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.

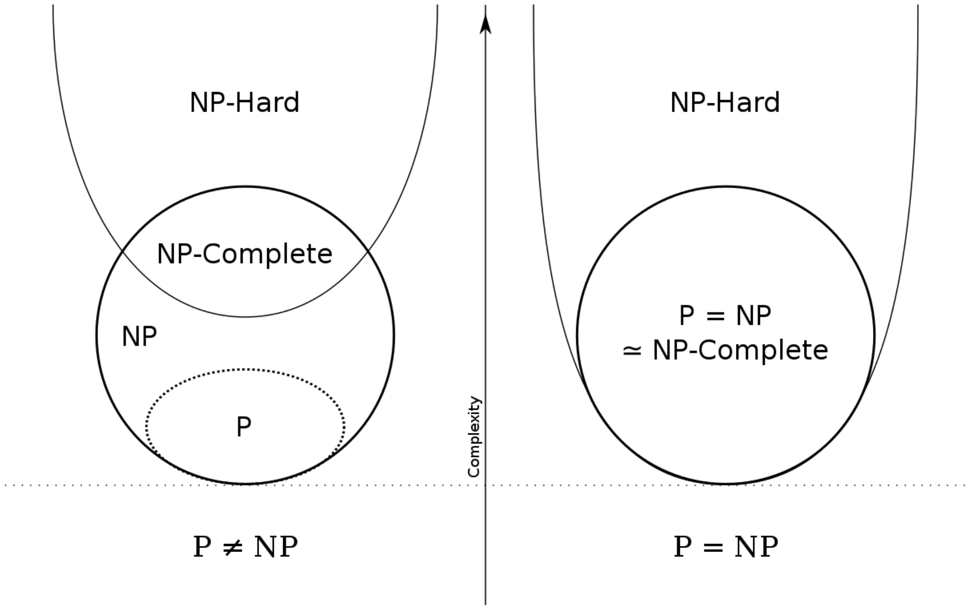


Figura 3.1: Diagramma di *Eulero Venn* per le classi di complessità

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 ($\wedge$) 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****)*

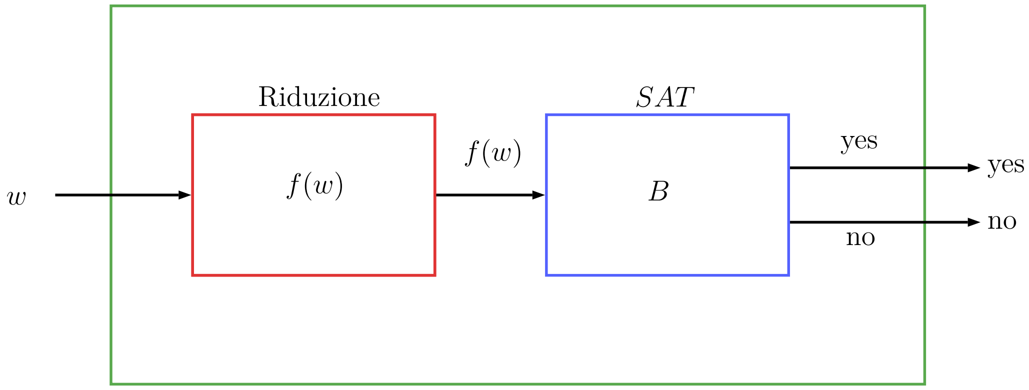


Figura 3.2: Rappresentazione grafica della riduzione

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****.*

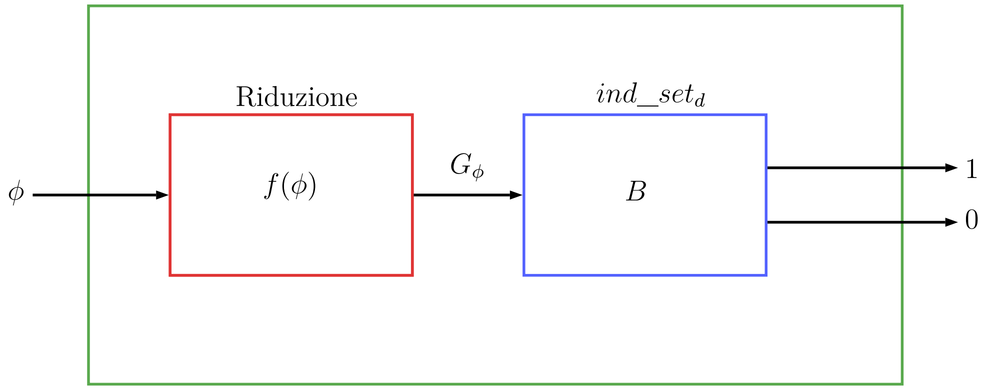


Figura 3.3: Rappresentazione grafica del'esempio 7

Definizione 15. *B è **NP-hard** sse $\forall A \in NP$ 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:*

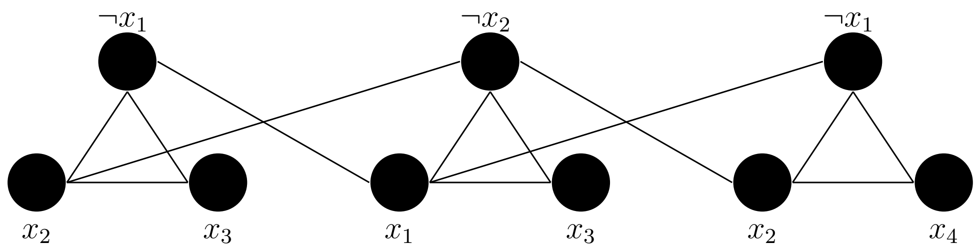
$$3SAT \leq_p ind\_set_d$$

*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:

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

Teorema 1. *Se un problema* Π ***NP-complete*** *è in* ***P*** *allora si può dire* $P = NP$*, implicando che ogni problema* $A \in NP$ *è risolvibile da* $\Pi \in P$*. Questa cosa non è stata ancora dimostrata (e probabilmente si riuscirà a dimostrare l'opposto).*

Dimostrazione. Per ogni problema A in **NP** so che $A \leq_p \Pi$, ovvero Π è una procedura che risolve A, con trasformazione dell'input x di A nell'input $f(x)$

di Π, con $f(x)$ calcolabile in tempo polinomiale.
Assumendo quindi che $\Pi \in P$ allora anche ogni $A \in P$, e quindi $NP = P$. □

Teorema 2. *La riduzione polinomiale è transitiva. Ovvero se $A \leq_p B$ e $B \leq_p C$ allora:*

$$A \leq_p C$$

Dimostrazione. Infatti $A \leq_p B$ implica l'esistenza di f tale che $x \in L_A$ sse $f(x) = L_B$. Ugualmente $B \leq_p C$ implica l'esistenza di g tale che $x \in L_B$ sse $g(x) = L_C$.
Devo dimostrare che $\exists\ f'$ tale che $x \in L_A$ sse $f'(x) \in L_C$. Per ottenere f' compongo f e g. Assumendo che x appartiene all'input di ha compongo le funzioni, quindi ho che $f' = g \circ f$, e ho che $x \in L_A$ e che $f(x) \in L_B$ (quindi $f(x)$ è un input per B) ma quindi $g(f(x)) \in L_C$ (quindi $g(f(x))$ è un input per C). Ho quindi dimostrato la transitività.
Se f e g sono costruibili in tempo polinomiale, la prima sulla dimensione di x e la seconda su quella di $f(x)$, allora anche $f' = g \circ f$ è costruibile in tempo polinomiale, proporzionalmente alla cardinalità di x. □

Quindi per dimostrare che un problema Π' è **NP-hard** devo ridurre Π' ad un problema qualsiasi **NP-hard** (in base alla somiglianza del problema), sapendo che $\forall A \in NP\ A$ si riduce ad un problema **NP-hard**, come SAT.
In modo equivalente per dire che è un problema è **NP-complete** faccio quanto fatto per **NP-hard** ma devo aggiungere che esso sia in NP, dovendo quindi aggiungere il "certificato".

Teorema 3. *Si ha che $SAT \leq_p 3SAT$*

Dimostrazione. **Dimostrazione solo parziale, solo l'inizio è stato fatto in aula.**
Prendo una ϕ con $k \geq 4$ letterali. Ogni clausola deve diventare una clausola con al più 3 letterali (introducendo nuovi letterali ogni volta che viene negato uno). □

3.1.1 Problema set-cover

Questo problema si applica bene allo studio, nel campo delle telecomunicazioni, dei ripetitori e dello studio della copertura per reti mobili.

Definizione 17. *Dato un universo U di n elementi sia $S = \{S_1, \dots, S_M\}$ una collezione di sottoinsiemi di U. Sia anche data una funzione di costo $c : S \to \mathbb{Q}^+$. Il problema **set-cover** consiste nel trovare una collezione C di sottoinsiemi di S di costo minimo che copra tutti gli elementi di U.*

Esempio 8. *Se ho* $U = \{1, 2, 3, 4, 5\}$*,* $S_1 = \{1, 2, 3\}$*,* $S_2 = \{2, 3\}$*,* $S_3 = \{4, 5\}$ *e* $S_4 = \{1, 2, 4\}$*, con* $S = \bigcup S_i$ *. Per praticità assumo costo uniforme, ovvero che* $c_1 = c_2 = c_3 = c_4 = 1$*.*
Quindi la soluzione C *è* $\{S_1, S_3\}$*, in quanto questi due insiemi coprono tutti gli elementi di* U*, con costo pari a* 5 *(che è il minimo che posso avere).*

Definizione 18. *Qualora il costo sia uniforme allora il* ***set-cover*** *diventa la ricerca di una sotto-collezione che copra tutti gli elementi di* U *con minima dimensione.*

Teorema 4. *set-cover, nella versione decisionale e nella versione con peso uniforme, è* ***NP-complete*** *(quindi se esiste una collezione* C *di sottoinsiemi di* S *la cui unione sia* U *con cardinalità minore uguale di* k*).*

Dimostrazione. Posso facilmente dimostrare che $|C| \leq k$ in tempo polinomiale e che l'unione degli insieme di C include tutti gli elementi di U, in tempo polinomiale su $|U|$. Posso aggiungere anche un "certificato", nella forma di una collezione che copre tutto U (e quindi la verifica è in tempo polinomiale sulla dimensione del "certificato" più quella di U). Quindi **set-cover** è in **NP**.
Per dimostrare che **set-cover** è **NP-hard** dimostro che:

$$\text{vertex-cover} \leq_p \text{set-cover}$$

ovvero uso un problema che so già essere **NP-hard**, appunto *vertex-cover* (avendo già dimostrato che $\forall A \in NP,\ A \leq_p$ vertex-cover, dimostrando che *set-cover* dimostra tutti i problemi in **NP**).
Faccio vedere che posso usare *set-cover* per dimostrare *vertex-cover*.
Devo quindi trasformare un'istanza di *vertex-cover* $C = \langle G = (V, E), j\rangle$ in un'istanza C' di *set-cover*, in tempo polinomiale, tale che C è soddisfacibile sse C' è soddisfacibile.
Procedo quindi con la trasformazione. Pongo innanzitutto $U = E$. Per quanto riguarda la collezione S procedo nel seguente modo. Etichetto i vertici in V da 1 a n. A questo punto S_i diventa l'insieme degli archi incidenti al vertice i-simo. A questo punto basta porre $k = j$ per concludere la costruzione polinomiale dell'istanza di *set-cover*.
In poche parole ciascun arco è un elemento di U e ciascun vertice è un insieme di S.
Vediamo anche la dimostrazione formale che *vertex-cover* risponde "yes", per j, sse istanza di *set-cover* risponde "yes" per $k = j$.
Innanzitutto se *vertex-cover* risponde "yes" per j allora trovo una collezione di *set-cover* buona di cardinalità j. Suppongo infatti G ha una copertura

C di al più j vertici e quindi C corrisponde ad una collezione di C' di sottoinsiemi U. Poiché assumo $k = j$ allora $|C'| \leq k$. Inoltre C' copre tutti gli elementi di U coprendo tutti gli archi di G, in quanto ogni elemento di U è un arco in G. Poiché C è una copertura, almeno un estremo dell'arco è in C e quindi l'arco è in un insieme di C'.
Dimostriamo anche l'altro verso della dimostrazione, ovvero devo garantire l'esistenza della copertura. Suppongo di avere un set cover C' di dimensione k. Dato che ad ogni insieme di C' ho associato un vertice in G allora $|C| = |C'| \leq k = j$. Inoltre, C è una copertura di G poiché C' è un set-cover, poiché, preso un arco e ho che $e \in U$ e quindi C' deve contenere almeno un insieme che contiene e e tale insieme è quello che corrisponde ai nodi che sono estremi di e. Quindi C deve contenere almeno un estremo di e. Quindi posso concludere dicendo che C è copertura di G. □

Si continua la ricerca di una dimostrazioni di **NP-completezza**, ambito di studio nato dopo la scoperta di Cook di $SAT \in$ **NP-complete**.
Partiamo ricordando che:

Teorema 5. *Se B è un problema tale che $A \leq_p B$, con A **NP-hard** (o ovviamente anche **NP-complete**), allora B è **NP-hard** e se inoltre $B \in NP$ allora B è **NP-complete**.*

Dimostrazione. Con la transitività della riduzione $\leq_p$ si ha che $\forall \pi \in NP,\ \ \pi \leq_p A$ e quindi A è **NP-hard**. Inoltre avendo $A \leq_p B$ ho che $\pi \leq_p B$ e quindi B è **NP-hard**, inoltre, avendo $B \in NP$, ho che è **NP-complete** □

3.1.2 Problemi di ottimizzazione

Clique-problem

Definizione 19. *Definisco **clique** (**cricca**) di un grafo non orientato $G = (V, E)$ come un sottoinsieme $V' \subseteq V$ di vertici tale che:*

$$\forall\, v_1, v_2 \in V'(v_1, v_2) \in E$$

quindi un sottoinsieme di vertici con solo vertici collegati da un arco.

Definisco quindi il problema.

Definizione 20. *Il **clique-problem** è un problema di ottimizzazione (nel dettaglio di massimo) in cui si cerca la **clique** di dimensione massima di un grafo (ovvero $|V'|$ è massimo). Nella versione decisionale chiedo se esiste una **clique** di dimensione k.*
*Il problema è **NP-complete***

Dimostrazione. Dimostriamo che sia **NP-complete** (nella versione decisionale). Cerco quindi un algoritmo polinomiale, con"certificato". Uso quindi un insieme $V' \subseteq V$ come vertici della **clique** come "certificato" per il grafo in input. Per verificare che V' è una **clique** controllo che:

$$\forall\, v_1, v_2 \in V'(v_1, v_2) \in E$$

e questa verifica è polinomiale, infatti è $O(|V'|^2)$ e quindi è quadratico nella dimensione dell'input.
Bisogna ora trovare un problema $A \in$ NP-complete tale che A si riduce in tempo polinomiale al **clique-problem** (quindi **clique-problem** risolve A).
Notiamo che una **clique** è l'opposto di **independent-set**, che avevamo dimostrato tramite $3SAT$ (che può risolvere **independent-set**).
Quindi per **clique-problem** provo ancora ad usare $3SAT$. Devo fare in modo che $\phi \in 3SAT$ sse il grafo G_ϕ ha una **clique** di dimensione uguale al numero di clausole di ϕ. Quindi avendo k clausole in ϕ avrò che ciascuna clausola sarà un gadget e da ogni gadget estrarrà un vertice che comporrà la **clique** di dimensione k.
Il grafo G_ϕ è costruito come nel caso di *independent-set* coi letterali come vertici.
Bisogna studiare il collegamento tra tali vertici (che sarà diverso al caso di *independent-set*, non avendo quindi i triangoli).
Ipotizzo:

$$\phi = c_1 \wedge c_2 \wedge c_3$$

con:

$$c_1 = x_1 \vee \neg x_2 \vee \neg x_3$$

$$c_2 = \neg x_1 \vee x_2 \vee x_3$$

$$c_3 = x_1 \vee x_2 \vee x_3$$

Per ogni clausola faccio i vertici per ogni letterale (distinti anche per lo stesso letterale, come nel caso di *independent-set*).
Per ora abbiamo vertici isolati e i tre vertici isolati di ogni clausola sono il gadget. A questo punto collego ogni letterale di ogni clausola con ogni altro letterale di ogni altra clausola (non della stessa) che non sia l'opposto, collegando quindi solo vertici **consistenti** (in quanto non potrei avere una **clique** connettendo due vertici non consistenti). Sto facendo esattamente l'opposto di *independent-set*. Costruito il grafo G_ϕ cerco almeno una **clique** di dimensione 3 per $3SAT$. Non potrò mai avere più di un letterale per clausola nella **clique** non essendo tra loro collegati.
Passare da ϕ a G_ϕ ha costo polinomiale in tempo, ottenendo quindi istanza, in tempo polinomiale.

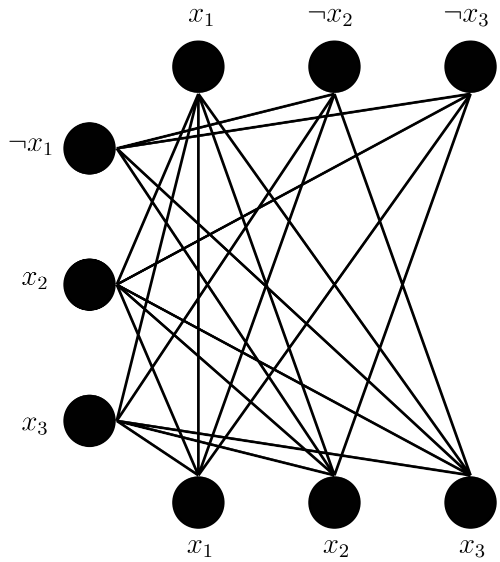


Figura 3.4: Grafo G_ϕ per **clique-problem**, con le due righe di nodi e la colonna che rappresentano i tre gadget

Inoltre bisogna dimostrare che se ϕ ha un assegnamento che la rende vera allora esiste una **clique** per G_ϕ di dimensione k. Se ϕ è vera allora per ogni clausola c_r esiste almeno un letterale che è vero e assumiamo che tale letterale sia $l_i^r = 1$. Questo letterale è associato al vertice v_i^r. Quindi data la clausola c_i, vera per il letterale l_j^i allora costruisco l'insieme dei vertici $V' \subseteq V$ tale che sia formato da quei singoli vertici corrispondenti ai singoli letterali veri. Questo V' è una **clique** infatti avrò solo letterali "collegabili" nel grafo (non essendo vero che un letterale sia vero e contemporaneamente falso, cosa che comporterebbe l'assenza dell'arco). Ho solo archi tra letterali consistenti tra loro e quindi, per costruzione, esiste l'arco tra i vertici corrispondenti $(\forall\, u, v \in V')$.
Inoltre bisogna dimostrare che, se esiste una **clique** di dimensione k, allora ϕ è vera e quindi le K clausole sono tutte vere.
Per ogni vertice di $v_{it} \in V'$ se appartiene al gadget della clausola c_t rendo vero il letterale associato (o falso se esso è negato). A questo punto rendo vera ogni clausola rendendo vero almeno un suo letterale e quindi so di ottenere un assegnamento di verità per ϕ in quanto i letterali, per come sono stati definiti, sono consistenti.
sistemare formalismi □

Dimostrazione. Dimostriamo che la riduzione del problema *clique*, che sap-

piamo essere **NP-complete**, è in tempo polinomiale.
Vediamo che la trasformazione della formula ϕ di $3SAT$ nel grafo G_ϕ è una riduzione in tempo polinomiale.
Dato un assegnamento che rende vera ϕ, che ha k clausole, dobbiamo costruire una *clique* per G_ϕ che abbia k vertici. Ciascuna clausola c_r ha almeno un letterale l_i^r che ha assegnato il valore 1, $\top$. Ciascun letterale l_i^r corrisponde ad un vertice v_i^r. Costruiamo V' insieme di vertici per ogni letterale l_i^r vero per ogni clausola c_r. Dimostriamo che V' è una *clique*. Si ha che per ogni coppia $v_i^r, v_j^s \in V'$, $r \neq s$ (due vertici rappresentanti due letterali di due clausole diverse) abbiamo che $(v_i^r, v_j^s) \in E$ essendo entrambi i letterali veri e,per costruzione, sono consistenti, cioè uno non è l'opposto dell'altro.
Vediamo l'altro verso.
Supponiamo che G_ϕ abbia una *clique* V' di dimensione k e bisogna vedere se ϕ è vera. Sappiamo che V', per costruzione ha esattamente un solo vertice per ogni clausola, v_i^r per la clausola c_r. Prendo quindi il letterale corrispondente l_i^r e assegno a tale letterale il valore 1 (se negato 0). Quindi so che il letterale complemento di l_i^r non verrà usato per rendere vera una clausola perché il vertice v_i^r non è collegato con un arco ad un vertice che rappresenta il c0complemento di l_i^r. Quindi riesco a costruire un assegnamento di verità *consistente* (ovvero ogni variabile è presa come $x_i = 1$ oppure $x_i = 0$ ma non entrambi i valori sono usati) per ϕ. $\square$

Si può dimostrare che:

$$clique \leq_p vertex\_cover$$

$$independent\_set \leq_p vertex\_cover$$

$$vertex\_cover \leq_p independent\_set$$

$$vertex\_cover \leq_p clique$$

Infatti vale il seguente teorema:

Teorema 6. *Se ho che:*

$$A \leq_p B$$

e $A, B \in$ NP-complete, allora:

$$B \leq_p A$$

Dimostrazione. Infatti, siccome B è in **NP-complete** allora è anche in **NP**, allora:

$$B \leq_p A$$

Ma posso fare lo stesso discorso con A, essendo in **NP-complete** e quindi sono interscambiabili e quindi:

$$A \leq_p B$$

$\square$

I problemi **NP-complete** richiedono quindi una soluzione efficiente. Bisogna quindi pensare ad algoritmi di approssimazione, algoritmi polinomiali con "garanzia", ovvero **euristiche** (ovvero un qualcosa che funziona in pratica) per il problema. Non si ha quindi una soluzione esatta al problema, l'euristica non fornisce una soluzione esatta.
L'obiettivo è quindi quello di risolvere i problemi **NP-complete**.

Definizione 21. *Definiamo un **problema di ottimizzazione**, dato un problema* Π *e un'istanza* x*, come la ricerca di un ottimo di* x*, detto* $opt(x)$*. Il costo di una soluzione ammissibile su* x *calcolato da un algoritmo* A *per il problema* Π *è indicato con* $A(x)$ *(che quindi è il costo della soluzione calcolato da* A*).*

Si cerca quindi la soluzione, tra tutte quelle di Π*, che massimizza o minimizza una funzione costo (appunto* $opt(x)$*). Non sappiamo chi calcoli questo costo.*

Definizione 22. *Definiamo un **algoritmo** ε**-approssimato** per un problema* Π *è un algoritmo* A *polinomiale che restituisce una soluzione ammissibile che dista da quella ottima di un fattore* ε*.*
Se Π *è un **problema di minimo** allora:*

$$A(x) \leq \varepsilon \cdot opt(x), \text{ con } \varepsilon > 1$$

Si raggiunge quindi un valore più grande del minimo, di una quantità fissata ε*. Devo quindi garantire che non sia troppo grande, tramite varepsilon.*
Si ha quindi che:

$$\frac{A(x)}{opt(x)} \leq \varepsilon$$

L'algoritmo lavora in tempo polinomiale e risolve in modo approssimato problemi Π ***NP-completi**. Se avessi* $\varepsilon = 1$ *avrei la soluzione ottima, ma non sarebbe possibile in quanto avrei un algoritmo polinomiale per un algoritmo **NP-completo**.*
Se Π *è un **problema di massimo** allora:*

$$A(x) \geq \varepsilon \cdot opt(x), \text{ con } 0 < \varepsilon < 1$$

In quanto raggiungo una soluzione più piccola del massimo ma devo garantire che non sia troppo piccola, tramite varepsilon. $\varepsilon \cdot opt(x)$ è infatti una "frazione" dell'ottimo.
Si ha quindi che:

$$\frac{A(x)}{opt(x)} \geq \varepsilon \Longrightarrow \frac{opt(x)}{A(x)} \leq \frac{1}{\varepsilon}$$

A seconda del problema devo trovare un ε, che viene fissata costante per ogni input (e quindi è indipendente dall'input stesso e dalla sua dimensione) che serve da "garanzia". So quindi che $A(x)$, ovvero il costo della soluzione approssimata ammissibile calcolata da A, in tempo polinomiale, si rapporta a $opt(x)$, calcolata dall'algoritmo esatto in tempo esponenziale (se fosse polinomiale si avrebbe $P = NP$), tramite una distanza ε (da un alto è un $\limsup$ *e dall'altro un* $\liminf$*).*
Tipicamente si ha come valore tipico per varepsilon 2, avendo quindi una ***2-approssimazione****, in quanto facilmente dimostrabile.*

Esempio 9. *Prendiamo vertex-cover nella versione di ottimo. Dato $G = (V, E)$ ha come soluzione ammissibile una copertura di vertici per G. Il costo di una soluzione è la cardinalità di V', se V' è una soluzione ammissibile.*
Il costo ottimo di vertex-cover invece è la minima cardinalità di V'.

Definizione 23. *Si ha il cosiddetto* ***approximation ratio*** *r dicendo che:*

$$\max\left\{\frac{A(x)}{opt(x)}, \frac{opt(x)}{A(x)}\right\} \leq r, \text{ con } r > 1$$

Il primo caso copre il caso di minimo mentre il secondo caso copre il massimo.

Non tutti i problemi ammettono una *r*-approssimazione, per *r* costante.
Ci sono problemi infatti dove il calcolo di:

$$\max\left\{\frac{A(x)}{opt(x)}, \frac{opt(x)}{A(x)}\right\} \leq \rho(n)$$

per una certa funzione ρ,con $|x| = n$, ovvero il massimo dipende dalla dimensione dell'input, non avendo quindi più un r costante. La dimensione del problema influisce sulla capacità di approssimazione e degradano all'aumentare della dimensione. Si ha, per esempio, $\rho(n) = \log n$.
Per capire ε fornisco un'euristica A per il problema e provo a dimostrare, senza conoscere l'ottimo (quindi per ogni possibile input), che in ogni caso:

$$\frac{A(x)}{opt(x)} \leq \varepsilon$$

dimostrando che si ha una ε-approssimazione (non sempre è dimostrabile o perlomeno per alcuni problemi si è ancora riusciti).

Teorema 7. *Esiste A polinomiale per vertex-cover che è 2-approssimante, quindi:*

$$\frac{A(x)}{opt(x)} \leq 2$$

Dimostrazione. Prendo il grafo $G = (V, E)$.
Cerco un A che calcoli in tempo polinomiale una soluzione ammissibile, ovvero una copertura di vertici del grafo. Si sceglie che A non deve essere un *algoritmo greedy.*
Prendo quindi un arco del grafo $e = (u, v) \in E$. Inizio a costruire una copertura C, all'inizio vuota ($C = \emptyset$), che ora diventa, aggiungendo i due vertici:

$$C = C \cup \{u, v\}$$

Inoltre rimuovo dall'insieme degli archi E tutti gli archi che hanno un estremo nell'attuale copertura C, e quindi nel nostro caso in u o v.
Procedo quindi iterativamente costruendo C fermandomi quando E risulti vuoto, ovvero fino a che $E = \emptyset$.
È quindi una 2-approssimazione in quanto al massimo posso prendere il doppio dei vertici per fare la copertura, infatti, per costruzione, seleziono ogni volta archi che non hanno estremi in comune. Per ognuno di questi archi scelti devo prendere i due estremi, quindi ho $2 \cdot k$ vertici in C per k archi e quindi:

$$A(x) = 2 \cdot k$$

Ma di $opt(x)$ so che nel casi migliore prende esattamente un estremo per ogni scelto dall'algoritmo, e quindi per archi che non hanno estremi in comune. Quindi nel caso migliore:

$$opt(x) \geq k$$

Dato che stiamo cercando di minimizza, quindi k è il *lower bound* e quindi:

$$\frac{A(x)}{opt(x)} \leq \frac{2k}{k} \leq 2$$

Come volevasi dimostrare. □

Algorithm 1 Algoritmo di vertex-cover approssimato

```
function VCApprox(G = (V, E))
    C ← ∅
    E′ ← E
    while E′ ≠ ∅ do
        let (u, v) ∈ E′
        C ← C ∪ {u, v}
        for every (u, z) ∈ E′ ∧ every (v, z′) ∈ E′ do
            delete (u, z), (v, z′) from E′
    return C
```

Capitolo 4

Macchina di Turing

4.1 Problemi intrattabili

Trovare una soluzione polinomiale ad un algoritmo **NP-hard** come TSP comporterebbe la rottura di tutte le chiavi crittografiche in quanto trovato un algoritmo polinomiale per uno lo trovi per tutti.
Ci serve a questo punto una definizione più rigorosa di **algoritmo**, per poterne calcolare meglio i tempi. Ricordiamo che per assunzione un algoritmo è **efficiente** se è in tempo polinomiale rispetto alla dimensione dell'input x, sapendo che $|x| = n$ (solitamente al più si arriva a $O(n^5)$ o poco più poi si salta ad algoritmi esponenziali). Un algoritmo esponenziale è un algoritmo **non efficiente**, il tempo cresce troppo velocemente all'aumentare dell'input (anche se magari in alcuni casi non è in tempo esponenziale). Si ricorda che si studia sempre il tempo nel **caso peggiore**, prenderemo quindi sempre l'O-grande sulla dimensione dell'input $O(f(x))$.
Vediamo degli esempi:

- un algoritmo che cerca l'arco minimo lavora in tempo polinomiale nel caso peggiore ed è quindi un **problema trattabile**
- problemi, come il test di primalità o TSP, che non hanno un algoritmo polinomiale sono **intrattabili**, infatti nessuno ha mai dimostrato che esiste un algoritmo efficiente

Tra i problemi intrattabili abbiamo però problemi che sono **dimostrabilmente intrattabili**. Banalmente un problema che mi chiede di stampare tutte le possibili sequenze per una certa proprietà è in questa categoria, dovendo stampare tutte le sequenze possibili si ha $O(2^n)$ e si può dimostrare che con meno operazioni non si stamperebbero alcune soluzioni corrette.

Ci concentreremo su problemi intrattabili ma non *dimostrabilmente intrattabili.*
Per anni problemi come il *test di primalità* potevano garantire che prima o poi si sarebbe trovato un algoritmo polinomiale (forse). Quindi abbiamo:

- problemi dimostrabilmente intrattabili
- problemi non dimostrabilmente intrattabili, sono, diciamo, "i più difficili" tra i problemi intrattabili
- tutti gli altri problemi

Un problema indecidibile non è un problema intrattabile, in quanto non si hanno proprio algoritmi che risolvono un certo problema e questo è dimostrabile (c'è almeno un input che manda in crisi un algoritmo, che va in loop infinito o sbaglia risposta) mentre un problema intrattabile comunque in qualche modo lo posso risolvere ma in tempi troppo elevati.

4.2 Definizione della TM

Definiamo quindi in modo più rigoroso il concetto di algoritmo per poter dimostrare che un certo algoritmo può anche non esistere. Non ci basta più la definizione di algoritmo come sequenza di passi logici, in quanto andrebbe anche definito un certo linguaggio (con scelte, cicli, operazioni aritmetiche, operazioni logiche) ma ancora non basterebbe, non si è ancora sicuri di poter trasformare un certo input in un certo output (per qualunque problema in input). Lo step mancante è la **macchina di Turing**, con essa si può garantire quanto appena detto, con essa si formalizza il processo di calcolo, ovvero la serie di passaggi che porta da un input ad un output. Turing ragionò dicendo che normalmente si risolve un problema partendo da carta e penna, ponendosi poi in un certo stato mentale in cui si risolve o si studia una parte dell'input (ad esempio in uno stato *leggi tutti* leggo "step by step" tutto l'input, senza cambiare sto mentale, che verrà cambiato quando finisco quello precedente). Divide quindi l'input in *caselle* su cui si fanno operazioni semplici, spostandosi a destra o a sinistra di una casella, o leggendo/scrivendo la casella corrente. Turing ipotizza di avere carta illimitata. Quindi la MT avrà un nastro infinito che permette di memorizzare informazioni e si ha una testina di lettura e scrittura. Si ha un meccanismo che si pone in uno *stato*, sulla base del contenuto letto dalla testina e dallo stato posso scegliere di spostarmi di una testina a destra o di una a sinistra, eventualmente dopo avere scritto e modificando, sempre eventualmente, lo stato. Tutto questo basta per il **calcolo**, infatti:

$\cdots$		$\triangleright$	b	b	a	a	a	a	$\sqcup$	$\cdots$	**Input/Output Tape**

Figura 4.1: Esempio di nastro di una TM

Teorema 8 (Tesi di Turing-Church)**.** *Non esiste nessun formalismo di calcolo che sia più potente della Macchina di Turing:*

"Se un problema è umanamente calcolabile, allora esisterà una macchina di Turing in grado di risolverlo (cioè di calcolarlo)"

È una tesi che non ha dimostrazione formale (non avendo chiara la definizione di ***calcolo****) ma è stata dimostrata empiricamente nel corso degli anni. Portando quindi a dire che il calcolo è ciò che può essere eseguito con un Macchina di Turing (anche se non tutti i meccanismi di calcolo sono equivalenti ad una TM, ad esempio gli automi a stati finiti).*
Quindi ciò che è computabile è computabile da una TM o da un suo equivalente (come un linguaggio di programmazione). Banalmente anche una rete neurale lo è.

Formalizziamo quindi la macchina di Turing.

Definizione 24. *Si definisce formalmente una TM come la quintupla:*

$$TM = (K, \Sigma, k_0, \delta, F)$$

- *insieme K di stati*
- *un alfabeto Σ*
- *uno stato di partenza k_0*
- *una funzione di transizione δ*
- *un insieme F di stati finali*

Si hanno inoltre i seguenti stati finali:

- *H, per l'halt*
- *Y, per lo yes*
- *N, per il no*

Il simbolo $\sqcup$ specifica che non ho un simbolo e il simbolo $\triangleright$ mi specifica che da lì parte l'input.

Definizione 25. *La funzione di transizione esprime cosa fa passo-passo la TM:*

$$\delta : K \times \Sigma \to K \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

Ovvero prende in input uno stato e un simbolo e può avere in input un cambio di stato oppure il cambio del simbolo in quel punto o lo spostamento della testina di una posizione (che può comunque restare ferma).
Si possono avere diverse varianti di implementazioni, obbligando al testina a spostarsi (andando avanti e indietro per indicare che deve stare ferma etc...).
Vedremo poi che avere questa funzione di transazione comporta l'avere una ***TM deterministica*** *e che si potrà sviluppare una* ***TM non deterministica****.*

Ogni operazione sulla TM ha lo stesso tempo e quindi posso usare il numero di passi per calcolare il tempo di risoluzione.
Per esprimere la computazione di una TM usiamo una **configurazione**, ovvero sulla base della definizione della TM e dello stato attuale devo definire tutti i passi.

Definizione 26. *Un* ***configurazione*** *di una TM è definita da:*

- *lo stato in cui si trova*
- *la stringa sul nastro (definita da tutti i simboli a destra della testina e tutti quelli a sinistra, ai quali viene aggiunto quello sotto la testina)*
- *la posizione delle testina*

In base alla configurazione la TM saprà come procedere.
La configurazione descrive in ogni istante lo stato della macchina e quindi si ha la seguente ***configurazione iniziale****, per la stringa X (che ha un carattere per posizione del nastro, al quale comunque viene aggiunto lo start triangleright):*

$$(k_0, \triangleright X, 1)$$

e ad un certo punto sia arriverà ad uno stato di arresto, per esempio dopo aver cambiato X in Y ed essere tornata nello stato 2:

$$(H, \triangleright Y, 2)$$

e quindi output sarà la stringa Y (senza il $\triangleright$).
Potrei avere anche Y o N al posto di H in problemi decisionali, per capire magari se una certa stringa ha le caratteristiche desiderate o meno.

Vediamo alcuni esempi di costruzione di TM:

Esempio 10. *Si scriva la TM che calcoli il successore di un numero binario, che sarà l'input (e si da per scontato che sia correttamente formattato avendo solo 0 o 1 come simboli). Si trascuri il riporto (nel senso che non aggiungo ulteriori bit).*

Vediamo nella pratica un esempio di somma binaria:
prendo 01101010 *e sommo 1:*

$$10010101+$$

$$00000001=$$

$$10010110$$

Definisco quindi la TM:

- $S = \{s_0, s_1\}$
- $\Sigma = \{\triangleright, \sqcup, 0, 1\}$
- *per la funzione di transizione si ha:*

$$\delta \to (s_0, [\triangleright, 0, 1]) \to (s_0, [\triangleright, 0, 1], \to)$$

$$\delta \to (s_0, \sqcup) \to (s_1, \sqcup, \leftarrow)$$

$$\delta \to (s_0, 0) \to (H, 1, -)$$

$$\delta \to (s_0, 1) \to (s_1, 0, \leftarrow)$$

$$\delta \to (s_1, \triangleright) \to (H, \triangleright, -)$$

  ovvero scorro fino alla fine e inverto l'ultimo numero (se è 0 diventa 1 e fine ma se è 1 lo rendo 0 e poi mi sposto a sinistra e se è un 1 diventa 0 e così via, fino alla fine dove metto 1, come prevede la somma binaria)

- s_0 *è lo stato iniziale*

Capitolo 5

Pattern matching

Il **pattern matching** si occupa di ricercare un pattern P in un testo T. Si hanno:

- ricerca esatta
- ricerca approssimata

Definizione 27. *Definiamo stringa come una sequenza di simboli appartenenti ad un dato alfabeto* Σ*, scritta come:*

$$X = x_1, \ldots, x_n, \ \ \forall x_i \in \Sigma$$

ovviamente non è necessario che la stringa contenga tutti i simboli dell'alfabeto. Diamo alcune definizioni utili:

- $|X|$ *indica la lunghezza della stringa*
- ε *indica la stringa vuota*
- $X[i]$ *indica il carattere all'indice i (partendo da 1 e non da 0)*
- $X[i, j]n$ *indica la sottostringa che parte dall'indice i e arriva all'indice j (estremi inclusi). Inoltre se si hanno* $i \neq j$ *e* $j \neq |X|$ *si ha una **sottostringa propria**.*
  (Esempio: per $X = bbaccbbaac$ *ho la sottostringa* $X[4, 8] = ccbba$*)*
- $X[1, j]$ *indico un **prefisso** (estremo finale incluso). Si ha inoltre il **prefisso proprio** se* $j \neq |X|$ *e si ha **prefisso nullo** se* $j = 0$ *e si indica con* ε*.*
  (Esempio: per $X = bbaccbbaac$ *ho il prefisso* $X[1, 4] = bbac$*)*

- $X[i, |X|]$ *indica un* ***suffisso*** *della stringa (estremi inclusi), di lunghezza* $k = |X| - i + 1$*, avendo quindi il suffisso* $X[|X| - k + 1, |X|]$*. Si ha inoltre che se ho* $X[|X|, |X|+1]$ *allora ho il* ***suffisso nullo****, ovvero* ε*.*
  (Esempio: per $X = bbaccbbaac$ *ho il suffisso* $X[8, 10] = aac$ *e il suffisso* $X[11, 10] = \varepsilon$*)*

Definizione 28. *Definiamo il* ***pattern matching esatto*** *come la ricerca di tutte le occorrenze esatte di un pattern* P *in un testo* T*.*
P *occorre esattamente in* T*, a partire dall'indice* i *in* T*, sse:*

$$T[i, i + |P| - 1] \text{ coincide con } P$$

Quindi in output ho tutte le posizione i *nel testo tali per cui:*

$$T[i, i + m - 1] \text{ coincide con } P$$

Esempio 11. *Dato il testo* $T = bbaccbbaac$ *e il pattern* $P = ccbb$ *ho un'occorrenza esatta a partire da* $i = 4$ *nel testo.*

L'algoritmo naive sarebbe quello di prendere una finestra W, di lunghezza P, da far scorrere sul testo (avanzando ogni volta di un carattere). Ogni volta tale finestra viene confrontata con P confrontando tutti i caratteri e, in caso di match, stampando in output l'indice di inizio della finestra sul testo. Questo algoritmo è $O(|T| \cdot |P|)$ nel caso peggiore.
Questo algoritmo ignora le caratteristiche di testo e pattern, ignorando aspetti che potrebbero accelerare la ricerca, trovabili tramite un'operazione di preprocessamento.

Definizione 29. *Definiamo* ***distanza di edit*** *come il minimo numero di operazioni di sostituzione, cancellazione e inserimento di un unico simbolo per trasformare una stringa in un'altra (o viceversa).*
La distanza di edit tra due stringhe è simmetrica (anche se le operazioni saranno diverse il loro numero minimo sarà uguale).

Esempio 12. *Calcoliamo la distanza di edit tra* $X_1 = agtgcgt$ *e* $X_2 = atgtgat$*.*
Partiamo cancellando la g in indice 2 di X_1*:*

$$X_1 = atgcgt$$

$$X_2 = atgtgat$$

Inserisco quindi una "a" al penultimo indice di X_1*:*

$$X_1 = atgcgat$$

$$X_2 = atgtgat$$

Infine sostituisco la "c" con una "t" all'indice 4 di X_1:

$$X_1 = atgtgat$$

$$X_2 = atgtgat$$

Ho quindi una distanza di edit pari a 3.
Per trasformare la seconda nella prima avrei dovuto inserire una "g" all'indice 2 di X_2, cancellare la "a" al penultimo indice di X_2 e infine sostituire la "t" con una "c" al terzultimo indice di X_2. Si vede quindi come anche in questo caso avrei distanza di edit pari a 3.

Definizione 30. *Definiamo il **pattern matching approssimato** come la ricerca di occorrenze approssimate di un pattern P in un testo T, con al più un certo errore k.*
Viene usata la distanza di edit*, che rappresenta l'errore che si commette nell'eguagliare una stringa con un altra.*
P ha un'occorrenza approssimata in T, che finisce all'indice i, se esiste almeno una sottostringa $T[i - L + 1, i]$ che ha distanza di edit con P di al più k. L è un certo valore non prevedibile in quanto sto ammettendo che il match sia su una stringa di lunghezza diversa da quella del pattern. Ovviamente in questo caso si possono avere più sottostringhe accettabili secondo i parametri richiesti.
In output ho quindi tutte le posizioni i di T in cui finisce almeno una sottostringa che ha con P distanza di edit al più uguale a k.

Esempio 13. *Prendiamo il testo $T = bbaccbbac$ e $P = ccbb$.*
Impongo $k = 1$ e vedo che un match è "ccbb" terminante all'indice 6 del testo (avendo errore 1 massimo). Ovviamente "ccbb" terminante all'indice 7 va bene avendo errore nullo.

Definizione 31. *Definiamo **bordo** della stringa X, $B(X)$, che è il più lungo prefisso proprio che occorre come suffisso di X.*

Esempio 14. *Vediamo qualche esempio:*

$$X = baacagahabaac \implies B(X) = baac$$

$$X = abababa \implies B(X) = ababa$$

$$X = aaaaaa \implies B(X) = aaaaa$$

$$X = aaaccbbaac \implies B(X) = \varepsilon$$

$$X = aaaaaaaa \implies B(X) = aaaaaaa$$

Ovviamente può essere ε (in primis se ho una stringa di lunghezza 1).

Definizione 32. *Definiamo concatenazione tra una stringa X e un simbolo σ come: $X\sigma$*

Esempio 15. *Dato $X = bba$ e $\sigma = d$ ho:*

$$X\sigma = bbad$$

5.1 Pattern matching con automi

Siamo nell'ambito del pattern matching esatto.
Ho un pattern di lunghezza m e un testo di lunghezza n.
Abbiamo una fase di preprocessamento in tempo $O(m|\Sigma|)$ per calcolare la funzione di transizione δ.
Definiamo δ definita sul pattern P come:

$$\delta : \{0, 1, \dots, m\} \times \Sigma \to \{0, \dots, m\}$$

Quindi ad ogni coppia tra un simbolo e un intero tra 0 e m corrisponde un intero tra 0 e m. Nel dettaglio si hanno due casi:

1. primo caso:
$$\delta(j, \sigma) = j + 1 \iff j < m \land P[j + 1] = \sigma$$
2. secondo caso:
$$\delta(j, \sigma) = k \iff P[j + 1] \neq \sigma \lor j = m, \text{ con } k = |B(P[1, j]\sigma)|$$
Con k che è la lunghezza del bordo tra il prefisso corrente a cui viene concatenato il carattere σ. Si ha che $k \leq j$ per definizione (dato che sto calcolando il bordo su una stringa di lunghezza $j+1$). Con questa transizione posso, eventualmente, andare "indietro" nel pattern ma comunque "in avanti" nell'automa.

La transizione dallo stato j allo stato $\delta(j, \sigma)$ avviene attraverso il simbolo σ.
Posso quindi passare allo stato $j + 1$ o allo stato k.
Nell'automa avremo quindi gli archi etichettati coi simboli e i nodi etichettati con gli indici da 0 a m.
Nella stringa gli indici partono comunque da 1 quindi l'indice 0 nelle formule rappresenta una sorta di posizione prima dell'inizio della stringa. Qualora k sia pari a 0 per la seconda formula mi sposto effettivamente in questo indice posto prima della stringa (anche se avrò due stati diversi etichettati come 0 nell'automa).

Esempio 16. *Prendiamo il pattern:*

$$P = acacbabbaabac$$

calcoliamo:

- $\delta(6, b) = 7$ *in quanto ho effettivamente* b *come simbolo all'indice successivo a* 6*, uso quindi la prima formula*

$$6 \xrightarrow{b} 7$$

- $\delta(0, a) = 1$*, in quanto* a *è il primo carattere, uso quindi la prima formula*

$$0 \xrightarrow{a} 1$$

- $\delta(6, c) = 2$ *in quanto devo usare la seconda formula e avere:*

$$|B(P[1, 6]c)| = |B(acacbac)| = |ac| = 2$$

$$6 \xrightarrow{c} 2$$

- $\delta(0, c) = 0$ *in quanto devo usare la seconda formula e avere:*

$$|B(P[1, 0]c)| = |B(c)| = |\varepsilon| = 0$$

$$0 \xrightarrow{c} 0$$

- $\delta(13, a) = 4$ *in quanto devo usare la seconda formula e avere:*

$$|B(P[1, 14]a)| = |B(acacbabbaabaca)| = |aca| = 3$$

$$13 \xrightarrow{a} 3$$

Facciamo due osservazioni:

1. dallo stato 0 si arriva allo stato 0 per qualsiasi simbolo $\sigma \neq P[1]$ e quindi si arriva allo stato 1 attraverso $\sigma = P[1]$
2. dallo stato m si arriva sempre ad uno stato $k \leq m$ e da uno stato m si può arrivare di nuovo ad uno stato m

Esempio 17. *Se ho $P = aaaaaa$:*

$$\delta(6, a) = 6$$

in quanto:

$$l = |B(aaaaaaa)| = |aaaaaa| = 6$$

Esempio 18. *Sia dato un pattern $P = acacbac$, con $m = 7$ e $\sigma = \{a, b, c\}$. Si hanno le seguenti transizioni $\delta(j, \sigma)$, calcolate solo sulla base delle formule. Parto con $\delta(j, \sigma) = j + 1$:*

$j \backslash \sigma$	a	b	c
0	1		
1			2
2	3		
3			4
4		5	
5	6		
6			7
7			

Proseguo con $\delta(j, \sigma) = k$:

$j \backslash \sigma$	a	b	c
0	1	0	0
1	1	0	2
2	3	0	0
3	1	0	4
4	3	5	0
5	6	0	0
6	1	0	7
7	3	0	0

In realtà algoritmo procede per induzione di riga in riga, riempendo la riga basandosi con quella precedente, in $O(m|\Sigma|)$.

Esempio 19. *Sia dato un pattern $P = acacbac$, con $m = 7$ e $\sigma = \{a, b, c, d\}$. Aggiungo quindi un simbolo non appartenente al pattern, si ottiene:*

$j \backslash \sigma$	a	b	c	d
0	1	0	0	0
1	1	0	2	0
2	3	0	0	0
3	1	0	4	0
4	3	5	0	0
5	6	0	0	0
6	1	0	7	0
7	3	0	0	0

Si aggiunge quindi una colonna di soli 0