Dimostrazioni di Algoritmi 2 Orale

# Grafi

## Incidenza di un Arco

Un arco è incidente su determinati vertici se toccano quei vertici

## Grado di Un Vertice

Numero di archi ad esso incidenti

## Sottografo

H è un sottografo di G che è composto da H**(V*, E*)** dove V è il sottoinsieme di vertici, E è il sottoinsieme di Archi

## Cammino

Percorso all'interno di un grafo, è una sequenza ordinata di vertici e quindi una sequenza ordinata di archi.

### Cammino Semplice

Cammino che contiene ciascun vertice una volta sola escludendo il primo e l'ultimo che possono essere lo stesso vertice (A B C D A)
**Il cammino minimo è un cammino semplice**.

### Cammino non Semplice

Non è il cammino più corto che collega il vertice di partenza con quello di arrivo. All'interno è presente una ripetizione del vertice.
(A B A C D)

## Ciclo

Anch'esso è un cammino semplice, al suo interno ci sono ripetizioni di vertici oltre che il primo e l'ultimo

# Dimostrazione sulla visita generica

Dimostrare le tre invarianti:

1. Se ( u, v ) esiste dentro **E**, e il vertice ( u ) è nero, allora il vertice ( v ) è grigio o nero.
    a. Risolvibile con il secondo invariante
    b. Quando ( V ) è  grigio si risolve con il terzo invariante

2. Tutti i vertici grigi o neri sono raggiungibili da s (sorgente)
    a. **Caso base**: Ho solo ( s ), S raggiungibile da se stesso.
    b. **Passo Induttivo**: Se ( u ) è raggiungibile da ( s ), lo è anche ( v ) se e soltanto se ( u ) è adiacente a ( v ).

3. Qualunque cammino da ( s ) a un vertice bianco deve contenere almeno un vertice grigio.
    a. Caso in cui ( s ) è grigio: Il cammino da ( s ) a un altro vertice, hai che ( s ) è grigio.
    b. Caso in cui ( s ) è  nero: se non ci fosse nessun vertice grigio tra ( s ) e il vertice bianco ci sarebbe un vertice nero adiacente a un altro vertice nero, che viola il primo invariante.

## Visita in ampiezza

L'ampiezza usa la tecnica FIFO, e calcola la distanza tra la sorgente ( s ) e ( v ). Quindi si calcola il cammino minimo.

d [ v ] è il vettore delle distanze stimate di ( v ), invece GAMMA è la distanza minima  tra ( s ) e ( v ).
Esistono due proprietà :

1. In d[ ] ci sono tutti e soli vertici grigi. Questo perché inserisco i vertici all'interno della coda quando diventano grigi, sono rimossi quando diventano neri.

2. Se <v1, v2, v3….vn> è il contenuto di d [ ] in un determinato momento, allora saranno vere queste cose:

a. d { v di i } <= d { v di i + 1 }  per 1 <= i <= n-1
   I vertici all'interno di d [ ] sono ordinati per livello nella coda secondo la loro distanza stimata.

b. d [ v di n ] <= d [ v1 ] + 1
   la coda contiene al massimo sempre 2 livelli.

## Dimostrazione d [ v ] == gamma(s, v)

Caso base:

Nella coda si ha solo la sorgente: valgono entrambe le due proprietà precedenti. Quindi all'interno della coda si ha solo un livello. Se si effettua la DEQUEUE, la coda rimarrà vuota e continueranno a valere le due precedenti proprietà.

Se si effettua la ENQUEUE (inserimento di ( v ) dentro la coda), inserisco un vertice nella coda, quindi d [ v ] = d[ u ] + 1

Per la proprietà 1, d [ v di n ] <= d [ v di n ] + 1 = d [ v ]
Per la proprietà 2,  d [ v ] == d [ v di n ] + 1 <= d [ v di 1 ] + 1

Passo induttivo:

Assegno a **d [ v ] = d [ u ] + 1**
d [ v ] >= gamma ( s, v )

La distanza stimata di un determinato vertice è maggiore uguale alla distanza effettiva (cammino minimo) tra ( s ) e  ( v )

Dato l'albero dei predecessori ( simbolo P greco) contiene solo archi appartenenti a G, il cammino da ( s ) a ( v ) nell'albero è un cammino che appartiene a G.

Altro caso d [ v ] <= gamma ( s, v )
Si definisce un insieme di vertici che sono a distanza K da S. gamma( u, w) = K
Questo insieme è :

Vk = { v appartiene a V tale che gamma( s, v ) = K }

Per qualsiasi ( v ) che appartiene a Vk, quando ( v ) viene inserito nella coda, viene assegnato la distanza che sarebbe <= K.

Caso base:

K = 0
L'unico vertice a distanza 0 da s è s stesso e d[s] = 0 ≤ k (uno dei primi passi dell'algoritmo è d[s] <- 0).

Passo Induttivo:

Per ogni W, esiste V(k - 1), quindi d [ w ] <= K - 1.
Per ogni V, esiste Vk quindi d [ v ] <= K. ( W vertice precedente a V)
Esisterà almeno un vertice ( W ) che appartiene al V(k - 1)

Quindi W ha distanza K -1 dalla sorgente. Avrà un arco che va da W a V.

Definiamo quindi Uk−1 = {w ∈ Vk−1| w, v ∈ E}
(l'insieme dei vertici appartenenti a Vk−1 con un arco entrante in v)
Tra questi, sia u il primo vertice di Uk − 1 ad essere scoperto e inserito nella coda

Per la politica FIFO, u sarà anche il primo ad essere estratto dalla coda Poiché u appartiene all'insieme (Uk−1) dei vertici che hanno un nodo entrante in v (e che hanno distanza da s uguale a (k - 1), ed è il primo ad essere estratto dalla coda, quando u viene estratto v non è ancora stato scoperto (è bianco), e v verrà inserito nell'albero come figlio di u (cioè π[v] = u) con $\mathbf{d[v] = d[u] + 1}$. Inoltre, per l'ipotesi induttiva, $\mathbf{d[u] \leq k - 1}$

$\mathbf{d[v]} = d[u] + 1$
$\mathbf{d[v]} \leq (k - 1) + 1$
$\mathbf{d[v]} \leq k$

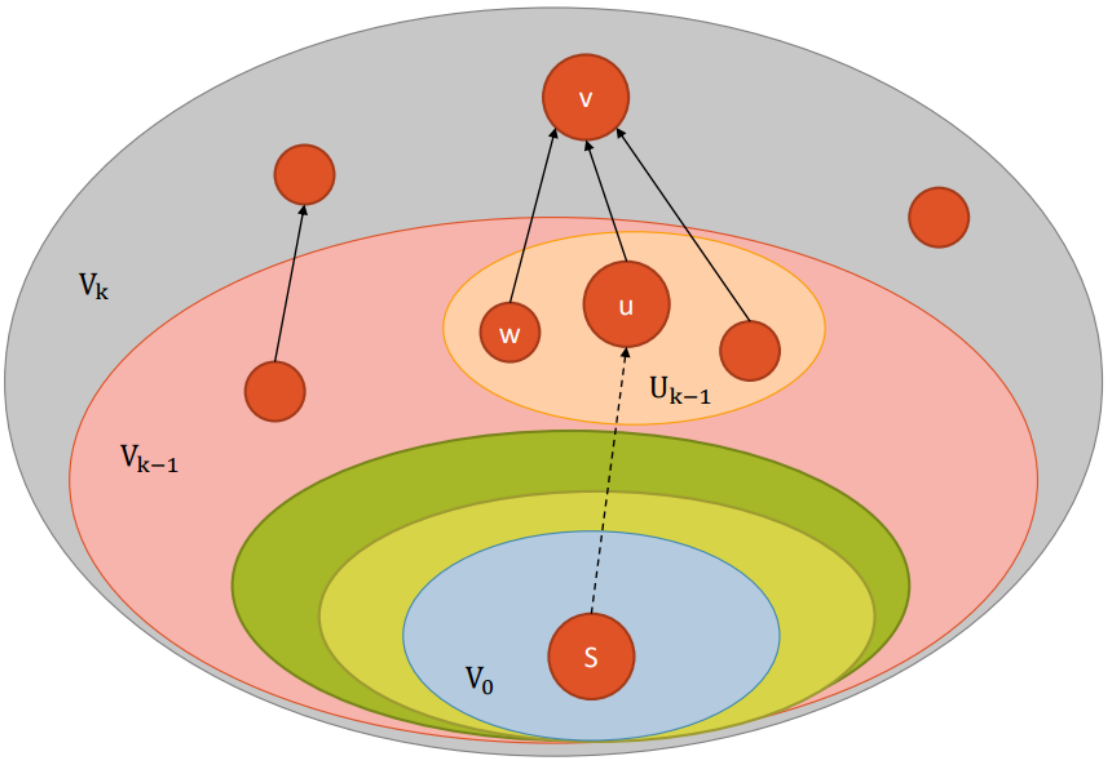
Manu Theory:
Questa cosa l'abbiamo dimostrata prima dimostrando che la distanza di questo vertice è maggiore o uguale e quello che c'è nel vettore d[v] è maggiore uguale alla distanza tra sorgente e vertice, poi dimostriamo che questo d[v] è minore o uguale alla distanza dalla sorgente e v.

Maggiore o Uguale:
Lo fai sul fatto che nell'algoritmo io come vado a dire che questa distanza, com'è definita questa distanza di d[v]? è definita così: Il mio albero di vista è un sottografo del grafo originario, quindi puo contenere soltanto degli archi che sono presenti nel mio grafo di partenza, quindi i miei cammini sono cammini che trovo dalla sorgente a tutti questi vertici, che effettivamente appartengono a questo grafo di partenza. Dal momento che la distanza dalla sorgente a un vertice è la lunghezza del cammino minimo dalla sorgente a quel vertice li, la distanza è sicuramente o maggiore o uguale alla distanza minima. Il primo caso è un caso di esistenza.

Minore o Uguale:
Dimostrazione la si fa ragionando sugli insiemi di vertici, noi definiamo un insieme Vk. (In questo insieme è un insieme di vertici in cui la distanza (minima) dalla sorgente a questi vertici è uguale a una costante K). Vogliamo dimostrare che un vertice che appartiene a Vk, quando viene inserito nella coda (diventa grigio) la distanza D[V] è minore o uguale a K. (Ricordati che K è la distanza minima dalla Sorgente a questi vertici).
Caso Base:
K = 0. Unico vertice che ha una distanza uguale a 0 è un vertice verso se stesso, quindi il cammino degenere della sorgente. Dimostrato.

Passo Induttivo:
Io parto dall'ipotesi che questo valore D[V] <= K è uguale al passo K - 1. Io faccio una serie di visite, scopro uno etc. AL PASSO K - 1, tutto quello che sto facendo è effettivamente MINORE o UGUALE di K - 1. Tutto quello che vale, vale fino a K-1. Voglio dimostrare che tutto quello che vale fino a K-1, vale e che valga anche K - 1 + 1.
Con K > 0:
Abbiamo l'insieme di vertici Vk. Dentro questo insieme, abbiamo un vertice V. Poi prendiamo un altro insieme di Vertici, che
Altro insieme Uk-1. Questo insieme ha una distanza K-1 dalla sorgente, è un sottoinsieme dell'insieme di vertici Vk-1. è un sottoinsieme che ha una particolarità, è un sottoinsieme di tutti i vertici che hanno un arco che va da loro al vertice V che fa parte dell'insieme Vk. Di questi tre vertici, quando facciamo la visita del nostro grafo, noi di questi tre vertici ne estraiamo uno per volta. immaginiamo che il vertice preso si chiama  U. Tra i suoi adiacenti di questo U, ci sarà anche V. Quando io ho fatto diventare U grigio, quindi trovo V come adiacente. Quando vado ad assegnare la distanza di V dalla sorgente, gli assegno la distanza di U + 1. Ma la distanza di U è <= K - 1.

D[ V ] = D [ U ] + 1
D[ U ] = K - 1
D[ V ] = K - 1 + 1
D[ V ] = K
D[ V ] <= K



# Teorema del cammino minimo per la visita in profondità

Il vertice V è discendente del vertice U se e solo se al Tempo d [ u ], V è raggiungibile da U con un cammino contenente vertici bianchi.

# Ordinamento Topologico

Un DAG è un grafo aciclico, sul quale è possibile calcolare un ordinamento topologico, che ordina i vertici del grafo per cui tutti gli archi vanno da sx verso dx e viceversa.
L'ordinamento topologico è una relazione di ordine totale (perchè vengono considerati tutti i vertici), Un Dag può possedere diversi ordini topologici, ma almeno uno.

L'ordinamento topologico dato un DAG G=(V,E) è una relazione di ordine totale <
sull'insieme V dei nodi tale che: <u,v> o il cammino u->v, segue che u< v, quindi u precede v

## Correttezza Algoritmo Astratto

Dobbiamo dimostrare che il vettore ORD è un ordinamento topologico di un Grafo G.
In ORD abbiamo tutti i vertici sorgenti che abbiamo eliminato da G' (copia di G).
L'invariante afferma che ad ogni istante del ciclo, non esiste un cammino in G che parte da G'i ad un vertice contenuto in Ord_i (cioè non esiste un arco all'indietro)

- Caso base: i=0, cioè G' = G e ord_i=0 (non contiene nessun nodo)
  quindi l'invariante vale perchè non esiste nessun cammino che parte da un vertice del grafo e raggiunge un vertice in ord_i
- Passo induttivo = al passo k-esimo, si sceglie un vertice che non possiede archi entranti nel grafo G'_k. Si considera che sia U il vertice sorgente, quindi al massimo posso raggiungere u passando per uno dei vertici contenuti in ord, ma questo non è possibile per ipotesi induttiva (i vertici contenuti in ord_k NON sono raggiungibili da nessun vertice in G_k.

Sicuramente ogni nodo di ord avranno un collegamento che li porta verso un grafo i esimo senza il nodo di Ord. Questo nodo, sarà in grado di raggiungere tutti i nodi del Grafo i esimo, ma non il contrario. I nodi del Grafo i esimo non avranno nessun arco che possano portarli verso alcun nodo dentro ORD.

## Teorema Dell'ordinamento Topologico

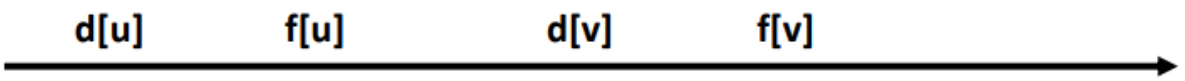
Il teorema dice che ogni visita in profondità di un grafo **orientato aciclico** associa ai vertici dei tempi di fine visita, in ordine decrescente, tali che:

**f [v] < f [u] per ogni arco <u, v> del grafo.**

Per dimostrare questo teorema dobbiamo supporre per **assurdo** che almeno un qualsiasi arco < u, v > abbia i tempi di fine visita in questo modo :

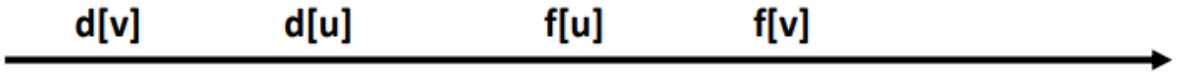
**f[u] < f[v]** ( Ovvero U viene chiuso prima di V)

Abbiamo due casi:




Questa supposizione è impossibile perche U non puo diventare NERO prima che V diventi grigio, cioé tutti gli adiacenti di V devono essere stati scoperti. L'arco parte da U e va verso V, non puoi chiudere prima il Padre e poi il figlio.

Altro caso:




Anche questo caso è impossibile, perchè stiamo aprendo prima il figlio e poi il padre nel tempo di visita del figlio, poi sempre dentro il tempo di visita del figlio stiamo chiudendo il padre. Questo viola le regole delle parentesi (ovvero non abbiamo rispettato l'ordine dell'arco <u, v>. Con questo metodo sembra che stiamo facendo una visita su <v, u>).

# Componenti Fortemente Connesse

Se esiste un cammino da ogni vertice verso ogni altro vertice.

## Lemma del cammino fortemente connesso

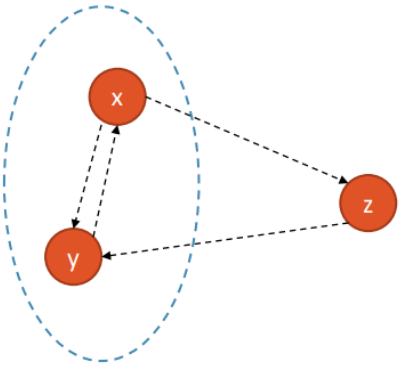
Il lemma dice che due vertici che appartengono a una stessa Componente fortemente Connessa, allora questi due vertici non potranno mai abbandonare tale componente. Per abbandonare intendiamo uscire con un cammino dalla CFC.

## Dimostrazione

Dimostriamo che X e Y appartengano alla stessa CFC. Utilizziamo un nodo ausiliare Z tale che:

Z sia raggiungibile da X ( $X \rightarrow Z$ ) e
Y sia raggiungibile da Z ( $Z \rightarrow Y$ )

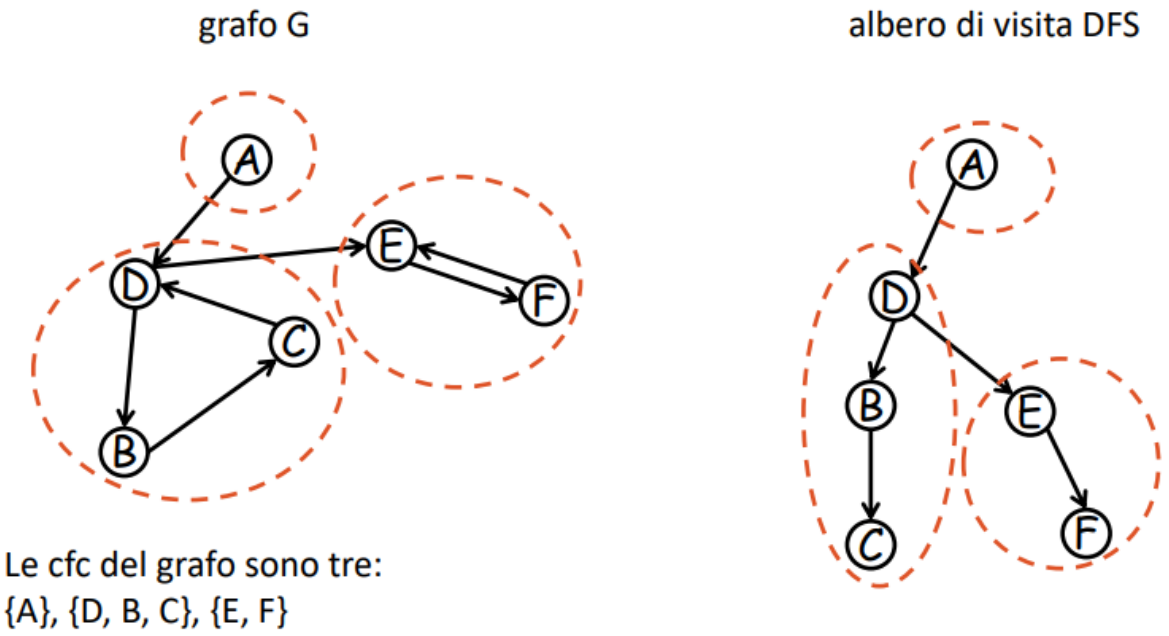
Dimostriamo che Z stesso appartenga a questa CFC. Siccome Sia X che Y sono entrambi in una CFC, ci sarà un cammino che porta entrambi a loro. Ovvero $X \rightarrow Y$ e $Y \rightarrow X$. DI conseguenza, se ci mettessimo una Z in mezzo, quest'ultima sarà raggiungibile sia da X che da Y. E quindi esisterà un cammino $Z \rightarrow X$.



# Teorema del sottoalbero fortemente connesso

In una qualunque visita in profondità di un grafo G orientato, tutti i vertici di una componente fortemente connessa vengono messi in uno stesso sotto albero.

**Dimostrazione**
Sia R il primo vertice di una CFC che viene scoperto nella DFS
Da R sono raggiungibili tutti gli altri vertici di questa CFC.
Siccome R è il primo, al momento della scoperta di R, tutti gli altri vertici chiaramente saranno bianchi.
Tutti i cammini da R agli altri vertici della CFC conterranno solo vertici bianchi che fanno parte della CFC
Di conseguenza, tutti i vertici appartenenti alla componente fortemente connessa di R saranno discendenti di R all'interno dell'albero DFS.

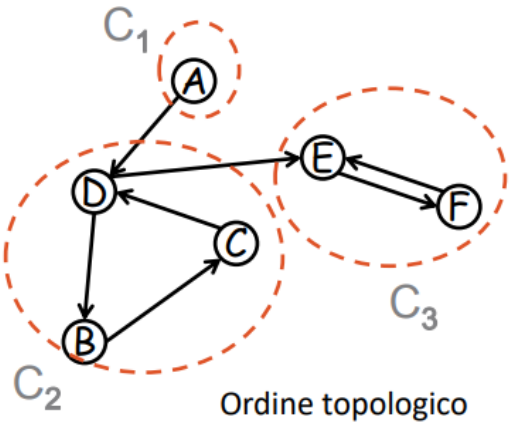



Tenendo questo a mente, allora è possibile trovare una CFC utilizzando solo una visita DFS. In particolare possiamo:

- Trovare un DFS del grafo e da esso estrarre le CFC
- Tagliare l'albero DFS in punti opportuni ed ottenere le CFC

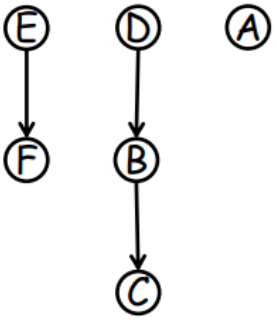
Problema 1: Trovare i punti per tagliare l'albero
Abbiamo due proprietà che ci possono aiutare.

Proprietà 1:
Esiste sempre, per ogni grafo **diretto**, almeno un ordine di visita DFS dei suoi nodi tale per cui le CFC sono già separate nella foresta di visita.




Ordine topologico del grafo quoziente: $C_1, C_2, C_3$

Visitandolo nell'ordine E F D B C A otteniamo




Proprietà 2:
Un grafo G e il suo trasposto G' hanno le stesse CFC.
La proprietà 1 vale anche per G trasposto.

Siccome sia G che G trasposto hanno le stesse CFC, cerchiamo di capire se possiamo sfruttare una DFS su G per trovare un ordine di visita DFS di G trasposto tale che DFS di Gt abbia le CFC tra di loro separate
Il motivo per cui facciamo una visita DFS sia su un G che sul suo trasposto è perche non non sappiamo l'ordine di visita DFS che ci dividerà i vertici in CFC.

Dati due vertici X e Y, quali visitare per primo nel grafo trasposto in modo che X e Y non stiano nello stesso Sottoalbero?

Assumiamo che X non sia nella stessa CFC di Y, quindi

1. Y è discendente di X in un albero della foresta DFS di G
2. X e Y non sono uno discendente dell'altro nella foresta DFS di G

Caso 1:
Y è  discendente di X, quindi Esiste in G un cammino da X a Y ma non esisterà l'opposto nello stesso Grafo.

Quindi nel Grafo trasposto, Y non puo essere discendente di X nella foresta DFS di Gt, quindi NON esisterà il cammino da X a Y in Gt..

Quindi se X precede Y nella visita DFS di Gt, sarò sicuro che X e Y non stanno nello stesso albero (o nella stessa CFC)

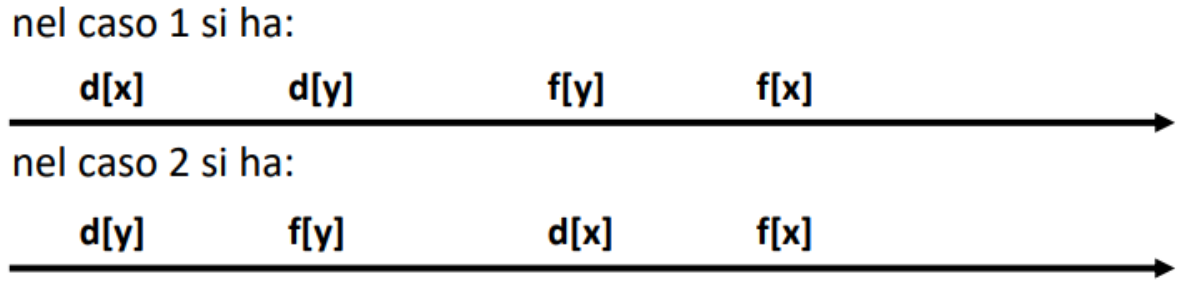
Caso 2:
In G normale X e Y non sono uno discendente dall'altro, quindi NON può esistere un cammino da X a Y.

In G trasposto, Y non può essere discendente di X nella foresta DFS, quindi non esisterà il cammino da X a Y.

Quindi se X precede Y nella visita DFS di Gt, sono sicuro che X e Y non sono nello stesso sottoalbero.

In entrambi i casi, vale la pena che la visita che inizia da X in Gt preceda quella che parte da y perché, se i vertici non sono nella stessa CFC, non saranno collocati nello stesso albero.
Sembra quindi che i vertici dela visita di Gt debbano essere considerati

- Dall'alto verso il basso
- Da destra Verso Sinistra




In entrambi I casi, Y viene chiuso prima di X.
QUINDI nella seconda visita (Gt) i vertici devono essere considerati in ordine decrescente di tempo di fine visita.

# Scelta Greedy

Un problema gode della proprietà della scelta greedy se è sempre possibile scegliere in esso una variabile x_h, attribuirgli un Valore ammissibile e localmente ottimo, rimuoverla dal problema ed ottenere un sottoproblema la cui soluzione unita al valore di x_h sia una soluzione ottima per il problema di partenza.
Se il problema gode di questa proprietà, si può definire per la variabile del problema un criterio/valore di **appetibilità**, che permette di effettuare la scelta locale.
Può essere fissa (non cambia durante l'esecuzione dell'algo) o modificabile (durante l'algo cambia, ci sono elementi che assumono maggior valore o minor valore a seconda dell'andamento dell'algo).
Un problema che gode della sottostruttura ottima, ammette che la soluzione del problema è ottima e al suo interno contiene le soluzioni ottime dei suoi sottoproblemi.

# Intervalli Disgiunti (greedy, appetibilità fissa)

## Dimostrazione di Correttezza

**MAX**: La sequenza di intervalli disgiunti già selezionati e appartenenti all'insieme S. è Una sequenza MASSIMALE.

**PrimaMAX**: tra gli intervalli disgiunti, il PrimaMAX è una sequenza che finisce per **prima** dentro l'insieme S. Finisce prima di tutte le altre sequenze possibili.

**PrimaVisti**: Sono gli intervalli che non ho ancora analizzato, quindi quelli che non appartengono a S.

Voglio dimostrare che, all'uscita di un ciclo del mio algoritmo, la mia sequenza che arriva fino a M di intervalli disgiunti scelti è sempre una sequenza MASSIMALE (**MAX**).
Voglio quindi dimostrare che **MAX è un Invariante.**

Caso Base:

Inizialmente S è vuoto. La sequenza massimale per S quindi è una sequenza vuota.
Gli invarianti sono tutti veri.
Max: S = 0/0, quindi S è massimale.
PrimaMAX: la sequenza vuota è l'unica possibile, quindi anche quella che finisce prima
PrimaVisti: ogni intervallo finisce dopo la sequenza vuota

Passo Induttivo:

Sia A un intervallo che termina per prima tra quelli da esaminare (i PrimaVisti, non appartenenti a S)
Considero quindi un ipotetico insieme S', che equivale all'unione della S di partenza unita con questo nuovo intervallo. Qual è il MAX di S' ?

Nel caso in cui A interseca con S, allora A verrà scartato. Oppure possiamo tentare di rimuovere l'ultimo intervallo del nostro MAX (o primaMax) da S.  e vedere se tutto funziona. Se l'inserimento di A viola primaMax (S' finisce DOPO S) allora A non si puo aggiungere e quindi S' non è la sequenza massimale.

Se invece l'inserimento all'inserimento di A non si verifica alcuna intersezione tra A e S, l'intervallo A verrà aggiunto alla MAX di S. Quindi PrimaVisti, PrimaMax non cambiano.

# Moore (appetibilità fissa)

## Dimostrazione di Correttezza

Moore ordina dei job in un vettore chiamato Sol.
I job nella soluzione devono rispettare **t_i <= s_i** (ammissibilità). Si ordina la sequenza di job in ordine crescente per istante di scadenza. Ma a differenza di un algoritmo tradizionale greedy, in moore i job inseriti nella soluzione possono essere rivalutati e scartati.

Sia S l'insieme ordinato di tutti i JOB fino ad ora esaminati:

1. (Max) SOL è uno SCHEDULING MASSIMALE di JOBs di S che rispetta le scadenze, cioè tra tutti gli scheduling che rispettano le scadenze, quello massimale è quello con il **NUMERO MASSIMO** di elementi.
2. (prima max) Inoltre SOL è tra tutti gli scheduling massimali di job di S, è quello che ha la **DURATA TOTALE MINIMA**

3. SOL è ordinato per tempi di scadenza CRESCENTI
4. (prima visti) Ogni JOB L che non appartiene a S ha una scadenza più GRANDE o UGUALE alle scadenze di tutti i JOB che appartengono a S

Caso base: insieme S e Sol sono vuoti, quindi gli invarianti valgono
\\Passo Induttivo:

Caso 1 = **t_k + d <= s_i**
Caso 2 = **t_k + d > s_i**.

**Caso 1:** Si aggiunge L (primo job non ancora esaminato, che ha scadenza minore rispetto gli alti jon ancora da esaminare) al fondo di Sol. Quindi sol contiene uno scheduling massimale. ⇒ Si dimostra per assurdo che: se esisterebbe uno scheduling per S U {L} con più di K+1 elementi, ma questo impossibile.
Si afferma anche che Sol ha durata minima, infatti se inserisco nella soluzione un job che ha scadenza superiore rispetto a L, otterrei comunque un insieme massimale, ma non minimo perchè L è il lavoro che finisce prima.
Gli altri 2 invarianti continuano a valere.
Caso 2: in questo caso non posso aggiungere L in fondo a Sol, perché non rispetterebbe la condizione di ammissibilità. Quindi si possono avere altri 2 casi.

a) **d_k >= d_massima**: cioè il job L che sto considerando è quello di durata Massima, quindi se si sostituisce al posto del job L il job Li, la soluzione che si ottiene comunque uno scheduling massimale di k elementi e di durata maggiore o uguale alla soluzione senza L. Cioè se aggiungo L a sol, ma per poterlo fare devo eliminare un job inserito nella soluzione, si otterrebbe comunque un insieme massimale, ma di durata maggiore rispetto alla durata di Sol senza l'inserimento di L.
b) **d_k < d_massima :** esiste un L_max all'interno di Sol, quindi se si elimina dalla soluzione questo lavoro di durata massima e si aggiunge il lavoro L in fondo a Sol, gli invarianti continuano a valere, perchè continuo ad avere uno scheduling massimale di k elementi e anche di durata minore rispetto alla durata precedente con L_max. Anche il terzo invariante continua a valere perchè in Sol, i job sono in ordine crescente perché il job che aggiunga in fondo ha scadenza maggiore o uguale rispetto alla scadenza del penultimo job inserito in Sol.

Quindi gli invarianti valgono alla fine di ogni esecuzione.

# Huffman (appetibilità modificabile)

Huffman vuole trovare la lunghezza media di decodifica di alcuni caratteri di un alfabeto.
Tutti I caratteri che sono usati frequentemente devono essere messi in alto, così che si usano più spesso (cammino più corto). Quelli che sono meno frequentemente sono in fondo all'albero.
Si vuole costruire l'albero di huffman, che è un albero binario pieno (binario bit decodifica).
La lunghezza media data dalla frequenza media e altezza(distanza) della sorgente.
Huffman contiene sottoalberi di huffman, all'inizio dell'algoritmo, nella foresta, ci sono solo foglie che conterranno il carattere e la frequenza.
Il passo Induttivo dice che ad ogni fine di un ciclo, si ha un sempre una foresta di Huffman.

# Dimostrazione di correttezza

Dobbiamo dimostrare che l'algoritmo restituisca un albero di Huffman (un albero binario pieno, cioè ogni nodi possiede entrambi i figli ), si vuole minimizzare la lunghezza di codifica L ( T ). (T = singolo carattere, L = Lunghezza di codifica) **(codifica = frequenza * distanza nell'albero)**

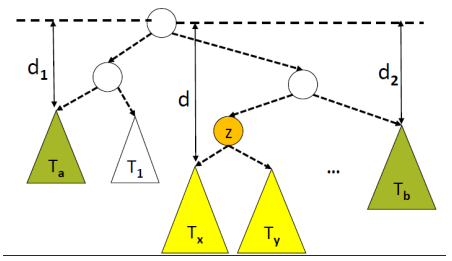
C = Alfabeto con caratteri f = frequenza (numero)
T = Sottoalbero di Huffman
Foresta Huffman= foresta contenenti T, ogni T è un albero per un alfabeto C con frequenza f
Le foglie degli alberi della foresta di Huffman sono tutti i soli i caratteri di C (con le frequenze). Ad ogni ciclo si uniscono in un nodo i 2 alberi di frequenza minore (appetibilità)

Partiamo **dall'ipotesi** che ciò che è mantenuto dall'algoritmo può essere completato in un albero di Huffman T. Si vuole dimostrare che se tale ipotesi vale all'inizio, essa continua a valere dopo ogni passo.
A fine algoritmo, la foresta (tanti sotto-alberi) può essere completata in un albero di Huffman T, che è composto da sottoalberi (T1, T2,...., Tn) e le foglie di questi sottoalberi sono i caratteri di C.

Dimostrazione per induzione:

- Caso Base: possiedo solo i nodi che rappresentano i singoli caratteri, quindi posso completare in qualsiasi albero (almeno uno esiste), quindi invariante vale
- Passo induttivo: si assume che prima della k-esima iterazione, l'invariante vale, quindi si ha una foresta di Huffman per quel determinato alfabeto C.
  Dopo la k+1-esima iterazione in cui si sta fondendo 2 alberi Ta e Tb con frequenze minime in un nuovo albero Tab, la nuova foresta risulterebbe così costituita
  F= (F - {Ta, Tb}) U {Tab}, si continua a possedere una foresta di Huff.

  Per ipotesi induttiva esiste un albero di Huff T' di cui gli alberi (T1, T2,..., Tn) sono i suoi sottoalberi, dove Ta e Tb sono nella foresta al passo considerato, essi possiedono le 2 più piccole frequenze. Allora esisterà un albero di Huff, T'' avente Tab come sottoalbero.
  Si considera fra i nodi non alla foresta, cioè quei nodi che non sono ancora stati creati nel passo, quello/i di *profondità massima*, sia Z che possiede 2 sottoalberi Tx e Ty. Si assume che **f(Ta) <= f(Tb)** e **f(Tx) <= f(Ty)** quindi si può affermare che **d1<=d** e **d2<= d.**
  Quindi scambiando la posizione di Ta e Tx e si ottiene un albero T'' di lunghezza media non superiore di T. ⇒
  L(T'')= L(T') - d1*f(Ta) - d*f(Tx) + d1*f(Tx) + d*f(Ta) =
  **L(T') + (f(Ta) - f(Tx)) (d - d1) <= L(T')**
  Allo stesso modo si scambiano la posizione Tb e Ty.
  Ottenendo alla fine che **L(T''') <= L(T'') <= L(T')**




  Quindi T''' che contiene come sottoalbero Tab, ed è un albero di Huff, che ho costruito al k+1 esimo passo, ed è completabile in un albero di Huff, quindi si continua ad avere un foresta di Huff.

# Dijkstra (appetibilità modificabile)

## Proprietà da considerare

Proprietà 1 : Sottocammini minimo di un cammino minimo
Un sottocammino di un cammino minimo è un cammino minimo (Sottostruttura ottima)

Dimostrazione:
Sia **u** → **V** un sottocammino minimo da S a T:
s → U → V → t

Se **u** → **v** non fosse minimo, ci sarebbe un altro cammino da U a V di costo inferiore.
Ma allora sostituendo tale cammino nel cammino da S a T si otterrebbe un cammino da S a T di costo inferiore rispetto al cammino minimo.

Proprietà 2:
Siano **S** l'insieme dei Vertici già considerati dalla visita e **D** l'insieme dei vertici ancora da considerare, le seguenti asserzioni sono tutte invarianti del ciclo:

2.1 $\forall v \in V: v \in S \Rightarrow d[v]$ non viene (più) modificata

2.2 $\forall v \in D: \pi[v] \neq NULL \Rightarrow \pi[v] \in S$

2.3 $\forall v \in V - \{s\}: d[v] \neq \infty \Leftrightarrow \pi[v] \neq NULL$

2.4 $\forall v \in V - \{s\}: d[v] \neq \infty \Rightarrow d[v] = d[\pi[v]] + W(\pi[v], v)$

Proprietà 3:
Se per ogni vertice, la distanza di questo vertice V non è infinita, allora esiste un vertice da S a V in G.

Dimostriamo che se la distanza del vertice U estratto dalla coda non è infinito, il cammino esiste
Per ipotesi induttiva, stabiliamo che esiste un cammino dalla Sorgente fino al PREDECESSORE di U estratto.
Allora il cammino dalla Sorgente al Pred. di U (assieme all'arco che va dal Pred(U) fino a U) costituisce un cammino da S a U.

Se esiste un cammino da S a V (dove V appartiene a S) nel Grafo, allora la distanza di V non è infinita.

## Dimostrazione di Correttezza (Dijkstra)

Dimostriamo che il seguente predicato è un invariante del ciclo while:
Per ogni T che appartiene a S : la distanza di T == al Peso tra Sorgente e T

Caso base:
Il predicato è vero all'inizio perchè la mia S soluzione è vuota.

Passo Induttivo:
Supponiamo vero che quando l'albero è stato costruito parzialmente, che il nuovo vertice U estratto da D (insieme dei vertici non considerati), allora la Distanza di U == al peso tra Sorgente e U.

Per ipotesi induttiva : Per ogni T che appartiene a S, d[T] == cammino minimo(Sorgente e T)

U Estratto, si hanno due casi:

- Caso 1 = Distanza di U diverso da Infinito **(d[u] ≠ ∞)**

Sia il predecessore di U = r =/= Null (deriva da 2.3)
Sappiamo allora che R appartiene a S, cioè R appartiene all'albero dei cammini minimi, sappiamo anche che la distanza di U è uguale alla somma tra la distanza di R e il peso da R a U.
Supponiamo per assurdo che tra S e U esista un cammino di peso minore della distanza di U. Questo deve contenere un arco tra un vertice S e uno in D. Supponiamo sia X il vertice tra Quelli neri e Quelli Grigi.

Questo cammino tra S e U può essere visto come la concatenazione di tre cammini
S → x → y → U

Se S → x → y → U è un cammino minimo, allora anche S → x → y è minimo.
Di conseguenza la distanza di Y, è uguale al peso dalla sorgente a Y (perche minimo)
Quindi :

$$W(s \rightsquigarrow x \rightsquigarrow y \rightsquigarrow u) = W(s \rightsquigarrow x \rightsquigarrow y) + W(y \rightsquigarrow u)$$
$$= d[y] + W(y \rightsquigarrow u)$$
$$\textbf{ma } d[y] + W(y \rightsquigarrow u) \geq d[y] \text{ (perché } W(y \rightsquigarrow u) \geq 0)$$
$$\textbf{e } d[y] \geq d[u] \text{ (perché u è stato estratto e ha d[u] minimo)}$$

Quindi, la distanza W è uguale a d[y] + W( y → U) NON è minore di Distanza di U.
Quindi d[u] == peso(sorgente, U).

- caso 2: d[u] = inf

# Minimo Albero Ricoprente

Il minimo albero ricoprente per un G, non orientato, connesso, e pesato; è un sottografo di G tale che:

- è un albero Libero
- Contiene tutti i nodi di G
- La somma dei pesi degli archi di G in questo sottografo è minima (ottimalità)

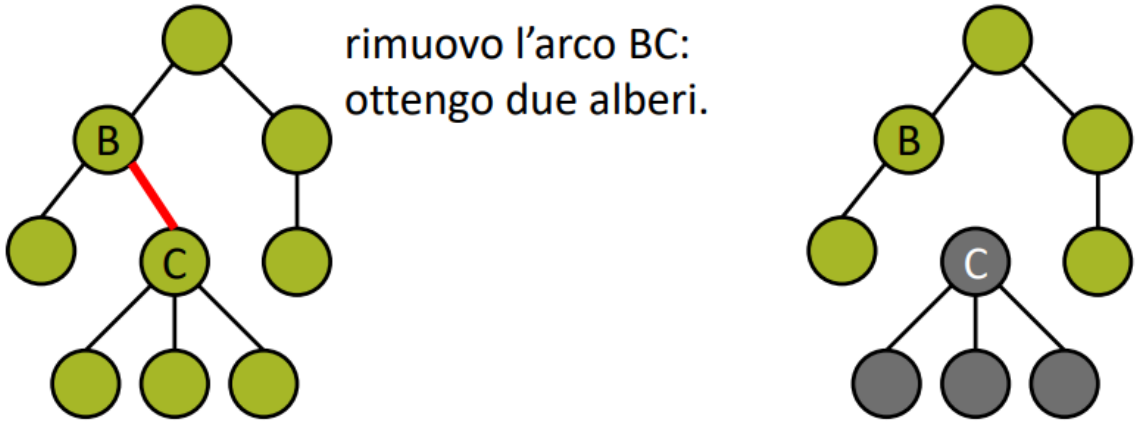
# Lemma del Taglio

Supponiamo di avere un insieme A contenente tutti gli archi di un MAR di un grafo G. Consideriamo un taglio non attraversato da alcun arco di A, siano S e V-S le sue due parti. Sia ( u, v) un arco con peso minimo tra tutti gli archi del grafo che Attraversa il Taglio. Allora se lo attraversa, (u, v) appartiene al MAR di G, che a sua volta estende l'insieme A. Ovverosia A unito con (u, v) è un sottoinsieme di un MAR.

Data una foresta A che costituisce un sottoinsieme di un MAR di grafo G, se facessimo un altro taglio sul GRAFO G, e questo taglio non comprende alcun Arco di A, e poi aggiungiamo ad A un arco di peso minimo fra quelli tagliati, otterremo cosi una nuova foresta A', anch'essa sottoinsieme di un MAR.

Proprieta 1:

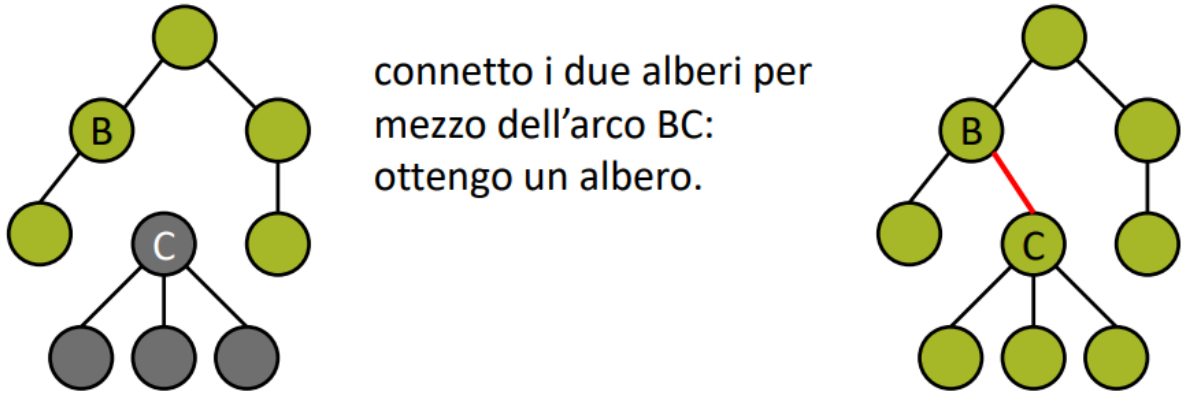
- Se rimuovo da un albero libero (grafo non orientato connesso aciclico) un suo arco, il grafo si divide in due sottografi distinti non connessi fra di loro.




Fra due nodi di un albero libero ci sarà soltanto UN SOLO cammino (altrimenti ci sarebbe un ciclo). Se si elimina un cammino, non ci saranno altri cammini fra i due estremi, e quindi tra i due sottoalberi.

Proprietà 2

- Se si connettono con un arco due nodi appartenenti rispettivamente a due alberi fra di loro non connessi, si ottiene un nuovo albero.



## Dimostrazione del Lemma

Ad ogni step, eseguo un taglio sul Grafo. Questo taglio, mi divide in due il grafo G in due aree. L'area S e l'area V-S. Tra gli archi tagliati, devo selezionare l'arco con peso minore. Una volta selezionato, il nodo di partenza di quell'arco finirà nell'area S. Continuo facendo nuovi tagli e aggiungendo passo per passo nuovi archi di peso minimo alla mia soluzione. Ad ogni passo, si ha sempre un nuovo MAR di G. Se dovessi selezionare un arco non minimo, si otterrebbe comunque un Albero Ricoprente, ma non sarebbe minimo.
Ipotizzando che l'arco (x,y) sia stato aggiunto al MAR, si possono verificare 2 casi:
--Se (x,y ) ≡(u, v), quindi il MAR contiene (u, v)l'arco==> il lemma vale
--Se (x,y ) diverso da (u,v ), si prende il MAR che non contiene l'arco (u, v) (altrimenti ci sarebbe un ciclo). Si prende la soluzione (dell'ipotesi ottima), si vuole ottenere la soluzione contenente l'arco (u,v), sostituendo l'arco (x,y), realizzando un MAR (T') che ha peso non maggiore a T. T' è quindi anch'esso un MAR che estende A, e che contiene l'arco (u, v) ed il lemma vale

## Teorema dell'unicità del MAR

Bisogna dimostrare che se tutti i pesi degli archi fossero tutti distinti, allora di MAR ce ne sarebbe soltanto uno.
Dobbiamo supporre per assurdo che un grafo G con archi tutti distinti avesse due alberi ricoprenti diversi M1 e M2.
Considero un arco dei due MAR che appartiene soltanto ad un MAR. Se dovessi aggiungere questo arco nell'altro MAR, si genererebbe un ciclo. Questo ciclo nega l'esistenza del secondo MAR (non sarebbe più un MAR). Di conseguenza, nel ciclo vado a togliere l'arco che ha peso Maggiore. Così facendo si ritorna ad avere un MAR vero e proprio, ma quest'ultimo sarebbe uguale al primo MAR, così violando l'ipotesi iniziale che si hanno 2 MAR. Quindi, un grafo con archi tutti distinti ha un solo unico MAR.

# Prim (appetibilità modificabile)

L'insieme di tutti i nodi che sono estremi di archi di A (Albero completo, contiene S, V-S) effettuando una visita inserisco un taglio che divide l'albero di visita e i vertici che non sono stati ancora aggiunti all'albero di visita. (Primm fa la visita, a sinistra contiene l'albero dei vertici visitati, a destra i nodi bianchi)
**D[v] > W(u,v)** faccio rilassamento

## Dimostrazione di Correttezza

Ci sia una partizione dell'insieme di vertici, S: contiene i nodi definitivi ed è un sottoinsieme di un mar ed V-S: contiene i nodi non definitivi.

**1 Invariante**: Tutti gli archi di S appartengono ad un sottoinsieme del Mar di G
**2 Invariante**: Per ogni nodo non definitivo, quindi contenuti in V-S, d[x] è il peso dell'arco più leggero che collega un nodo x ad un nodo nero in S.

- Caso base: nell'insieme S si ha solo il nodo sorgente e i 2 invarianti sono banalmente verificati.
- Passo induttivo: si vuole dimostrare che all'iterazione k-esima in S, continuiamo ad avere un sottoalbero del Mar. Si scegli un taglio che separi i nodi neri dagli altri nn ancora definitivi. Al passo k.esimo si sceglie un nodo U, con distanza minore tra i nodi contenuti in V-S e un nodo in S.
  Per sapere che esiste questo arco mediante il lemma del taglio, ed inoltre mi permette di affermare che in continuo ad avere un sottoalbero del Mar, aggiungendo l'arco e il nodo in S.
  Ma il secondo invariante non vale perchè sono cambiate alcune distanze per i nodi non definitivi, in particolare per i nodi che sono adiacenti al nuovo nodo definitivo.
  L'algo però effettua questo controllo e mediante il rilassamento ed effettua la modifica solo se la condizione (**D[v] > W(u,v)** ) è verificata.

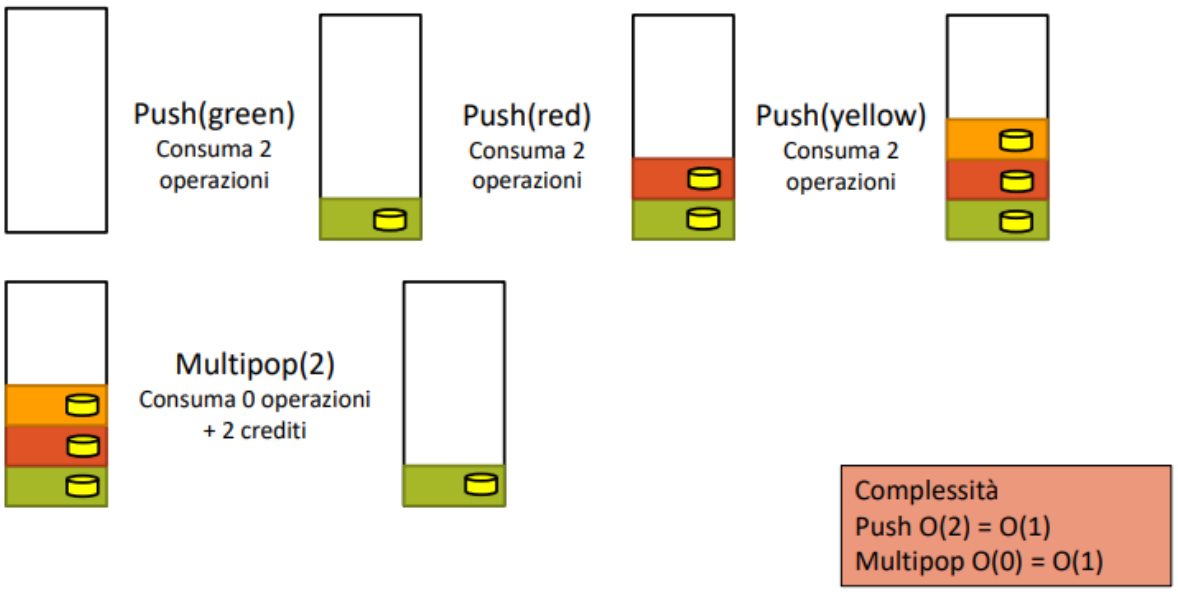
# Union Find

## Metodo dei crediti

Vengono utilizzati dei crediti per determinare il costo ammortizzato di una sequenza di operazioni.

- 1 credito vale O(1) passi di esecuzione

Le funzioni "meno costose" depositano dei crediti sugli oggetti.
I crediti depositati possono essere prelevati dalle funzioni più costose.
Il costo ammortizzato di ogni operazione in una sequenza è poi dato dalla somma di tutti i costi diviso il numero di operazioni



## Analisi Ammortizzata

Con questo in mente, qual è la complessità della gestione di una QuickFind bilanciata?
m find ha costo O(m)
n makeSet (ho n nodi) e n-1 union(la union toglie un nodo che lo unisce a un altro)
analizzato con il metodo dei crediti.

La makeSet singola inizialmente ha costo O(1)
La union inizialmente ha costo O(n), dove n sta per il numero di nodi che dovranno essere spostati.
In quanto la union sposta tutti i nodi da un set all'altro, il secondo set sarà grande sicuramente almeno il doppio di quello che era inizialmente. Una foglia quindi dopo K cambi di padre (rappresentante) apparterrà alla fine a un insieme grande 2^k elementi.
Siccome il numero totale di elementi è N, allora 2^k <= n, quindi K sarà <= log_2 n.

Con questo possiamo depositare dei crediti nella MakeSet cosiche la Union preleva i crediti dalla makeSet. I crediti depositati sono log_2 n crediti. Se ci fossero N nodi allora avremmo dei makeSet con n * log_2 n crediti.

Ogni union, per ogni foglia consumerà 1 credito per ogni costo operazione. Ma siccome abbiamo messo dei crediti di log_2 n crediti per foglia, la union preleva questi crediti all'esecuzione dell'operazione. Questo nullifica completamente il costo dell'operazione a 0. La union ora costerà O(0).

Di conseguenza, il costo ammortizzato di ogni operazione in una sequenza di N nodi con makeset e N - 1 union sarà O(log_2 n), ovvero O(n * log_2 n).

Esempio: 50 makeSet, 49 Union. Costo: 50 * log_2 50 * 0 + (eventuali find)

# Kruskal (appetibilità modificabile)

L'insieme di tutti i nodi di un sottoalbero T contenuto in A e non connesso ad altri alberi contenuto in A, cioè mantiene in maniera efficiente la foresta costituita con tutti gli archi che inserisco nella soluzione. Utilizzo il lemma quando devo inserire un arco in A, so che ottengo un sottoalbero del mar mediante il lemma. Mediante il lemma del taglio, che attraverso la partizione, il quale taglia un arco che è il minimo, con l'inserimento dell'arco e del vertice, vado ad aggiornare le distanze degli altri vertici non ancora considerati, se le nuove distanze hanno peso minore rispetto alle distanze stimate in precedenza.

## Dimostrazione di Correttezza

Invariante: gli archi in A, definiscono una foresta che è sottoinsieme di un Minimo Albero Ricoprente.
Caso Base: Invariante verificato: A è vuoto, e un insieme vuoto sicuramente è sottoinsieme di un MAR.

Passo Induttivo:

- Caso 1:

U e V appartengono allo stesso albero (stessa Find),
Se entrambi i rappresentanti di questi due nodi sono uguali, significa che sono nello stesso albero. Se noi dovessimo aggiungere questo arco all'insieme A, si andrebbe a creare un ciclo. Un ciclo non è ammissibile in un MAR. Di conseguenza U e V vengono scartati.

- Caso 2

U e V appartengono a alberi diversi T_1 e T_2
Eseguiamo un TAGLIO (Lemma del taglio) sull'arco U e V.
Questo arco, se ha peso minimo tra tutti gli archi del grafo che connettono due alberi distinti allora sarà anche di peso minimo fra tutti gli archi che attraversano il taglio. Di conseguenza, grazie al lemma del taglio, sappiamo che l'arco U e V uniti all'insieme A formeranno un sottoinsieme di MAR.
Aggiungendo (u, v) ad A si mantiene intatto l'invariante.

Per finire ad ogni ciclo, l'invariante non cambia. A alla fine di un ciclo sarà sempre sottoinsieme di MAR. A avrà un'albero **connesso aciclico** e contiene **tutti** i vertici di G. All'uscita del ciclo, l'insieme A è MAR.

# Programmazione Dinamica

La programmazione dinamica è una delle tecniche algoritmiche per risolvere un certo problema. SI suddivide il problema in N sottoproblemi. Per effettuarla bisogna che il problema gode della sottostruttura ottima e per risolverlo si crea una struttura di memorizzazione. Si utilizza una strategia bottom up.

Un problema ha la proprietà della sottostruttura ottima se la soluzione ottima del problema contiene le soluzioni ottime dei suoi sottoproblemi. Cioè si ottiene la soluzione del problema di partenza a partire dalle soluzioni dei suoi sottoproblemi. Mediante le soluzioni dei sottoproblemi riesci a costruire la soluzione del problema di partenza.

## LCS - Longest Common SubSequence

Date due sequenze S1 e S2, dobbiamo trovare la più lunga sottosequenza S3 che è inclusa sia in S1 che in S2. Si prova a risolvere questo problema con la tecnica di forza bruta, ovvero creando ogni possibile sotto sequenza sia in S1 e sia in S2.
Questo procedimento è troppo pesante perché oltre a generare ogni sottosequenza sia di S1 che di S2, devo poi confrontarla gli uni agli altri.
Di conseguenza si usa la tecnica di programmazione dinamica, dove innanzitutto si verifica che questo problema gode della sottostruttura ottima, data la sequenza S1 (a1, a2, … an) e

S2 (b1, b2 …., bn) e la sequenza S3 (c1, c2, … ck), la quale conterrà la piu lunga sottosequenza tra S1 e S2.
Si hanno due casi:

- Caso 1: sottosequenza finale di S1 == sottosequenza finale di S2

In questo caso si sa che la sottosequenza sarà contenuta in S3 e occuperà l'ultima posizione. ck == am == bn
S3 = LCS ( ( S1 - {am}, S2 - {bn} ) + {am} )
I prossimi confronti si faranno nelle due sottosequenze SENZA gli ultimi elementi verificati.

- Caso 2: sottosequenza finale si S1 =/= sottosequenza finale di S2

Qui le due sottosequenze non saranno contenute in S3. Possiamo toglierla dal confronto e dire che:
S3 = LC ( S1 - {am}, S2 ) → Nel caso in cui la sottosequenza finale di S1 non sta in S3
S3 = LC ( S1, S2 - {bn} ) → Nel caso in cui la sottosequenza finale di S2 non sta in S3

Mediante questi casi possiamo affermare che LCS gode della proprietà della sottostruttura ottima.
Per risolvere questo problema si deve avere una struttura di input/output del problema.
La struttura di memoizzazione
Effettuare il popolamento.

# Bellman-Ford

Condizione di Bellman: Per ogni arco (u,v ) e per ogni vertice s.
$\delta(s, v) \leq \delta(s, u) + W(u,v) \Rightarrow \delta(s, vk) \leq \delta(s, vk-1) + W(vk-1,vk)$
Da esso deriva il seguente Lemma: un arco u,v appartiene ad un cammino minimo da s a v se e solo se u è raggiungibile da S e la condizione di Bellman è soddisfatta con l'uguaglianza per ( u,v ), cioè se vale: $\delta(s, v) = \delta(s, u) + W(u,v)$

$\delta(s, v) = D(s,v)$ sovrastima della distanza tra s e v.

## Verifica di correttezza

Sia gamma(s, Vk) la distanza da S a Vk.
Definiamo l'invariante KPath:

Dopo la K-esima iterazione del ciclo esterno (quindi da 1 fino a n-1) si ha che d[Vk] = gamma(s, Vk) per ogni nodo Vk, per cui il cammino da S a Vk è composto al massimo di K archi.

Al passo k+1, ci sono 2 casi:
Caso 1: $d[v_{k+1}]$ viene aggiornata,
Allora $d[v_{k+1}] = d[v_k] + W(v_k, v_{k+1}) = \delta(s, v_k) + W(v_k,v_{k+1}) = \delta(s, v_{k+1})$
Caso 2: $d[v_{k+1}]$ non viene aggiornata,
poiché $d[v_{k+1}]$ è il peso di un cammino da s a $v_{k+1}$ in G,
$d[v_{k+1}] \geq \delta(s, v_{k+1})$
poiché non è stata aggiornata,
$d[v_{k+1}] \leq d[v_k] + W(v_k,v_{k+1}) = \delta(s, v_{k+1})$,
possiamo avere solo
$d[v_{k+1}] = \delta(s, v_{k+1})$ quindi il cammino trovato è comunque minimo.

Quindi, dopo la k-esima iterazione, KPATH vale e $d[v_k] = \delta(s, v_k)$ per ogni nodo $v_k$ per cui il cammino minimo da s a $v_k$, che è sicuramente un cammino semplice, è composto al più da k archi.

Ma in un grafo, un cammino minimo può contenere al più tutti i nodi appartenenti a V, quindi un cammino minimo non può contenere più di n-1 archi.

Allora, dopo n-1 iterazioni $d[u] = \delta(s, u)$ per ogni nodo u. Quindi i cammini restituiti dall'algoritmo sono effettivamente quelli minimi.

**CVD.**

# Floyd-Warshall

## Distanze cammini minimi vincolati da K

Cammino K-vincolato vuol dire un cammino che non contiene da K + 1 vertici.
Denotiamo i vertici di un grafo con v1, v2, …., vn
Fissiamo un K che sta tra 1 <= K <= N (con N numero dei nodi)

**Il cammino minimo vincolato da K** è quello tra un vertice X e Y che ha **COSTO MINIMO** tra tutti i cammini che non contengono i vertici X e Y stessi. (Tutti i cammini esclusi gli estremi del cammino X Y).

**La distanza vincolata ta K** è il peso del cammino se il cammino esiste, altrimenti il cammino vale INFINITO.

Questa affermazione di Vk e K possiedono la caratteristica di sottostruttura ottima. Ogni cammino di un cammino minimo (K-Vincolato) è anch'esso un cammino minimo K-Vincolato.
Si definisce l'equazione

Distanza K-Esima di X e Y =

minimo ( distanza K-esima -1 di XY,
distanza K-esima -1 di Xvk + distanza K-esima - 1 di Yvk )

- Caso base: con K=0, considero solo i vertice di partenze x e il vertice destinazione
  $\pi xy$^0 = (x,y ) e $dxy$^**0**= W(x,y ) se (x,y ) ∈E
  $\pi xy$^0 = {} e $dxy$^**0**= 0 se x = y ,
  $\pi xy$^0 non esiste e $dxy$**0**=inf, ⇒ perchè non ho un arco da x a y
  Infine, $dxyn=\delta(x,y)$

- Passo Induttivo:

Si hanno due possibili casi:

- Caso 1: Vk **non** è un cammino (K non è un vertice intermedio).

In questo caso, se K non è un vertice intermedio, allora significa che il percorso piu corto che va da i a j e che utilizza i vertici inclusi in { 1, 2, ,..... k -1 } è anche il piu corto per il percorso che usa i vertici in { 1, 2, ….. k }

- Caso 2: Vk è un cammino (K è un vertice intermedio)

In questo caso, K è un vertice intermedio e quindi la strada puo essere divisa in due sottoparti. Ognuna di queste sottoparti utilizza i vertici inclusi in {1, 2, …., k -1} per creare una strada che utilizza tutti i vertici inclusi in {1, 2, …., k}. Questo perchè appunto K è un punto medio.

La formula ricorsiva di questi due casi sarà quindi:

$$d_{xy}^{k} = \min(d_{xy}^{k-1}, d_{xv_k}^{k-1} + d_{v_k y}^{k-1})$$

Preso un K, si va a vedere se la concatenazione die cammini minimi che vanno da i a vk e da vk a j hanno peso minore rispetto al cammino diretto che va da i a j.
Quindi si va a verificare la disuguaglianza triangolare:
D(x, y) > D(x,k) + D(k, j)

Se questa disuguaglianza non è verificata, si applica un rilassamento facendo passare il cammino minimo attraverso Vk (Il punto intermedio K)
D(x, y) = D(x, K) + D(K, y)

# Complessità

Nella teoria della complessità algoritmica, la classe di problemi PSPACE (da polynomial space) è l'insieme di tutti i problemi che possono essere risolti da una macchina di Turing deterministica usando una quantità di memoria di **O(n^{k})**, dove **n** è la dimensione dei dati di ingresso e **k** è un qualsiasi valore finito.

In altre parole, PSPACE include quei problemi che possono essere risolti da un algoritmo che utilizzi uno spazio di memoria la cui dimensione sia al più funzione polinomiale della dimensione dell'input.

## P contenuto in PSPACE

Inoltre si osserva facilmente che la ben più conosciuta classe di complessità P è inclusa in PSPACE. Infatti è evidente che se un algoritmo ha tempo di esecuzione **t**, potrà utilizzare al più **t** celle di memoria; se quindi sappiamo che **t = O(n^{k})** per un qualche **K,** necessariamente lo spazio utilizzato è limitato allo stesso modo.
Il problema contiene il quantificatore esiste, quindi posso assegnare a z o il valore 0 o 1 (x1 = 1, x2=0, x3= 0, x4=1), possendo così un certificato lineare, nella dimensione dell'input, se tutti i risultati sono verificati.

## NP contenuto in PSPACE

La classe NP contiene quei problemi risolvibili in tempo polinomiale da un algoritmo non deterministico (cioè un algo che non riesce a trovare la soluzione in tempo lineare, ma mediante la funzione oracolo è possibile); possiede una fase di verifica del certificato, in cui si determina se la soluzione è corretta, questo controllo viene effettuato in tempo polinomiale.
Quindi NP contiene i problemi che non possiamo risolvere in tempo polinomiale, ma sono quei problemi che potranno essere risolti in **t** polinomiale, mediante la funzione oracolo.

Il problema contiene il quantificatore di esistenza e l'identificatore del $\forall$,
le variabili associate all' $\exists$, assumono un solo valore (come il problema precedente); con il per ogni è possibile percorrere 2 strade (0 e 1). Quindi ottengo un albero che possiede delle biforcazioni ogni volta che ho $\forall$, quindi la dimensione dell'albero è esponenziale nel numero di variabili.
La verifica del certificato ha una complessità esponenziale O(2^n).
⇒ quindi il problema non appartiene a NP, perchè il certificato non è verificato in t polinomiale.

# Problema Indecidibile

Il problema indecidibile è un tipo di problema che appartiene ai problemi matematici. Un problema matematico può avere una complessità, cioè la complessità del miglior algoritmo in grado di risolverlo, sia se l'algoritmo esiste, sia se è immaginario. Un problema P è una relazione di tipo:

$$P \subseteq I \times S$$

Dove I sta per l'insieme di tutte le possibili istanze in INGRESSO e S è l'insieme delle possibili soluzioni del problema P.

Quindi P problema contiene il prodotto cartesiano delle istanze con le soluzioni.

I problemi indecidibili sono quindi dei problemi in cui non si conoscono né gli algoritmi per risolverli, né l'esistenza di algoritmi in grado di risolvere il problema. Neanche con tempo o spazio infiniti.

## Halt

Halt è un problema indecidibile.
Definiamo un programma Halt(P, i) che restituisce 1 se il programma (con un certo input i) termina con un numero **finito** di passi. Altrimenti restituisce 0.

Se fosse possibile scrivere il programma Halt, sarebbe scritto nel seguente modo:

```
int g(Prog)
      while (halt(Prog, Prog)) {
            do nothing;
      }
      return 0;
```

(si può dare in input ad un algoritmo se stesso)
A questo punto, quale sarebbe l'output di g(g)?
g(g) termina se e soltanto se g(g) non termina, e questa è una contraddizione.

Per questo problema non esiste né un algoritmo ne esiste un modo per idearlo.

# Algoritmo di Approssimazione (si può affermare che sia una sorta di tecnica algoritmica)

Questi algoritmi associano ad un problema di Ottimizzazione, un problema Decisionale, afferma che una determinata soluzione per il problema è ottima.
Quindi permette di risolvere quei problemi complessi (np-completo o extime), infatti l'algo di approssimazione privilegia l'efficienza a costo dell'ottimalità, tale che la soluzione restituita non è per forza ottimale, ma ha un costo **C** che differisce dal costo **C*** di quella ottima di al massimo di un valore, che è definito fattore di approssimazione (p(n)).
Quindi se il fattore è 2, si ha un algoritmo 2-approssimato, cioè la soluzione che si troverà, costerà al massimo 2 volte il costo della soluzione ottima.
Il costo deve rispettare questa formula: $\max\{C/C^*, C^*/C\} \leq \varrho(n)$
(il rapporto tra soluzione restituita e soluzione ottima e il rapporto tra soluzione ottima e soluzione restituita ⇒ è <= al fattore di approssimazione.)

- X un problema di Minimizzazione il costo approssimato della soluzione trovata dall'algoritmo è al massimo **2C***

- X un problema di Massimizzazione il costo approssimato della soluzione sarà la metà del costo della soluzione ottima ⇒ **C* / 2**

# Copertura (minima) di Vertici (problema NP-Completo)

Questo problema è un problema di Minimizzazione.
Dato un grafo non orientato G = (V, E), una copertura di vertici è un sottoinsieme $\mathbf{V'} \subseteq \mathbf{V}$ tale che, se ( u,v ) $\in \mathbf{E}$, allora $\mathbf{u} \in \mathbf{V'} \vee \mathbf{v} \in \mathbf{V} \Rightarrow$ quindi la copertura contiene almeno uno dei 2 vertici incidenti a un determinato arco. (**Ammissibilità**)
La dimensione di una copertura V' è il numero di vertici che contiene.
(**Ottimalità**) che la copertura di vertici contenga il minor insieme possibile di vertici che coprono tutti gli archi.

Essendo un algoritmo np-completo, non è possibile trovare un algoritmo deterministico che lo risolve in maniera ottima, ma è possibile individuare un algoritmo che trova una soluzione approssimata :

```
APPROX-COVER (G)
  C <- {}    //C conterrà la soluzione
  E' <- E
  while E' ≠ Ø
    sia (u,v) un arco in E' //E'∈ A
    C <- C ∪ {u,v}
    rimuovi da E' ogni arco incidente in u o v
  return C
```

ad ogni iterazione del ciclo, si sceglie un arco, si considerano i 2 vertici che sono incidenti a quell'arco, che vengono inseriti in C. Successivamente vengono rimossi dall'insieme degli archi, tutti gli archi incidenti per quei vertici (u e v)

Quindi il costo della soluzione dell'algoritmo approssimato, ha il fattore di approssimazione a 2, quindi al massimo la soluzione conterrà 2 volte quello della soluzione ottima.
(Soluzione ottima = 2 soluzione approssimata= 4)

**Dimostrazione**: che l'algo sia un 2-approssimato
Sia A l'insieme di archi ( u,v ) scelti all'inizio di ogni ciclo.
I vertici u e v, che sono stati inseriti nella coperture, non sono incidenti con nessun arco che esiste in A, cioè in A non esistono archi coperti dallo stesso vertice.
Per coprire essere una copertura ottimale deve contenere **almeno** un vertice di ciascun arco in A.
Infatti l'algoritmo una volta considerato un arco (u,v), vengono tolti da E* tutti gli archi incidenti ad U o V, ottenendo così in A solo archi che non possiedono dei vertici in comune.
|A| è il limite inferiore della dimensione di una copertura ottima C*: $C* \geq |A|$
Se consideriamo però l'inserimento di entrambi i vertici incidenti ad un arco nella copertura restituita dall'algoritmo, avendo così $C=2|A|$ si può affermare che
$|C| = 2|A| \leq 2\,|C^*|$ significa che l'algo è 2 approssimato rispetto alla soluzione ottima

## Problema del commesso viaggiatore (problema NP-Completo)

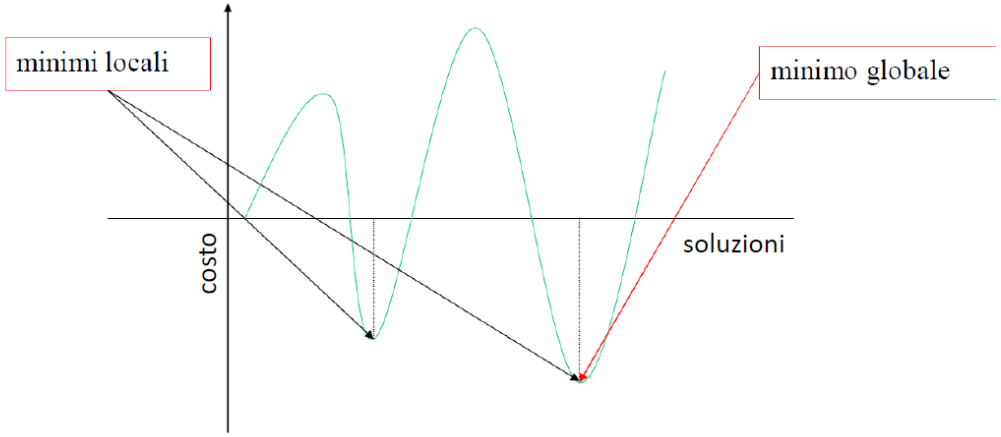
Questo problema è un problema di Ottimizzazione e di minimizzazione.
Il problema richiede di trovare un ciclo Hamiltoniano (è un ciclo semplice che contiene tutti i vertici, al suo interno non contiene ripetizioni dei vertici) di peso minimo.
Considerando un grafo pesato, **completo** (ogni vertice possiede un arco che collega con tutti gli altri vertici) e non orientato.
TSP rispetta la disuguaglianza triangolare: **W(u,w) <= W(u,v) + W(v,w)**

Per costruire un ciclo hamiltoniano, si realizza innanzitutto un Mar, essendo un grafo che contiene tutti i vertici del grafo, la differenza però e che il MAR non possiede dei cicli. Infatti il numero di archi nel Mar (n-1) è < del numero di arche nel ciclo hamiltoniano (n archi).
Il MAR però è la struttura che possiede tutti i vertici ed è quella che pesa di meno in assoluto, infatti si considera il peso del Mar come limite inferiore del ciclo Hamiltoniano.
Mediante la circumnavigazione del Mar si ottiene che il costo della soluzione è 2C* (il doppio del costo della soluzione ottimale del Mar).
Infine per ottenere un ciclo Hamiltoniano, sapendo che stiamo lavorando su un grafo completo, possiamo eliminare gli archi che provocano la ripetizione dei nodi, questo infatti è possibile grazie alla disuguaglianza triangolare che sostituisce al posto degli archi (u,v) w (v,w), l'arco (u,w).
Così facendo il costo della soluzione è **C <= 2C***
Quindi questo problema è possibile risolverlo se si realizza un Mar dal grafo di partenza e successivamente si realizza la visita in profondità tenendo traccia dei tempi di inizio visita.

# Ricerca Locale

Esistono problemi che risultano difficili da risolvere utilizzando l'algoritmo di approssimazione, quindi si utilizza la tecnica della ricerca locale, essa però per essere utilizzata su un determinato problema,si deve essere in grado di generare almeno una **soluzione** e bisogna definire un criterio per generare in maniera efficiente il **vicinato** N(x); il vicinato sono delle soluzione ammissibile simili a x.

Questa ricerca permette di trovare il **Minimo o Massimo locale**; perchè ad ogni soluzione che è migliore rispetto alla precedente, si deve analizzare le soluzioni del suo vicinato.




Nella maggior parte dei casi **non possiamo fare assunzioni su** che minimo abbiamo raggiunto, né sulla **distanza tra il suo costo e quello della soluzione ottima**.

Con questa tecnica è possibile risolvere il problema TSP (senza però l'utilizzo delle disuguaglianza triangolare degli archi) utilizzando però la tecnica dei **K-scambi**.

1. Si genera un ciclo Hamiltoniano casuale
2. Se ne cancellano k archi non adiacenti e se ne aggiungono altri k in modo da ricreare il ciclo
3. Se così facendo si crea un ciclo di peso minore, si ricomincia l'iterazione dal punto precedente (punto 2)
4. Altrimenti, si è trovata una soluzione minima localmente