiamo quindi delle categorie informali per i problemi:

- **facili**, so risolverli in modo efficiente. È la **classe P**
- **difficili** o, più formalmente, **intrattabile**, so risolverli ma non in modo efficiente e non ho una dimostrazione che mi assicuri che non siano risolvibili in modo efficiente. È la **classe NP** e la sua sottoclasse **NP-complete**
- **dimostrabilmente difficili**, so risolverli ma so che non esiste un algoritmo efficiente in quanto è stato dimostrato che non può esistere
- **impossibili**, non so risolverli sempre neanche in modo non efficiente (esiste almeno un input che manda in crisi l'algoritmo ma esiste almeno un caso in cui funzioni)

Come formalismo useremo la **Macchina di Turing (*TM*)**, *deterministica* e *non deterministica*. Analizzeremo in primis **problemi sui grafi**. Un grafo è definito come $G = (V, E)$, con V insieme dei vertici e E insieme degli archi. Un grafo può essere *orientato* o *non orientato*. Un **cammino** tra due vertici è una sequenza di archi che mi porta da un vertice all'altro. Un cammino è detto **ciclo** se il vertice sorgente coincide con quello di destinazione. Due vertici sono **connessi** se esiste un cammino che li collega. Un **grafo connesso** è un grafo dove per ogni coppia di vertici si ha che essi sono connessi. Se questo cammino è di un solo arco si parla di **grafo completo**, ovvero ogni

vertice è **adiacente** ad ogni altro. Si parla di **grafo pesato** se si ha una funzione W che associa un peso ad ogni arco.
Useremo anche la teoria dei linguaggi formali con V alfabeto e stringhe costruite su V. Con ε abbiamo la stringa vuota e con V^* è l'insieme di tutte le possibili stringhe costruibili con quell'alfabeto, inclusa la stringa vuota. V^* è un insieme infinito. Con V^+ indico V^*/ε, ovvero senza la stringa vuota. Un **linguaggio** L è un sottoinsieme di V^*, quindi $L \subseteq V^*$, che comprende tutti gli elementi di V^* che seguono una certa **proprietà** (o più proprietà). Anche L è un insieme infinito.
Un'altra nozione è quella di **problema**. Un problema computazionale è una "questione" a cui si cerca risposta. Più formalmente un problema è specificato da **parametri** (l'input del problema) e le **proprietà** che deve soddisfare la **soluzione** (l'output). L'**istanza** di un problema specificando certi parametri in input al problema (input che devono essere coerenti ai parametri richiesti).
Cominciamo con degli esempi di problemi comunque risolvibili.

Esempio 1. *Considero il problema* arco minimo*. Come parametro ho un grafo pesato sugli archi $G = (V, E)$. Le proprietà della soluzione è che voglio l'arco con peso minimo.*
*Per risolvere guardo tutti gli archi è vedo quello di peso minimo. Dato che basta iterare su tutti gli archi quindi la soluzione è in $O(n)$ (in realtà $\Theta(n)$), quindi in **tempo lineare** sul numero di archi (è quindi in **tempo polinomiale**)*

Esempio 2. *Considero il problema* raggiungibilità*. Come parametro ho un grafo non pesato $G = (V, E)$ e due vertici, uno sorgente e uno destinazione, tali che $v_s, v_d \in V$. Le proprietà della soluzione è che voglio sapere se posso arrivare a v_d partendo da v_s.*
*Per risolvere studio tutti i cammini che partono da v_s e posso dare la risposta. Una soluzione del genere è in tempo $O(2^{|E|})$. Il tempo quindi cresce in **modo esponenziale**. Una soluzione migliore è quella di usare un **algoritmo di visita** che richiede tempo $O(|V| + |E|)$, ovvero un **tempo polinomiale**. Quindi per quanto all'inizio si pensi che sia un **problema intrattabile** si scopre che è un **problema facile***

Esempio 3. *Considero il problema* TSP*. Come parametro ho un grafo pesato sugli archi e completo $G = (V, E)$. Le proprietà della soluzione è che voglio sapere il* cammino minimo *(in realtà un ciclo) che tocca tutti i vertici una e una sola volta (una volta trovata la soluzione non mi interessa la sorgente essendo il grafo completo).*
*Sarebbe facile determinare **un** ciclo ma non quello di peso minimo e per*

farlo devo trovare tutti i cicli e trovare quello di peso minimo. Ho quindi un algoritmo che è $O(2^n)$ *(nella realtà è circa* $O(n!)$ *che è comunque esponenziale per l'**approssimazione di Stirling**). In questo caso non si riesce a pensare ad una soluzione che non sia esponenziale nel tempo (anche se per alcuni input sia di facile risoluzione, basti pensare ad avere tutti gli archi di peso 1, ma mi basta avere un input problematico). Non potendo però dimostrare che sia irrisolvibile si dice che è un **problema intrattabile**. TSP è uno dei 10 problemi famosi per i quali ti danno un milione di dollari se dimostri che è o* facile *o* impossibile

Per completezza definiamo un **algoritmo** come una sequenza di **istruzioni elementari** (supportate dal calcolatore) che, eseguite in sequenza, mi portano alla soluzione di un problema. Si ha quindi che un algoritmo A risolve un problema Π se per ogni possibile istanza di Π l'algoritmo A mi da la risposta corretta. Distinguo però:

- **algoritmo efficiente**, che mi da la soluzione in **tempo polinomiale** rispetto alla **dimensione dell'input**. Ho un *caso peggiore* limitato superiormente da un **polinomiale**: $O(p(n))$. Ho una crescita di tempo accettabile all'aumentare dell'input. Diciamo comunque che è dura anche solo raggiungere $O(n^{10})$ quindi anche se dire polinomiale potrebbe voler dire $O(n^{10000000})$ non si hanno casi reali di questo tipo

- **algoritmo non efficiente**, che mi da la soluzione ma in tempo superiore a quello **polinomiale**. Ho un *caso peggiore* limitato superiormente da un **esponenziale**: $O(2^n)$. Ho una crescita di tempo assolutamente non accettabile (esponenziale appunto) all'aumentare dell'input

Se ho anche solo un caso di input che porta a tempo esponenziale ho comunque un algoritmo non efficiente.
Spesso **problemi intrattabili** vengono risolti tramite approssimazioni per arrivare ad una soluzione accettabile anche se non la migliore ma non sempre è possibile effettuare delle approssimazioni.
Anche se avessi ha che fare con un computer mille volte migliore di quelli attuali, un problema esponenziale avrà comunque tempi non accettabili in proporzione ad un problema polinomiale. Quindi non sarà il miglioramento hardware a permettere di rendere accettabile la soluzione di problemi esponenziali.
Un **algoritmo intrattabile** quindi non risolve in modo efficiente tutti gli input. Bisognerà trovare un modo per capire se un algoritmo è **intrattabile**

per davvero.
Studiamo ora il tempo di calcolo di una **macchina di Turing non deterministica (*NTDM*)**:

Definizione 6. *Sia $T : \mathbb{N} \to \mathbb{N}$ una funziona calcolabile da TM. Dato $L\pi$ un linguaggio di decisione allora una $NDTM$ M. M accetta $L\pi$ in tempo $T(n)$ se per ogni x in $L\pi$,con $|x| = n$, M accetta x in $T(n)$ mosse (o configurazioni).*
*Quindi la **classe NP** è la classe dei linguaggi di decisione accettati in tempo $T(n) = cn^p$, $p \in \mathbb{N}$ da una $NDTM$*

Diamo una definizione alternativa di NP:

Definizione 7. *Sia $L\pi$ un linguaggio di decisione, $N : n \to n$ una funzione calcolabile, y una stringa di lunghezza polinomiale nell'input, allora un algoritmo A con "certificato" accetta $L\pi$ in tempo $t(n)$ se per ogni x in $L\pi$, con $|x| = n$, A termina su input (x, y) dopo $t(|x|)$ passi di calcolo (istruzioni eseguite) producendo 1 in output. Quindi la **classe NP** è la classe dei linguaggi (o problemi) di decisione accettati in tempo $T(n) = cn^p$, $p \in \mathbb{N}$ da un algoritmo A "certificato"*

Resta da capire il significato del termine "certificato".

Definizione 8. *Preso un algoritmo A definiamo cosa significa che sia "certificato" si compone di un input (x, y) con y che è una stringa, nel dettaglio una **dimostrazione**, che garantisce che $x \in L\pi$.*

Vediamo un esempio:

Esempio 4. *Uso un problema NP come esempio, quindi $\pi \in NP$, per avere un algoritmo che ammette certificato (non ho alternative).*
Prendo il problema π vertex-cover*. Come input si ha un grafo $G = (V, E)$ e un intero k. Come output ho o 1 o 0, essendo un problema di decisione, e si cerca di capire se $\exists V' \subseteq V$ tale che V' è una copertura di G. Il sottoinsieme è copertura di un grafo quando $forall\, e \in E$ almeno un estremo dell'arco $e = (u, v)$ è in V'. La copertura con il minor numero di vertici è detta **minima copertura**.*
Il problema vertex-cover *chiede se esiste una copertura del grafo di dimensione k. È quindi un **problema di decisione**. Qualora trovassi una copertura di cardinalità minore di k mi basterà aggiungere vertici arbitrari fino al raggiungimento di k. Ovviamente può non esistere una copertura di cardinalità k.*
Il problema di vertex-cover *è il problema di decisione del problema di trovare*

la minima copertura di un grafo, che è un ***problema di ottimizzazione (o problema di ottimo)*** *(ogni problema di ottimizzazione può essere trasformato in uno di decisione aggiungendo un parametro k e richiedendo un risultato booleano).*
vertex-cover *decisionale è nella classe* ***NP*** *con "certificato" e, in questo caso, il parametro y è la dimostrazione che posso dare risposta affermativa, per esempio la certezza di avere un sottoinsieme di vertici che effettivamente copre tutto il grafo, quindi $y = V'' \subseteq V$ tale per cui V'' copre G con cardinalità k. Bisogna capire come trovare e come usare y, sapendo che A lavora in tempo polinomiale e che la lunghezza di y è polinomiale in dimensione di x. Vedendo che la cardinalità di y è minore di k l'algoritmo A verifica che con i vertici passati con y si è in grado di rispondere al problema in modo affermativo, problema che ha in input G e k. In poche parole A, per ogni arco, esamina se ogni estremo è in y e, se tutti gli archi hanno un estremo in y allora si ha che usando y si è in grado di verificare l'input G, k è* ***accettato****. Questa verifica è in tempo polinomiale e si ha che $|y| = O(|G, k|)$, soprattutto se $|y|$ è una costante.*
Ad oggi non si è dimostrato che vertex-cover *si risolvibile in tempo polinomiale. È quindi un problema* ***NP-hard****.*

Esempio 5. *Per l'algoritmo TSP la versione di ottimo è trovare il* tour *di costo minimo. Il problema di decisione è se esiste un* tour *di massimo peso k. In questo caso già il problema di decisione è già in* ***NP*** *e lo è anche il problema di ottimo. La verifica con "certificato" ha comunque costo polinomiale nel caso di richiesta di verifica di un dato* tour *con costo minore di k.*

Il problema di decisione è una restrizione di quello di ottimo.
Per notazione dato un algoritmo A chiamiamo A_d la sua versione di decisione.

Senza "certificato" non potrei accettare un problema NP.
Un problema di **classe NP** quindi ammette verifica in tempo polinomiale, ammettendo un "verificatore" in tempo polinomiale per i problemi specificati.
Non è così scontato trovare "certificato", di dimensione polinomiale nell'input, e trovare il "verificatore". Per esempio la classe **exptime** non esiste A con "certificato" che sia polinomiale (la verifica potrebbe richiedere tempo esponenziale).
Quindi per un problema **NP** con "certificato" non posso trovare soluzione in tempo polinomiale ma posso verificare una soluzione data in tempo polinomiale.
Posso quindi dimostrare che un problema π, con in input una struttura combinatoria discreta (come un grafo) e un intero, è in **NP**.

Si ha quindi che **P** è vista come sottoclasse la **NP** dove i problemi di decisione sono risolvibili in tempo polinomiale anche senza "certificato". Per dimostrare che $P \subseteq NP$ devo far vedere che ogni $L\pi \in P$ ammette un algoritmo A con "certificato" in tempo polinomiale. Ma questo è vero perché $L\pi$ è accettato da un algoritmo A, in tempo polinomiale con $y = \emptyset$ e quindi y non è necessario.
Si ha quindi che $P \subseteq NP$. Pensando poi, per esempio, all'*isomorfismo di grafi* nessuno sa se sia P o NP. Questo problema ha in input due grafi e come output se sono isomorfi tra loro.

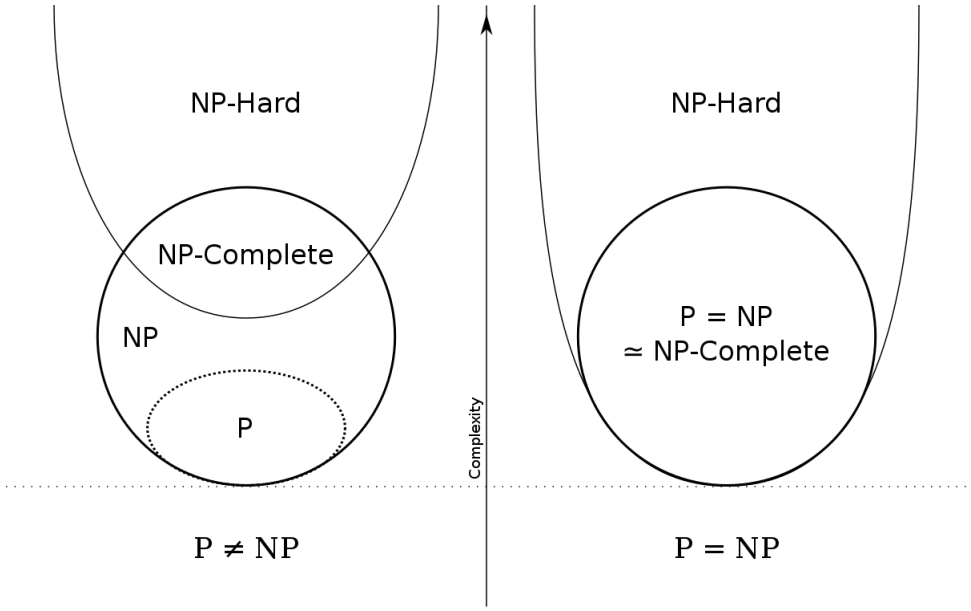


Figura 3.1: Diagramma di *Eulero Venn* per le classi di complessità

Tratto dagli appunti di Metodi Formali:

Si parla di isomorfismo quando due strutture complesse si possono applicare l'una sull'altra, cioè far corrispondere l'una all'altra, in modo tale che per ogni parte di una delle strutture ci sia una parte corrispondente nell'altra struttura; in questo contesto diciamo che due parti sono corrispondenti se hanno un ruolo simile nelle rispettive strutture.

Diamo ora una definizione formale di isomorfismo tra sistemi di transizione etichettati, che possono quindi essere grafi dei casi o grafi dei casi sequenziali.

Definizione 9. *Siano dati due sistemi di transizione etichettati:*
$A_1 = (S_1, E_1, T_1, s_{01})$ *e* $A_2 = (S_2, E_2, T_2, s_{02})$.
e siano date due ***mappe biunivoche****:*

1. $\alpha : S_1 \to S_2$, *ovvero che passa dagli stati del primo sistema a quelli del secondo*
2. $\beta : E_1 \to E_2$, *ovvero che passa dagli eventi del primo sistema a quelli del secondo*

allora:

$$\langle \alpha, \beta \rangle : A_1 = (S_1, E_1, T_1, s_{01}) \to A_2 = (S_2, E_2, T_2, s_{02})$$

è un ***isomorfismo*** *sse:*

- $\alpha(s_{01}) = s_{02}$, *ovvero l'immagine dello stato iniziale del primo sistema coincide con lo stato iniziale del secondo*
- $\forall s, s' \in S_1, \forall e \in E_1 : (s, e, s') \in T_1 \Leftrightarrow (\alpha(s), \beta(e), \alpha(s')) \in T_2$ *ovvero per ogni coppia di stati del primo sistema, tra cui esiste un arco etichettato e, vale che esiste un arco, etichettato con l'immagine di e, nel secondo sistema che va dall'immagine del primo stato considerato del primo sistema all'immagine del secondo stato considerato del secondo sistema, e viceversa*

Definizione 10. *Si definiscono due* ***sistemi equivalenti*** *sse hanno grafi dei casi sequenziali, e quindi di conseguenza anche grafi dei casi,* isomorfi.
Due sistemi equivalenti accettano ed eseguono le stesse sequenze di eventi

3.1 Riduzioni polinomiali

Si hanno alcuni problemi che sono in grado di risolvere qualunque problema di decisione in **NP**. Serviranno prima le definizioni di **NP-hard** e **NP-complete**.
Vediamo innanzitutto il problema *independent-set* che ci aiuterà analisi.

Definizione 11. *L'independent-set di un grafo non orientato è un sottoinsieme* $I \subseteq V$ *tale che* $\forall u, v \in I \ (u, v) \notin E$. *Il problema ind_set, nella versione di ottimo, è quello di trovare l'independent-set di cardinalità massima di un grafo non orientato. Nella versione di decisione* ind_set_d *si ha*

anche il parametro k intero e si cerca se esiste un independent-set di cardinalità uguale a k. L'independent-set di cardinalità massima può essere usato come "certificato".

Questo problema è legato alla *copertura dei vertici*, infatti sappiamo che se dall'insieme dei vertici togliamo un sottoinsieme di minima copertura troviamo un *independent-set* di cardinalità massima perché sto facendo il complemento di un insieme di copertura di cardinalità minima, infatti tra i vertici non nell'insieme di copertura di cardinalità minima, ovvero nel complemento, non posso avere un arco per definizione e quindi se il primo è di cardinalità minima allora il secondo, che è l'*independent-set*, è di cardinalità massima. Infatti i vertici nella copertura sono vertici che toccano tutti gli archi.
Dal punto di vista delle applicazioni pratiche questi problemi si prestano allo studio, per esempio, delle telecomunicazioni.

Dimostrazione. Dimostriamo che ind_set_d è **NP**, infatti esiste un algoritmo A che in costo polinomiale prende in ingresso il grafo, k, e un "certificato" y e i vertici in y, che sono vertici di un *independent_set* per il grafo G di cardinalità k. L'algoritmo verifica che y è un *independent-set* e il costo della verifica è quadratico su $|y|$, ovvero $|y|^2$ che nel caso peggiore è $|V|^2$. So anche che, per l'input x, $O(|x|) = O(|E| + |V|) = O(|V^2| + |V|)$ nel caso peggiore, quindi il tempo di verifica è **polinomiale**. □

Definizione 12. *Vediamo ora il problema di **soddisfacibilità** SAT. Questo problema prende in input una formula booleana ϕ in **forma normale congiunta (CNF)**, ovvero che ha una congiunzione (d) come legame tra le **clausole**. Una clausola è un $\vee$ di **letterali**, ovvero di variabili booleane x_i o $\neg x_i$. In output ho se la forma sia soddisfacibile o meno.*

Esempio 6. *Prendo 3 variabili, x_1, x_2, x_3. Creo i letterali x_1, x_2, x_3 e anche $\neg x_1, \neg x_2, \neg x_3$. Creo quindi le clausole $c_1 = x_1 \vee x_2$, $c_2 = x_1 \vee \neg x_2$ e $c_3 = x_1 \vee \neg x_2$. Definisco quindi la CNF ϕ:*

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) = c_1 \wedge c_2 \wedge c_3$$

Quindi in ogni clausola almeno un letterale deve essere vero, cosicché tutte le clausole siano vere rendendo vera la CNF.
Avendo due letterali a clausola si è definito un $2SAT$.

*Il numero di letterali k che compongono la clausola definsice un problema $kSAT$. Si ha che $2SAT \in P$ ma con $k > 2$ si ha che $kSAT \in NP$ (in realtà è in **NP-hard**)*

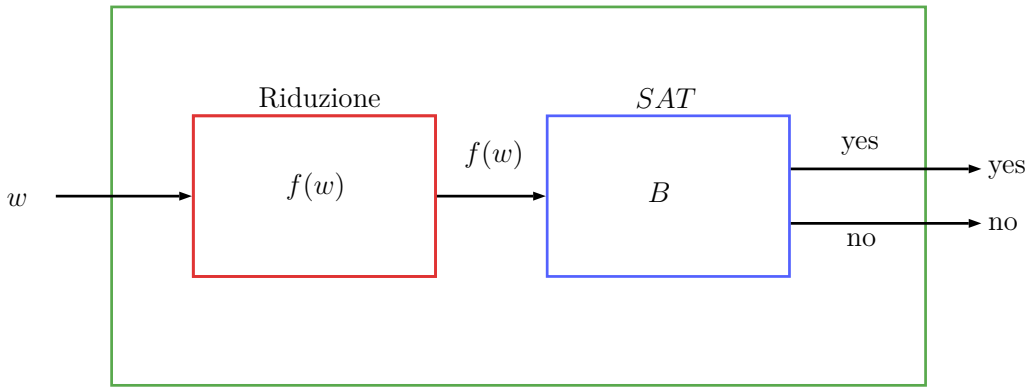


Figura 3.2: Rappresentazione grafica della riduzione

Definizione 13. *Definiamo un problema **NP-hard** come un problema difficile almeno quanto un problema **NP**. Ogni che problema A in **NP** può essere risolto con un chiamata di procedura a B, che è un problema **NP-hard** a cui tutti gli altri "chiedono aiuto" per trovare una soluzione.*
*Ad esempio vertex-cover è un problema **NP-hard** e quindi posso risolvere ogni problema A in **NP** con il problema vertex-cover B.*
Trasformo quindi l'input w di A in un input $f(w)$ per B in tempo polinomiale. La risposta di B con input $f(w)$ è la stessa che A da su input w.
*Non tutti i problemi **NP-hard** sono dentro la classe **NP***

Definizione 14. *Definiamo quindi il concetto di **riduzione**, rappresentato in figura 3.2.*
La riduzione è la trasformazione dell'input di w in A in un input $f(w)$ per B, in tempo polinomiale. La risposta di B con input $f(w)$ è la stessa che A da su input w.
Si ha che A si riduce polinomialmente a B, e si scrive:

$$A \leq_p B$$

se $\exists\, f$ tale che:

$$w \in L_A \text{ sse } f(w) \in L_B$$

con f calcolabile in tempo polinomiale, infatti il calcolo di $f(w)$ è $= (|w|^p)$, con $p \in \mathbb{N}$ per semplicità. Non devo introdurre una complessità superiore nel contesto di confronto tra problemi (??).
*Ogni problema A che in **NP** può essere risolto con una chiamata di procedura a B, quindi posso risolvere ogni problema $A \in NP$ con vertex-cover, essendo esso un problema **NP-hard**.*

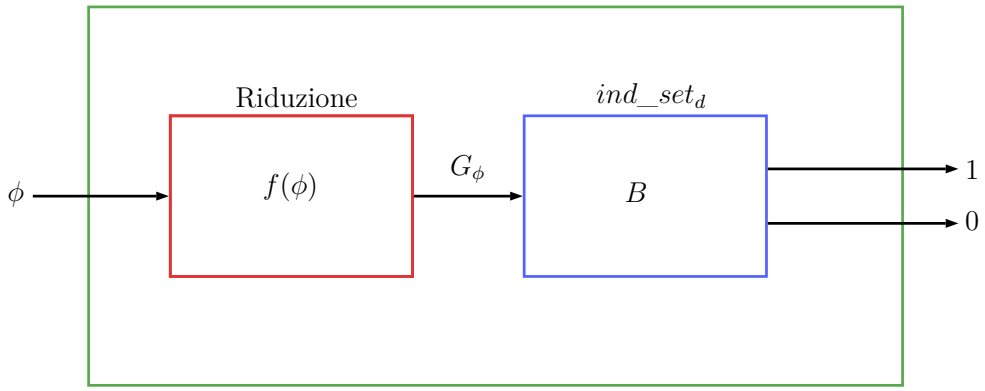


Figura 3.3: Rappresentazione grafica del'esempio 7

Definizione 15. *B è **NP-hard** sse $A \in NP$ A si riduce a B in tempo polinomiale:*

$$A \leq_p B, \ \forall A$$

Un problema $NP-hard$ può non essere in NP, in quanto potrebbe non avere un "certificato" per consentire la verifica in tempo polinomiale.

Definizione 16. *Un problema **NP-hard** e anche **NP** si dice che il problema è **NP-complete***

$kSAT$ è il primo problema che si è dimostrato essere anche **NP-complete**.

Esempio 7. *Vediamo un esempio di riduzione, rappresentata in figura 3.3:*

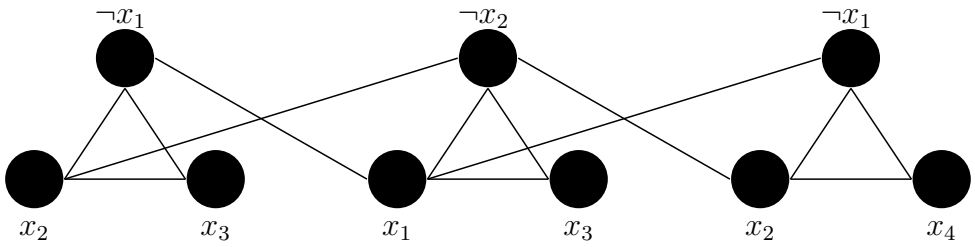
$$3SAT \leq_p ind\_set_d$$

*arrivando e alla conclusione che ind_set_d è **NP-completo**.*
e vediamo che ϕ è soddisfacibile sse G_ϕ ha un independent-set di dimensione $k = |\phi|$, con $|\phi|$ pari al numero di clausole della formula.
Costruisco quindi un grafo che ha un vertice per ogni letterale della clausola. Collego i tre letterale della clausola ottenendo un "triangolo", detto gadget, che rappresenta una clausola. Infine collego ogni letterale al suo negato.

Quindi per la formula:

$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

avrò il grafo G_ϕ che codifica la formula ϕ:



*Quindi un **gadget** è una rappresentazione dell'input del problema A di partenza e ogni **gadget** rappresenta una clausola.*
*Ricordiamo che ϕ è soddisfacibile sse esiste un assegnamento delle variabili della formula tale per cui almeno un letterale di ogni clausola è vero. Nel grafo relativo alla formula lego quindi un letterale ad ogni suo complemento al fine di poter identificare i valori di verità e avendo il calcolo di independent-set (per la **riduzione**) prendo uno solo degli estremi di un arco, e quindi uno solo tra x_i e $\neg x_i$, codificando l'assegnamento di verità. Il concetto di **riduzione** è quindi ritrovabile nella capacità di rappresentare un problema in un'altra forma, studiabile con un altro algoritmo.*

Dimostrazione. Indichiamo con A $3SAT$ e con B *independent-set*.
Effettuiamo quindi la prova finale, la dimostrazione vera e propria. A questo livello di comprensione abbiamo dimostrato l'esistenza della funzione f, che trasforma ϕ (ovvero l'input di A) in $\langle G, k\rangle$, ovvero l'input di B, e che essa è in tempo polinomiale. Abbiamo quindi che $w \in L_A$ sse $f(w) \in L_B$. Se ϕ è vera allora esiste un *independent-set* di dimensione k per $\langle G, k\rangle$.
Nell' "altro verso" abbiamo che se esiste un *independent-set* di dimensione k per $\langle G, k\rangle$ allora ϕ è vera. Dimostrare questi due "versi" equivale a dimostrare la riduzione.
Il **primo verso** si dimostra dicendo che dato un assegnamento di verità si seleziona un letterale vero da ogni triangolo. Questi letterali veri scelti formano l'*independent-set* S, che ha dimensione k. Questo può accadere sse ϕ è vera, infatti per ogni clausola c_i $\exists$ l_{ij}, letterale, che rende vera c_i, a questo punto tale letterale è un vertice del triangolo, ovvero del gadget g_{c_i}, (mi basta infatti un letterale vero per triangolo) e, poiché tutte le clausole sono vere, ho la scelta di k vertici se k è il numero delle clausole. Esiste quindi un *independent-set* di dimensione k.

Per questo si potrebbe fare una **dimostrazione per costruzione**, ovvero se rendo vera c_y con l_y significa che la variabile x_i può essere usata o come 1 o come 0 nell'assegnamento di verità in un altro letterale l_z, che rende vera la clausola c_z. Ma x_i, se già usata, non posso più usarla con valore opposto a quello scelto per c_y e quindi non esiste un arco di collegamento tra l_y e l_z. Si può dimostrare anche per **assurdo**. Se esiste un arco tra i due letterali l_z e l_y che rendono vere le clausole c_z e c_y, allora ottengo una contraddizione sugli assegnamenti di verità, asserendo che i due letterali sono uno la negazione dell'altro ma entrambi sono veri, per poter rendere vere le clausole.

Il **secondo verso** si dimostra dicendo che, dato un *independent-set* S di dimensione k, S deve contenere un vertice per triangolo. Ponendo quindi i letterali contenuti in S come veri si ottiene un assegnamento di verità che è **consistente** e tutte le clausole sono soddisfatte. Quindi se esiste un *independent-set* di dimensione k allora trovo un assegnamento alle variabili $x_i,\ \forall\ 1 \dots n$ che rende vera ϕ, ovvero assegno 0 o 1 a ciascuna variabile (ovviamente o 1 o 0, non entrambi). Se esiste l'*independent-set* di dimensione k pari al numero delle clausole, allora per ogni gadget g_{c_i}, che rappresenta una clausola, esiste un vertice nel gadget che si trova anche nell'*independent-set*, quindi esiste un letterale, per ogni clausola c_i tale che non è collegato ad un altro letterale dell'*independent-set*, ovvero non è collegato ad un altro letterale di un'altra clausola (ovvero ho un nodo per gadget che non ha un arco verso un nodo di un altro gadget). Quindi per ogni letterale vedo la variabile che rende vero il letterale e con il valore dato alla variabile costruisco l'assegnamento o 0 o 1 a quella variabile (se $l_i = \neg x_j$ allora $x_j = 0$ e se $l_i = x_j$ allora $x_j = 1$). Trovo quindi l'assegnamento delle variabili che è di verità per ϕ, dimostrando quindi che ϕ è vera. □

Siccome $3SAT$ è **NP-complete** allora anche *independent-set* è **NP-complete**, infatti:

$$\forall\ A_{\in NP} \leq_p 3SAT \leq_p\ \textit{independent-set}$$

in quanto la riduzione $\leq_p$ è **transitiva**, e quindi:

$$\forall\ A_{\in NP} \leq_p\ \textit{independent-set} \in NP$$