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.

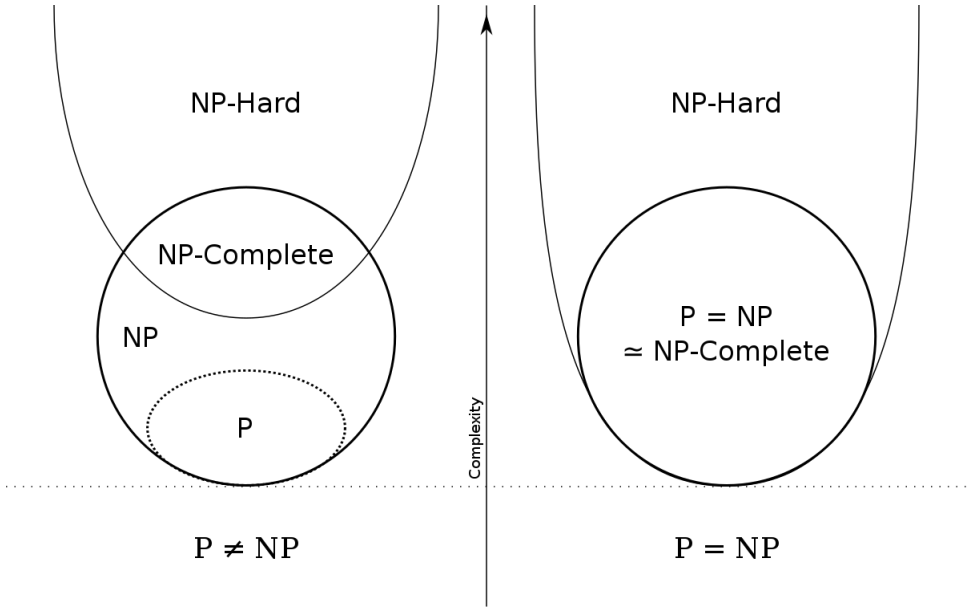


Figura 3.1: Diagramma di *Eulero Venn* per le classi di complessità

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**)*

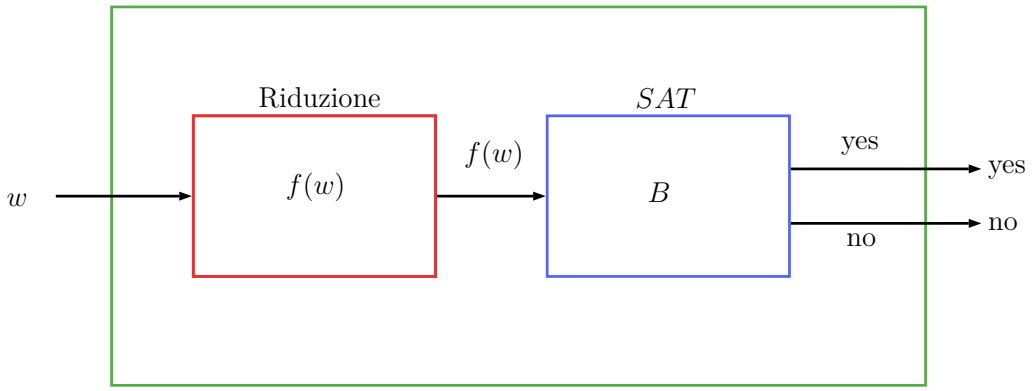


Figura 3.2: Rappresentazione grafica della riduzione

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**.*

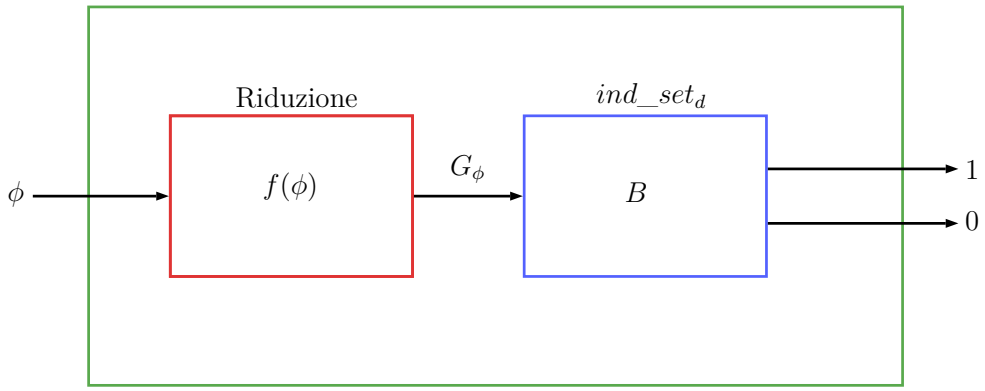


Figura 3.3: Rappresentazione grafica del'esempio 7

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:*

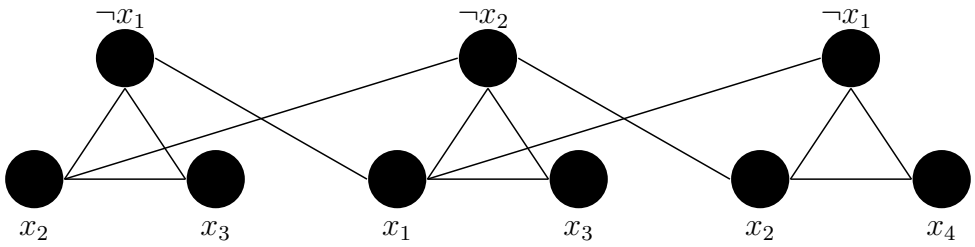
$$3SAT \leq_p ind\_set_d$$

*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 $S = \{S_1, \ldots, S_M\}$ 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.

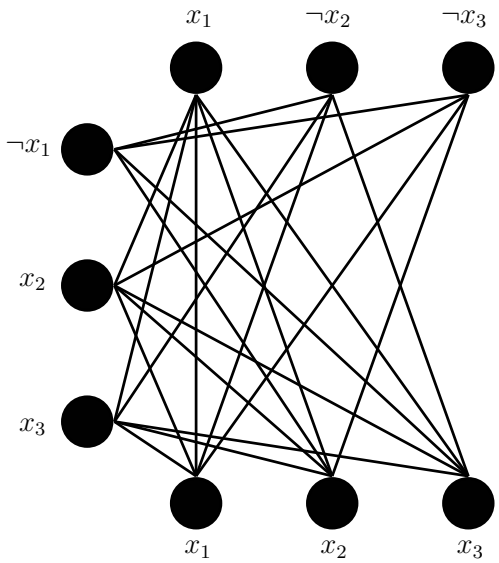


Figura 3.4: Grafo G_ϕ per **clique-problem**, con le due righe di nodi e la colonna che rappresentano i tre gadget

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. □

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:

$$A \leq_p B$$

□

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
```

L'idea della 2-approssimazione è quella che si cerca un limite inferiore all'ottimo per il problema, ad esempio, appunto, *vertex-cover*. Si ha che:

$$opt(x) \geq k$$

con k che è il lim inf mentre $opt(x)$ è la dimensione dell'ottimo su istanza X, nel caso di *vertex-cover* ho $x = G = (V, E)$. Sviluppo quindi un algoritmo polinomiale che produce una soluzione ammissibile per il problema e tale che abbia costo di tale soluzione sia ε volte il limite inferiore dell'ottimo, con $\varepsilon > 1$:

$$A(x) = \varepsilon \cdot \liminf, \;\; \varepsilon > 1$$

Inoltre ho che:

$$A(x) \leq \varepsilon \cdot k, \;\; \varepsilon > 1$$

e quindi:

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

avendo quindi una ε-approssimazione.
Nel caso di *vertex-cover* costruisco un **matching perfetto**, selezionando archi del grafo che non condividono i vertici estremi. Utilizzo quindi tutti i vertici, per il *matching perfetto*. Una copertura minima del grafo deve per forza ottenere esattamente un vertice per ogni arco di matching, perché non hanno estremi in comune. Quindi il limite inferiore dell'ottimo è il numero di archi del matching (visto che ho un vertice per arco):

$$opt(x) \geq |E_{matching}|$$

Posso usare una tecnica greedy per costruire il matching, mettendo nelle soluzioni ammissibili entrambi gli estremi di ogni arco, per l'ammissibilità, per poi fare la rimozione degli archi incidenti.

L'algoritmo A quindi costruisce in modo greedy (o meglio "in modo incrementale", incrementantalmente) un matching per G così:
prende un arco e_i e rimuove tutti gli archi incidenti agli estremi di e_i inserendo nella soluzione ammissibile entrambi gli estremi di e_i. Sicuramente l'arco successivo scritto da A non condivide estremi con e_i.
L'unico problema è che $A(x)$ ha dimensione doppia rispetto al numero di archi:

$$A(x) = 2 \cdot |E_{matching}|$$

del matching mentre:

$$opt(x) \geq |E_{matching}|$$

avendo quindi la 2-approssimazione.

3.1.3 Problemi NP-hard non NP

Vediamo un algoritmo **NP-hard** che non è in **NP**, o meglio che ancora non sia dimostrato esserlo. Si congettura che non sia possibile che tutti i problemi **NP-hard** siano nella classe **NP**.
Si parla comunque di problemi di decisione decidibili.
Non posso al momento attuale dimostrare che un problema **NP-hard** non sia in **NP** ma si hanno problemi per cui non si riesce a mostrare che siano in **NP**, ovvero nessuno è riuscito a costruire un algoritmo con "certificato" per questi problemi. Si ricorda la figura 3.1.

Quantified boolean formula problem

Un esempio è decidere se una formula ϕ con *quantificatori* è vera. Per questo problema **NP-hard** non esiste un algoritmo con certificato, non ancora perlomeno.
Una **Quantified Boolean Formula (*QBF*)** ϕ è una formula proposizionale con dei *quantificatori*, ovvero della forma:

$$\phi = Q_{1p_1} Q_{2p_2} \dots Q_{1n_n}$$

Con un quantificatore Q_{ip_i} relativi alle proposizioni p_i.
Si ha:

$$Q_i \in \{\forall, \exists\}$$

Esempio 10. *Vediamo un esempio di QBF:*

$$\forall\, p_1 \exists\, p_2\, \exists\, p_3 (p_1 \to (p_1 \wedge p_3))$$

Decidere se la formula ϕ è vera è un problema **PSPACE-complete**, ovvero un problema che si risolve con spazio polinomiale rispetto all'input x, ma al momento attuale nessuno ha mostrato che il problema QBF appartiene a **NP**.
Diversi problemi possono essere espressi in QBF (come ad esempio le mosse degli scacchi) ma questi per ora sono tutti in tempo esponenziale.
Aumentando il numero di quantificatori mi sposto in una nuova gerarchia, in un contesto di gerarchie infinite dove ad ogni aumento di un quantificatore corrisponde una classe di complessità nuova che contiene tutte quelle precedenti.
Noi sappiamo che:

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME \subseteq NEXPTIME$$

La classe PSPACE è la classe di tutti i problemi accettati da un Turing Machine in spazio polinomiale. La classe**EXPTIME** ha tempo esponenziale e **NEXPTIME** se il problema è risolvibile da una Turing Machine non deterministica.

Definizione 24. *Un problema* Π *è* ***complete*** *per una classe di complessità* k *sse* $\Pi \in k$ *ed è* ***difficile*** *per quella classe.*
SI hanno:

- ***NP-complete***
- ***NPSPACE-complete***

3.2 Complessità parametrica

La **complessità parametrica** è stata introdotta negli anni novanta e viene studiata molto in ambito di **bioinformatica** e studio di **reti/grafi**.
Un problema **NP-complete** posso dire che alcune istanze sembrano risolvibili in tempo polinomiale, fissando determinati parametri che caratterizzano l'istanza. Ad esempio prendo *vertex-cover* con al più una certa dimensione k in un grafo comunque molto complesso, magari con 10^8 nodi e k fissato a un numero piccolo, tipo 5. Non cerco quindi una minima copertura ma una di una certa dimensione, piccola. In questo caso, con k piccolo, un tempo esponenziale in k è accettabile (esempio 2^5 è accettabile).
Mi serve quindi un algoritmo polinomiale sull'input ma esponenziale su k.

Definizione 25. *Un problema decidibile* Π *è* ***trattabile fissato il parametro, Fixed Parameter Tractable (FPT)****, con parametro* k*, sse esiste*

un algoritmo A, una costante c è una funzione computabile f tale che per tutti gli input $\langle x, k\rangle$ A risolve Π con tempo di calcolo:

$$T(n) = f(k) \cdot |x|^c = f(k) \cdot n^c$$

Si arriva quindi ad un $O(f(k) \cdot n^c)$ con f esponenziale in k fissato.
L'input diventa quindi (x, k).

Quindi ho comunque un tempo polinomiale in x ed esponenziale in k, $|x|^c \to 2^k$. Divido quindi l'input in due dimensioni ed esprimo il tempo evidenziando i due parametri.
Se avessi un algoritmo A, con input x, per un problema **NP-complete** avrei $2^{|x|} = 2^n$ ma avendo in input $\langle x, k\rangle$ diventa polinomiale in $|x| = n$ ma esponenziale in k. Nella risposta quindi considero una parte limitata dell'input, per questo serve k piccolo.
Fissato k ho una **soluzione esatta** e quindi gli algoritmi parametrici sono vantaggiosi rispetto agli algoritmi di approssimazione.
Non tutti i problemi possono avere un problema parametrico associato.
Queste tecniche vengono usate per trattare problemi **NP-hard**, considerando problemi di ottimizzazione e non di decisione.

3.2.1 Vertex-cover parametrico

Prendendo *vertex-cover* **decisionale** cerco una copertura di dimensione massima k. Ho quindi in input $\langle G = (V, E), k\rangle$ e trovo un algoritmo che decide se esiste tale copertura minima in tempo $T(n)$, in funzione di n e k.
Si ha il seguente teorema:

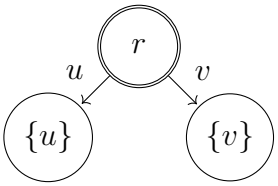
Teorema 8. *Il problema vertex-cover (G, k) è risolvibile in tempo $O(2^k \cdot |V|)$ dove V è l'insieme dei vertici di G, con la costante che non dipende da k e da $|V|$. Si ha quindi:*

$$O^*(f(x)) = O(2^k \cdot |V|)$$

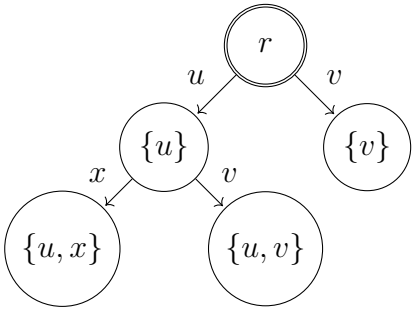
Avendo O^* che specifica indipendenza da k e da $|V|$.

Si vuole quindi $s^{|V|} = 2^k$.
Si parte quindi cercando di costruire insiemi con al più k vertici esplorando un albero binario di ricerca, sviluppandolo fino a profondità 2^k, che avrà come foglie degli insiemi *vertex-cover* di dimensione k.
Si quindi ricorre ai **Bounded search trees (*alberi di ricerca limitati*)**.
Si costruisce quindi un albero binario di profondità k, con il numero di nodi che quindi cresce circa per $O(2^k)$ e che sono una copertura candidata.
Si prende la radice r dell'albero e dato l'arco $(u, v) \in E$ ho due figli di r,

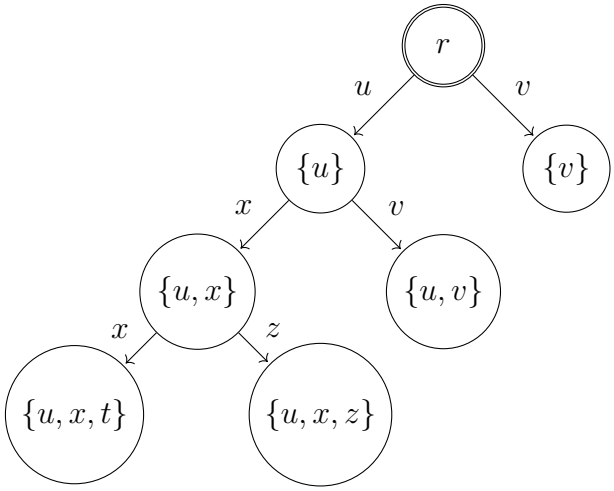
uno con u e l'altro con v, costruisco quindi due figli a partire dalla radice. Questo si fa in quanto basta avere o u o v nella copertura e quindi separo le due situazioni possibili date dalla scelta di uno dei due estremi. Scelto un arco arco (x, v) non coperto da u e estendo il nodo di u collegandolo ad altri due nodi, uno con $\{u, v\}$ e uno con $\{u, x\}$ (fisso $k = 3$):



Proseguo poi via via con ulteriori archi che non sono coperti dai vertici foglia, etichettando gli archi dell'albero con l'identificativo del vertice che ho aggiunto.
Aggiungo ad esempio (x, v):



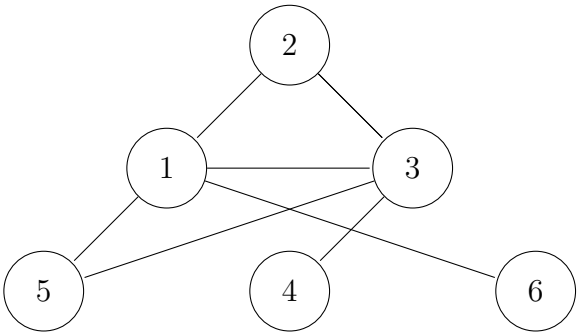
aggiungo poi (z, t):



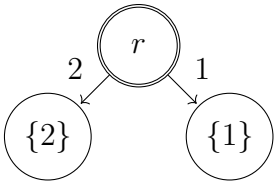
(mi fermo che sono a $k = 3$)
Scriviamo il processo formalmente.

Ricorsivamente si ha che sia S l'insieme che etichetta un vertice, scelgo (u', v') in E che non è coperto da S, quindi creo due figli con etichetta $S + u'$ e $S + v'$.
Si ha che se c'è un percorso di lunghezza k da r ad un nodo S che è una copertura per G allora non c'è bisogno di esplorare oltre l'albero, facendo quindi il **bound**, ovvero il taglio dell'albero.
Ovviamente ogni split non esclude che i due risultati coprano lo stesso insieme.
La scelta di ordine di archi da esplorare è casuale.
Ovviamente se ad altezza k non trovo una copertura minima significa che non esiste una copertura di k nodi.
Vediamo un esempio:

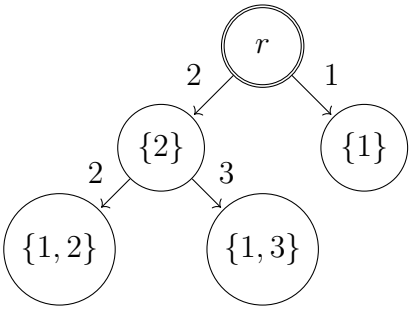
Esempio 11. *Sia dato il grafo:*



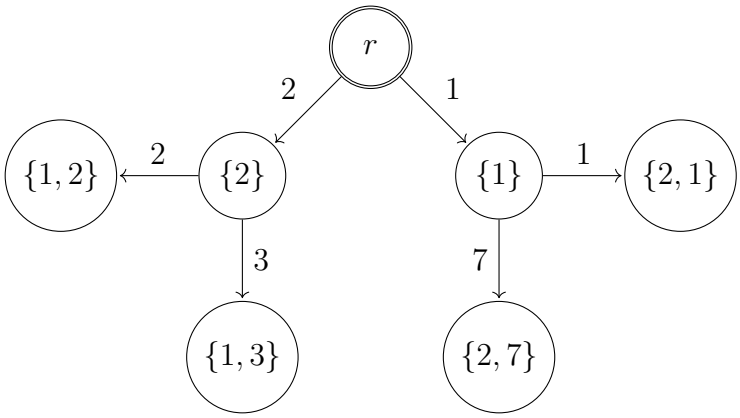
Scelgo di imporre $k = 2$.
Inizio con l'arco $(1, 2)$ *(la scelta è casuale):*



Scelgo poi $(2, 3)$*:*



scelgo poi $(1, 7)$



e mi fermo essendo a profondità 2. Tra le foglie cerco le coperture e noto che $\{1, 3\}$ *è effettivamente una copertura minima.*

su elearning appunti extra con algoritmo e calcolo dei tempi.

3.2.2 TSP metrico

Definizione 26. *Definiamo* ***ciclo di Eulero*** *come un ciclo che tocca tutti gli archi del grafo una e una sola volta. Posso tornare allo stesso vertice e quindi i vertici hanno grado entrante pari a quello uscente (numero di archi entranti pari a quello di quelli uscenti):*

$$indegree = outdegree$$

Vediamo quindi un algoritmo parametrico per TSP. Il problema TSP consiste nella pratica nel trovare un **ciclo hamiltoniano** di costo minimo. Ricordiamo che un ciclo hamiltoniano è un ciclo semplice, non orientato, che passa per ogni nodo di G. Il costo del ciclo è pari alla somma dei costi dei suoi archi.
Parliamo di **TSP simmetrico** se il grafo non è orientato, altrimenti è detto **TSP asimmetrico**. TSP è un problema **NP-hard**.
Studiamo quindi un'euristica per TSP. Si definisce w_{xy} come il peso dell'arco tra i nodi x e y.
Definiamo **TSP metrico** come un TSP dove vale la **distanza** "vera", ovvero i costi sono attribuiti tramite la **disuguaglianza triangolare**. Si ha che $\forall (u, v) \in E \ w_{uv} = 0$ sse $u = v$. Inoltre, dati tre nodi distinti, vale appunto la disuguaglianza triangolare:

$$w_{uv} \leq w_{uz} + w_{zv}$$

e quindi w introduce una semimetrica, che accettiamo come metrica. Applicando questo concetto ad un intero cammino posso dire che un arco tra due

vertici ha costo sicuramente minore uguale di quello di un cammino tra i due archi (praticamente estendo ogni volta il conto facendo una serie di triangoli usando sempre la disuguaglianza triangolare).
Una classe molto importante di metriche èquella delle metriche indotte dalle varie norme $\|\cdot\|_p$:

$$d_{\|\cdot\|_p}(i,j) = \|i-j\|_p = \left(\sum_{k=1}^{m} |i_k - j_k|^p\right)^{\frac{1}{p}}$$

e al variare di p si hanno varie metriche:

- **distanza di Manahattan** se $p = 1$:

$$d_{\|\cdot\|_1}(i,j) = \sum_{k=1}^{m} |i_k - j_k|$$

- **distanza Euclidea** se $p = 2$:

$$d_{\|\cdot\|_2}(i,j) = \left(\sum_{k=1}^{m} |i_k - j_k|^2\right)^{\frac{1}{2}}$$

- **distanza di Lagrange** se $p = \infty$:

$$d_{\|\cdot\|_\infty}(i,j) = \max_k\{|i_k - j_k\}$$

Per descrivere la 2-approssimazione, detta di **Christofides**, per TSP usiamo il ciclo di Eulero.

Definizione 27. *Definiamo multigrafo come un grafo che prevede la possibilità di avere archi diversi tra due nodi (possono anche essere cappi).*

Teorema 9 (teorema di Eulero)**.** *Un (multi)grafo ammette un ciclo euleriano se e solo se è connesso e ogni nodo ha grado pari (quindi per un arco entrante ho un arco uscente).*
Un (multi) grafo connesso e con ogni nodo di grado pari è detto grafo euleriano.

Esiste un algoritmo polinomiale per costruire un ciclo euleriano da un grafo, che se lo ammette è chiamato **grafo euleriano**. Si procede scomponendo il grafo in cicli che poi vengono fusi secondo un meccanismo che esclude archi

già visitati.

Studiamo quindi l'**algoritmo di Christofides**. Viene dato in input un grafo $G = (V, E)$, non orientato, completo, con costi dati dalla funzione c e dotato di semimetrica. L'algoritmo trova un ciclo hamiltonianodi costo al piùdoppio rispetto al ciclo di costo minimo procedendo tramite alberi di copertura minima, ovvero:

1. si trova il minimo albero ricoprente T di G, questo ha costo polinomiale tramite l'**algoritmo di Prim**. Per questo problema l'algoritmo greedy garantisce l'ottimo, basta infatti scegliere gli archi in ordine di costo crescente ma saltando quelli che chiudono ciclo

2. si raddoppia ogni arco di T ottenendo un grafo euleriano

3. si costruisce un ciclo euleriano ϵ di questo grafo, questo ha costo polinomiale, basta infatti percorrere uno dopo l'altro tutti i cicli ottenuti dai rami dell'albero

4. si visita ϵ partendo da un nodo iniziale u

5. si costruisce una permutazione π dei nodi V di G *sequenziando i nodi* nell'ordine della loro prima apparizione in ϵ nella visita (potrei avere comunque archi che accorciano il percorso essendo G completo (???)).

6. si restituisce il ciclo hamiltoniano H associato a π

In pratica si ha un'euristica che si dimostra essere 2-approssimante.
Inoltre si ha che c soddisfa la disuguaglianza triangolare (saltando dei nodi seguo delle scorciatoie) allora si ha che:

$$c(H) \leq c(\epsilon)$$

Teorema 10. *Sia H^* il ciclo hamiltonianodi costo minimo, e sia T l'albero di copertura minima di G di costo minimo. Si ha quindi che:*

$$c(H^*) \geq c(T)$$

Teorema 11. *Sia H^* il ciclo hamiltoniano di costo minimo, e sia H il ciclo ritornato dall'algoritmo di Christofides, allora si ha che:*

$$c(H) \leq 2 \cdot c(H^*)$$

Avendo così la 2-approssimazione.

Dimostrazione. Si ottiene quindi che $c(H)$ approssima la soluzione di minimo costo, che indichiamo con $c(H^*)$, avendo H^* come il ciclo hamiltonianodi costo minimo.
Si ha inoltre che, dato $c(T)$ come costo dell'albero di copertura:

$$c(\epsilon) = 2 \cdot c(T)$$

perché ϵ è il ciclo di eulero del grafo G_T ottenuto duplicando gli archi di T e ϵ attraversa una sola volta tutti gli archi di tale grafo:

$$c(\epsilon) = c(\epsilon(G_T)) = 2 \cdot c(T)$$

Si può quindi anche dire che:

$$c(T) \geq c(H^*)$$

dato che H^* è ottenuto da un certo albero T', che non sappiamo se è di copertura minimo, a cui viene aggiunto un arco e'. Ma sappiamo che:

$$c(T') \geq c(T)$$

riassumendo un ciclo hamiltoniano ottimo è quindi costituito da un albero di copertura T' (che non sappiamo se è di copertura minimo e quindi $c(T) \leq c(T')$ più un arco e' che chiude l'albero ad un ciclo per cui:

$$c(H^*) \geq c(T')$$

avendo che:

$$c(H^*) = C(T') + c(e')$$

A questo punto si ha:

$$c(T) \leq c(T') \leq c(H^*)$$

Sapendo quindi che:

$$c(\epsilon) = 2 \cdot c(T)$$

e che:

$$2 \cdot c(T) \leq 2 \cdot c(H^*)$$

e quindi si ottiene che:

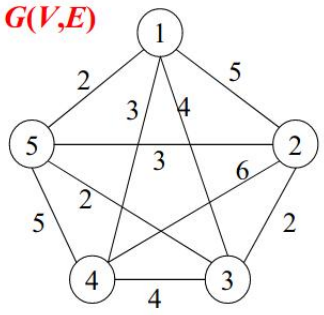
$$c(H) \leq c(\epsilon) \leq 2 \cdot c(H^*)$$

e quindi:

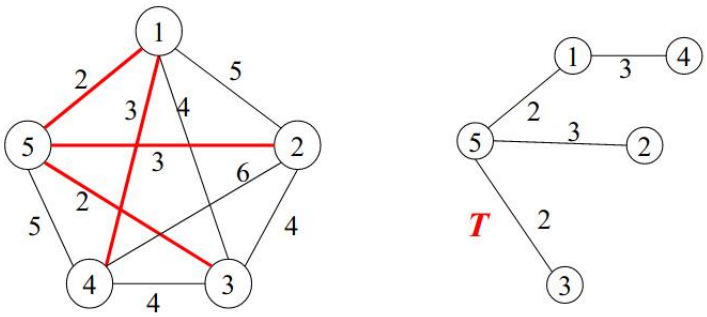
$$c(H) \leq 2 \cdot c(H^*)$$

dimostrando la 2-approssimazione. □

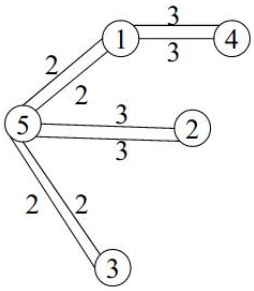
Esempio 12. *Partiamo da:*



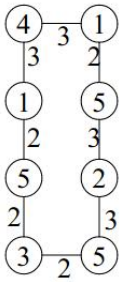
Procediamo con lo step 1, ottenendo T:



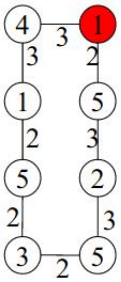
Procediamo con lo step 2, raddoppiando ogni arco di T:



Procediamo con lo step 3, ottenendo ϵ:



e con gli step 4 e 5 si ha:



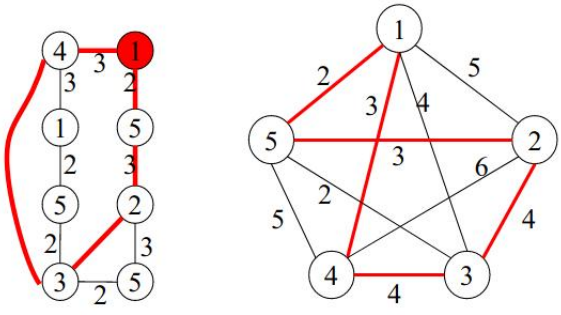
che deve essere "letto" in senso orario.
Si ha quindi la sequenza di visita:

$$\epsilon = \{1, 5, 2, 5, 3, 5, 1, 4\}$$

con la permutazione associata:

$$\pi = \{1, 5, 2, 3, 4\}$$

Si arriva quindi, con lo step 6, a:



(avendo che comunque un arco tra due nodi che sostituisce un cammino tra quei nodi sicuramente costa meno o la più costa uguale, avendo la semimetrica)

L'**algoritmo di Christofides** trova quindi un'euristica cercando un *lower bound*, tramite l'albero di copertura.

Cammino hamiltoniano e TSP

Abbiamo visto che il cammino hamiltoniano è un problema **NP-complete** e sappiamo che (si veda lezione nel capitolo sulle macchine di Turing, cronologicamente fatta prima):

$$hamiltonianPath \leq_P TSP$$

(aggiungendo gli archi mancanti con costo maggiore dell'unità)

Teorema 12. *TSP (non metrico) non ha un'approssimazione costante a meno che $P = NP$.*

Dimostrazione. Sia $G = (V, E)$ il grafo in analisi su cui costruire il ciclo hamiltoniano. Costruiamo un $G' = (V', E', W)$ tale che:

$\Big\{ w((u,v)) = 1$ sse $e = (u,v) \in E w(e) = \varepsilon \cdot |V| + 1$ altrimenti (pesando più di 1)

Si assume quindi di avere un algoritmo ε-approssimato. Si hanno due casi:

1. l'algoritmo restituisce un tour di costo $|V|$ e quindi esiste il ciclo hamiltoniano
2. l'algoritmo A restituisce un tour con almeno un arco di costo $\varepsilon \cdot |V| + 1$ e quindi $A(x) > \varepsilon \cdot |V|$

Ma per definizione di ε-approssimazione ho che:

$$\varepsilon \cdot opt(x) \geq A(x)$$

e quindi si ha che:

$$opt(x) > \frac{A(x)}{\varepsilon} > |V|$$

cioèil tour ottimoèdi costomaggiore di $|V|$ e quindi non esiste un ciclo hamiltoniano (non c'è modo di avere Un tour di costo $|V|$).
Pertanto A diventa un algoritmo per deciderese esiste o meno un cammino hamiltoniano dove A è polinomiale cioè è nella classe P. L'algoritmo è fatto come segue:

- se $A(x) = |V|$ allora ho che $opt(x) = |V|$ ed esiste il ciclo Hamiltoniano, altrimenti deve essere che $A(x) > \varepsilon \cdot |V|$ ma per costruzione ho che $opt(x) > |V|$ e quindi non esiste il ciclo hamiltoniano ma poiché il problema del ciclo hamiltoniano è in NP, ne consegue che P= NP, dimostrando il teorema.

□

Esistono quindi problemi che non hanno approssimazione costante.
Esistono anche problemi che hanno ε-approssimazione costante $\forall\, \varepsilon$ piccolo a piacere (si parla di **Polynomial time approximation scheme (*PTAS*)**).

3.2.3 Closest string

Fatta velocemente e sembra fuori esame.
In questo problema si ha in input una collezione di k stringhe $s_1, \ldots, s_k$ di lunghezza l. Si ha anche un input d e in output si ha una stringa s tale che:

$$d_H(s, s_j) \leq d, \;\; \forall\, j = 1, \ldots, k$$

dove con $d_H(s, s')$ indichiamo la **distanza di Hamming** tra due stringhe s e s', ovvero il numero di simboli diversi tra esse.

Teorema 13. *Closest string è risolvibile in tempo:*

$$O(d+1)^d |G|$$

e se d è piccolo, allora il tempo $O(d+1)^d$ è accettabile (esempio per distanza Hamming $d = 5$) anche quando l'input è molto grande

Dimostrazione extra su slide.

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.

L'algoritmo per il test di primalità funziona in $\log(n^{12})$, nella versione più efficiente sviluppata da un gruppo di indiani, che arriva ad ottenere un tempo polinomiale.

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

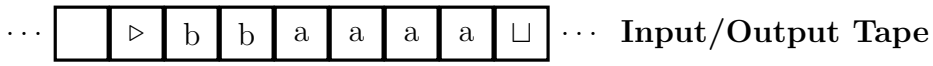


Figura 4.1: Esempio di nastro di una TM

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:

Teorema 14 (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 28. *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 29. *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 30. *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 13. *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:*

$$\begin{array}{r} 10010101+ \\ 00000001= \\ \hline 10010110 \end{array}$$

Definisco quindi la TM:

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

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

$$\delta \to (s_0, \sqcup) \to (s_1, \sqcup, \leftarrow)$$
$$\delta \to (s_1, 0) \to (H, 1, -)$$
$$\delta \to (s_1, 1) \to (s_1, 0, \leftarrow)$$
$$\delta \to (s_1, \triangleright) \to (H, \triangleright, -)$$

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*

Volendo ad un certo punto si arriva allo stato finale H. In quel momento ho una nuova stringa Y (il risultato delle modifiche su X):

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

e la testina non deve più spostarsi. Questa può essere interpretata come uno **stato di arresto**.
Un altro caso è avere una computazione infinita nel caso in cui, letto un simbolo e lo stato, la testina ricopia il simbolo ma non si sposta ne a destra ne a sinistra. Oppure una testina potrebbe "rimbalzare" infinitamente tra due posizioni. Quest'ultima cosa può anche accadere tra molte posizioni, entrando comunque in **loop infinito**, ritornando sempre, prima o poi, in una configurazione (e si noti "configurazione", non coppia stato-simbolo) già avuta. In questo caso la computazione è **non terminante** e la computazione non produce alcun output. *Quindi un insieme finito di elementi (stati, simboli e possibili transazioni) può comunque descrivere un comportamento infinito (infiniti passi).*
Analizziamo meglio gli stati terminanti Y e N il primo indicante che la stringa in input è accettata, avendo certe caratteristiche richieste, il secondo che la stringa viene rifiutata. Non ho quindi più bisogno della stringa in output, quindi posso lasciar scritto quello che voglio quando finisco, mi basta sapere lo stato finale Y e N.

Esempio 14. *Sistemiamo l'esempio precedente aggiungendo il riporto, dovendo aggiungere un bit.*
Si indica solo la funzione di transizione.
Scorro fino alla fine (quindi vado a destra indipendentemente dal simbolo fino ad un blank):

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

Sono a destra dell'ultimo carattere di X (avendo un blank):

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

eseguo il conto tornando indietro:

$$\delta \to (s_1, 0) \to (H, 1, -)$$

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

sono tornato all'inizio:

$$\delta \to (s_1, \triangleright) \to (s_2, 1, \leftarrow)$$

devo "shiftare" tutti i caratteri (per farlo semplicemente metto $\triangleright$ nel primo blank a sinistra del precedente $\triangleright$ arrivando nello stato s_2, come indicato nello step precedente):

$$\delta \to (s_2, \sqcup) \to (H, \triangleright, -)$$

(avrei comunque potuto shiftare tutte le unità a destra di una posizione, o semplicemente, sapendo che ora sul nastro ho soli 0 dovrei tornare a destra di un passo, cambiare il primo 0 in 1 e aggiungere uno 0 in fondo alla stringa).

Esempio 15. *Vediamo un esempio in cui una TM riconosce una stringa binaria che è palindroma o meno.*
Si indica solo la funzione di transizione.
Se sono sullo start vado a destra di uno:

$$\delta(s_0, \triangleright) \to (s_0, \triangleright, \to)$$

se vedo 0 vado nello stato zero, scrivo $\sqcup$ e vado a destra:

$$\delta(s_0, 0) \to (zero, \sqcup, \to)$$

analogo leggendo 1, andando nello stato 1 scrivendo blank:

$$\delta(s_0, 1) \to (one, \sqcup, \to)$$

vado in fondo alla stringa (qualsiasi incrocio tra gli stati one e zero e simboli 1 e 0 legga vado a destro riscrivendo lo stesso simbolo):

$$\delta \to ([zero, one], [0, 1]) \to (\delta \to ([zero, one], [0, 1], \to)$$

sono in fondo alla stringa, se sono in stato zero scrivo blank e torno indietro, in uno stato $zero'$:

$$\delta(zero, \sqcup) \to (zero', \sqcup, \leftarrow)$$

idem per stato one

$$\delta(one, \sqcup) \to (one', \sqcup, \leftarrow)$$

se sono in $zero'$ e leggo 0 vado in stato s_1 e vado a sinistra:

$$\delta(zero', 0) \to (s_1, \sqcup, \leftarrow)$$

se sono in $zero'$ e leggo 1 la stringa non è palindroma, esco con stato N:

$$\delta(zero', 1) \to (N, \sqcup, \leftarrow)$$

idem per stato one', andando in stato s_2:

$$\delta(one', 1) \to (s_2, \sqcup, \leftarrow)$$

$$\delta(one', 0) \to (N, \sqcup, \leftarrow)$$

ora proseguo a sinistra fino ad un blank riscrivendo quanto letto:

$$\delta(s_1, [0, 1]) \to (s_1, [0, 1], \leftarrow)$$

sono nel blank e torno allo stato iniziale, potendo ricominciare la computazione:

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

ma se la stringa (di cardinalità pari) è palindroma cancello tutto, devo quindi specificare che se in s_0 ho blank la stringa è valida:

$$\delta(s_0, \sqcup) \to (Y, \sqcup, -)$$

se invece la stringa è di cardinalità dispari, allo stato attuale, entra in loop, dobbiamo quindi aggiungere un uscita d:a $zero'$ o one' in questo caso:

$$\delta([zero', one'], \sqcup) \to (Y, \sqcup, -)$$

Esempio 16. *Scrivo una TM per decidere se la stringa in input è del tipo $a^n b^n c^n$.*
Si indica solo la funzione di transizione.
Se leggo a vado nello stato A e vado a destra

$$\delta(s_0, a) \to (A, \sqcup, \to)$$

Se leggo in s_0 b o c significa che non ho a quindi esco:

$$\delta(s_0, [b, c]) \to (N, \sqcup, -)$$

Se in A trovo un'altra a la lascio e vado a destra:

$$\delta(A, a) \to (A, a', \to)$$

Se in A leggo b vado in B, e vado a destra:

$$\delta(A, b) \to (B, b', \to)$$

se invece leggo c non ho b e esco:

$$\delta(A, c) \to (N, \sqcup, -)$$

se in B e trovo a non va bene:

$$\delta(B, a) \to (N, \sqcup, -)$$

Se in B trovo un'altra b la lascio e vado a destra:

$$\delta(B, b) \to (B, b, \to)$$

Se in B leggo c vado in C e vado a sinistra a controllare:

$$\delta(B, c) \to (C, c', \leftarrow)$$

controllo tramite i sentinella indicati con $'$.
Siamo in C quindi:

$$\delta(C, c') \to (C, c', \leftarrow)$$

Se però vedo b o b' resto in C e riscrivo:

$$\delta(C, [b, b']) \to (C, [b, b'], \leftarrow)$$

Per A cambia, se vedo a scorro:

$$\delta(C, a) \to (C, a, \leftarrow)$$

ma se vedo a' torno in s_0:

$$\delta(C, a') \to (s_0, a', \to)$$

ma ora A può incontrare b', che va bene, o c' che non va bene:

$$\delta(A, b') \to (A, b', \to)$$

$$\delta(A, c') \to (N, \sqcup, -)$$

Per C:

$$\delta(C, [b, c]) \to (C, [b, c'], \to))$$

Se sono in s_0 e trovo a' ho finito le a:

$$\delta(s_0, a') \to (s_0, a', \to))$$

idem:

$$\delta(s_0, [b', c') \to (s_0, [b', c'], \to))$$

Vado in stato check se:

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

e proseguo fino a trovare solo b' o c', se trovo invece b o c esco. Diventa quindi molto lungo.

L'esempio sopra mostra quanto possa diventare lunga una semplice computazione sulla TM, ci servirebbe quindi una sorta di *TM alternativa*.

Teorema 15 (teorema di Böhm-Jacopini)**.** *Qualunque algoritmo può essere implementato utilizzando tre sole strutture, la sequenza, la selezione e il ciclo, da applicare ricorsivamente alla composizione di istruzioni elementari*

Definizione 31. *Si ha la* ***TM a*** k ***nastri****, ovvero la* ***TM multi-nastro*** *dove ho k nastri di lettura e scrittura, magari avendo l'input solo su un nastro o su multipli. La macchina quindi legge uno stato e k simboli $\{\sigma_1 \dots, \sigma_k\}$. Prima dello spostamento quindi scrive tutti i simboli. Lo spostamento sarà uno per ogni nastro.*

Esempio 17. *Risolviamo il precedente problema (decidere se la stringa in input è del tipo $a^n b^n c^n$) con k nastri.*
Potrei usare 4 nastri, sul primo c'è l'input. Finché trovo a scrivo sul primo, quando trovo b passo a secondo e faccio lo stesso se leggo c. In ogni caso mi interrompo se dopo delle b trovo a o dopo c trovo a, b. Infine torno indietro sui tre nastri e vedo se sono allineati.

Una **TM a k nastri** non è più potente di una **TM a singolo nastro** (vale il rapporto linguaggio di programmazione e linguaggio macchina). Esiste sempre una traduzione verso la TM a singolo nastro.
Per esempio invece un automa a stati finiti è meno potente di un TM, che può spostarsi e scrivere dove vuole, a differenza degli automi.
Vediamo cosa quindi può fare una TM:

- una TM può **computare** funzioni su stringhe
- una TM può **decidere** (rispondendo Y o N) un linguaggio (ovvero un insieme finito o infinito di stringhe, come il caso sopra $a^n b^n c^n$), ovvero data una stringa in input è sempre in grado di dire se appartiene o meno al linguaggio. Un linguaggio è **decidibile**, o, più formalmente, **ricorsivo**, se esiste almeno una TM che decide il linguaggio, fermandosi sempre in Y o N. Solitamente è un linguaggio finito, ma potrebbe anche essere infinito (ma tutte le stringhe sarebbero comunque riconoscibili)
- una TM può **accettare** un linguaggio, ovvero se fornisco una stringa in input che appartiene al linguaggio la riconosce come tale e prima o poi arriva allo stato Y, altrimenti la macchina potrebbe:
  - fermarsi in uno stato N

– entrare in un loop infiniti, non avendo mai la risposta ma magari in realtà non è un loop infinito ma solo servono anni per avere la risposta

Quindi la TM non è in grado di distinguere completamente le stringhe che non fanno parte di un linguaggio. Un linguaggio è **linguaggio ricorsivamente enumerabile** è un linguaggio per cui data una stringa in input buona la TM si ferma, altrimenti la TM potrebbe o fermarsi o andare avanti all'infinito nella computazione. Posso enumerare tutte le stringhe che fanno parte del linguaggio, tramite una certa procedura. Potrei quindi avere un linguaggio infinito ma con stringhe che mandano in loop la TM, anche se comunque non sono in grado di capire se sono in un loop o se prima o poi la stringa data in input verrà riconosciuto. Ci sono linguaggi che si può dimostrare essere ricorsivamente enumerabili (e alcuni che non sono nemmeno ricorsivamente enumerabili)

Ricordiamo comunque che esistono problemi irrisolvibili, dimostrati tali, come la soluzione di *equazioni Diofantee*.
I linguaggi possono essere quindi ricorsivi, ricorsivamente enumerabili o non ricorsivamente enumerabili. Un linguaggio finito è ricorsivo.
Se ho l'alfabeto Σ, presa la *-chiusura di Σ (che quindi comprende la stringa vuota e tutte le stringhe che posso ottenere concatenando uno o più simboli di Σ), ovvero σ^*, la TM riconosce subito la stringa in input in quanto costruita per funzionare su Σ, quindi il linguaggio è ricorsivo. Se prendo Σ^+, ovvero Σ^* senza ε, ho comunque un linguaggio ricorsivo per lo stesso motivo, dovendo solo controllare in più se s_0 è $\sqcup$.

Teorema 16. *Preso un linguaggio L costruito alfabeto Σ si ha che se $L \subseteq \Sigma^*$ se L è ricorsivo è anche ricorsivamente enumerabile.*

Dimostrazione. Se L è ricorsivo esiste una TM che riconosce se, data una stringa x, $x \in L$, rispondendo Y e risponde N se $x \notin L$.
Costruisco ora una TM M' che se il linguaggio non è ricorsivo allora vado in loop, indicato con $\infty, \uparrow, \perp$. Quindi quando la macchina sta per andare in N cambio δ per ottenere il loop, ottenendo una macchina che va in loop se $x \notin L$, mentre riconosce con Y se $x \in L$ e quindi L è ricorsivamente enumerabile. □

Teorema 17. *L è ricorsivo sse L è ricorsivamente enumerabile e il complementare di L, $\overline{L}$ è ricorsivamente enumerabile.*

Dimostrazione. Rispetto alla dimostrazione precedente, che garantisce che se L è ricorsivo allora è ricorsivamente enumerabile, devo definire, per il complementare, una TM che va in loop se $x \in L$, andando altrimenti in loop. Presa quindi una TM M che decide L, definisco M' che accetta L, che quindi è ricorsivamente enumerabile. Definisco ora M'' che restituisce Y se $x \in \overline{L}$ ovvero $x \notin L$ (e ∞ se $x \notin \overline{L}$, ovvero $x \in L$). Quindi M'' fa la stessa cosa di M ma quando M sta per rispondere Y faccio andare M'' in loop e se M va in N faccio andare M'' in Y, tutto tramite un cambio di δ. □

Teorema 18. *Preso un linguaggio L, se è ricorsivamente enumerabile e lo è anche il complementare $\overline{L}$ allora L è ricorsivo.*

Dimostrazione. Presa una TM M', se questa risponde Y allora M risponde Y. Presa M'' se questa risponde Y allora M risponde N. M' e M' accettano L e $\overline{L}$ quindi non possono dire Y contemporaneamente e non rispondono N, andando in loop. M non va mai in loop perché $x \in L \vee x \notin L$ per cui prima o poi una tra M' e M'' si ferma. Devo quindi simulare l'esecuzione delle due macchine, raggruppandole in una sola, appunto M. Posso usare 4 nastri (due per le macchine e due per gli input) oppure posso mettere le due δ delle due macchine nella stessa (facendo prima un'operazione di m' e poi una di M'' e così via), fino a che non si arriva ad uno Y e, segnando tramite un nastro le configurazioni di M' e M'', posso capire se far uscire M con Y o N, in base a chi tra M' e M'' è uscito con Y. □

A noi interessano comunque i **problemi** ma i linguaggi sono comodi per parlare di **intrattabilità**, legandosi bene ai **problemi di decisione**.

Definizione 32. *Un problema di decisione è un problema con solo due possibili risposte, Y e N.*
*I problemi di decisione sono restrizioni di problemi di ottimo, ai quali viene aggiunto un **bound**, cambiando la domanda in una richiesta di esistenza di un caso che soddisfi il bound nel problema di ottimo (ma comunque potrei non avere tempi polinomiali, anche se il problema decisionale è più facile di quello di ottimo).*

Teorema 19. *Un problema di decisione è sempre più facile del corrisponde problema di ottimo.*

Posso associare un problema di decisione allo studio di un linguaggio, avendo essi la stessa risposta, Y o N. Un problema di decisione ha quindi un corrispondente linguaggio, con l'input del problema che ha una corrispondente stringa (di cui bisogna studiare appartenenza ad un linguaggio).

Esempio 18. *Il problema* ***Hamiltonian cycle problem (HCP)*** *(esiste anche la variante non con il ciclo ma con il cammino:* ***Hamiltonian path problem (HPP)****).*
Dato un grafo non completo e non pesato ci si chiede se c'è un modo per partire da un nodo e tornarci dopo aver toccato tutti i vertici una e una sola volta.
Questo problema è di decisione ed è risolvibile, ma è difficile da risolvere in termini temporali (restringersi ai problemi decisionali non sempre implica miglioramenti in termini di tempo). HCP si risolve solo in tempo esponenziale.

Teorema 20. *Se il problema di decisione non è risolvibile in modo efficiente sicuramente il problema di ottimo associato non è risolvibile in modo efficiente.*

Teorema 21. *Preso π problema di decisione. Posso passare da π ad un linguaggio $L(\pi)$ attraverso uno* ***schema di codifica****, che prende in input un'istanza del problema è mette in output una stringa x del linguaggio. Se istanza ha risposta Y ho $cod(x) \in L$ (e se risponde N ho $cod(x) \notin L$, con cod che indica una* ***codifica****, ovvero una traduzione dell'istanza nel linguaggio.*

Risolvere un problema di decisione vuole dire essere in grado di riconoscere o meno le stringhe del corrispondente linguaggio.

Esempio 19. *Pensando a HCP ho:*

$$L_{HCP} = \{y \in \Sigma^* | \text{ che corrispondono ad un'istanza con risposta } Y\}$$

Sapendo che HCP è risolvibile so che esiste una TM che risolve il problema e il grafo può essere tradotto in stringa, ad esempio tramite una matrice di adiacenza.
HCP è risolvile sse il linguaggio associato è decidibile, e quindi ricorsivo.

Teorema 22. *Quindi un problema decisionale è risolvile sse il linguaggio associato è decidibile, e quindi ricorsivo.*

Definizione 33. *Il tempo di esecuzione di una TM è il conto dei passi di esecuzione nel caso peggiore.*

4.3 Problemi non risolvibili

Ci serve, in primis, una nuova TM:

Definizione 34. *Definiamo la* ***Universal Turing Machine (UTM)*** *come una TM che non fa un compito specifico in base alla δ ma prende in input un'altra TM M, un separatore ";" e l'input x di M e da in output la stessa cosa che darebbe M con input x:*

$$UTM(M;x) \to M(x)$$

Quindi calcola quello che calcola qualunque altra TM.

La UTM è simile a quello che fa tramite un linguaggio di programmazione, dove si da una sequenza di istruzioni e un input, ottenendo un output. Un calcolatore moderno è quindi una sorta di UTM (con l'hdd che corrisponde al nastro infinito, non avendo potenzialmente limite potendo aggiungere altri dischi).
Esistono le **Small UTM (*SUTM*)** che riusano gli stati per ridurre lo spazio usato.
Stati e simboli sono mediamente numeri naturali, e gli spostamenti pure (con magari degli stati particolari). Un moderno calcolatore infatti ragiona solo su numeri per definire tutto, dagli stati ai simboli.
Normalmente si ha che $UTM(M;x)$ ha due nastri sul primo $(M;x)$ e sul secondo la configurazione attuale di $(M;x)$. Quindi una TM può simularne un'altra salvando le configurazioni. La UTM tramite la sua δ poi copia lo stato di uscita di M, magari scrivendo su un terzo nastro l'output.
Posso quindi costruire una TM (la UTM) che simuli un'altra TM, quindi si è in grado di scrivere un algoritmo che prende in input un altro programma scritto nello stesso linguaggio, che fa le stesse cose, e quindi il primo algoritmo, per esempio, può dare le risposte opposte.
Si ha quindi che esistono **linguaggi non ricorsivi**, ovvero esistono **problemi non decidibili** (come i dieci problemi proposti da Hilbert ad inizio novecento).
Si ha che le TM sono **enumerate** in quanto possono essere associate ai numeri naturali.
Ci sono problemi che possono essere chiaramente descritti per i quali sappiamo che non esiste alcun algoritmo in grado di risolverli (di risolverli **sempre**, basta quindi anche solo un caso per cui non posso avere tale algoritmo). **Un problema non decidibile non è un problema intrattabile**, che sarebbe risolvibile ma solo in tempo esponenziale.
Non si può comunque dimostrare a priori che non esista un certo algoritmo (?)

4.3.1 Halting problem

Turing riconosce da subito come interessante l'**halting problem**. Si cerca un algoritmo che data una certa coppia $(M; x)$, TM/input, mi dica se quella macchina arriva a termine computazione o entra in loop infinito.

Definizione 35. *Definisco formalmente l'**halting problem** con il linguaggio H:*

$$H = \{M; x |\, M(x) \neq \perp\}$$

Quindi se la stringa fa parte del linguaggio M termina, altrimenti no.

Teorema 23. *Si ottiene però che H non è ricorsivo e quindi l'halting problem non è decidibile.*

Dimostrazione. **DIMOSTRAZIONE DA SISTEMARE!**
Immaginiamo per assurdo che H sia ricorsivo e quindi esiste una TM M_H che prende in input $(M; x)$ e mi da sempre in output $M(x)$, ma così sto facendo la stessa cosa della UTM. Ma M_H non fa quello che fa la UTM, infatti fa:

$$\begin{cases} Y & \text{se } M(x) \neq \perp \\ N & \text{se } M(x) = \perp \end{cases}$$

Quindi fa qualcosa in più della UTM, perché se finisce in loop deve capire che è in loop infinito, interrompere e restituire N.
Assumo che H quindi è ricorsivo e quindi M_H esiste, anche se magari non la so costruire. Se esiste M_H deve esisterne un'altra, chiamata D, che presa in input solo M produce:

$$D(M) \to M_H(M; M)$$

Dando in input alla TM la descrizione della TM stessa (che è comunque ragionevole). Significa che:

$$M_H(M; M) = \begin{cases} Y & \text{se } M(M) \neq \perp \\ N & \text{se } M(M) = \perp \end{cases}$$

ma voglio una macchina D per la quale i risultati siano invertiti (è questa la chiave della dimostrazione per assurdo):

$$D(M) = \begin{cases} Y & \text{se } M(M) \neq \perp \to \infty \\ N & \text{se } M(M) = \perp \to Y \end{cases}$$

Quindi D cambia N in Y e Y in ∞, presa in input una macchina M.
Provo quindi a dare in input a D D stessa, ottenendo o il loop o Y. Ipotizziamo di andare in stato Y, quindi:

$$M_H(D; D) = N$$

che ci dice se D su input D termine e quindi M_H, per la definizione di M_H deve rispondere N, Ma M_H risponde N sse $D(D)$ deve andare in loop infinito. Siamo arrivati ad un assurdo avendo detto che $M_H(D;D) = N$ accade sse $D(D) = Y$.
Inoltre se ipotizziamo che $D(D)$ va in loop allora $M_H(D;D) = Y$ che implica che $D(D)$ termina la computazione, sempre per definizione. Ho ottenuto quindi un'altra contraddizione, confermando la prova per assurdo. □

Si è dimostrato quindi che anche problemi ben definiti possono non essere decidibili, problemi come il definire il comportamento di un programma dato un input. Il problema comunque non è una potenza limitata della TM ma un limite intrinseco al problema stesso.

Teorema 24. *Il linguaggio H è ricorsivamente enumerabile, ovvero il problema è **parzialmente decidibile**. La macchina quindi riconosce una macchina che termina, altrimenti non si sa.*

Dimostrazione. Devo trovare una macchina che termina in Y che riconosce una stringa appartenente al linguaggio H.
Costruisco quindi una M'_H che prende in input una TM generica col suo input e si ha che:

$$M'_H(M;x) = \begin{cases} Y & \text{se } M(x) \text{ termina} \\ N/\infty & \text{altrimenti} \end{cases}$$

Quindi se non termina non si sa che succede, ma potrebbe anche essere sempre in loop.
Pensando a UTM ho che:

$$UTM(M;x) = \begin{cases} Y & \text{se } M(x) = Y \text{ avendo } M'_H = Y \\ N & \text{se } M(x) = N \text{ avendo } M'_H = Y \\ H & \text{se } M(x) = H \text{ avendo } M'_H = Y \\ \infty & \text{se } M(x) = \infty \text{ avendo } M'_H = \infty \end{cases}$$

Quindi M'_H è una UTM con gli stati finali leggermente modificati (quindi non più una UTM), avendo solo Y (se termina) e ∞ (altrimenti). Abbiamo quindi dimostrato che è **parzialmente decidibile**, avendo che la macchina da Y se la computazione è terminante e va in loop altrimenti. □

Ho quindi problemi non decidibili che sono parzialmente decidibili. Esistono anche problemi legati a linguaggi **non ricorsivamente enumerabili**, per cui quindi non si può **mai** dare risposta. Si hanno quindi problemi nemmeno parzialmente decidibili. Pensiamo alle coppie $(M;x)$ che non terminano.

Teorema 25. *Esistono linguaggi* ***non ricorsivamente enumerabili*** *avendo quindi problemi non decidibili.*

Dimostrazione. Basti pensare che se L è ricorsivo sse L è ricorsivamente enumerabile. Quindi, pensando all'halting problem, L_H non è ricorsivo ma è è ricorsivamente enumerabile. Ma allora $\overline{L_H}$, il linguaggio complementare non è ricorsivamente enumerabile:

$$\overline{L_H} = \{M; x | M(x) = \perp)$$

□

4.3.2 Problemi decidibili

Studiamo ora i problemi decidibili catalogandoli in base ai tempi, nei casi peggiori. Si hanno:

- algoritmi in tempo polinomiale rispetto alla dimensione dell'input
- algoritmi in tempo esponenziale rispetto alla dimensione dell'input, sono dimostrabilmente intrattabili

Bisogna capire se un problema cade in queste due categorie.
Il tempo di esecuzione viene calcolato tramite il numero di passi della TM.

Definizione 36. *Definiamo $t_M(x)$ come il tempo di calcolo di un TM M su input x. Non è un caso peggiore ma dipendente dal singolo input specifico. Il tempo di calcolo è il numero di passi che esegue M su input x per dare una risposta. Concentrandosi sui problemi di decisione decidibili avrò sempre Y o N se x fa parte o meno del linguaggio. Ci concentriamo solo sui linguaggi ricorsivi.*

Non si usa comunque il numero di passi nella realtà ma si usa l'O-grande, studiando il caso peggiore, il numero massimo di passi.

Definizione 37. *Definiamo $T_M(n)$ come la* ***funzione di complessità temporale*** *come:*

$$T_M(n) = \max\{t_m(x) | \, |x| = n\}$$

Definizione 38. *Definisco la classe P in base alle TM. La* ***classe P*** *è definita come:*

$$P = \{L | \ L \text{ è deciso da una DTM in tempo polinomiale } O(p(n))\}$$

*Con DTM che indica una TM deterministica, in cui per ogni coppia stato/-simbolo la macchina può fare una sola cosa (leggere/*scrivere/spostarsi di uno)m e con p una certa funzione polinomiale.*
È quindi la classe di problemi che so risolvere velocemente.

Definizione 39. *Definiamo la classe* ***time(f(n))*** *come la classe dei linguaggi decisi da una TM entro un tempo $f(n)$. Quindi* ***time(n)*** *sono tutti quelli decisi in tempo lineare, ad esempio (ma potrei anche per un qualsiasi n^k). Quindi:*

$$P = \cup_{i \geq 0} time(n^i)$$

Infatti P è l'unione di tutte le classi time con funzioni polinomiali.
La maggior parte dei problemi non supera, per il tempo polinomiale, un esponente pari a 10 (N^{10}).

Teorema 26. *Se un problema è nella* ***classe P*** *allora è risolvibile in un* ***tempo efficiente****.*

Devo anche definire lo **spazio**, oltre al **tempo**.

Definizione 40. *Definisco lo* ***spazio di calcolo*** *come:*

$$s_M(x) = \text{ numero di celle del nastro usate dalla TM } M \text{ con input } x$$

$$\text{durante la computazione}$$

Il calcolo non è semplice come per il tempo, avendo anche decrementi. Quindi più che "celle usate" studiamo le "celle visitate".

Definizione 41. *Definisco $S_M(n)$ come la* ***funzione di complessità spaziale****:*

$$S_M(n) = \max\{s_M(x) | \, |x| = n\}$$

Spazio e tempo sono legati per una TM.
Innanzitutto se una computazione dura n passi (tempo) posso dire che al più ho usato n celle (spazio), perché magari in qualche passo la testina rimane ferma ma nel caso peggiore si sposta sempre. Si ha quindi:

$$S_M(n) \leq T_M(n) + n$$

con $+n$ perché sul nastro abbiamo comunque l'input di lunghezza n (anche se potrebbe non essere letto dalla TM). In alcuni casi non viene nemmeno considerato come spazio usato in quanto non cambia la risposta in merito agli studi di tempo (a meno di studiare tempi inferiori al tempo lineare, dove in caso si specifica a parte di avere un input che occupa spazio n).

Teorema 27. *Se il tempo è limitato allora lo spazio è limitato ma non vale l'opposto (potrei avere banalmente un loop tra poche celle, avendo l'uso limitato di poche celle per un tempo illimitato).*

Teorema 28. *Se ho una TM M che lavora in spazio finito e tempo infinito, esiste una TM M' che fa la stessa cosa di M in tempo limitato.*
Quindi se lo spazio è limitato allora il tempo è limitato.

Dimostrazione. Infatti la macchina M' può trovarsi in un numero finito di stati K e avendo spazio limitato ho un numero limitato $S_M(n)$ di celle in cui si trova la testina. Ho anche un numero finito di simboli in alfabeto σ e quindi:

$$S_{M'}(n) \leq |k| \cdot |S_M(n)| \cdot |\Sigma|^{|S_M(n)|}$$

avendo che prima o poi ritorno a stati già visti quindi la macchina se supera la quantità appena definita capisce di essere in loop (questo perché si ha a che fare con insiemi limitati e quindi tale ragionamento non va bene per l'halting problem). Quindi $|k| \cdot |S_M(n)| \cdot |\Sigma|^{|S_M(n)|}$ è anche un limite temporale per la seconda macchina (**rivedere quest'ultima affermazione**). □

Quindi data una certa macchina che lavora in un certo spazio $S(n)$ posso costruire una macchina equivalente che da la stessa risposta in tempo limitato $T(n)$. Si ha che se ho un problema che si risolve in spazio polinomiale, per la formula appena scritta avrò tempo esponenziale. Invece, al contrario, tempo polinomiale comporta spazio polinomiale.

4.3.3 NonDeterministic Turing Machine

I problemi nella **classe EXP** sono ancora in studio per capire se sono **dimostrabilmente intrattabili**. Per il loro studio si usano le **NonDeterministic Turing Machine (NDTM)** che sono comunque puramente teoriche, in quanto non si è in grado fisicamente di costruirle. Nella DTM deterministica avevo una δ mentre nella NDTM ho:

$$\Delta \subseteq (k \times \Sigma) \times (k \times \Sigma \times \{\leftarrow, \rightarrow, -\})$$

Con Δ, detta **relazione di transizione**, che non è una funzione (come era δ) ma una relazione.
Nella DTM dato uno stato potevo passare ad un solo stato tramite δ, passando di stato in stato fino ad uno stato finale. Era una sequenza di transizioni. Nella NDTM ho delle **scelte**, ovvero la computazione non è una sequenza di computazioni ma un **albero di computazione**. Da ogni stato posso passare a uno tra più stati, a seconda della *scelta*, formando così un albero. Il singolo passo di computazione non è univocamente definito. Ogni singolo ramo comunque è equivalente al passo di computazione della DTM. Tramite Δ definisco il passaggio tra stati.
Per capire se una stringa è accettata o meno dalla NDTM, visto che potrei

andare sia in Y che in N (che in H) a seconda della varie computazioni dell'albero, si ha che:

Definizione 42. *Una stringa x è **accettata** da una NDTM se esiste una computazione tale per cui:*

$$(s_0, x, 1) \to (Y, z, \rho_G)$$

quindi se almeno un ramo dell'albero di computazione che termina nello stato Y. Tutti gli altri rami possono fare quello che vogliono, anche restare in loop. Se non c'è almeno un ramo Y non è accettata. Gli altri possono andare in loop o in uno stato N.

Definizione 43. *Un linguaggio L è **accettato** da una NDTM N se per tutte le stringhe che fanno parte del linguaggio esiste almeno una computazione che termina nello stato Y, ovvero esiste una computazione per cui:*

$$\forall x \in L \Rightarrow N(x) = Y$$

Definizione 44. *Un linguaggio L è **deciso** da una NDTM N se, qualora la stringa x appartenga il linguaggio, esiste **almeno una** computazione tale per cui:*

$$x \in L \to N(x) = Y$$

*altrimenti, se x non appartiene al linguaggio, per **tutte** le computazioni, si ha che:*

$$x \notin L \to N(x) = N$$

Non devo quindi avere loop in questo caso, tutte devono dare N.

Esempio 20. *Si ha:*

$$L = \{a^n b^{2n}\} \cup \{a^{2n} b^n\}$$

Studiamo una DTM che riconosca tale linguaggio. Dalle DTM controlla prima il primo formato senza cambiare i simboli. Se vede che le a sono più delle b passo al secondo step, dopo aver riconvertito i simboli in a e b. Dopo aver analizzato il secondo insieme restituisce Y o N. Tale funzione di transazione non è scontata.
Una NDTM potrebbe essere:

$$(s_0, x, 1)$$

che va in

- $(s_1, x, 1)$*, da cui parte lo studio del primo insieme*
- $(s_2, x, 2)$*, da cui parte lo studio del secondo insieme*

Data una stringa che non fa parte di L tutti i rami terminano con N. Altrimenti almeno un ramo porta a Y a seconda dei tipi di stringa accettati, se appartengono al primo o al secondo insieme.
La scelta del ramo non è definita formalmente, è solo una scelta. La macchina sa il ramo da scegliere dopo aver analizzato la stringa iniziale.

Teorema 29. *Una NDTM non è "più potente" di una DTM in termini di capacità di riconoscimento di un linguaggio. Se ho una NDTM che accetta un linguaggio sicuramente ho una DTM che lo fa.*

Dimostrazione. La NDTM non viene simulata da una DTM procedendo per rami, concludendoli, ma si procede per singoli passi su ogni ramo (onde evitare di incappare prematuramente in loop). Si scende quindi per livelli simulando poi tutti i rami (dovendo però "tornare indietro" ogni volta per passare al ramo successivo). Quindi muovendomi tra i rami posso simulare tramite una DTM quanto fa una NDTM. □

Le NDTM sono "più potenti" in termini di tempo di computazione.

Definizione 45. *La NDTM N decide il linguaggio L in tempo $T(N)$ se:*

- *N decide L*
- $\forall\, x \in \Sigma^*$ *se;*

  $$(s_0, x, 1) \to (s_G, z, \rho_G)$$

  in M^k passi, avendo $k \leq F(N)$

Trovo quindi uno Y in ogni caso in meno di $F(n)$ passi.
Ho quindi un ramo che in almeno k passi porta in Y, per ogni stringa accettata, se L è accettato. ***Rivedere definizione***

Il problema diventa quindi trovare il ramo giusto. Nei problemi dimostrabilmente esponenziali richiedono che il ramo Y sia lunghissimo mentre ci sono altri problemi per i quali tale ramo richiederebbe tempo polinomiale per arrivare a Y ma non si sa quale sia tale ramo da seguire, la soluzione diventa quindi provarli tutti, arrivando a tempi esponenziali.
Ponendo il limite $F(n)$ posso studiare che ogni singolo ramo non superi $F(n)$ quindi la NTDM impiega il tempo richiesto da un solo ramo per decidere e accettare, perché di passo in passo capisce su quale ramo non proseguire (???).
un altro modo di vedere la NDTM è pensare che parallelamente si sposti in tutti i livelli fino a cercare uno Y e se non lo trova risponda N. Il tempo è comunque in termini di livello dell'albero. La NDTM è una macchina **massicciamente parallela**.

Quindi la NTDM, per quanto non più potente in termini di riconoscimento, lo è in base al tempo, basandosi sui livelli dell'albero. Si parla di **guess&check**, per il metodo che individua il ramo da seguire e lo controlla, controllando lo stato Y.
Ci interessa quindi il tempo di verifica. Ci sono problemi difficili da risolvere e facili da verificare e altri che sono pure difficili da verificare.
La classe dei problemi facilmente verificabili di cui non si ha una facile risoluzione altro non è che la **classe NP**, che sono problemi polinomiali in modo non deterministico.

Definizione 46. *la **classe NP** corrisponde all'insieme di linguaggi L che sono decisi da una NDTM in tempo polinomiale:*

$$F(n) = O(p(n))$$

Si ha un rapporto tra P e NP.

Teorema 30. *Data una NDTM N che decide L in tempo $F(n)$ esiste una DTM M che decide L in tempo $O(d^{F(n)})$, dove d è una caratteristica particolare della NDTM, ovvero il **grado massimo di divisione dell'albero di computazione** di N.*

Dimostrazione. M ha vari nastri:

- il primo contiene N
- il secondo simula N
- il terzo contiene un numero in base d, inizialmente 1

La macchina fa un passo di simulazione di N, andando nel primo ramo ($d = 1$) a sinistra. A questo punto:

- trova Y e si ferma
- trova N quindi va avanti
- trova un altro stato, incrementa d e passa al ramo indicato da d

e così via. Nello step successivo farà due passi e non uno, per ogni ramo. Valutando quindi eventuali nuove ramificazioni per ogni ramo.
Se N arriva a Y allora troverò certamente uno Y per M. Se arriva a N dopo un po' tutti i rami terminano in N ma devo verificare la cosa e a quel punto M termina con N.

La simulazione richiede d passi per il primo livello, d^2 per il secondo, d^i per l'i-simo livello. Mi fermo a $d^{F(N)}$, al livello $F(n)$ e quindi ho tempo:

$$O(d^{F(N)})$$

capire meglio □

La NDTM è quindi più potente per i tempi, per quanto ne sappiamo infatti che se la NTDM lavora in $F(n)$, quindi in tempo polinomiale, la DTM che la simula lavora in $d^{F(n)}$, quindi in tempo esponenziale. Quindi la NDTM è "più veloce", anche se non lo sappiamo con certezza.
Rispetto al rapporto tra P e NP possiamo dire che $P \subseteq NP$, in quanto una NDTM è un caso generale della DTM (dove la relazione non ha mai scelte). Si ha inoltre che:

- ci sono problemi che non sappiamo dire se sono in P o meno ma sappiamo che sono in NP (vedasi test di primalità). Ma questo non aiuta rispetto agli altri problemi
- ci sono problemi che sappiamo essere in NP ma nessuno ha mai trovato la dimostrazione che sia in P

Capire qualcosa in merito ai problemi del secondo tipo potrebbe portare a $P = NP$ o $P \not\subseteq NP$. Questo è uno dei 10 problemi aperti della matematica ed è il più importante per l'informatica teorica.
A questo conosciamo quindi P, NP, EXP (poste in ordine di "è contenuto in"):

$$P \subseteq NP \subseteq EXP$$

Anche se $p \subseteq NP$ è al centro di studi.
Ordiniamo i problemi in base alla difficoltà in queste due classi, tramite le **riduzioni polinomiali** tra problemi, che ipotizziamo solo di decisione per praticità, $A \to B$. Cerco una procedura che per ogni istanza del problema A la trasforma in un'istanza per un problema diverso B. Quindi un'istanza di A, I_A, ha due risposte, Y o N, ma posso passare tramite una certa $f : I_A \to I_B$ avendo poi I_B tale che B avrà risposte, uguali a quelle di A, Y o N. Dato l'input, una stringa, x di A, ho che lo trasformo in input per B potendo dire se la stringa appartiene o meno al linguaggio associato ad A, mascherando il passaggio a B. La cosa ha tempo pari alla somma tra il tempo della trasformazione più quello dell'algoritmo B.
Vediamo, ad esempio, se possiamo risolvere il problema del ciclo hamiltoniano usando TSP, entrambi decisionali. Partendo da un'istanza del grafo per il ciclo hamiltoniano creo un'istanza con gli stessi vertici, aggiungo gli archi per

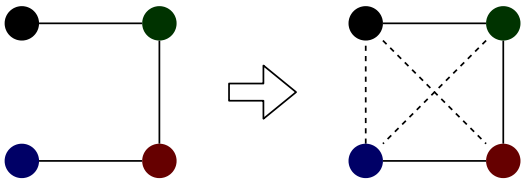


Figura 4.2: Esempio di riduzione da ciclo hamiltoniano a TSP, con archi tratteggiati di peso 2 e archi normali di peso 1. Il colore dei vertici specifica che dalle due parti della trasformazione si ha lo stesso vertice

ottenere un grafo completo (visto che TSP ha un grafo completo mentre ciclo hamiltoniano no). Aggiungo anche i pesi (TSP ha un grafo pesato mentre ciclo hamiltoniano no), mettendo il peso a 1 per i vecchi archi e K, con $k \neq 1$, a quelli appena aggiunti. Manca quindi solo il bound, che è $B = |V|$ per il ciclo hamiltoniano che ipotizziamo con archi solo pesati a 1 ma quindi in TSP non posso usare quelli aggiunti in quanto uscirei per forza dal bound se usassi uno degli archi aggiunti, basta $K = 2$. Ho un ciclo hamiltoniano sse $B = |V|$ anche con TSP, che avrà un giro con gli stessi archi del ciclo hamiltoniano presente nel primo problema, posto che esista, avendo che se TSP rende Y sse $B = |V|$ allora avrò un ciclo hamiltoniano nel problema iniziale, avendo Y in risposta anche per questo. Ho quindi ridotto il ciclo hamiltoniano a TSP, e possiamo dire che quest'ultimo è più difficile di ciclo hamiltoniano. Per esempio vedere figura 4.2 Controlliamo quindi i tempi. Voglio un costo della riduzione polinomiale e si ha che l'aggiunta di archi, pesi e bound sono operazioni polinomiali.

Definizione 47. *Un problema B è più difficile di un problema A, che ha in input la stringa x, sse:*

$$A \to B$$

infatti ho:

$$T(A(x)) \leq F(x) + T(B(x))$$

Definizione 48. *Definiamo formalmente riduzione polinomiale tra un linguaggio L_1 e un linguaggio L_2 come:*

$$f : \Sigma_1^* \to \Sigma_2^*$$

tale che:

$$x \in L_1 \iff f(x) \in L_2$$

con f calcolabile in tempo polinomiale ($O(|x|)$) dalla DTM. SI ha quindi che:

$$L_1 <_T L_2$$

Teorema 31. *Se si ha che $L_1 <_T L_2$ e si ha che $L_2 \in P$ allora sicuramente $L_1 \in P$.*
Se avessi avuto $L_2 \in NP$ o $L_2 \notin P$ non avrei informazioni su L_1.

Dimostrazione. Infatti e esiste un algoritmo polinomiale per L_2 allora, avendo una trasformazione polinomiale ho che $f(x)+L_2$ è ancora polinomiale. □

Teorema 32. *Se si ha che $L_1 <_T L_2$ e si ha che $L_1 \notin P$ allora ho che $L_2 \notin P$.*

Dimostrazione. Basta vedere "al contrario" il teorema precedente. □

Le riduzioni godono di proprietà:

- riflessiva, $A \to A$
- transitiva, $A \to B \ \wedge \ B \to C \implies A \to C$, che essendo due trasformazioni in tempo polinomiale avrò comunque una trasformazione polinomiale

Le riduzioni polinomiali **non sono sempre simmetriche** e quindi non sono una relazione di equivalenza.
Abbiamo però una classe di equivalenza interna ai problemi NP, dove i problemi contenuti sono i più difficili di NP e si riducono l'uno all'altro. Tali problemi sono i problemi **NP-complete**.

Definizione 49. *Un linguaggio è **NP-complete** sse:*

- $L \in NP$
- $L' <_T L, \forall\, L' \in NP$, *tutti i linguaggi in NP si riducono a* $L \in NP - complete$ *e quindi i problemi in* $NP - complete$ *sono i più difficili di* NP

Se ne risolvessi uno in tempo polinomiale risolverei tutti i problemi in NP in tempo polinomiale (ma ormai è assunto che non possa succedere anche se non si può dimostrare).

Teorema 33. *Ho che $P = NP$ sse esiste un linguaggio L tale che:*

$$L \in NP - complete \cap P$$

Dimostrazione. Se $P = NP$ allora sappiamo già che sarebbero polinomiali.
Per l'altro verso ho che:

$$\forall L' \in N,\ L <_T L$$

e quindi esisterebbe un algoritmo polinomiale che risolve L per una DTM e quindi avrei un algoritmo polinomiale per ogni $L' \in NP$ per una DTM. □

Quindi se trovassi un algoritmo polinomiale per un NP-complete lo avrei per tutti gli NP.

Teorema 34. *Se esiste un $L \in NP$ ma $L \notin P$ allora:*

$$\forall L'' \in \textit{NP-complete}$$

ho che $L'' \notin P$.

Dimostrando quindi che $P \neq NP$.
Andrebbe benissimo avere le prove di una di queste due situazioni, ma non esiste ancora.
Per gli NP-complete vale anche la simmetria in quanto ogni problema NP-complete è riducibile ad ogni altro problema NP-complete. Riprendendo l'esempio sopra so che ciclo hamiltoniano è TSP, anche nelle forme decisionali, e quindi so che TSP decisionale è NP-complete. La difficoltà è trovare almeno un problema NP-complete, poi saprò che ogni altro a cui posso ridurlo è NP-complete.
Conosciamo moltissimi problemi NP-complete, molti riconosciuti tramite riduzioni ma il primo è ottenuto tramite il **teorema di Cook**:

Teorema 35 (teorema di Cook)**.** *SAT (che è stato già definito), che richiede un assegnamento di verità sulla formula ϕ in base all'assegnamento delle variabili $x_1, \ldots, x_n$, è NP-complete.*

Dimostrazione. La dimostrazione completa è complessa ma vediamo qualche spunto.
Si ha che SAT può essere risolto in tempo esponenziale, $O(2^n)$, provando tutti i possibili assegnamenti di verità possibili.
Posso usare inoltre una NDTM che fa tutti i rami in parallelo, con ogni possibilità di assegnamento fatta in un ramo oppure posso dire che "spara" un assegnamento a caso, che si suppone giusto, su un ramo e verifica che è giusto. In ogni caso con una NDTM avrei tempo polinomiale, $O(n \cdot m)$ per n variabili e m clausole.
Per vedere che ogni problema NP, per semplicità decisionali, si riduce a SAT vedo che tutti i problemi in NP hanno in comune di avere un NDTM che li risolve in tempo polinomiale, che decide il linguaggio associato. Basta quindi dire che $\forall \Pi' \in NP$ ha associato una NDTM N'_Π che lavora in tempo polinomiale e mi basta vedere che ogni NDTM che lavora in tempo polinomiale si può trasformare in una formula CNF, presa in ingresso da SAT, sfruttando poi gli stati finali della computazione di SAT. Se SAT dice che è soddisfacibile allora lo è il problema iniziale e quindi il problema è riducibile a SAT. □

SAT è stato il primo problema identificato come NP-complete ed è stato usato per riconoscere gli altri problemi NP-complete.
La descrizione di una NDTM deve quindi corrispondere ad una ϕ di SAT per far vedere che tutti i problemi NP si riducono a SAT. Si avrà una variabile per ogni stato, una variabile per ogni carattere e una variabile per la posizione della testina. Queste variabili saranno quelle che comporranno la ϕ e il loro assegnamento comporterà la veridicità della formula.
Bisogna far vedere che $SAT <_T \Pi$ per far vedere che Π è NP-complete. Facendo poi vedere che un altro problema si riduce a Π o SAT dimostro che anche questo secondo problema è NP-complete.
Si ha che il problema del ciclo hamiltoniano e TSP, decisionali, siano NP-complete.
Vediamo che $SAT <_T 3SAT$, che ha tre letterali in ogni clausola. Ho in input una formula ϕ che deve diventare una ϕ_3 tale che ϕ è soddisfacibile sse ϕ_3 lo è. La formula ϕ ha tante clausole in congiunzione tra loro con un numero arbitrario di letterali ma ϕ_3 deve averne altrettante ma con soli tre letterali per clausola. Qualora si abbiano clausole in ϕ con un solo letterale si aggiungono due letterali (che indichiamo con z) in modo però che la veridicità sia basata solo sulla variabile iniziale (questo vale per tutte le trasformazioni che stiamo per mostrare, ovvero la veridicità deve valere in base solo alle variabili iniziali della formula ϕ). Tale clausola che conteneva solo x_1 diventa una quadrupla clausola:

$$(x_1 \vee z_1 \vee z_2) \wedge (x_1 \vee \neg z_1 \vee z_2) \wedge (x_1 \vee z_1 \vee \neg z_2) \wedge (x_1 \vee \neg z_1 \vee \neg z_2)$$

Se invece avessi una formula con due letterali diventa, aggiungendo una sola variabile:

$$(x_1 \vee x_2 \vee z_1) \wedge (x_1 \vee x_2 \vee \neg z_1)$$

Una formula a tre letterali resta ovviamente invariata mentre quelle a più letterali vanno spezzate (aggiungendo poi variabili). Per esempio, avendo quattro letterali, otterrei:

$$(x_1 \vee x_2 \vee z_1) \wedge (\neg z_1 \vee x_3 \vee z_2) \wedge (\neg z_2 \vee x_4 \vee z_3) \wedge \cdots \wedge (z_{n-3} \vee x_{n-1} \vee x_n)$$

Si vede che ogni formula di SAT può essere trasformata in una di 3SAT e quindi anche quest'ultimo è NP-complete. Si dimostra che invece 2SAT non è NP-complete ma ogni KSAT, con $K > 2$, è NP-complete.
Anche, per esempio, il *problema della colorabilità* è NP-complete, in quanto può essere ridotto a SAT (non vedremo come).
Torniamo a parlare di TSP, non di decisione. Ho capito che serve tempo esponenziale per risolverlo quindi si può pensare che sia NP-complete. Passo

quindi alla versione di decisione. Vedo però che ciclo hamiltoniano, che è NP-complete, si riduce a TSP decisionale che quindi è anch'esso NP-complete. Abbiamo comunque che TSP è più difficile del suo problema decisionale, infatti D-TSP si riduce a TSP. Si ha quindi che TSP è **NP-hard**, ovvero tutti i problemi NP si riducono ad esso ma il problema stesso non è in NP (essendo un problema di ottimo e quindi non gli posso associare un linguaggio) e quindi non è NP-complete (non essendo in NP). Per TSP servirebbe una **macchina di Turing ad oracolo**.

4.3.4 Rapporto spazio e tempo

Aggiungiamo qualcosa a quanto già detto nelle sezioni precedenti.

Definizione 50. *Definisco la* ***funzione di complessità spaziale*** *come:*

$$f : \mathbb{N} \to \mathbb{N}$$

con $f(n)$ che è il massimo numero di caselle utilizzate da una TM per decidere un certo linguaggio L tutte le istanze di una certa dimensione, $n = |x|$.

Sappiamo che, dal punto di vista del tempo (in notazione si ha $dtime = time$):

$$P = \bigcup_{k \geq 1} dtime(n^k)$$

$$NP = \bigcup_{k \geq 1} ntime(n^k)$$

Definizione 51. *Definiamo la* ***classe DSPACE*** *come l'insieme di tutti i linguaggi tali che sono decisi da una TM in spazio $O(f(n))$. Si ha che;*

$$PSPACE = \bigcup_{k \geq 1} dspace(n^k)$$

avendo quindi limite polinomiale.

Definizione 52. *Definiamo la* ***classe NSPACE*** *come l'insieme di tutti i linguaggi tali che sono decisi da una NDTM in spazio $O(f(n))$. Si ha che:*

$$NPSPACE = \bigcup_{k \geq 1} nspace(n^k)$$

avendo quindi limite polinomiale.

Come vale:

$$P \subseteq NP$$

Vale che:

$$PSPACE \subseteq NPSPACE$$

e per dimostrarlo mi concerto su una macchina che si preoccupa solo dello spazio e non del tempo.
Prendiamo nuovamente il problema SAT e quindi ragioniamo sulla NDTM. Vediamo che SAT è in NPSPACE per ovvi motivi (capisco che ramo scegliere e tale ramo ha una casella per letterale, avendo spazio n) ma vediamo se appartiene anche a PSPACE. Una DTM potrebbe provare tutte le possibilità in tempo esponenziale ma in spazio polinomiale in quanto si sovrascrivono ad ogni tentativo le varie caselle indicanti gli assegnamenti di verità ma la cardinalità di tali caselle è sempre la stessa, una per ogni letterale, ovvero n. Quindi ho che $SAT \in PSPACE$ anche se non è un'informazione così eclatante a causa del tempo esponenziale.

Teorema 36 (Teorema di Savitch)**.** *Si dimostra che:*

$$PSPACE = NSPACE$$

Quindi la DTM e la NDTM hanno la stessa potenza dal punto di vista dello spazio di calcolo.

Dimostrazione. Un problema abbiamo visto essere definito come **configurazione raggiungibile**, ovvero se una macchina può andare da una configurazione iniziale c_i ad una finale c_f in un certo numero di passi t, indicato con:

$$(c_i, C_f, t)$$

Prendo una configurazione intermedi c_m e vedo se vale:

$$\left(c_i, C_m, \frac{t}{2}\right) \wedge \left(c_m, C_f, \frac{t}{2}\right)$$

Spezzo poi in modo D&I di volta in volta, vedendo ricorsivamente se entrambi i rami sono validi.
Vedo che lo stack delle chiamate ricorsive ha una certa profondità. Una NDTM andrebbe in spazio $f(n)$ ha associata una DTM che lavora in spazio $O(f^2(n))$ in quanto salvo le varie configurazioni dello stack ricorsivo, avendo $2^{\log f(n)} = f(n)$.
(**rivedere tutta la dimostrazione, quanto scritto è errato in buona parte!**)

□

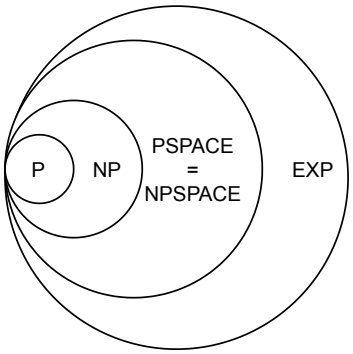


Figura 4.3: Diagramma di Venn delle classi fin'ora descritte

Abbiamo quindi $PSPACE = NPSPACE$ e che $P \subseteq NP$ quindi certamente:

$$P \subseteq PSPACE$$

$$NP \subseteq NPSPACE$$

Si ha quindi la figura 4.3. Abbiamo anche le classi CO_P e CO_NP ovvero le classi dei problemi complementari P e NP. Si ha che, a causa del determinismo:

$$CO\_P = P$$

ma quando si introduce il non determinismo le cose cambiano. La domanda iniziale era del tipo *esiste almeno uno* quindi il completamento è *non esiste nessuno*, quindi devo verificare tutti i casi, andando in un caso che nemmeno la NDTM riesce a risolvere, si finisce infatti nella classe EXP. Quindi si ha che:

$$P \subseteq CP\_NP \wedge P \subseteq NP$$

non avendo un rapporto diretto e definito tra NP e CO_NP.

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 53. *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 54. *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 21. *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 55. *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 22. *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 56. *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 23. *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 57. *Definiamo* ***bordo*** *della stringa* X*,* $B(X)$*, che è il più lungo prefisso proprio che occorre come suffisso di* X*.*

Esempio 24. *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 58. *Definiamo concatenazione tra una stringa X e un simbolo σ come: $X\sigma$*

Esempio 25. *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:

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

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:

$$\delta(j, \sigma) = j + 1 \iff j < m \wedge P[j + 1] = \sigma$$

2. secondo caso:

$$\delta(j, \sigma) = k \iff P[j + 1] \neq \sigma \vee j = m, \text{ con } k = |B(P[1, j]\sigma)|$$

   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.

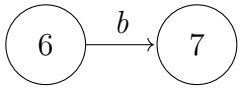
La transizione dallo stato j allo stato $\delta(j, \sigma)$ avviene attraverso il simbolo σ. Posso quindi passare allo stato $j + 1$ o allo stato k.
Nell'automa avremo quindi gli archi etichettati coi simboli e i nodi etichettati con gli indici da 0 a m.
Nella stringa gli indici partono comunque da 1 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 26. *Prendiamo il pattern:*

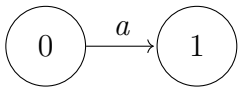
$$P = acacbabbaabac$$

calcoliamo:

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

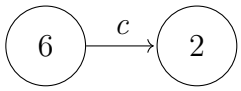


- $\delta(0, a) = 1$*, in quanto* a *è il primo carattere, uso quindi la prima formula*



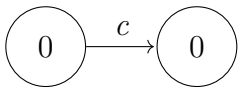
- $\delta(6, c) = 2$ *in quanto devo usare la seconda formula e avere:*

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



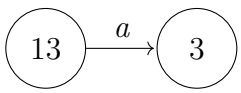
- $\delta(0, c) = 0$ *in quanto devo usare la seconda formula e avere:*

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



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

$$|B(P[1, 13]a)| = |B(acacbabbaabaca)| = |aca| = 3$$



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 27. *Se ho $P = aaaaaa$:*

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

in quanto:

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

Esempio 28. *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 29. *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

Esempio 30. *Individuare un pattern sconosciuto di 6 caratteri e la tabella incompleta con solo $\delta(j, \sigma) = j + 1$:*

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

Quindi banalmente il pattern è:

$$P = acbaad$$

Completo quindi la tabella con $\delta(j, \sigma) = k$:

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

Esempio 31. *Individuare un pattern sconosciuto di 6 caratteri e 3 simboli e la tabella incompleta:*

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

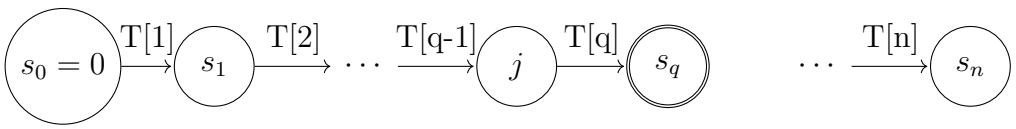
specifico i primi tre caratteri e gli ultimi due (il quarto non è ritrovabile). Quindi il pattern è:

$$P = cac.ca$$

per gli ultimi due sappiamo che avendo bordo 3 quindi gli ultimi due sono sicuro ca. Per le prime due uso la classica formula $\delta(j, \sigma) = j+1$. *Per il terzo sfrutto l'ultima cella che ha segnato 3, che non è uno stato successivo ma ho il passaggio dallo stato 6 al 3 con c, implicando la lunghezza del bordo che calcolo sul pattern intero con c aggiunto è lungo 3 e quindi i primi tre simboli del pattern si ripetono negli ultimi 3 di questo pattern a cui è concatenato c. Quindi in 3 avrò c e in 5 e 6 ca. Sul quarto carattere non posso dire nulla.*

Vediamo quindi l'uso della funzione di transizione per trovare tutte le occorrenze esatte nel testo. Si scandisce il testo per scandire il testo. Ricordiamo che il preprocessamento (che non vedremo) ha tempo $O(m|\Sigma|)$ mentre la scansione del testo ha tempo $O(n)$ in T. Il procedimento è:

- si parte dallo stato iniziale $s_0 = 0$
- il testo T viene letto dalla posizione 1 alla n
- per ogni posizione q da 1 a n è associato uno stato s_q ottenuto con la transizione dallo stato precedente s_{q-1} tramite il simbolo $T[q]$:



Si hanno due supposizioni:

1. si suppongache j sia la lunghezza del più lungo prefisso di P che ha un'occorrenza in T che finisce in posizione $q-1$ (e che inizia quindi in posizione $q-j$)

2. si supponga che j sia lo stato s_{q-1} a cui l'algoritmo arriva dopo aver letto il simbolo $T[q-1]$

A questo si passa a $s_q = \delta(j, T[q])$. Il nuovo stato s_q deve essere calcolato.
Il calcolo di $s_q = \delta(j, T[q])$ si divide in due casi (anche se in realtà poi sono tre):

1. $j < m \ \wedge \ P[j+1] = T[Q]$, significa avere in $j+1$ lo stesso simbolo che ho appena letto:

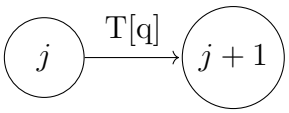
   Esempio 32. *Si ha:*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T=	a	a	a	b	a	c	c	a	b	a	c	c	c	a	c	b	a

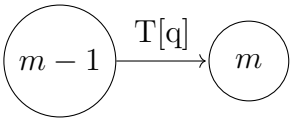
   matchando dal carattere 2 di T:

	1	2	3	4	5	6	7	8
P=	a	b	a	c	c	a	b	.

   Il prefisso $P[1, j+1]$ (quindi quando ho $j+1 = m$) si ha un'occorrenza di T che inizia in $q-(j+a)+1$, dopo aver letto il simbolo $T[q]$:



   Ho quindi che **il prefisso** $P[1, m] = P$ **ha un'occorrenza su** T **che inizia in posizione** $q-m+1$:



2. $j < m \ \vee \ P[j+1] \neq T[Q]$, che a sua volta si divide in:

   (a) $j < m \ \wedge \ P[j+1] \neq T[Q]$ ad esempio:

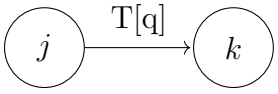
Esempio 33. *Si ha:*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T=	a	a	a	b	a	c	c	a	b	a	c	c	c	a	c	b	a

matchando dal carattere 2 di T:

	1	2	3	4	5	6	7	8
P=	a	b	a	c	c	a	c	.

Si ha, con il bordo $k = B(P[1, j]T[q])$:



Quindi **il prefisso** $P[1, k]$ **ha occorrenza su** T **che inizia con il carattere** $(q - k + 1)$ **del testo**. Faccio quindi un salto

(b) $j = m$, ad esempio:

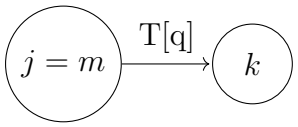
Esempio 34. *Si ha:*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T=	a	a	a	b	a	c	c	a	b	a	c	c	c	a	c	b	a

matchando dal carattere 2 di T:

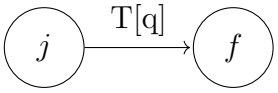
	1	2	3	4	5	6	7	8
P=	a	b	a	c	c	a	b	a

Ho quindi un'occorrenza di P in T che finisce in $q - 1$. Il prefisso $P[1, k]$ ha occorrenza in T che un'occorrenza che inizia in $q - m$ e una che inizia in $q - k + 1$. Lo stato iniziale m indica un'occorrenza di P in T. Lo stato di arrivo k è sicuramente minore o uguale a m. Se $k = m$, allora esiste una nuova occorrenza di P in T che risulta essere sovrapposta con la precedente di $m - 1$ simboli. **Il prefisso** $P[1, k]$ **ha occorrenza in** T **che un'occorrenza che inizia in** $q - k + 1$



Per rendere più semplice lo studio sono in grassetto i punti chiave dei tre casi.

Gli ultimi due casi possono essere riassunti con $j \geq m$ e $f \geq j$:



Il prefisso di P di lunghezza f occorre in T in posizione $q - f + 1$ e se $f = m$, allora P occorre in T alla posizione $q - m + 1$.
In sostanza quindi l'algoritmo procede con tre step:

- Si parte dallo stato iniziale $s_0 = 0$ e si effettua una scansione di T dal primo all'ultimo simbolo
- per ogni posizione q di T si effettua la transizione dallo stato corrente s al nuovo stato $f = \delta(s, T[q])$
- ogni volta che il nuovo stato f coincide con $m = |P|$, viene prodotta in output l'occorrenza $q - m + 1$

Avendo quindi lo pseudocodice: Si ha quindi complessità lineare nella lun-

```
function SCAN-TEXT(δ, T, m)
    n ← |T|
    s ← 0
    for q ← 1 to n do
        f ← δ(s, T[q])
        if f == m then
            print q − m + 1
        s ← f
```

ghezza del testo sia nel caso migliore che peggiore.

Esempio 35. *Vediamo un esempio di scansione del testo dati:*

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	c	a	b	a	c	a	c	b	a	c	a	b	a

	1	2	3	4	5	6	7
P=	a	c	a	c	b	a	c

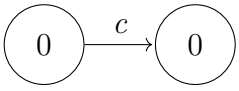
Sono dati, secondo la solita divisione:

$\delta(i,\sigma)$	a	b	c
0	1		
1			2
2	3		
3			4
4		5	
5	6		
6			7
7			

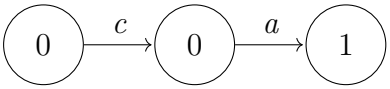
$\delta(i,\sigma)$	a	b	c
0	1	0	0
1	1	0	2
2	3	0	0
3	1	0	4
4	3	5	0
5	6	0	0
6	1	0	7
7	3	0	0

inizio con solo lo stato iniziale:

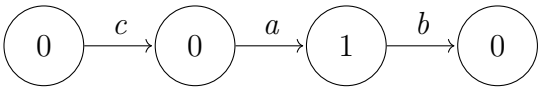
leggo c seguo la tabella e vado in 0 *da* 0*:*



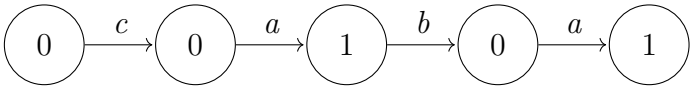
leggo a seguo la tabella e vado in 1 *da* 0*:*



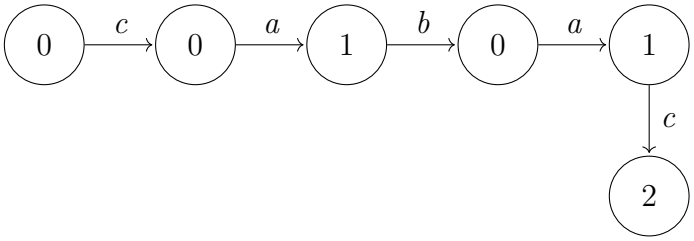
leggo b seguo la tabella e vado in 0 *da* 1*:*



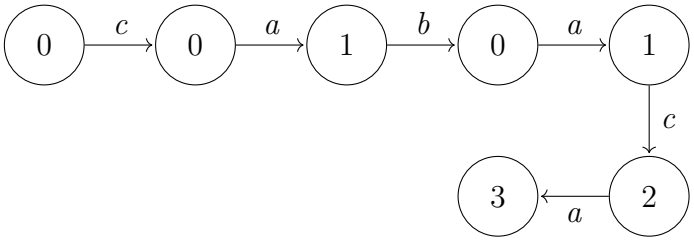
leggo a seguo la tabella e vado in 1 *da* 0*:*



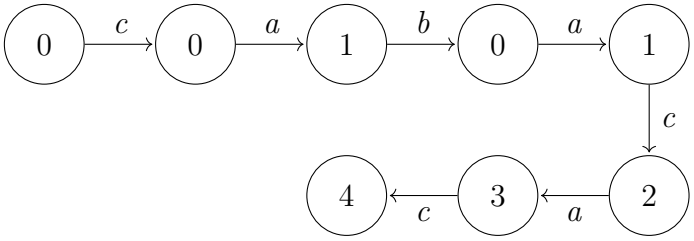
leggo c seguo la tabella e vado in 2 *da* 1*:*



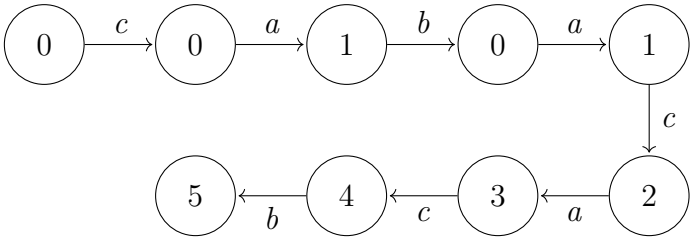
leggo a seguo la tabella e vado in 3 *da* 2*:*



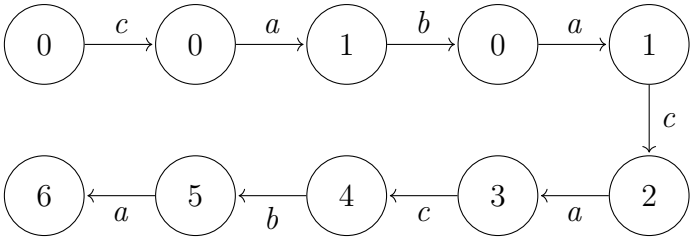
leggo c seguo la tabella e vado in 4 *da* 3*:*



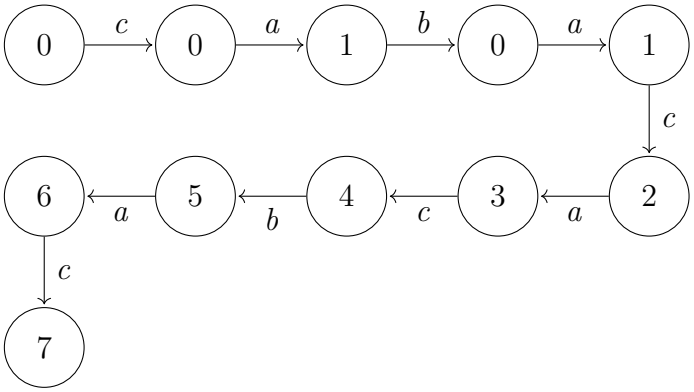
leggo b seguo la tabella e vado in 5 *da* 4*:*



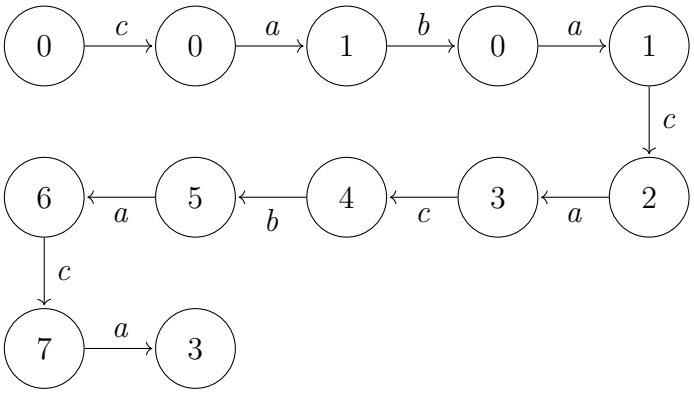
leggo a seguo la tabella e vado in 6 *da* 5*:*



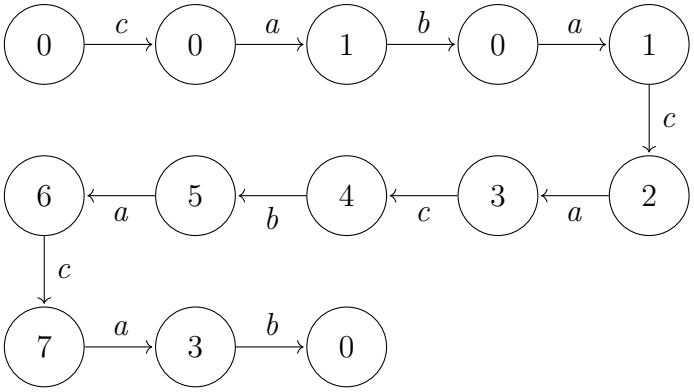
leggo c seguo la tabella e vado in 7 *da* 6*:*



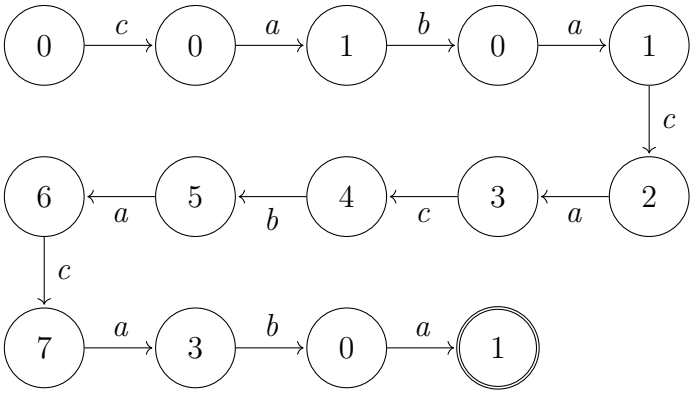
Essendo arrivati allo stato $m = 7$*, esiste un'occorrenza di* P *in posizione* $10 - 7 + 1 = 4$*.*
leggo a seguo la tabella e vado in 3 *da* 7*:*



leggo b seguo la tabella e vado in 0 *da* 3*:*



leggo a seguo la tabella e vado in 1 *da* 0*:*



e mi fermo avendo concluso la scansione del testo. Ho trovato quindi solo una occorrenza che inizia all'indice 4.

5.2 Algoritmo Knuth-Morris-Pratt

Vediamo ora l'**algoritmo Knuth-Morris-Pratt (*KMP*)**.
Anche in questo caso si ha un preprocessing del pattern in tempo lineare alla lunghezza del pattern $O(m)$. Si calcola una funzione di fallimento φ per avere il preprocessamento. Si ha poi la scansione del testo in tempo $O(n)$.

Definizione 59. *Definiamo la funzione di fallimento, detta prefix function* φ*:*

$$\varphi : \{0, 1, \ldots, \mathrm{m}\} \to \{-1, 0, 1, \ldots, \mathrm{m} - 1\}$$

tale che:

- $\varphi = |B(P[1,j])|$ *se* $1 \leq j \leq m$
- $\varphi = -1$ *se* $j = 0$

Esempio 36. *Dato il pattern lungo 13:*

	1	2	3	4	5	6	7	8	9	10	11	12	13
P=	a	b	c	a	b	a	a	b	c	a	b	a	b

ottengo:

	1	2	3	4	5	6	7	8	9	10	11	12	13
φ=	-1	0	0	0	1	2	1	1	2	3	4	5	2

Ci serve una definizione ricorsiva di bordo:

Definizione 60. *Si hanno 4 punti:*

1. *il bordo della stringa vuota è la stringa vuota stessa:*
$$B(\varepsilon) = \varepsilon$$
2. *il bordo di una stringa di un solo carattere è la stringa vuota:*
$$B(\sigma) = \varepsilon$$
3. *il bordo* $B(X\sigma)$ *della concatenazione tra una stringa* X *e il simbolo* σ *è:*
$$B(X\sigma) = B(X)\sigma$$
4. *il bordo* $B(X\sigma)$ *della concatenazione tra una stringa* X *e il simbolo* σ *è anche:*
$$B(X\sigma) = B(B(X)\sigma)$$

Definizione 61. *Vediamo il calcolo di* ϕ *per induzione.*
I primi passi sono immediati:

$$\phi(0) = -1$$

$$\phi(1) = 0$$

Possiamo quindi calcolare $\phi(j)$ *per* $j > 1$ *supponendo che* ϕ *sia nota per tutti gli indici in* $[0, j-1]$.
Si ha:
$$B(P[1,j]) = B(P[1,j-1]P[j])$$

e diciamo che:

$$k_1 = |B(P[1, j-1])| = \phi(j-1)$$

Scriviamo anche:

$$B(P[1, j-1]) = P[1, k_1]$$

e chiamiamo:

$$k_2 = |B(P[1, k_1])| = \phi(k_1) = \phi(\phi(j-1))$$

Si hanno due casi:

1. *si ha* $P[k_1 + 1] = P[j]$ *quindi si ha:*

   $$B(P[1, j]) = B(P[1, j-1])P[j]$$

   Ne segue che:

   $$|B(P[1, j])| = |B(P[1, j-1])| + 1$$

   Ma sappiamo che:

   $$|B(P[1, j])| = |B(P[1, j-1])| + 1 = k1 + 1 = \phi(j-1) + 1$$

   e quindi:

   $$|B(P[1, j])| = \phi(j-1) + 1$$

   e quindi possiamo dire che:

   $$\phi(j) = \phi(j-1) + 1$$

2. *si ha* $P[k_1 + 1] \neq P[j]$ *quindi:*

   $$B(P[1, j]) = B(B(P[1, j-1])P[j])$$

   e sostituendo con $B(P[1, j-1]) = P[1, k_1]$*:*

   $$B(P[1, j]) = B(P[1, k_1]P[j])$$

A questo punto dovrei separare due sottocasi per il secondo caso:

1. $P[k_2 + 1] = P[j]$
2. $P[k_2 + 1] \neq P[j]$

e andrei avanti ricorsivamente (su slide approfondimento anche su secondo livello di ricorsione).
In generale possiamo dire che aggiungere un valore k_p tale per cui:

$$P[k_p + 1] = P[j]$$

implica che:

$$\phi(j) = k_p + 1$$

mentre trovare tutti i valori k_p tali per cui:

$$P[k_p + 1] \neq P[j]$$

implica che a un certo punto si raggiunge un $k_p = -1$, che implica che il bordo di $P[1, j] = 0$ e quindi:

$$\phi(j) = 0 = k_p + 1$$

Riassumo quindi la procedura:

- *inizializzo $\phi(0) = -1$ e $\phi(1) = 0$ e $j = 2$ (per gli argomenti di ϕ che sono quindi maggiori di 1)*
- *setto $k = \phi(j - 1)$*
- *divido nel due casi:*

    1. *se $P[k + 1] = P[j]$:*
        - *setto $\phi(j) = k + 1$*
        - *$j = j + 1$*

       e ricomincio
    2. *se $P[k + 1] \neq P[j]$:*
        - *setto $k = \phi(k)$*
            - *se $k = -1$ setto $\phi(j) = k + 1$ e $j = j + 1$ e ricomincio*
            - *altrimenti torno a testare i due casi*

Possiamo quindi scrivere lo pseudocodice:

```
function PREFIX(P)
    m ← |P|
    φ(0) ← −1
    φ(0) ← 0
    for j ← 2 to m do
        k ← φ(j − 1)
        while k ≥ 0 ∧ P[k + 1] ≠ P[j] do
            k ← φ(k)
        φ(j) ← k + 1
    return φ
```

su slide esempio di esecuzione

Passiamo quindi alla scansione del testo.
La scansione del testo T alla ricerca di P avviene tramite una finestra W di confronto lunga m simboli che scorre lungo T da sinistra a destra, inizialmente posizionata in corrispondenza della posizione $i = 1$ di T. Per ogni posizione i di W, ogni simbolo di P viene confrontato con il corrispondente simbolo in W, procedendo da sinistra a destra. Se tutti i simboli sono confrontati con successo, allora nella posizione i di T esiste un'occorrenza di P. La finestra viene poi spostata verso destra in una nuova posizione j e si ripete il confronto.
La finestra viene spostata a destra utilizzando la funzione ϕ. Il confronto riparte dal simbolo del testo che aveva dato un mismatch con il pattern nella precedente posizione di W, avendo quindi una scansione in $O(n)$.
Studiamo meglio lo spostamento di W.
Se ho W in i su T e ho il primo carattere di P che determina un mismatch con T in posizione $j > 1$ ho che il simbolo di mismatch su T è in $i + j - 1$.
Per definizione sappiamo che $k = phi(j - 1)$ è la lunghezza del bordo del prefisso $P[1, j - 1]$. Si ha quindi che:

- $P[1, k]$ ha un'occorrenza su T che inizia in posizione i
- $P[1, k]$ ha un'occorrenza su T che inizia in posizione

$$p = i + j - k - 1 = i + j - \phi(j - 1) - 1$$

Quindi il salto di W da i a p garantisce di non perdere occorrenze di P e garantisce che i primi k simboli di P hanno un matching con T a partire

dalla nuova posizione p. Posso quindi far ripartire il contorno per la nuova posizione p di W dalle posizioni:

- $i + j - 1$ su T
- $k + 1$ su P

Sposto quindi W a $p = i + j - \phi(j - 1) - 1$ e i primi $k = \phi(j - 1)$ simboli di P non vengono più confrontati con i corrispondenti simboli su T nella nuova posizione di W.
Si hanno dei casi particolari:

- se $\phi(j-1) = 0$ sposto W a $p = i+j-1$ e i primi $k = 0$ simboli di P non vengono più confrontati con i corrispondenti simboli su T nella nuova posizione di W quindi il confronto riparte da $i+j-1$ su T e dal primo simbolo su P
- se $j = 1$ sposto W a $p = i + 1$ e i primi $k = -1$ simboli di P non vengono più confrontati con i corrispondenti simboli su T nella nuova posizione di W quindi il confronto riparte da $i + 1$ su T e dal primo simbolo su P
- se $j = m + 1$ sposto W a $p = i + m - \phi(m)$ e i primi $k = \phi(m)$ simboli di P non vengono più confrontati con i corrispondenti simboli su T nella nuova posizione di W quindi il confronto riparte da $i + m$ su T e dal simbolo in posizione $\phi(m) + 1$ su P

Questo algoritmo, tramite uno studia di complessità ammortizzata, nonostante il doppio ciclo, ha complessità $O(n)$.
Su slide esempio di esecuzione.

Algorithm 2 Algoritmo Knuth-Morris-Pratt

```
function KMP(P, T, φ)
    m ← |P|
    n ← |T|
    j ← 0
    for q ← 1 to n do
        while j ≥ 0 ∧ P[j + 1] ≠ T[1] do
            j ← φ(j)
        φ(j) ← j + 1
        if j = m then
            return q − m + 1
```

5.3 Algoritmi shift-and

5.3.1 Algoritmo di Baeza-Yates e Gonnet

Con questo algoritmo si ha il preprocessing del pattern P di lunghezza m in tempo $O(|\Sigma| + m)$. Questo preprocessamento avviene tramite il calcolo di una tabella B di $|\Sigma|$ word di di bit B_σ, ognuna di lunghezza m. Alla fine la scansione del testo T di lunghezza n è in $O(n)$.
Si sfrutta il paradigma **shift-and (esatto)** confrontando quindi delle word binarie e non dei simboli.

Definizione 62. *L'operazione di* ***AND*** *definita su due word:*

$$w_1 \ \textbf{\textit{AND}} \ w_2 = w$$

produce:

- $w[i] = 1$ *sse* $w_1[i] \ \wedge w_2[i] = 1$
- $w[i] = 0$ *altrimenti*

Definizione 63. *L'operazione di* ***or*** *definita su due word:*

$$w_1 \ \textbf{\textit{or}} \ w_2 = w$$

produce:

- $w[i] = 1$ *sse* $w_1[i] \ \vee w_2[i] = 1$
- $w[i] = 0$ *altrimenti*

Definizione 64. *L'operazione di* ***RSHIFT*** *(shift a destra con MSB a 0) definita su una word:*

$$\textbf{RSHIFT} = w$$

produce:

- $w[1] = 0$
- $w[i] = w_1[i-1]$ *con* $2 \leq i \leq |w|$

In pratica è lo spostamento dei bit di una posizione verso destra con bit più significativo a 0 (quindi visto che avrei il buco a sinistra lì metto 0).

Definizione 65. *L'operazione di* ***RSHIFT1*** *(shift a destra con MSB a 1) definita su una word:*

$$\textbf{RSHIFT} = w$$

produce:

- $w[1] = 1$
- $w[i] = w_1[i-1]$ *con* $2 \leq i \leq |w|$

In pratica è lo spostamento dei bit di una posizione verso destra con bit più significativo a 1 (quindi visto che avrei il buco a sinistra lì metto 1).

Si ha quindi che ogni word B_σ è una word di m bit $\{0, 1\}$:

$$B_\sigma = b_1 b_2, \ldots b_m$$

tale che:

$$b_i = B_\sigma[i] = 1 \iff P[i] = \sigma$$

Esempio su slide.
La tabella prodotta B è quindi l'insieme delle word B_σ, $\forall\, \sigma \in Sigma$ In pratica ho la tabelle con un numero di righe pari al numero di simboli. In ogni riga ho una word dove a 1 ho solo gli elementi corrispondenti al carattere della riga nel pattern (se $P = aba$ ho due righe: $B_a = 101$ e $B_b = 010$).
Si ha quindi il procedimento di calcolo:

1. tutte le word di B_σ sono inizializzate a bit di soli 0
2. si inizializza una maschera M, che è una word di m bit tutti uguali a 0 tranne il più significativo che è pari a 1
3. si esegue una scansione di P da sinistra verso destra, e per ogni posizione i faccio:
   - $B_{P[i]} = M$ **OR** $B_{P[i]}$
   - $M =$ **RSHIFT**(M)

Per la scansione del testo introduciamo la notazione:

$$P[1, i] = suff(T[1, j])$$

per indicare il prefisso lungo i sul pattern P occorre esattamente come il suffisso di $T[1, j]$.

Definizione 66. *Dato un indice j, con $0 \leq j \leq n$, con $n = |T|$, D_j è una word di bit:*

$$D_J = d_1 d_2 \ldots d_m$$

tale che:

$$d_i = D_j[i] = 1 \iff P[1, i] = suff(T[1, j])$$

Algorithm 3 Algoritmo per il calcolo di B

```
function COMPUTE-B(P)
    m ← |P|
    for every σ ∈ Sigma do
        B_σ ← 00...0
    M ← 10...0
    for i ← 1 to m do
        σ ← P[i]
        B_σ ← M OR B_σ
        M = RSHIFT(M)
    return
```

Esempio 37. *Preso:*

$$T = abcbcbcabaadc$$

e:

$$P = cbcbca$$

Ottengo, ad esempio:

$$D_7 = 101010$$

In pratica avanzo sul pattern è verifico se il prefisso mano a mano occorre come suffisso nella prima parte del testo (parte definita da j di D_j che voglio calcolare). Avanzando sul pattern se ho il prefisso coincide con quel suffisso coincide metto 1, altrimenti 0 e poi aggiungo un carattere al prefisso avanzando fino alla fine del pattern.
La word D_0 (ricordiamoci per l'indice inizia con 1 quindi è la word relativa ad un indice prima di quello reale) è per definizione:

$$D_= = 00\ldots0$$

poiché nessun prefisso di P occorre come suffisso del prefisso nullo $T[0,0]$.
La word $d_m = D_j[m] = 1$ sse $P[1,m] = suff(T[1,j])$ quindi tutto il pattern ha un'occorrenza che finisce in posizione j sul testo T e che inizia in $j-m+1$.
Vediamo quindi come funziona la scansione del testo.

1. si inizia dalla word $D_0 = 00\ldots0$
2. per ogni valore di j da 1 a $n = |T|$ calcolo D_j a partire da D_{j-1} che è stata calcolata al passo precedente
3. ogni volta che D_j ha l'ultimo bit uguale a 1 ho in output $j-m+1$ che è l'inizio dell'occorrenza di P su T

Bisogna capire come calcolare D_j partendo da $j-1$:

- se $i > 1$:
$$D_j[i] = 1 \iff P[1,i] = suff(T[1,j])$$
ma questo può essere riscritto come:
$$D_j[i] = 1 \iff P[1,i-1] = suff(T[1,j-1]) \textbf{ AND } P[i] = T[j]$$
ma la cosa può essere semplificata sapendo che $P[1,i-1] = suff(T[1,j-1]) \iff D_{j-1}[i-1] = 1$ e quindi posso dire che:
$$D_j[i] = 1 \iff D_{j-1}[i-1] = 1 \textbf{ AND } P[i] = T[j]$$
ma sapendo che $P[i] = T[j] \iff B_{T[j]}[i] = 1$ ho che:
$$D_j[i] = 1 \iff D_{j-1}[i-1] = 1 \textbf{ AND } B_{T[j]}[i] = 1$$
Avendo un calcolo basato sulla logica, con verità a 1, possiamo anche riscrivere come:
$$D_j[i] \iff D_{j-1}[i-1] \textbf{ AND } B_{T[j]}[i]$$
e quindi:
$$D_j[i] = D_{j-1}[i-1] \textbf{ AND } B_{T[j]}[i]$$
Si ha quindi che l'i-simo bit di D_j è l'**AND** logico tra l'(i-1)-esimo bit di D_{j-1} e l'i-esimo bit di B_σ in corrispondenza di $\sigma = T[j]$
- se $i = 1$:
$$D_j[1] = 1 \iff P[1,1] = suff(T[1,j])$$
ma questo può essere arricchito come:
$$D_j[1] = 1 \iff P[1,1] = suff(T[1,j]) \iff P[1] = T[j]$$
e quindi posso dire, aggiungendo in **AND** un 1 che non cambia il risultato:
$$D_j[1] = 1 \iff P[1,1] = suff(T[1,j]) \iff 1 \textbf{ AND } P[1] = T[j]$$
Quindi il risultato non cambia se si effettua l'**AND** logico con un bit uguale a 1 e quindi si ha:
$$D_j[1] = 1 \iff 1 \textbf{ AND } P[1] = T[j]$$

ma si ha che $P[1] = T[j] \iff B_{T[j]}[1] = 1$ e quindi:

$$D_j[1] = 1 \iff 1 \textbf{ AND } B_{T[j]}[1] = 1$$

e quindi:

$$D_j[1] \iff 1 \textbf{ AND } B_{T[j]}[1]$$

che avendo logica booleana su bit diventa:

$$D_j[1] = 1 \textbf{ AND } B_{T[j]}[1]$$

Quindi il primo bit della parola D_j è uguale all'**AND** logico tra 1 e il primo bit della parola B_σ in corrispondenza del simbolo $\sigma = T[j]$

Esempio di calcolo su slide.
Nell'esempio si arriva a dire che:

$$D_j = \textbf{RSHIFT1}(D_{j-1}) \textbf{ AND } B_{T[j]}$$

Posso quindi approfondire l'algoritmo di scansione:

1. si inizializza una maschera del tipo $M = 00\ldots1$ (solo l'ultimo bit è pari a 1)
2. si inizializza $D_0 = 00\ldots0$
3. per j compreso tra 1 e n calcolo:

$$D_j = \textbf{RSHIFT1}(D_{j-1}) \textbf{ AND } B_{T[j]}$$

4. ogni volta che D_j **AND** M è diverso da $00\ldots0$ ho in output la posizione $j - m + 1$ di inizio occorrenza P su T

Avendo quindi nel complesso complessità:

$$O(|\Sigma| + m + n)$$

Su slide due esempi, uno di esecuzione e uno di calcolo più mirato.

Algorithm 4 Algoritmo di Baeza-Yates e Gonnet

```
function BYG(P, T)
    B ← Compute-B(P)
    n ← |T|
    D ← 00...0
    M ← 00...1
    for j ← 1 to m do
        σ ← T[j]
        D ← RSHIFT(D) AND B_σ
        if D AND M ≠ 0 then
            return j − m + 1
```

5.3.2 Algoritmo di Wu e Manber

Vediamo ora un algoritmo di **ricerca approssimata** con paradigma **shift-and** (*esempi per ogni step su slide*).
Dato un pattern P e un testo T, su alfabeto Σ, e una soglia d'errore k, si ha che P ha un'occorrenza approssimata su T in posizione finale j se esiste una sottostringa $[j - L + 1, j]$ che ha distanza di edit con il pattern di al più k.
In merito all'algoritmo **Wu Manber (*WM*)**, implementato ad esempio da *grep*, si ha un preprocessing del pattern di lunghezza m in $O(|\Sigma|+m)$ tramite il calcolo di una matrice B di $|\Sigma|$ parole di m bit. La scansione del testo per cercare le occorrenze approssimate è in $O(nk)$. Il preprocessing è come in Baeza-Yates.
Studiamo la scansione. L'algoritmo effettua $k + 1$ scansioni per un indice h, tale che $0 \leq h \leq k$, dove la h-sima iterazione trova tutte le occorrenze del pattern con al più h errori. Con h nullo ho le occorrenze esatte. La generica iterazione $h = k$ trova tutte le occorrenze in output all'algoritmo, ritrovando anche tutte le soluzioni degli step precedenti di h. Sembra ci sia ridondanza di calcolo ma si vedrà che si ha memorizzazione di tutte le occorrenze con un numero più basso di h rispetto a quello che si sta verificando. La generica iterazione h calcola n parole D_j^h lunghe m bit tali che $D_j^h[m] = 1$ sse per ha un'occorrenza approssimata che finisce in j con al più h errori (a conferma che con $h = 0$ ho lo shift-and esatto).
Si denoti con:

$$P[1, i] = suff_h(T[1, j])$$

il fatto che $P[1, i]$ occorre come suffisso di $T[1, j]$ **con al più h errori**.
Definiamo quindi, per un pattern P lungo m:

$$D_j^h = d_1 d_2 \dots d_m \;\; 0 \leq j \leq n, \;\; 0 \leq h \leq k$$

tale che:

$$d_i = D_j^h[i] = 1 \iff P[1,j] = suff_h(T[1,j])$$

ricordando che D_j^h è una parola di m bit.
In particolare:

- D_0^h è per definizione h bit più significativi pari a 1 e gli altri 0, ovvero:

  $$\begin{cases} D_0^h = 1 & \text{se } i \leq h \text{ (prime } h \text{ posizioni da sinistra)} \\ D_0^h = 1 & \text{se } i > h \text{ (ultime } m-h \text{ posizioni da destra)} \end{cases}$$

  infatti sui ha che:

  $$\begin{cases} \forall i \leq h, P[1,i] = suff_h(T[1,0]) \\ \forall i > h, P[1,i] \neq suff_h(T[1,0]) \end{cases}$$

  Ne segue che D_0^0, avendo h nullo, è per definizione $00\ldots0$

- $d_m = D_j^h[m] = 1 \iff [1,m] = suff_k(T[1,j])$, avendo un'occorrenza che finisce in j con al più h errori e quindi con al più k errori, avendo:

  $$D_J^h[m] = 1 \implies D_j^k[m] = 1$$

  per cui non devo ogni volta ricalcolare

- D_j^0 corrisponde a Baeza-Yates

Nel dettaglio vediamo l'algoritmo:

- alla prima iterazione $h = 0$ T viene scandito per j da 1 a n per calcolare tutte le word D_j^0 (ovvero Baeza-Yates)
- alla generica iterazione $h > 0$ per ogni posizione j del testo T viene calcolata D_j^h
- all'ultima iterazione $h = k$, ogni volta che la word D_j^h calcolata ha il bit meno significativo pari a 1 viene prodotta in output la posizione j di fine match con al più k errori (non potrei sapere quella iniziale a causa degli errori)

Bisogna quindi calcolare D_j^h per qualsiasi $h > 0$ e $j > 0$, avendo che per $h = 0$ e $j = 0$ ho word note.
Abbiamo che,per $i > 1$:

- ho un primo caso:

  $$D_j^h[i] = 1 \iff P[1,i] = suff_h(T[1,j]) \impliedby P[1,i-1] = suff_h(T[1,j-1]) \wedge P[i] = T[j]$$

  e quindi:

  $$D_j^h[i] = 1 \impliedby P[1,i-1] = suff_h(T[1,j-1]) \wedge P[i] = T[j]$$

  Ma sappiamo che:

  $$[1,i-1] = suff_h(T[1,j-1]) \iff D_{j-1}^h[i-1] = 1$$

  e quindi posso riscrivere:

  $$D_j^h[i] = 1 \impliedby D_{j-1}^h[i-1] = 1 \wedge P[i] = T[j]$$

  ma so anche che:

  $$P[i] = T[j] \iff B_{T[j]}[i] = 1$$

  e quindi:

  $$D_j^h[i] = 1 \impliedby D_{j-1}^h[i-1] = 1 \wedge B_{T[j]}[i] = 1$$

  tolgo gli 1 per lo stesso ragionamento fatto con Baeza-Yates (avendoch a e fare con bit), ottenendo:

  $$D_j^h[i] \impliedby D_{j-1}^h[i-1] \wedge B_{T[j]}[i]$$

  implicazioni a sinistra $\impliedby$ non la posso togliere avendo altri casi

- ho un secondo caso: ho un primo caso:

  $$D_j^h[i] = 1 \iff P[1,i] = suff_h(T[1,j]) \impliedby P[1,i-1] = suff_h(T[1,j-1])$$

  avendo quindi un errore in meno nel primo caso e quindi $P[i]$ e $T[j]$ possono essere diversi.
  Riduco:

  $$D_j^h[i] = 1 \impliedby P[1,i-1] = suff_h(T[1,j-1])$$

  ma ho che:

  $$P[1,i-1] = suff_h(T[1,j-1]) \iff D_{j-1}^{h-1}[i-1] = 1$$

  quindi:

  $$D_j^h[i] = 1 \impliedby D_{j-1}^{h-1}[i-1] = 1$$

  e quindi:

  $$D_j^h[i] \impliedby D_{j-1}^{h-1}[i-1]$$

- caso 3:

  $D_j^h[i] = 1 \iff P[1,i] = suff_h(T[1,j]) \impliedby P[1,i] = suff_{h-1}(T[1,j-1])$

  e quindi $T[j]$ genera un errore.
  Compatto in:

  $$D_j^h[i] = 1 \impliedby P[1,i] = suff_{h-1}(T[1,j-1])$$

  ma so che:

  $$P[1,i] = suff_{h-1}(T[1,j-1]) \iff D_{j-1}^{h-1}[i] = 1$$

  e quindi:

  $$D_j^h[i] = 1 \impliedby D_{j-1}^{h-1}[i] = 1$$

  che è:

  $$D_j^h[i] \impliedby D_{j-1}^{h-1}[i]$$

- caso 4:

  $d_J^H[i] = 1 \iff P[1,j] = suff_h(T[1,j]) \impliedby P[1,i-1] = suf_{h-1}(T[1,j])$

  avendo che $P[i]$ genera un errore.
  Ho quindi che:

  $$d_J^H[i] = 1 \impliedby P[1,i-1] = suf_{h-1}(T[1,j])$$

  ma:

  $$P[1,i-1] = suf_{h-1}(T[1,j]) \iff D_j^{h-1}[i-1] = 1$$

  quindi:

  $$d_J^H[i] = 1 \impliedby D_j^{h-1}[i-1] = 1$$

  ovvero:

  $$d_J^H[i] = \impliedby D_j^{h-1}[i-1]$$

Metto quindi in or i 4 casi:

$$D_j^h[i] \impliedby D_{j-1}^h[i-1] \wedge B_{T[j]}[i]$$

$$\vee$$

$$D_j^h[i] \impliedby D_{j-1}^{h-1}[i-1]$$

$$\vee$$

$$D_j^h[i] \impliedby D_{j-1}^{h-1}[i]$$
$$\vee$$
$$d_J^H[i] = \impliedby D_j^{h-1}[i-1]$$

e quindi:

$$D_j^h[i] \iff D_{j-1}^h[i-1] \wedge B_{T[j]}[i]$$
$$\vee$$
$$D_{j-1}^{h-1}[i-1]$$
$$\vee$$
$$D_{j-1}^{h-1}[i]$$
$$\vee$$
$$D_j^{h-1}[i-1]$$

e quindi posso mettere l'uguale:

$$D_j^h[i] = D_{j-1}^h[i-1] \wedge B_{T[j]}[i]$$
$$\vee$$
$$D_{j-1}^{h-1}[i-1]$$
$$\vee$$
$$D_{j-1}^{h-1}[i]$$
$$\vee$$
$$D_j^{h-1}[i-1]$$

Passando al caso $i = 1$ ho che:

$$D_j^h[1] = 1$$

per $h > 0$ e $j > 0$, ovvero $P[1,1]$ occorre sempre come suffisso di $T[1,j]$ con al più h errori.
Sostituisco $i = 1$ nella formula per $i > 1$:

$$D_j^h[1] = D_{j-1}^h[0] \wedge B_{T[j]}[i]$$
$$\vee$$
$$D_{j-1}^{h-1}[1]$$
$$\vee$$

$$D_{j-1}^{h-1}[0]$$
$$\vee$$
$$D_{j}^{h-1}[1]$$

Ma sapendo che:

$$D_{j-1}^{h}[0] = D_{j-1}^{h-1}[0] = D_{j}^{h-1}[0] = 1$$

ho che:

$$D_{j}^{h}[1] = 1 \wedge B_{T[j]}[i]$$
$$\vee$$
$$1$$
$$\vee$$
$$D_{j-1}^{h-1}[0]$$
$$\vee$$
$$1$$

Quindi il risultato è sempre 1 come deve essere sulla base della definizione. Posso quindi unire il tutto in un solo conto per $i \geq 1$ usando **RSHIFT**:

$$D_{j}^{h}[i] = RSHIFT(D_{j-1}^{h})[i-1] \wedge B_{T[j]}[i]$$
$$\vee$$
$$RSHIFT(D_{j-1}^{h-1})[i-1]$$
$$\vee$$
$$RSHIFT(D_{j-1}^{h-1})[i]$$
$$\vee$$
$$RSHIFT(D_{j}^{h-1})[i-1]$$

e quindi, usando **RSHIFT1**:

$$D_{j}^{h}[i] = RSHIFT1(D_{j-1}^{h})[i-1] \wedge B_{T[j]}$$
$$\vee$$
$$RSHIFT1(D_{j-1}^{h-1})$$
$$\vee$$
$$RSHIFT1(D_{j-1}^{h-1})$$
$$\vee$$

$$RSHIFT1(D_j^{h-1})$$

In generale si ricorda che:

$$D_j^0 = 1 \implies D_j^h[i] = 1, h > 0$$

o addirittura:

$$D_j^{h'} = 1 \implies D_j^{h'}[i] = 1, h > h'$$

In generale si ha anche che:

$$D_j^h[i] = 1 \implies D_j^{h+1}[i+1] = 1, i < m$$

o analogamente:

$$D_j^h[i] = 1 \implies D_j^{h+1}[i-1] = 1, i > 1$$

In generale si ha anche che:

$$|B(P)| = i \iff D_j[m] = 1$$

e i è la più grande posizione $< m$ tale che $D_j[i] = 1$.
Vari esercizi e una simulazione intera dell'algoritmo su slide.

Algorithm 5 Algoritmo di WU Manber

```
function WM(P, T)
    D_0^0 ← 00...0
    M ← 00...1
    D_j^0 = BYG(P, T)
    for h ← 1 to k do
        D_0^h = word con h bit più significativi a 1, resto a 0
        for j ← 1 to n do
            calcolo D_j^h con la formula dei 4 OR, usando RSHIFT1
            if h = k AND (D_j^h AND M) ≠ 00...0 then
                return j
```

Capitolo 6

Text indexing

Definizione 67. *Una **struttura di indiczzazione** è una struttura dati che riorganizza un testo in maniera da rendere facile ed efficiente un compito che compiuto sul testo originale risulta essere difficile e inefficiente.*

Si effettua quindi il preprocessing sul testo.
Prima di tutto bisogna fissare un ordinamento dei simboli dell'alfabeto Σ e si fissa l'ordinamento lessicografico. Esempio si ha $a < c < g < t$.
Bisogna estendere Σ con un simbolo sentinella lessicograficamente minore dei simboli dell'alfabeto e usiamo il \$, avendo $\$ < a < c < g < t$. Tale sentinella è posto a fine testo e non compare in altri punti del testo.

Definizione 68. *Si definisce il suffisso di indice j come il suffisso che inizia in j di T (bisogna specificare che inizia in j):*

$$T[j, |T|]$$

Il suffisso di indice iniziale pari a quello di \$ è nullo.

Posso stabilire quindi l'ordinamento dei suffissi di un testo T. Per ordinare due suffissi s_1 e s_2 si ha la regola:

- sia i la prima posizione a sinistra tale che $s_1[i] \neq s_2[i]$, quindi l'indice del primo carattere per cui ho un mismatch
- si ha che:
  - se $s_1[i] < s_2[i]$ allora $s_1 < s_2$
  - se $s_1[i] > s_2[i]$ allora $s_1 > s_2$

In altri termini è il modo in cui cerco le parole in un vocabolario. Terminando con il \$ non posso avere suffissi uguali e non avere che un suffisso sia suffisso dell'altro (visto che \$ non può trovarsi nel testo).

Definizione 69. *Definiamo la rotazione di indice j come la concatenazione del suffisso $T[j, |T|]$ con il suffisso $T[1, j-1]$.*
La rotazione con j pari all'indice del carattere \$ coincide con lo spostare il dollaro dalla fine all'inizio.

6.1 Suffix Array

Definizione 70. *Il **Suffix Array (SA)** di un testo T \$-terminato di lunghezza n è un array S di n interi tale che $S[i] = J$ sse il suffisso di indice j è l'i-esimo suffisso nell'ordinamento lessicografico dei suffissi di T, avendo $T[S[i], n]$ come i-esimo suffisso di T.*
Se ho $i < i'$ allora:

$$T[S[i], n] < T[S[i'], n]$$

Si vede che in spazio si ha:

$$O(n \log n)$$

avendo n valori interi di lunghezza a scalare da 1 a n.
L'algoritmo di calcolo può essere fatto in tempo $O(n)$.

Esempio su slide.
La ricerca esatta avviene in tempo $O(m \log n)$ con pattern P lungo m.
Andiamo avanti per step per capire la ricerca.
Se il pattern P occorre k volte in T allora PP è prefisso di k suffissi di T in cui indici sono le occorrenze di P in T. Si ha anche che gli indici dei k suffissi di cui P è prefisso sono successivi nel SA, essendo ordinati lessicograficamente, per cui mi muovo su finestre tramite ricerca binaria nel SA.
Se P occorre in posizione j di T ed è lessicograficamente minore/maggiore di un suffisso di indice j', $j' \neq j$, allora il suffisso di indice j è lessicograficamente minore/maggiore del suffisso di indice j':

$$P = \text{prefisso di } T[j, n] \wedge P < T[j', n] \implies T[j, n] < T[j', n]$$

$$P = \text{prefisso di } T[j, n] \wedge P > T[j', n] \implies T[j, n] > T[j', n]$$

Si ha quindi il seguente algoritmo:

1. si parte con tutto l'intervallo $[1, n]$ di posizioni su S, definendo tale intervallo come $[L, R]$

2. si considera il suffisso di indice $S[p]$ relativo alla posizione di mezzo, p, di $[L, R]$, e si verifica in tempo $O(m)$:
    - se $P < T[S[p], n]$ ripeto il punto 2 considerando la prima metà dell'intervallo corrente di ricerca come nuovo intervallo di ricerca $[L, R]$
    - se $P > T[S[p], n]$ ripeto il punto 2 considerando la seconda metà dell'intervallo corrente di ricerca come nuovo intervallo di ricerca $[L, R]$
    - P è prefisso di $T[S[p], n$ allora ho l'occorrenza di P in T all'indice $S[p]$. Questo trova una sola occorrenza ma può essere modificato per trovare tutti i suffissi di T che hanno P come prefisso

Si nota che:

$$p = \left\lfloor \frac{L + R}{2} \right\rfloor$$

Se P non occorre in T si raggiunge un intervallo vuoto di ricerca.

6.2 BWT

Definizione 71. *Definiamo la* ***Borrows Wheeler Transorm (BWT)*** *come una permutazione reversibile del testo T (e quindi è lunga n come il testo). È stata creata per comprimere il testo in quando tende ad avvicinare simboli uguali. La BWT rende possibile il pattern matching rendendo lineare sul pattern la ricerca esatta.*

Vediamo come costruire la BWT:

- bisogna costruire una matrice quadrata di lato n e alla riga i mettere la rotazione di indice i del testo (nella diagonale secondaria avrò tutti i \$)
- ordinare lessicograficamente le rotazioni spostando le righe ma salvando gli indici di partenza (possiamo dire che abbiamo una colonna degli indici). La nuova prima riga avrà indice n, iniziando con \$
- L'ultima colonna di questa matrice è la BWT del testo (il \$ non si può sapere dove sia)

Definizione 72. *Operazionalemnte definiamo la BWT B di un testo TT come la permutazione di T tale che $B[i]$ è l'ultimo simbolo della i-sima rotazione nell'ordinamento lessicografico delle rotazioni di T.*
Si ha quindi spazio:

$$O(n \log |\Sigma|)$$

La prima colonna F contiene i caratteri, che formano poi la BWT, ordinati e ha come primo simbolo il \$. Dato un indice i ho che nel testo T, ad un certo punto, il simbolo $B[i]$ precede il simbolo $F[i]$. Possiamo estendere che, avendo le rotazione, il \$, ultimo simbolo del testo, precede il primo simbolo. Questa proprietà è detta **proprietà p1**.
Il simbolo $B[1]$ è sempre l'ultimo simbolo di T prima di \$.
Per costruzione $B[i]$ è un simbolo di T in un certo indice k, avendo che $B[i]$ coincide con $T[k]$ (più forte di un solo =). Si arriva quindi anche alla **proprietà p2**, che lega B e F, infatti l'r-simo simbolo (nelle colonne ho r caratteri uguali di fila) σ in B e l'r-simo simbolo σ in F sono lo stesso simbolo in T . Questa proprietà è detta anche **last-first mapping**.
La funzione last-first (LF) è la funzione che corrispondere una posizione i sulla BWT B la pozione j su F in modo tale che $B[i]$ e $F[j]$ siano lo stesso simbolo del testo T:

$$j = LF(i)$$

Questo calcolo verrà approfondito con FM-index.
Le due proprietà mi permettono di ricostruire T da B, avendo la reversibilità di BWT. Si hanno vari step (**esempio su slide**):

1. prima cosa calcolo F da B (ordinando semplicemente i caratteri)
2. si mette il \$ in ultima posizione del testo. So anche che \$ è il primo carattere di F
3. inizio con $i = 1$
4. applico $p1$ sapendo che $B[i]$ precede in T il carattere $F[i]$ e quindi posso trovare il carattere prima di \$ nel testo con $i01$ su B e F
5. applico $p2$ e so che devo trovare di F la prima occorrenza del carattere $B[i]$ e poi ricomincio con lo step 4 fino a che non ho riempito il testo

Quindi passando ad ogni step da B a F e viceversa ricostruisco il testo.
Relazioniamo ora BWT e SA.
Consideriamo anche di avere la famosa colonna degli indici che indica gli indici delle rotazioni secondo il loro indice lessicografico. Si ha che essi sono

le posizioni sul testo dei simboli di F e sono anche il SA S del testo. Posso quindi calcolare la BWT di un testo a partire dal SA in $O(n)$.

Definizione 73. *Possiamo quindi ridefinire meglio la BWT.*
La BWT B, un array di lunghezza n, è la permutazione dei simboli di un testo T tale che $B[i]$ è il simbolo che precede l'i-esimo suffisso in ordine lessicografico. Ovvero ho che $B[i]$ è il simbolo (esattamente quel simbolo, è una relazione più forte di $=$, che trascurerebbe la ripetizioni di simboli in un testo):

$$T[S[i] - 1]$$

Esempio su slide.

6.3 FM-index