Bibbia

Algoritmi 2

28esimo libro del Nuovo Testamento:

Definizioni principali per i grafi

Un grafo G:

è una coppia di insiemi finiti G=(V,E), dove V rappresenta l'insieme finito dei nodi o vertici e le coppie di nodi in E ⊆ VxV sono chiamate ARCHI.

ORDINE DEL GRAFO (n) :

numero di nodi del grafo n= |V|.

NUMERO DI ARCHI:

m= |E| questo insieme è finito.

DIMENSIONE DEL GRAFO:

è data dal numero n+m totali di nodi e di archi

LIMITAZIONE SU ARCHI DEL GRAFO:

Vale *non orientato* $0 <= m <= (n * (n-1))/2$ (nel caso di un grafo *orientato* sarà $(n * (n-1))$ poichè il numero massimo di archi è dato dal numero $O(n^2)$ di tutte le coppie possibili di nodi: il grafo è *sparso* se $m= O(n)$ e *denso* se $m=O(n^2)$.

GRAFO PESATO:

Un grafo viene detto pesato se è definita una funzione $W: E \rightarrow R$ che assegna un valore reale a ogni arco del grafo. (Un grafo non pesato ha semplicemente peso unitario)

ADIACENZA DEI NODI:

se esiste un *arco (u,v) che è incidente a ciascuno di essi* (cioè se esiste un arco (u,v) che li collega). Il numero di archi adiacenti è detto **GRADO TOTALE** del nodo e un nodo con grado 0 è detto **ISOLATO.**

GRADO USCENTE:

è il numero di archi che partono da quel nodo

GRADO ENTRANTE:

è il numero di archi che entrano in quel nodoìì

CAMMINO:

Definizione libro: da un nodo u a un nodo z, definito come la sequenza di nodi x0,x1,x2,...xk tale che x0 = u e xk = z e (xi,xi+1) ∈ E per ogni 0<=i <=k: l'intero k>=0 è detto *lunghezza*

u ⇝ v

E' un percorso di nodi che vengono attraversati la cui lunghezza è pari al numero di nodi attraversati. Risulta evidente che per ogni nodo attraversato deve esistere una arco fra i due.

CICLO:

E' un cammino per cui vale $x_0=x_k$ ovvero un cammino che ritorna al nodo di partenza. Un ciclo viene detto banale se attraversa lo stesso arco.

CICLO SEMPLICE:

x_i tutti diversi tranne il primo e l'ultimo che coincidono (cioè non contiene un'altro ciclo al suo interno)

MASSIMA LUNGHEZZA CICLO SEMPLICE:

Per quanto riguarda un ciclo semplice al massimo potrà essere lungo n.

CAMMINO SEMPLICE:

Un cammino è semplice se non attraversa alcun nodo più di una volta, ossia se non esiste un ciclo annidato al suo interno.

MASSIMO CAMMINO SEMPLICE:

In un grafo con n nodi, il cammino semplice potrà al massimo essere lungo n-1.

CAMMINO MINIMO:

u ⇝ z è caratterizzato dall'avere lunghezza minima tra tutti i possibili cammini da u a z.

DISTANZA TRA DUE NODI:

u ⇝ z è pari alla lunghezza di un cammino minimo che li congiunge e, se tale cammino ∄, è pari a +∞.

PESO DI UN CAMMINO:

somma dei pesi degli archi attraversati.

NODI CONNESSI:

due nodi u⇝ z sono detti connessi se esiste un cammino fra essi sia da u a z che da z verso u.

GRAFO CONNESSO

è un grafo dove ogni coppia di nodi è connessa (cioè esiste un cammino per ogni coppia di nodi).

SOTTOGRAFO:

di G è un grafo G'=(V',E') composto da un sottoinsieme dei nodi e degli archi presenti in G: ossia V' ⊆ V ed E' ⊆ V' x V' e, inoltre vale E' ⊆ E.

SOTTOGRAFO INDOTTO:

se oltre alle condizioni elencate nel sottografo vale anche la condizione aggiuntiva che in E' appaiono tutti gli archi di E che connettono nodi di V', allora G' è denominato sottografo indotto da V', è sufficiente specificare solo V' in tal caso pochè E' = E ∩ (V' x V')

COMPONENTE CONNESSA di G:

È un sottografo G' connesso e massimale ovvero con tutti i nodi connessi fra loro e che non può essere esteso in quanto non esistono ulteriori nodi in G che siano connessi ai nodi di G'.

MATRICE DI ADIACENZA:

Indica quali nodi sono adiacenti fra loro.

- Spazio $O(n^2)$ poco conveniente per i grafi sparsi i quali hanno complessità lineare O(n)
- $A[i][j] = 1$ se e solo se (i,j) ∈ E per i>=0,j<=n-1 sennò $A[i][j] = 0$
- Se vogliamo rappresentare i pesi dobbiamo fare un'altra matrice.
- La presenza di un nodo(arco) è verificabile in O(1)
- Dato un nodo "i" è richiesto O(n) per trovare gli adiacenti
- Nel caso di un grafo non orientato la matrice sarà simmetrica rispetto alla diagonale

LISTA DI ADIACENZA:

Ad ogni nodo i del grafo è associata una lista dei nodi adiacenti.

- Spazio O(n+m)
- Se m=O(n) allora O(n+m) = O(2n) = O(n) (*grafo sparso*)
- Se m=O(n^2) allora O(n+m) = O(n+n^2) = O(n^2) (*grafo denso*)
- Risulta ottima per scandire i nodi adiacenti
- La lunghezza della lista per ogni nodo è pari al suo grado
- Ogni elemento della lista corrisponde ad un arco i,j incidente
- Per un grafo pesato mi basterà aggiungere un campo "peso" nella lista
- La presenza di un arco tra una coppia i e j richiede la scansione della lista i e j mentre nella matrice è O(1).
- Un nodo ha grado massimo n-1 quindi nel caso peggiore dipende dal grado della lista(del nodo), nella matrice invece dipende da n.

ARCHI NON APPARTENENTI ALL'ALBERO DI VISITA MA PRESENTI NEL GRAFO:

- **ARCO ALL'INDIETRO:**

  Se v è antenato di u.

- **ARCO IN AVANTI:**

  Se v è discendente di u(ma non figlio perché l'arco apparterrebbe all'albero).

- **ARCO TRASVERSALE:**

  Se v e u non sono uno antenato dell'altro.Cioè c'è un arco che va da un ramo a un'altro.

POSSIBILI ARCHI

- **BFS ORIENTATO:**

  Se grafo orientato sono tutti possibili.

- **DFS ORIENTATO:**

  Se grafo orientato sono tutti possibili

- **BFS NON ORIENTATO:**

  Solo trasversali.

- **DFS NON ORIENTATO:**

  Solo indietro/avanti.

CICLI IN UN ALBERO DFS:

Se non ci sono archi all'indietro non ci sono cicli nel grafo.

FORESTA:

L'algoritmo di visita genera più alberi di visita se un grafo non connesso.

Algoritmi di visita

VISITA GENERICA:

G=(V,E)
V=Ordine del grafo
E=Archi del grafo
S=Scoperti
S⊆V : I nodi scoperti sono contenuti nell'ordine.
nodi ancora da visitare= V \ S
Codice:

```
{
        S={sorgente};
        finché è possibile scelgo un arco (u,v) tale che u∈S, v∉S
        S=S U {v};
}
```

A ogni iterazione all'insieme S viene aggiunto un nodo, al massimo fino a quando S è uguale a V cioè l'ordine del grafo.
Oppure si ferma prima se il grafo non è connesso.
Se G(grafo) è connesso allora tutti i nodi sono stati visitati.

COMPONENTE CONNESSA:

Si chiama componente connessa di G un *sottografo G' di G che è connesso e massimale cioè nel quale non posso aggiungere altri nodi/archi.*
Perchè se si aggiungessero altri nodi del grafo non sarebbe più connesso.

In una visita generica si riescono a visitare solo i nodi raggiungibili dalla sorgente.

Consideriamo un insieme di archi E'={}.
Ogni volta che aggiungiamo un nodo si aggiunge un arco all'insieme E'
Codice:

```
{
        S={sorgente};
        finché è possibile scelgo un arco (u,v) tale che u∈S, v∉S
        S=S U {v};
        E'=E' U {(u,v)};
}
```

Si creerebbe così un nuovo grafo G'=(S,E')
Il quale è un vero e proprio albero di visita, infatti non presenta cicli ed è connesso.

Dimostrazione per induzione creazione albero:

Base:
S=1→ (S,E') è un albero. Per E'={} è verificato.

Ip. Induttiva:
S=K → (S di k,E' di k) è un albero // dove S sono i nodi scoperti=K e E' sono gli archi utilizzati per visitare i K nodi

Tesi induttiva:
S=K+1→ (S di k+1,E' di K+1) è un albero//dove S sono i nodi scoperti=K+1 e E' sono gli archi utilizzati per visitare i K+1 nodi

Nello pseudocodice S(k+1) corrisponde a S(k) U {v}
Mentre E'(k+1) corrisponde a E'(k) U {(u,v)}
Ad ogni passo del ciclo abbiamo controllato cicli e connessione, non ci devono essere cicli e deve essere connesso.

FRONTIERA:

Sono tutti i nodi già visitati ma che hanno almeno un arco uscente che va verso nodi non ancora visitati

VISITA IN AMPIEZZA (Breadth First Search)

Visita che esplora i nodi in ordine crescente di distanza da un nodo iniziale, la sorgente, tenendo presente l'esigenza di evitare che la presenza di cicli possa portare a esaminare ripetutamente gli stessi cammini.
Gestiamo tutti i nodi di frontiera come una coda così da capire nell'algoritmo quale arco andremo a scegliere.
La coda ci garantisce l' ordine di visita in ampiezza.
d[] è stato inizializzato a -1 per ogni nodo.

Pseudocodice:

```
VISITA BFS(sorgente s) {
S={s};  E'={};  d[s]=0 //d -> distanza dalla sorgente
coda ← s
finché la coda non è vuota {
        u ← estratto dalla coda;
        per ogni arco (u,v) ∈ E {
                if v∉S {
                        coda ← v;
                        d[v]=d[u]+1;
```

```
                    S = S U {v};
                    E' = E' U {(u,v)}
                }
        }
}
```

E' un algoritmo di visita perchè prendiamo nodi non ancora visitati e incrementiamo S con i nodi man mano. Quando termina tutti nodi raggiungibili dalla sorgente sono stati scoperti.

Gli archi che conducono a vertici non ancora visitati, permettendone la scoperta, formano un albero detto ALBERO BFS, la cui struttura dipende dall'ordine di visita.
G'(S,E') è un albero di visita, e si dice ricoprente se copre tutto il grafo di partenza(solo i nodi).
La ricerca BFS sceglie sempre i cammini minimi in quanto opera a livelli.

COMPLESSITÀ':

visitiamo ogni nodo e ogni arco una volta sola quindi la complessità è: O(n+m)

DIMOSTRAZIONE PER INDUZIONE:

Vogliamo dimostrare che un nodo è a livello k dell' albero di visita se e solo se la sua distanza dall' origine nel grafo è k.
$d_G(s,v) = k$ allora $liv_A(v) = k$

Lemma: $d_G(s,v) <= liv_A(v)$ il cammino minimo(distanza) nel grafo può' essere minore o uguale al livello di quel nodo in qualsiasi tipo di albero.

SOTTOSTRUTTURA OTTIMA: Quando abbiamo soluzioni ottime, al suo interno abbiamo soluzioni ottime per i sottoproblemi.
Per esempio se abbiamo un cammino minimo da un nodo u a un nodo v, allora anche tutti i cammini intermedi sono minimi.

Tesi: tutti i cammini interni sono ottimali/minimali

Base: k=0, $d_G(s,v)=0 \dashrightarrow liv_A(v)=0$, se la distanza nel grafo è 0 implica che il livello nell'albero di visita è 0.
Perchè se la distanza è zero vuol dire che stiamo considerando la sorgente, quindi il nodo s=v.
E il livello della sorgente è 0.

Ipotesi induttiva: $d_G(s,v)=h \rightarrow liv_A(v)=h$
Tesi induttiva: $d_G(s,v)=h+1 \rightarrow liv_A(v)=h+1$

Supponiamo che: $d_G(s,v) = h+1$

per il teorema della sottostruttura ottima se $d_G(s,u)=h$ allora $d_G(s,v)=h+1$ dato che v è figlio di u

se $d_G(s,u)=h$ allora u è a livello h nell'albero per l'ipotesi induttiva.
Utilizziamo il lemma:
Supponiamo $d_G(s,v) < liv_A(v)$.
Se il livello del nodo v fosse > della distanza del nodo v sul grafo, allora vuol dire che v è figlio di un altro nodo w che si trova a un livello > h (dove h era il livello di u dato dall'ipotesi induttiva).
Questa supposizione viene esclusa per il fatto che l'algoritmo fa uscire dalla coda prima u di w dato che si trova a un livello inferiore di w.
Quindi v uscirà come figlio di u.
Quindi abbiamo dimostrato che $d_G(s,v) = liv_A(v)$.

Se volessi dimostrare l'inverso cioè che $liv_A(v) = k \rightarrow d_G(s,v) = k$
lo si dimostra gratuitamente, infatti dato che a ogni distanza corrisponde un livello, dalla dimostrazione precedente allora vale anche l'inverso perché tutte le relazioni sono occupate:
dist → liv
0 → 0
1 → 1
2 → 2…… e così via

DIMOSTRAZIONE PER ASSURDO

$liv_A(v)= k$ allora $d_G(s,v) = k$
Se volessimo dimostrare l'inverso basterebbe dire che se fosse $liv_A(v)= k$ e $d_G(s,v) =h \neq k$, per la parte precedente della dimostrazione risulterebbe $liv_A(v)=h \neq k$, assurdo perchè abbiamo supposto che $liv_A(v)=k$.

VISITA IN PROFONDITÀ (DEPTH FIRST SEARCH):

Si utilizza una pila in modo implicito, grazie allo stack (record di attivazione)
visita in profondità (DFS). In questo tipo di visita viene costruito un **albero DFS** in cui gli archi vengono indicati alla scoperta di nuovi vertici.

Pseudocodice:

```
{
        S = {}
        E' = {}
```

```
        inizioDFS(Sorg)
                DFS(sorg)
}

DFS(v)
{
        S = S ∪ {v}
        per ogni vicino u di v  // per ogni arco (v,u) ∈ E
        {
                se u ∉ S
                {
                        E' = E' ∪ {(v,u)}
                        DFS(u)
                }
        }
}
```

Al massimo ho tante chiamate DFS quanti sono i nodi(n), per ogni nodo si considerano tutti i vicini (2m se non è orientato).
Non è sicuro che, effettuando una DFS, la distanza tra sorgente e un nodo sia anche il suo cammino minimo.
Un nodo vicino, in caso di ambiguità, viene scelto in base all'implementazione.
Termina quando tutti i vicini sono stati scoperti e visita tutti i nodi raggiungibili dalla sorgente.

COMPLESSITÀ':

Se rappresentato con lista di adiacenza O(n+m)
Se rappresentato come un elenco di archi O(n·m)
Se matrice $O(n^2)$

Caratteristica principale:

La visita di un nodo finisce quando tutti i suoi discendenti sono stati scoperti e hanno finito la propria visita.
I "sottonodi" iniziano la loro visita dopo l'inizio della visita del nodo principale e la terminano prima di esso.

Differenza BFS e DFS:

- **BFS**: i vertici sono esaminati in ordine crescente di distanza dal nodo di partenza S, per cui la *visita risulta adatta in problemi che richiedono la conoscenza della distanza e dei cammini minimi* (non pesati).
- **DFS:** l'algoritmo raggiunge rapidamente i vertici lontani dal vertice di partenza S e quindi la *visita è adatta per problemi collegati dalla percorribilità, alla connessione e alla ciclicità dei cammini.*

Grafi non orientati non connessi:

Consideriamo grafi non orientati. Una visita di un grafo scopre tutti i nodi raggiungibili dalla sorgente. Quindi se il grafo è connesso tutti i nodi vengono raggiunti. Questo si può sfruttare per riconoscere se un grafo non è connesso.
Se non `e connesso, come visitarlo tutto?
Basta aggiungere un ciclo for esterno:

Pseudocodice:

```
S = ∅;
 for v = 0 to n-1 {
          if (v∉ S) {
                   DFS(v); //analogo con la BFS
          }
 }
```

Complessità:

O(m+2n) Sarebbe il costo della dfs l'aggiunta di " n" a causa del ciclo esterno.

Naturalmente la stessa idea si può usare per visitare completamente un grafo orientato (non è detto che da un nodo scelto come sorgente si raggiungano tutti i nodi del grafo).
Foresta:Quando un grafo è formato da tanti alberi, succede quando il grafo non è connesso e avviene una visita dfs o bfs che riesce a visitare tutti i nodi.

Componenti connesse

Calcolo delle componenti connesse in un grafo *non orientato.*(Non va bene per un grafo orientato)
Come determinare il numero e la composizione delle componenti connesse?
Le identificheremo con dei numeri 0, 1, 2, ... Usiamo un array cc con n elementi, il cui significato sarà: cc[v] = i significa che il nodo v appartiene alla componente connessa i.
Come gestirla?

Pseudocodice:

```
 int[] cc; //inizializzato ad un valore non significativo, per esempio -1
int contatore = 0;
S = ∅;
 for v = 0 to n {
   if (v ∉ S) {
          DFS(v);
          contatore++;
           }
```

```
}

void DFS(nodo){
        S =S U nodo;
        cc[nodo]=contatore;
        per ogni vicino di nodo {
                se vicino ∉ S{
                        DFS(vicino);
                        }
                }
}
```

Cicli in grafi non orientati

Se un grafo non è orientato ogni arco (u, v) in linea di principio è un ciclo u, v, u. Poiché tutti i grafi non orientati hanno sempre questi cicli, non ci interessano. Ci interessano solo cicli che non attraversano più di una volta lo stesso arco. Se il grafo ha un solo arco in più rispetto all'albero di visita, il grafo ha un ciclo. Come scoprirlo? Un arco in più significa che durante la visita si trova un nodo già visitato; però anche il padre è un nodo già visitato e, come abbiamo detto, non ci interessa quel tipo di cicli. Un modo per non considerare il padre, è ricordare chi è il padre di un nodo nell'albero di visita e decidere che esiste un ciclo solo se si incontra un nodo già visitato che non è il padre. Usiamo per esempio un array padre tale che padre[v] = w se w è il padre di v nella visita.

Esempio con la visita DFS (pseudocodice parziale):

```
int[] padre; //inizializzato ad un valore non significativo, per esempio -1
hasCycle = false;

void DFS(nodo){
S =S U nodo;
per ogni vicino di nodo {
        se vicino ∉ S {
                padre[vicino] = nodo;
                DFS(vicino);
        }
        else se vicino != padre[nodo] {
                hasCycle = true;
                }
}
}
```

Cicli in grafi orientati

Considerate l'albero di visita DFS.

Lemma 1:

Nell'albero di visita DFS c'è un arco all'indietro se e solo se nel grafo esiste un ciclo.

Dimostrazione: Se esiste un arco all'indietro (v, u), allora poichè u è un antenato di v, esiste un cammino nell'albero da u a v. L'arco all'indietro chiude tale cammino, creando un ciclo. Se viceversa esiste un ciclo nel grafo, allora tutti i nodi del ciclo sono discendenti e antenati di tutti gli altri nodi del ciclo, nel grafo. Chiamiamo v il primo nodo del ciclo che viene scoperto. Prima che la sua visita sia terminata vengono scoperti tutti i suoi discendenti e in particolare tutti i nodi del ciclo. Sia u l'ultimo nodo del ciclo prima di v (esiste quindi un arco (u, v) nel grafo).
Il nodo u è discendente di v nell'albero di visita e quindi (u, v) è un arco all'indietro.

Lemma 2

Nell'albero di visita DFS c'è un arco all'indietro se e solo se durante la visita di un nodo si trova un suo vicino che è già stato scoperto ma la cui visita non è terminata.

Dimostrazione: La visita DFS di un nodo termina solo quando tutti i suoi vicini e i suoi discendenti sono stati scoperti. Se esiste un arco all'indietro (u, v), durante la visita di u, uno dei vicini di u è v che essendo antenato di u è già stato scoperto ma la sua visita non è terminata. Viceversa, se non esistono archi all'indietro, gli archi uscenti da un nodo u che viene visitato sono archi dell'albero se il vicino non è stato ancora scoperto; oppure trasversali e in questo caso il vicino è stato scoperto ma la sua visita è già terminata; oppure archi in avanti, ma di nuovo il vicino è stato scoperto e la sua visita è già terminata. Quindi nessun nodo ha vicini scoperti la cui visita non sia già terminata.

Dai due lemmi si ricava che:

Proprietà 1

Nell'albero di visita DFS esiste un ciclo se e solo se durante la visita di un nodo si trova un suo vicino che è già stato scoperto ma la cui visita non è terminata.
Da questo si deduce lo pseudocodice per la ricerca di un ciclo su un grafo orientato (T `e l'insieme dei nodi la cui visita è terminata):

PSEUDOCODICE:

```
T = ∅;
S = ∅;
void DFS(nodo){
        S ← nodo;
        per ogni vicino di nodo {
                se vicino ∉ S {
                        DFS(vicino);
```

```
                }
                else se vicino ∉ T {
                        hasCycle = true;
                }
        }
        T ← nodo;
}
```

Dai due lemmi si ricava anche la proprietà fondamentale della DFS:

Proprietà 2 :

In una visita DFS di un grafo orientato, in assenza di cicli, la visita di un nodo termina dopo che è terminata la visita di tutti i suoi discendenti. (Se sono presenti cicli, questa proprietà non è valida.)

DAG:

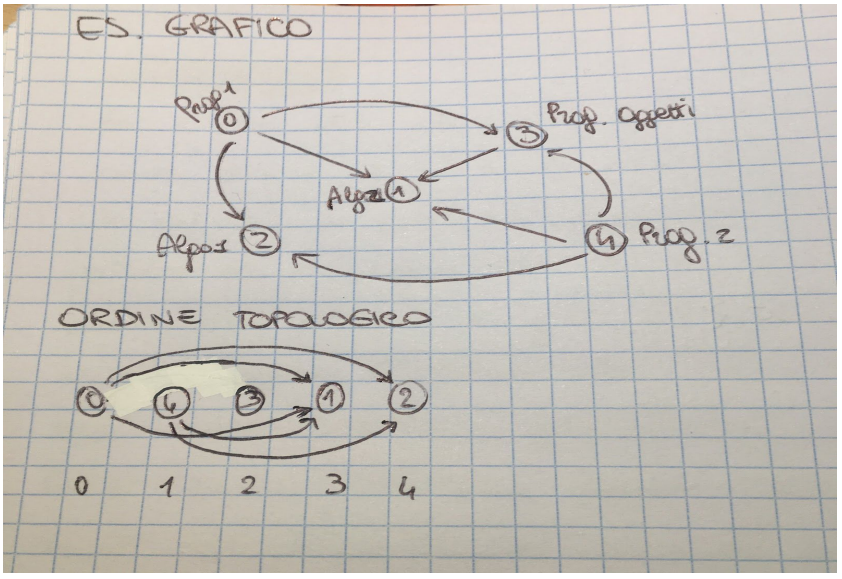
Un Directed Acyclic Graph, quindi un grafo orientato aciclico.

ORDINE TOPOLOGICO

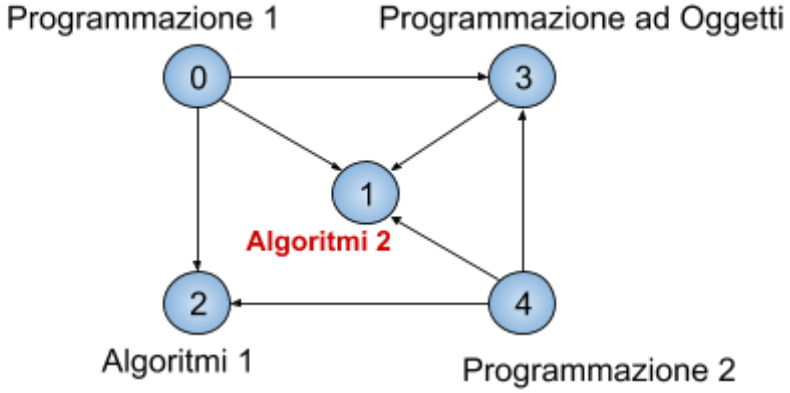
Quando si parla di ordine topologico stiamo sicuramente parlando di grafi orientati e aciclici (DAG).
Se un grafo ha un ciclo allora non ha un ordine topologico.
Gli archi vanno solo da sinistra verso destra.
Si parla quindi di DAG (Directed Acyclic Graph).

Dato un **DAG** G=(V,E) un **OT** di G è una numerazione η:V -> {0,1,..., n-1} dei suoi vertici tale che per ogni arco (u,v) ∈ E vale η(u) < η(v). In altre parole, se disponiamo i vertici lungo una linea orizzontale in base alla loro numerazione η, in ordine crescente, otteniamo che gli archi risultano tutti orientati da sinistra verso destra.

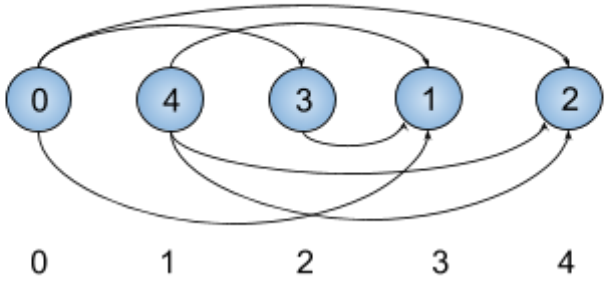
I grafici ciclici NON hanno un ordinamento topologico



GRAFO DI ESEMPIO

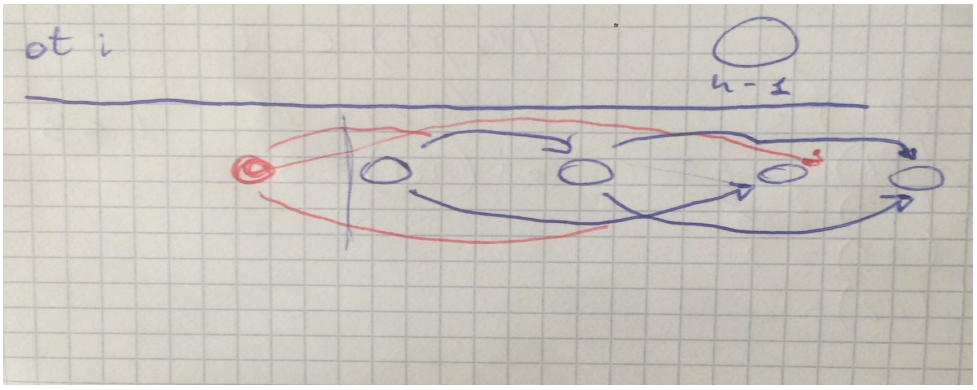


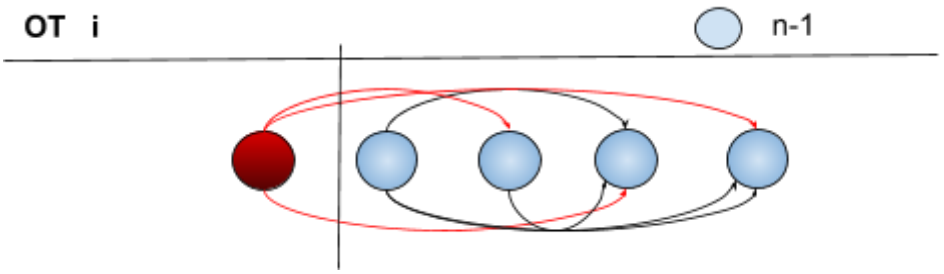
ORDINE TOPOLOGICO



ot : V -> {0,1,2,...,|V|-1} ot{3}=2 ot{2}=4

Se si considera l'arco (u,v) appartenente a E l' ordine topologico ot(u)<ot(v).
Infatti ot(u) corrisponde alla posizione di u nell'ordine topologico, e dato che è presente l'arco (u,v) e dato che gli archi vanno solo da sinistra verso destra(nell'ordine topologico) allora ot(u)<ot(v).
Per scoprire l'ordine topologico di un DAG si utilizza la visita DFS. Precisamente si utilizza la proprietà della dfs nei grafi aciclici, cioè che la terminazione della visita di un nodo avviene solo dopo la terminazione dei suoi discendenti, in assenza di cicli.





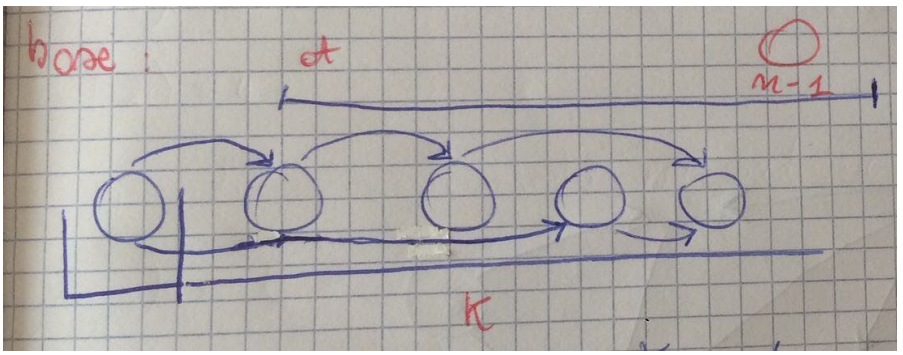
Se il primo nodo termina vuol dire che tutti gli altri nodi sono terminati perché sono tutti suoi discendenti.
Nella visita DFS il primo nodo a terminare sarà un nodo che non ha archi uscenti e corrisponderà all'ultimo nodo nell'ordine topologico.
Come vedremo poi nello pseudocodice.

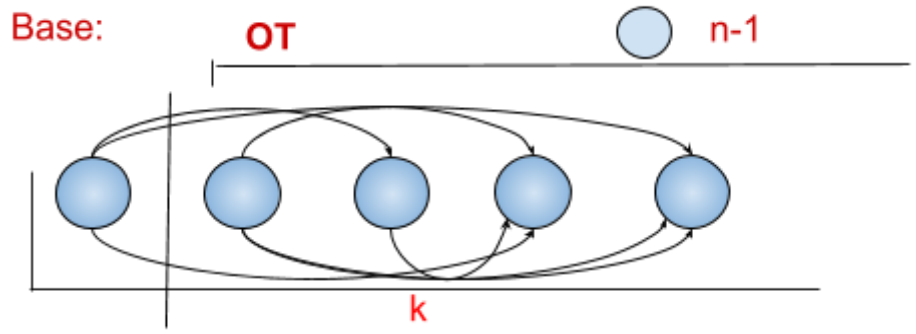
Dimostrazione per induzione:

Cerchiamo di dimostrare che con la visita DFS si riesce a ricavare l'ordine topologico di un grafo aciclico in quanto tutti gli archi vanno verso destra.

Caso base:

Il primo nodo che termina ha come ordine topologico n-1 ed è corretto perché il primo nodo che termina non ha discendenti, se invece fosse presente un ciclo non sarebbe possibile farlo.Quindi è banalmente verificato che tutti gli archi vanno verso destra in quanto quel nodo avrà solo archi entranti.

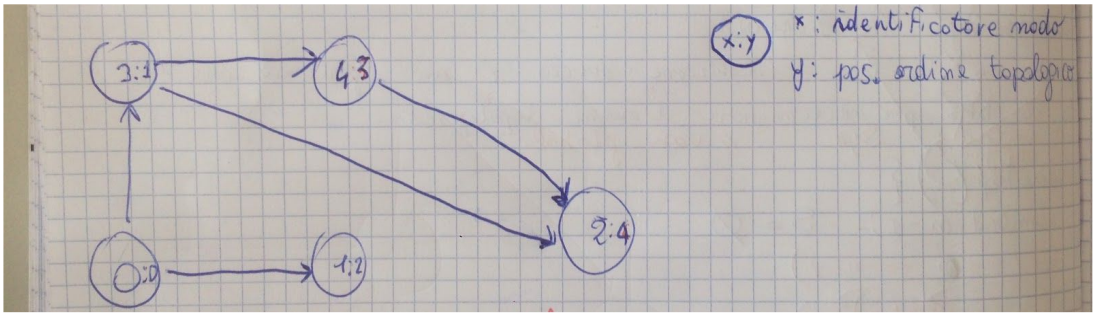




Ipotesi induttiva:
Per K nodi è verificato che tutti gli archi vanno verso destra.

Passo induttivo:
Una volta che K nodi sono stati sistemati e tutti gli archi vanno verso destra, con l'aggiunta del k+1 esimo nodo siamo sicuri che gli archi vadano verso destra, perchè usiamo come k+1 esimo nodo il primo nodo presente in figura(cioè il primo antenato), il quale dato che ha solo archi uscenti avrà solo archi che vanno verso destra.
Il primo nodo che vediamo in figura corrisponde all'ultimo nodo che termina il quale avrà sicuramente solo archi uscenti perchè se avesse degli archi entranti allora finirebbe prima di qualche altro nodo(il quale è un suo antenato), oppure vorrebbe dire che c'è un ciclo e non sarebbe applicabile l'ordine topologico.(in quanto il grafo non sarebbe un DAG)
Quindi per il nodo u che è l'ultimo a terminare ot(u)< ot(di tutti gli altri).

Esempio di ordine topologico:



In questo caso il nodo di partenza è il nodo 4.

Correttezza ordinamento topologico

L'algoritmo per l'ordinamento topologico funziona correttamente se facciamo l'ipotesi che il grafo non abbia cicli. La correttezza dell'algoritmo si basa sul fatto che, in assenza di cicli, una DFS termina la visita di un nodo quando la visita di tutti i discendenti di quel nodo è già

terminata. Notate che se il grafo avesse un ciclo, esso conterrebbe un arco all'indietro rispetto all'albero di visita DFS e la visita di un nodo del ciclo terminerebbe prima che sia terminata quella del suo antenato che è all'altra estremità dell'arco all'indietro. L'arco all'indietro rimarrebbe quindi anche tra i nodi ordinati dall'algoritmo, e quindi il risultato non sarebbe un ordinamento topologico.

Pseudocodice:

```
// variabili globali

int numOrd;
S; // non viene specificato il tipo
int[ ] ot; // ordine topologico nel vettore inizializzato a un valore non significativo, es -1


void ordineTop()
{
        numOrd= n-1;
        S= Ø;
        for(int i=0; i<n; i++)
        {
                DFS(i);
        }
}

void DFS(v)
{
        S=S U {v};
        ∀ vicino u di v  {
                if (u ∉ S)
                {
                        DFS(u);
                }
        }
        ot[v]= numOrd--;
}
```

Complessità:
tempo O(n+m) e spazio O(n)

GRAFO ORIENTATO FORTEMENTE CONNESSO

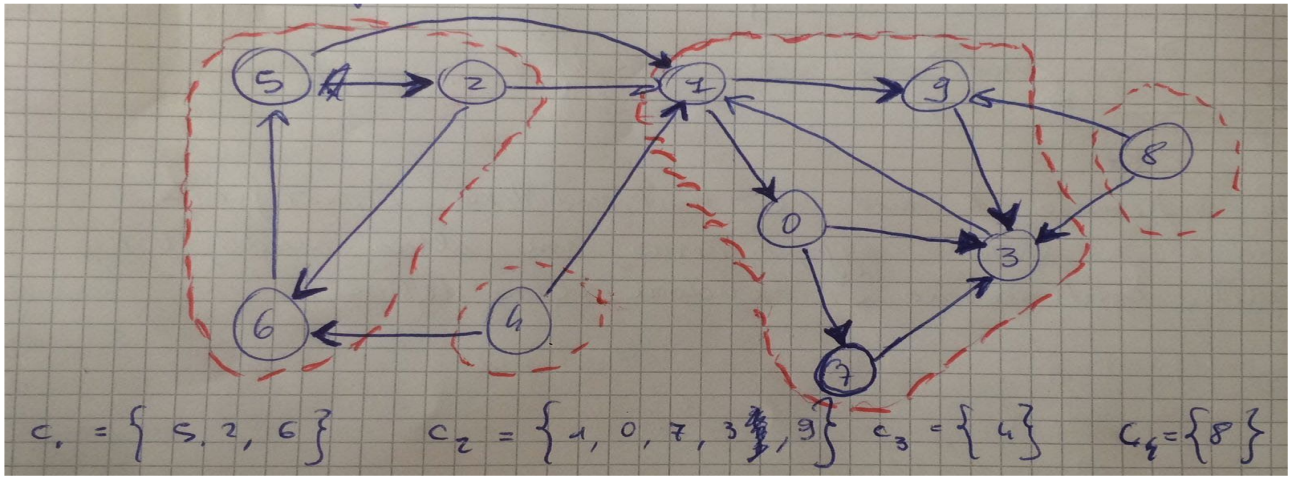
Un grafo G orientato è fortemente connesso se per ogni coppia di nodi (u,v) ∈ V esiste un cammino orientato u->v.
Quindi deve esistere un cammino tra 2->3 ma anche tra 3->2.

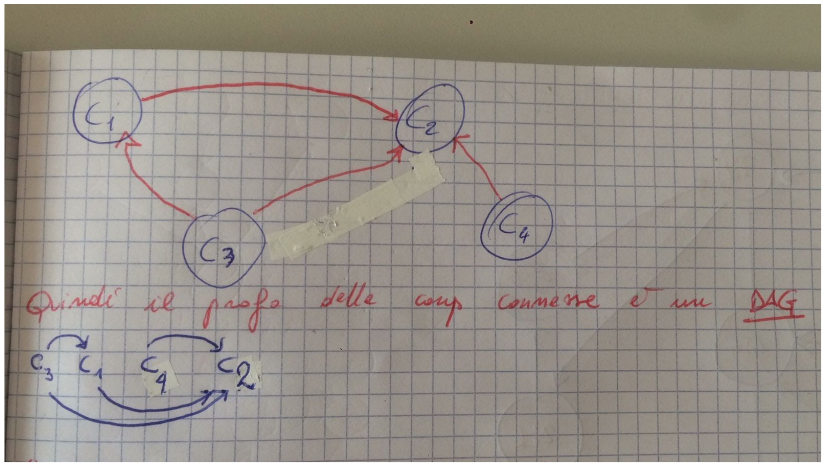
COMPONENTE FORTEMENTE CONNESSA(SCC)

Strongly Connected Component SCC

G'⊆G é una componente fortemente connessa se G' è un sottografo massimale G'=(V',E') e ∀ coppia di nodi (u,v) ∈ V' esiste un cammino u->v .
Devono valere entrambe queste condizioni.

GRAFO DELLE COMPONENTI CONNESSE

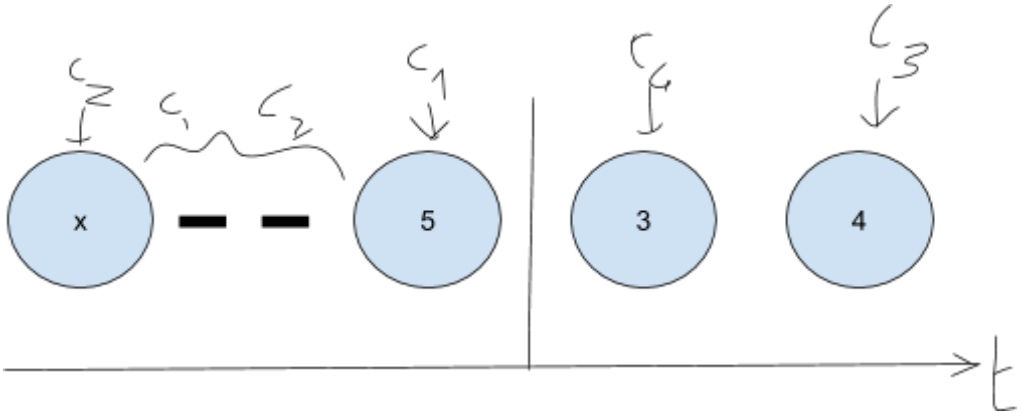




Il grafo delle componenti fortemente connesse è un grafo orientato aciclico.(DAG)
È sicuramente aciclico perchè se no non sarebbero componenti connesse distinte, per esempio se ci fosse un arco (C2,C3) le componenti C1, C2,C3 potrebbero essere considerate come un'unica componente.

Lemma
Se consideriamo 2 componenti fortemente connesse e avviene una visita dfs completa su queste 2 componenti possiamo dire con certezza che il nodo che termina per il ultimo appartiene alla componente connessa che ha un arco uscente verso l'altra componente.

Quindi nel nostro esempio si può dire che uno dei nodi che termina per ultimo sarà in C3 o C4 (non hanno archi entranti ma solo uscenti).
Ricordiamo inoltre che esiste un ciclo esterno che serve per fare in modo che vengano analizzate tutte le componenti connesse.
In figura si può illustrare un esempio di ordine di visita, l'ordine topologico di questa visita sarà quindi: C3,C4,C1,C2.(ma sarebbe potuto essere anche C3,C1,C4,C2)

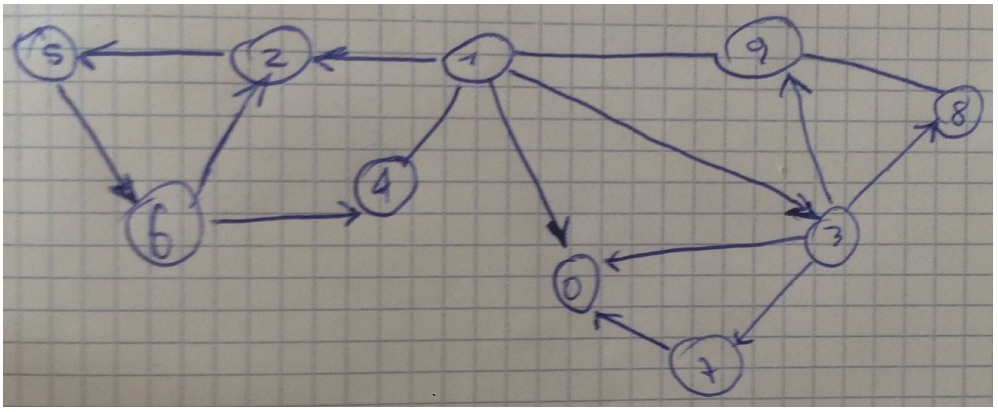


ALGORITMO DI KOSARAJU

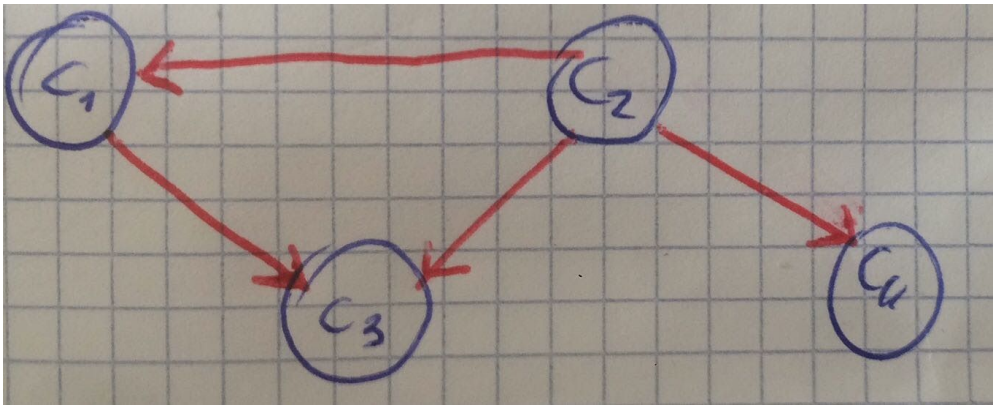
L'algoritmo di Kosaraju si occupa di distinguere le diverse componenti fortemente connesse, per farlo ci sono dei passaggi da seguire:

1. Visita Dfs di G
2. Memorizzare l'ordine di fine visita
3. G'=G trasposto
4. Visita di G' completa in ordine di fine visita(partendo dall'ultimo nodo visitato all'indietro(cioè dal nodo che ha terminato per ultimo))

La prima dfs serve per memorizzare l'ordine di fine visita sfruttando la proprietà della DFS che un nodo termina la sua visita solo dopo la terminazione dei suoi discendenti (considerando anche che parlando di componenti fortemente connesse l'ultimo nodo a terminare sarà della componente che ha solo archi uscenti e le componenti che termineranno prima di lui probabilmente saranno discendenti dell'ultima componente che abbiamo appena considerato, oppure non ci sono "legami tra queste componenti"(considerando il fatto che è presente un ciclo esterno)).
A questo punto si calcola il grafo trasposto G' il quale manterrà le stesse componenti connesse ma solo con archi invertiti.



Grafo delle componenti connesse trasposto.



A questo punto si può utilizzare indipendentemente una visita dfs o bfs (perchè utilizzando l'ordine di fine visita trovato grazie alla visita dfs nella prima visita con l'utilizzo di una bfs, anche se opera a livelli, la visita partirà da nodi di componenti le quali non hanno archi uscenti verso altre componenti ma solo entranti, si riuscirebbe a distinguere comunque le varie componenti fortemente connesse) per calcolare le componenti connesse a partire dall'ordine di fine visita.
In questo modo partiremo nel nostro caso dalla componente C3, il quale ha solo archi entranti quindi una volta terminata la visita su C3 non saranno ancora stati scoperti nodi delle altre componenti e quindi non ci sono problemi, successivamente guarderemo un nodo della componente C4 il quale anch'esso ha solo archi entranti.
A questo punto sempre seguendo l'ordine di fine visita vedremo un nodo di C1 il quale ha solo un arco uscente che va verso C3 il quale però è già stato tutto visitato quindi anche qui non ci sono problemi per etichettare i nodi della componente C1.
Infine bisogna visitare un nodo di C2 la cui componente anche se ha ben tre archi uscenti non da problemi in quanto tutte le componenti a parte C2 sono state scoperte.

Primo esonero fino a qua. 3 settimana di aprile.
Lezione del 12/04/18

Tecnica Greedy

Algoritmo di Dijkstra

Abbiamo visto che con la BFS possiamo calcolare le distanze dai grafi Orientati e non.
Supponiamo di dover costruire un navigatore: tra le due località dovrò attraversa delle strade che hanno una certa lunghezza (peso degli archi).
Possiamo utilizzare le idee della BFS se voglio sapere le distanze dalla sorgente quindi voglio sapere la lunghezza del cammino minimo pesato(distanza).
Trasformiamo allora il grafo in un grafo con tanti nodi e archi di peso unitario per poter utilizzare l'algoritmo di visita BFS. Questa operazione si può fare sempre ma è inefficiente in termini di spazio (anche se rimane lineare in tempo)
Questo algoritmo è quindi esponenziale nella dimensione dell'input del problema.
Introduciamo quindi l'algoritmo di Dijkstra che è un algoritmo greedy ovvero uno di quegli algoritmi che adottano la strategia di prendere quella decisione che, al momento, appare come la migliore. In altri termini, viene sempre presa quella decisione che, localmente, è la decisione ottima, senza preoccuparsi se tale decisione porterà a una soluzione ottima del problema nella sua globalità.
Questo tipo di algoritmo è un classico esempio che risolve i problemi di ottimizzazione(risposta ottimale secondo qualche criterio).
NOTARE: non funziona con archi pesati negativi.Funziona su grafi orientati e non orientati.

Pseudocodice algoritmo di Dijkstra:

```
Dijkstra(Sorg)
{
        S= {sorg};
        d[sorg]=0; // inizializzato a d[i]=+∞
        A= {Ø}; // serve per costruire l'albero

        finché è possibile:
        {
                scegli arco(u,v) u ∈ S , v ∉ S t.c  d[u]+peso(u,v) è minimo{
                        S= S U {v};
                        d[v]= d[u]+peso(u,v);
                        A=A U {(u,v)}
                }
        }
}
```

Complessità senza Heap: O(n*m) -> Analizza tutti i nodi e i suoi archi costa troppo!

Crea l'albero dei cammini minimi (S,A).

Dimostrazione per induzione che prende le distanze minime dalla sorgente

Caso base: S={sorg} |S|=1 (cardinalità nodi visitati)
dist{sorg}=ϑ(sorg,sorg)
La distanza della sorgente nell'albero di visita è uguale alla distanza nel grafo.
Infatti l'algoritmo la pone=0;
Ip Induttiva:|S|=k
Sono stati visitati k nodi e vale che: per ogni v ∈ S dist[v]=ϑ(sorg,v).
Tesi:|S|=k+1: per ogni v ∈ S dist[v]=ϑ(sorg,v).
Viene visitato il nodo v perchè viene attraversato un arco che attraversa la frontiera.
E' stato scelto quell'arco perchè era quello con peso minimo tra gli altri archi connessi alla sorgente.
Consideriamo qualsiasi altro cammino qualunque che arriva a v.
Questo cammino passa dai nodi scoperti e oltrepassa la frontiera.
Qualsiasi cammino io prenda avrà distanza >= di quello preso dall'algoritmo:
Dist[x]+peso(x,y)>= dist[u]+peso(u,v)

Supponendo 2 percorsi:
c=lungh(sorg,u)+peso(u,v)
c'=lungh(sorg,x)+peso(x,y)+lungh(y,v)
Abbiamo che il percorso
lungh(c')=lungh(sorg,x)+peso(x,y)+lungh(y,v)>=lungh(sorg,x)+peso(x,y)
essendo una minoranza (grazie al fatto che i pesi sono tutti positivi) siccome la distanza dalla sorgente a x nel grafo è il cammino minimo sarà >=δ(sorg,x)+peso(x,y) il quale per ipotesi induttiva è =dist[x]+peso(x,y) il che è >=dist[u]+peso(u,v)=lungh(c) perchè l'algoritmo di dijkstra usa una tecnica greedy, sapendo infatti che la distanza[u] per ipotesi induttiva è calcolata correttamente allora il percorso c risulta essere migliore del percorso c'.

Pseudocodice algoritmo di Dijkstra con heap:

```
Dijkstra(Sorg)
{
        S= {sorg};
        d[sorg]=0; // inizializzato a d[i]=+∞
        per ogni (sorg,v)∈ E t.c v non appartiene a S
                heap←-((sorg,v),d[sorg]+peso(sorg,v))

        while heap non vuoto
        {
                (u,w)<--- heap.ExtractMin;
                if  w ∉ S {
                        S= S U {w};
                        d[w]= d[u]+peso(u,w);
                        per ogni (w,z) ∈ E, t.c. z ∉ S
                                heap←---((w,z),d[w]+peso(w,z))
```

```
                }
        }
}
```

Complessità con Heap: O(2m*log(n))
Dove 2m corrisponde a m inserimenti di archi ed estrazioni.
Mentre log(n) è il costo dell'operazione nell'heap.
Se il grafo è denso --->O((n^2)*2*log(n))--->(((n^2)*log(n))
Se il grafo è sparso-->O(n*log(n))

Lezione 19/04/18

Sequenziamento di processi fatto in classe

Si considerino n processi p1,p2..pn con lunghezze positive $l_1,l_2.....,l_n$.
I processi vengono eseguiti sequenzialmente uno dietro l'altro una volta sola.
S:($P_{TT(0)}$,$P_{TT(1)}$,.....$P_{TT(n-1)}$)
π=Indica la posizione del processo.
Quindi $P_{TT(0)}$indica il processo in posizione 0, e non è detto che il processo P1 sia nella posizione $P_{TT(0)}$

$T_{TT(i)}=\Sigma^{i}_{j=0}\, l_{\pi(j)}$
Indica il tempo di attesa del processo in posizione i-esima.
Da questa formula possiamo ricavare il tempo di attesa medio
Am= 1/n($\Sigma^{n-1}_{i=0}\, T_{\pi(i)}$*);*
Quindi il tempo di attesa medio corrisponde alla sommatoria dei tempi di attesa di tutti i processi diviso per il numero di processi.

Esempio

Dati due processi P_0 di lunghezza lunghezza 3 e P_1 di lunghezza 100, eseguendoli nell'ordine (P_0, P_1), il tempo di attesa medio `e:

Am= ½(3+103)=53

se li si eseguono in ordine inverso, il tempo di attesa medio diventa:

Am1=½(100+103)=101,5 >Am

Questo esempio suggerisce di eseguire prima i processi che hanno lunghezza inferiore.

L'obiettivo è minimizzare il tempo di attesa medio:

Si utilizza così un Algoritmo di tipo greedy: si sceglie iterativamente come prossimo processo da eseguire il processo di lunghezza minima non ancora eseguito.
Permutazione greedy $\pi_{(G)}$

$l_{\pi G(0)} < l_{\pi G(1)} < .... < l_{\pi G(n-1)}$

Cerchiamo una soluzione diversa da quella greedy
Algoritmo dello scambio
$S:(P_{TT(0)},.....,P_{TT(i)},P_{TT(i+1)}.....P_{TT(n-1)})$
Dove : $\mathbf{l_{TT(i)}>l_{TT(i+1)}}$

$Am=1/n(T_{TT(0)}+.....+T_{TT(i-1)}+T_{TT(i)}+T_{TT(i+1)}+.....+T_{TT(n-1)})$

$Am=1/n(T_{TT(0)}+.....+T_{TT(i-1)}+(T_{TT(i-1)}+l_{TT(i)})+(T_{TT(i-1)}+l_{TT(i)}+l_{TT(i+1)})+.....+T_{TT(n-1)})$

Consideriamo una soluzione

$S':(P_{TT(0)},.....,P_{TT(i-1)}P_{TT(i+1)},P_{TT(i)},P_{TT(i+2)}....P_{TT(n-1)})$

$Am'=1/n(T_{TT(0)}+.....+T_{TT(i-1)}+(T_{TT(i-1)}+l_{TT(i+1)})+(T_{TT(i-1)}+l_{TT(i+1)}+l_{TT(i)})+.....+T_{TT(n-1)})$

Eseguendo la sottrazione Am'-Am si semplificano vari addendi cioè quelli evidenziati in rosso e in blu
Quindi rimane
$Am'-Am=1/n(l_{TT(i+1)}-l_{TT(i)})$
Come ipotesi iniziale avevamo che $\mathbf{l_{TT(i)}>l_{TT(i+1)}}$ quindi Am'<Am
Abbiamo preso una soluzione non greedy abbiamo fatto uno scambio e ne abbiamo ottenuta una migliore.
Quindi qualsiasi altra soluzione non greedy non è ottimale.
Se al posto di avere delle lunghezze tutte diverse ce ne fossero di uguali:
$l_{\pi G(0)} <= l_{\pi G(1)} <= .... <= l_{\pi G(n-1)}$
Allora avremmo più soluzioni ottimali in quanto ci sono più combinazioni.
Infatti se si scambiano due lunghezze uguali non miglioriamo nulla.

Sequenziamento di processi della prof

Si considerino n processi $j_1,....,j_n$ con lunghezze positive $l_1,.....,l_n$. Se i processi vengono eseguito nell'ordine $j_1,j_2….,j_{n,}$ definiamo il tempo di attesa T_k del processo $j_k=j_{is}$ (cioè del processo j_k che viene eseguito come s-esimo processo dopo i processi $j_{i1,}j_{2,},.....,j_{is-1}$) come:
$T_k=T_{IS}= \Sigma^{S}_{H=1}\, l_{IH}$
ovvero il tempo dall'inizio dell'esecuzione di tutti i processi fino al completamento di j_k. Il tempo di attesa medio è dato da
$A= 1/n(\Sigma^{n}_{H=1}\, T_H);$

Lo scopo `e minimizzare il tempo di attesa medio.

Problema del sequenziamento di processi: Dati n processi $j_1, \ldots, j_n$ con lunghezze positive $\ell_1, \ldots, \ell_n$, determinare un ordine di esecuzione dei processi che minimizzi il tempo di attesa medio.

1.1 SOLUZIONE GREEDY

Esempio semplice: dati due processi j_1 di lunghezza 4 e j_2 di lunghezza 7, eseguendoli nell'ordine (j_1, j_2), il tempo di attesa medio `e:

A= ½(4+11)=7.5

se li si eseguono in ordine inverso, il tempo di attesa medio diventa:

A=½(7+11)=9 > A

Questo esempio suggerisce di eseguire prima i processi che hanno lunghezza inferiore:

Algoritmo di Sequenziamento di processi: si sceglie iterativamente come prossimo processo da eseguire il processo di lunghezza minima non ancora eseguito.

DIMOSTRAZIONE CORRETTEZZA:

Teorema 1: *L'algoritmo* Sequenziamento di processi *determina un ordinamento dei processi che minimizza il tempo di attesa medio.*

Dimostrazione: Si supponga per assurdo che la soluzione g calcolata dall'algoritmo greedy non sia ottimale. Allora esiste un'altra soluzione s che produce tempo di attesa medio A_s inferiore a quello di g, A_g.

Per semplicità si supponga per il momento che le lunghezze ℓ_j siano tutte distinte. Dunque è possibile riordinare i processi in ordine crescente di ℓ_j e rinominarli in modo che $\ell_j < \ell_{j+1}$. Di conseguenza, la soluzione greedy `e

$$g = (j_1, j_2, \ldots, j_n).$$

Se la soluzione s ottimale è diversa da g, esistono due processi j_h e j_k, con $h > k$ (e quindi $\ell_h > \ell_k$) che in s sono programmati uno dopo l'altro ma con j_h prima di j_k:

$$s = (j_{i_1}, \ldots, j_{i_u}, j_h, j_k, j_{i_{u+3}}, \ldots, j_{i_n}).$$

Scambiando j_h e j_k, si ottiene una soluzione s' uguale a s eccetto che per i due processi scambiati:

$$s' = (j_{i_1}, \ldots, j_{i_u}, j_k, j_h, j_{i_{u+3}}, \ldots, j_{i_n}).$$

Il tempo di attesa media A_s `e dato dalla seguente espressione:

$$A_s = 1/n(\Sigma^u_{a=1} T_{ia} + T_h + T_k + \Sigma^n_{a=u+3} T_{ia})$$

Si osservi che lo scambio non altera il valore delle due sommatorie e il tempo di attesa medio di s' `e dunque:

$$A_{s'} = \frac{1}{n}\left(\Sigma_{a=1}^{u} T_{i_a} + T'_k + T'_h + \Sigma_{a=u+3}^{n} T_{i_a}\right),$$

dove T' e T' sono i tempi di attesa di j_h e j_k in s'. Si noti che $T' = T_h + \ell_k$ e $T' = T_k - \ell_h$. Dunque

$$A_s - A_{s'} = \frac{1}{n}(T_h + T_k - T'_k - T'_h) = \frac{1}{n}(\ell_h - \ell_k).$$

Poichè per ipotesi $\ell_h > \ell_k$, abbiamo $A_s - A_{s'} > 0$, contraddicendo l'ipotesi che s sia ottimale.

Nel caso generale in cui esistono coppie di processi di uguale lunghezza, la dimostrazione è analoga, ma non per assurdo.
Partendo da una soluzione ottimale s, diversa da quella greedy, si osserva che, scambiando due processi j_k e j_h programmati uno dopo l'altro ma in ordine diverso da quello previsto dall'algoritmo greedy, si ottiene una nuova soluzione s' non peggiore della precedente (ma non necessariamente migliore perch´e ci sono coppie di processi con uguale lunghezza) e piu´ simile alla soluzione greedy.
Ripetendo il procedimento un numero sufficiente di volte, si ottiene la soluzione greedy, il cui tempo di attesa medio non `e dunque peggiore di quello della soluzione ottimale.

Si noti che, nel caso in cui le lunghezze non sono tutti distinte, non esiste una soluzione ottimale unica, e quindi esistono strategie di scelta equivalenti (che specificano come ordinare processi di uguale lunghezza).

Grafi non orientati connessi pesati

Minimo albero ricoprente(Minimum Spanning Tree)

MST:

Dato un grafo G=(V,E) non orientato e connesso, si può trovare il minimo albero ricoprente T = (V,E') dove ci sono in V tutti i nodi del grafo.

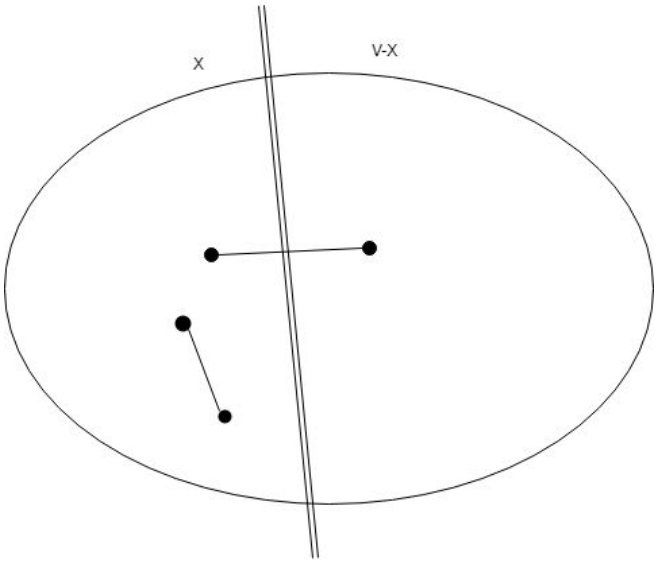
E' minimo se la somma dei pesi dei suoi archi è minore rispetto tutti i possibili alberi ricoprenti

Il costo è la somma dei pesi sui singoli archi.

Teorema del taglio

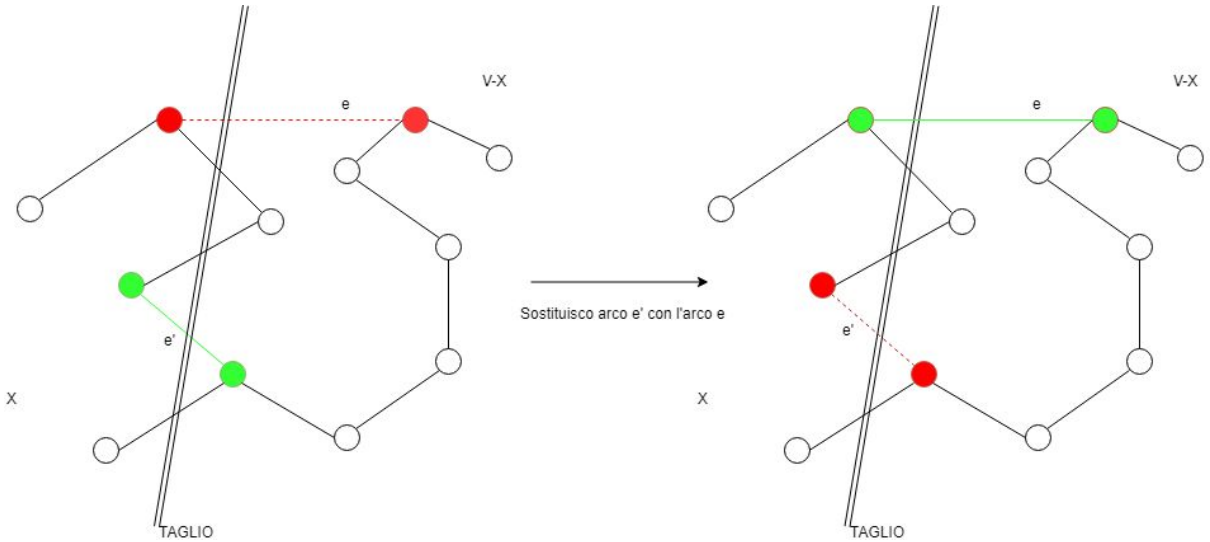
Dato un grafo G(V,E)

X=Sottoinsieme dei nodi
V-X=Secondo sottoinsieme dei nodi.
X != {} L'insieme X non deve essere un insieme vuoto
X != V L'insieme X non deve essere la totalità del grafo



Attraversano il taglio gli archi che hanno un estremo in un sottoinsieme X e l'altro estremo nel sottoinsieme V-X.
Teorema del taglio: *L'arco di peso minimo che attraversa il taglio appartiene al minimo albero ricoprente.(MST)*

T: albero ricoprente di G



Verde→ *arco presente nel grafo*
Rosso→ *arco non presente nel grafo*
e → *arco di peso minimo che attraversa il taglio*
Che succede se io aggiungo l'arco di peso minimo 'e' che non faceva parte dell'attuale albero di visita?
Non è più un albero perché si crea un ciclo.
Siamo sicuri che si crei un ciclo perchè ci deve essere almeno un secondo arco che attraversa il taglio.
Degli archi che attraversano il taglio sappiamo che **peso(e') > peso(e)** perché e è il minimo arco che attraversa il taglio.

Quindi con la sostituzione di e' con e si crea un nuovo albero ricoprente del quale sappiamo che:
P(T')= P(T) - P(e') +P(e)
Quindi dato che P(e) < P(e') siamo sicuri che P(T') < P(T).
Quindi l'albero da cui siamo partiti non era il minimo albero ricoprente.(MST)
Il teorema del taglio dice che tra tutti gli archi che attraversano il taglio, il minimo appartiene per forza al minimo albero ricoprente(MST).
Se così non fosse si può sempre fare uno scambio (di archi) che lo migliora.

Algoritmo di Prim/Jarnik

Grafi connessi e non orientati

Enunciato:
L'algoritmo di Prim calcola correttamente il minimo albero ricoprente di un grafo non orientato e connesso.
Ipotesi:
Le ipotesi sono: il grafo è non orientato e connesso, l'algoritmo è una visita di grafo in cui la scelta del nuovo nodo da visitare è greedy e minimizza il peso del nuovo arco scelto.
Tesi:
La tesi è che questo algoritmo calcola il minimo albero ricoprente del grafo (ovvero che costruisce un albero, che tale albero ricopre il grafo, e che l'albero è di peso minimo tra tutti gli alberi ricoprenti).

Prim genera un minimo albero ricoprente o MST(minimum spanning tree) e ciò è dimostrato dal teorema del taglio che divide il grafo in due sottoinsiemi e di questi va a prendere l'arco minimo che attraversa il taglio. Questo sicuramente apparterrà al MST.

- E' ricoprente poichè, essendo un algoritmo di visita di un grafo connesso, vengono visitati tutti i nodi del grafo.

- E' un albero poichè è aciclico e connesso → E' connesso poichè vi è un cammino da una radice a tutti gli altri nodi e anche perchè ad ogni iterazione viene aggiunto un arco tra un nodo visitato e un nodo non ancora visitato. Non ha cicli poichè ogni nodo ha uno e un solo padre.

Si può quindi dire che, essendo un algoritmo di visita, costruisce un albero ricoprente.

Pseudocodice Prim/Jarnik (funziona anche con pesi negativi)

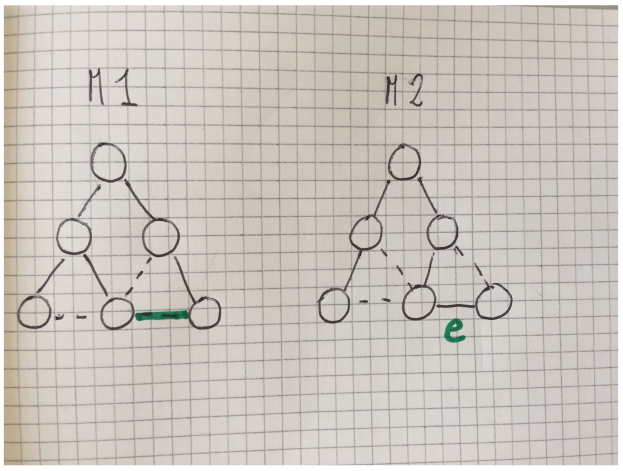
```
Prim(sorg){
        S={sorg};
        A=∅;
        per ogni (sorg,v) appartiene ad E
                heap ← ((sorg,v), peso(sorg,v))
        while(heap non vuoto){
                (u,v) =arco di valore minimo presente nell'heap
                se v non appartiene a S{
                        S = S u {v}
                        A = A u {(u,v)}

                        per ogni (v,w) appartenente ad E // con w si intendono tutti i vicini di v
                                if(w non appartiene a S )
                                        heap ← ((v,w), p(v,w))
                }
        }

}
```

Complessità: *O(2m*log(n))*
La complessità è la stessa di dijkstra.
M estrazioni e inserimenti moltiplicati per il costo dell'operazione sull'heap.

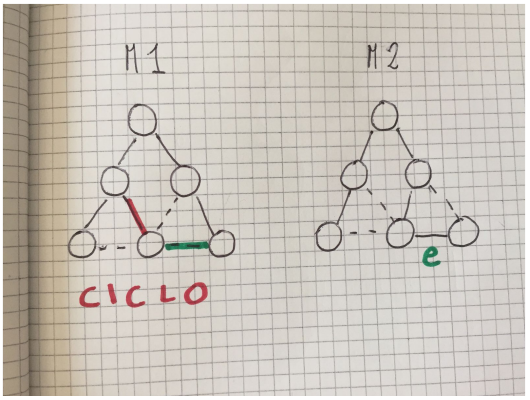
Unicità del Minimo Albero Ricoprente

Teorema: Sia G un grafo pesato, non orientato. Se non ci sono due archi di G che hanno lo stesso peso, G ha un unico albero ricoprente minimo.

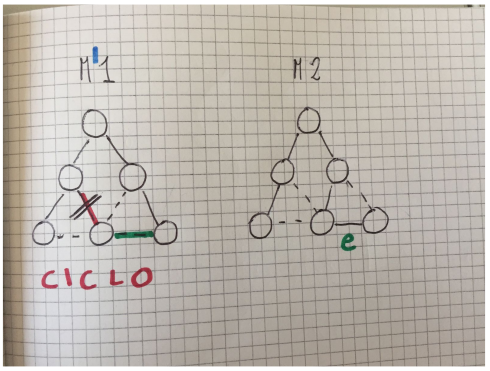
Dimostrazione: Per assurdo, supponiamo che G abbia due minimi alberi ricoprenti distinti **M1** e **M2**. Poiché sono distinti, esiste almeno un arco in uno dei due alberi che non appartiene anche all'altro.



Sia ***e*** l'arco di peso minimo che appartiene ad un albero ma non all'altro. Supponiamo, senza perdita di generalità, che appartenga a **M2**. Allora, se si aggiunge ***e*** ad **M1**, si genera un ***ciclo C***. Questo perché **M1** è un albero ricoprente, quindi è connesso e contiene tutti i nodi del grafo.



Consideriamo il ***ciclo C***. Poiché **M2** non contiene cicli, nel ciclo c'è almeno un arco ***e'*** che non appartiene a **M2**. Tale arco ha peso maggiore di ***e*** perché abbiamo scelto ***e*** come arco di peso minimo che appartiene ad uno dei due alberi ma non all'altro. Togliendo ***e'*** dal ciclo (e quindi da **M1** modificato) si torna ad avere un albero **M1'** , diverso da **M1** perché contiene l'arco ***e***, e non l'arco ***e'*** .



Si tratta di un albero ricoprente di G perché il ciclo non esiste più, **M1'** ricopre il grafo ed è connesso perché ***e'*** faceva parte di un ciclo.

Ma **M1'** ha peso minore di **M1** (poiché ***e'*** è stato sostituito con un arco di peso minore), fatto che contraddice l'ipotesi che **M1** sia un albero ricoprente minimo di G.

✷ Si noti che, **se esistessero nel grafo due archi con lo stesso peso**, allora non potremmo affermare che e ′ ha peso strettamente maggiore di e. Avremmo solo la garanzia che e ′ abbia peso maggiore o uguale ad e; dunque M1' potrebbe avere peso uguale a M1 e quindi non si giungerebbe ad un assurdo. In effetti, se i pesi non sono tutti distinti un grafo può avere più di un minimo albero ricoprente.

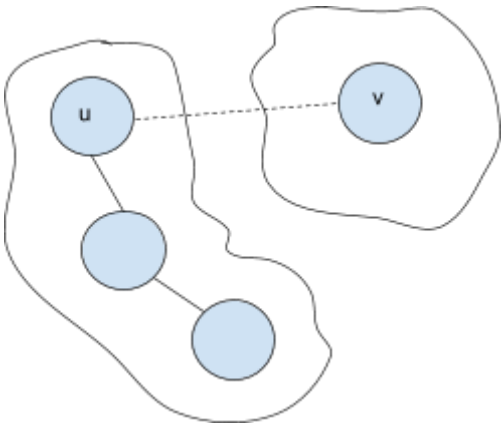
Algoritmo di Kruskal

E' un altro algoritmo per calcolare un MST.
Si parla quindi di grafi non orientati e connessi.
La base dell'algoritmo è ancora una volta il teorema del taglio.
L'algoritmo di Kruskal prende sempre l'arco di peso minimo tra tutti gli archi presenti nel grafo.(considera solo gli archi non i nodi)
Quindi considera qualsiasi taglio che attraversa il grafo.
(Può prendere qualsiasi taglio che prende una componente connessa non già presa.)
Non è un algoritmo di visita.
Nel caso in cui sia presente un ciclo nel grafo,l'arco che prenderà in considerazione verrà scartato dall'algoritmo.

Pseudocodice:

```
A= Ø;
While(finchè si può scegliere un arco)
{
        Sia (u,v) arco di peso minimo disponibile
                se (u,v) non forma un ciclo in (V,A)
                        A= A U { (u,v) };
                altrimenti
                        scarta (u,v);
}
```

Dimostriamo che Kruskal calcoli il minimo albero ricoprente:



MINIMO:

E' minimo perchè ogni volta che considera degli archi, prende in considerazione un arco (u,v) dove u appartiene a una componente connessa e v a un'altra componente connessa(che non forma cicli).
Riusciamo a definire quindi un taglio che ha "u" in una componente e "v" in un'altra, *e l'algoritmo non va a prendere in considerazione altri archi che appartengono a componenti connesse già prese in considerazione.*
Quindi gli archi sono tutti quelli che non avremo ancora preso in considerazione, e di questi prende il minimo.
Siccome prende il minimo grazie al teorema del taglio sicuramente apparterrà al MST.
Intanto va specificato che quando Kruskal sta per aggiungere (u,v) al grafo (albero) che sta costruendo, lo fa perché è l'arco di peso minimo che non ha ancora considerato e che non forma cicli.
Poi il punto è che si riesce a definire un taglio attraversato dall'arco (u,v) ma non da archi già presi in considerazione, perché questo taglio non spezza alcuna componente connessa.
Quindi gli archi già presi in considerazione hanno ambedue gli estremi da una parte o dall'altra del taglio. Quindi (u,v) è il minimo arco che attraversa il taglio.

ALBERO:

Per essere un albero deve essere ***connesso e senza cicli***

- ***Connesso e aciclico:*** è connesso perchè se così non fosse avremmo almeno 2 componenti connesse distinte, ma dato che il grafo di partenza è connesso esiste sicuramente un arco che va da una componente all'altra, il primo arco che connette le 2 componenti connesse sicuramente non crea un ciclo perché collega 2 componenti connesse distinte.

RICOPRENTE:

Si, per costruzione l'algoritmo tocca tutti i nodi del grafo poichè MST=(**V**,A)

Questo algoritmo applica una strategia greedy, in quanto prende l'arco di peso minimo e non torna indietro.

Costo dell'algoritmo

Ho "m" cicli while, e per le operazioni potremmo usare un heap quindi con costo log(n).

Costo complessivo:

O(m • log(n))

Se un arco prende 2 nodi che appartengono già alla stessa componente connessa allora forma un ciclo. Per tenere memoria di quali nodi ci sono nella componente connessa ci servono delle strutture, introduciamo quindi la **UNION FIND.**

Struttura Union Find: Tiene memoria degli insiemi su n oggetti e ci permette di fare la Find(i) la quale trova l'insieme a cui appartiene l'elemento e Union(i,j) che unisce 2 insiemi.

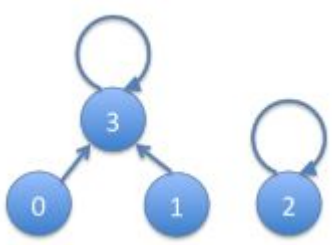
Pseudocodice:

```
A= Ø;
While(finchè si può scegliere un arco)
{
        Sia (u,v) arco di peso minimo disponibile
        se Find(u)!=Find(v)    // Costo O(1) //controlla che abbiano 2 componenti diverse
                A= A U { (u,v) };
                Union(Find(u),Find(v));
        altrimenti
                scarta (u,v);
}
```

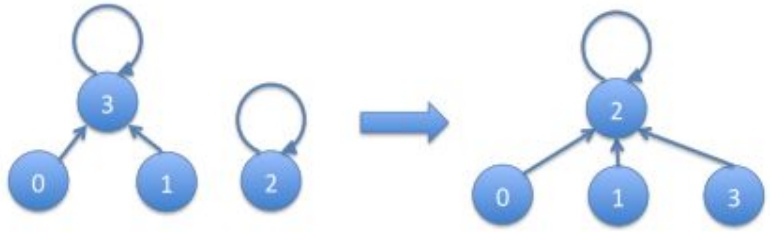
Quick Find

Un primo modo per realizzare la struttura richiesta, consiste nel rappresentarla utilizzando una sequenza di lunghezza n nella quale nella posizione i si trova il rappresentante dell'insieme al quale appartiene l'elemento i.

Ad esempio, se U = {0, 1, 2, 3} e la collezione consiste nei due insiemi {0, 1, 3} e {2}, se 3 è il rappresentante del primo insieme, la collezione sarà rappresentata dalla sequenza (3, 3, 2, 3). La sequenza può essere interpretata come una sequenza di puntatori, nella quale ciascun elemento punta al rappresentante dell'insieme cui appartiene.



Si noti che l'albero che deriva da questa rappresentazione è sempre un albero di profondità al massimo 1, e che è sempre necessario attraversare un solo arco per conoscere il rappresentante dell'insieme cui appartiene un elemento. La sequenza può essere implementata con un array di n elementi, **occupando spazio O(n)**.
La **Union** viene definita in questo documento come funzione che opera sui rappresentanti degli insiemi. Pertanto una **Union(e, f)** non ha alcun senso se e oppure f oppure tutti e due non sono rappresentanti di insiemi.
L'operazione **Find(e)** si implementa in questo caso con **costo O(1)** perché si realizza con un (unico) accesso alla posizione e dell'array. A questo si deve il nome **Quick Find**.
L'operazione di unione deve rinominare tutti gli elementi di uno dei due insiemi assegnando loro il nome del rappresentante dell'altro insieme. Quale dei rappresentanti degli insiemi di partenza diventi rappresentante dell'insieme unione dipende dall'implementazione.
Supporremo nei nostri esempi che il rappresentante di **Union(a, b)** sia sempre a per l'algoritmo **Quick Find**, ma anche la scelta opposta è legittima purché rimanga sempre coerente. Nel caso della collezione (3, 3, 2, 3) dell'esempio precedente, **Union(2, 3)** risulta nella collezione rappresentata da (2, 2, 2, 2). L'effetto dell'operazione è il seguente:

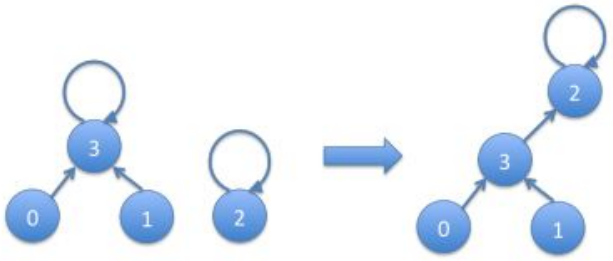


Seguendo la strategia Quick Find l'operazione di unione può avere **costo nell'ordine di O(n)** nel caso peggiore (ad esempio nel caso illustrato nella figura appena descritta è necessario aggiornare n − 1 puntatori).
La complessità finale anche in questo caso rimane quadratica.

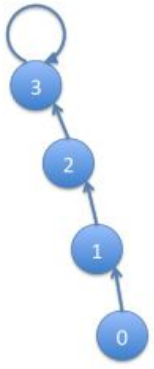
Quick Union

Per ovviare al costo computazionale della Union, si può pensare ad una strategia diversa, detta Quick Union, nella quale si rende più efficiente proprio la Union. L'idea alla base della Quick Union è di aggiornare solo il puntatore del rappresentante che non sarà rappresentante dell'insieme unione, lasciando inalterati i puntatori di tutti gli altri elementi. Cioè Union(a, b) modifica solo l'elemento b della sequenza, 2 ponendolo uguale ad a, se

manteniamo la convenzione di scegliere sempre il primo argomento come rappresentante dell'unione. Quindi adesso l'operazione di unione ha costo costante poiché si realizza con un solo accesso e una modifica. Nell'esempio del paragrafo precedente, Union(2, 3) nella collezione (3, 3, 2, 3) ha come risultato (3, 3, 2, 2) seguendo la strategia Quick Union.
L'effetto dell'unione in questo esempio :



Union(2, 3) nella collezione (3, 3, 2, 3) con strategia Quick Union, poichè è stata adottata la convenzione di scegliere il primo argomento come rappresentante dell'unione. La collezione risultante è rappresentata dalla sequenza (3, 3, 2, 2). Si noti che, con la nuova strategia, l'elemento "e" della sequenza non è più necessariamente il rappresentante dell'insieme al quale "e" appartiene, come nel caso della Quick Find, ma è il padre di "e" in un albero che rappresenta l'insieme "e" che può avere profondità maggiore di 1.
Difatti, nell'esempio, gli elementi 0 e 1 della sequenza hanno valore 3, mentre il rappresentante dell'insieme è 2. Il rappresentante dell'insieme è la radice dell'albero. Di conseguenza, l'operazione Find(e) ha un costo che dipende dalla profondità dell'albero, perché comporta la risalita dell'albero dal nodo etichettato con e fino alla radice. Si osservi che, nel caso peggiore, l'albero può avere profondità O(n).
Si consideri ad esempio il caso in cui, viene effettuata la sequenza di unioni Union(i+1, i) con i = 0, . . . , n−2. Se le unioni vengono effettuate in ordine di i crescente, e si mantiene la convenzione che il rappresentante dell'insieme unione è il primo argomento, l'insieme finale sarà una catena di nodi, rappresentata dalla sequenza (1, 2, 3, 4, . . . , n − 1, n − 1), ovvero da un albero con un unico ramo lungo n − 1 .



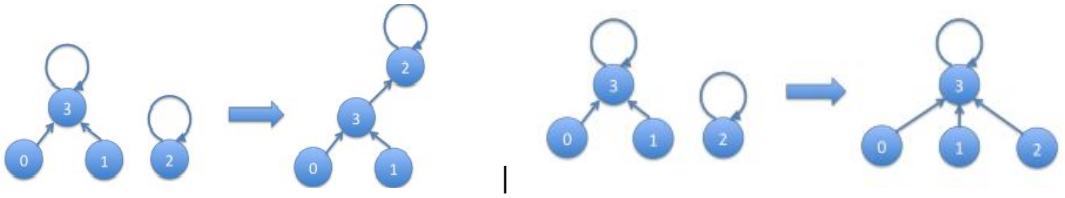
Pertanto in questo caso una Find(0) avrà costo n.
Poichè la FIND deve analizzare dalla foglia alla radice possiamo provare a limitare la profondità con la Union By Rank.

Union By Rank

La strategia Quick Find dunque realizza una Find in tempo costante e una Union nel caso peggiore in tempo lineare nella cardinalità dell'universo U. La strategia Quick Union viceversa realizza una Union in tempo costante ma una Find in tempo lineare nel caso peggiore. E possibile però ottenere un risultato intermedio, realizzando la Union ancora con costo costante e la Find con costo O(log n), al prezzo di mantenere una struttura addizionale che occupa spazio O(n). Si ottiene questo risultato modificando la strategia Quick Union con un bilanciamento dell'albero unione. Difatti, ricordiamo che la Find realizzata seguendo la strategia Quick Union ha costo O(n) perchè la profondità dell'albero che rappresenta un insieme può diventare lineare in n. Analizziamo come cambia la profondità dell'albero unione in funzione della profondità degli alberi di partenza. Notiamo intanto con un esempio che la profondità dell'albero unione può cambiare a seconda della scelta del rappresentante dell'insieme unione. Questo si vede chiaramente confrontando la figura A con la figura B. Il lemma seguente dimostra questa intuizione.

Esempio

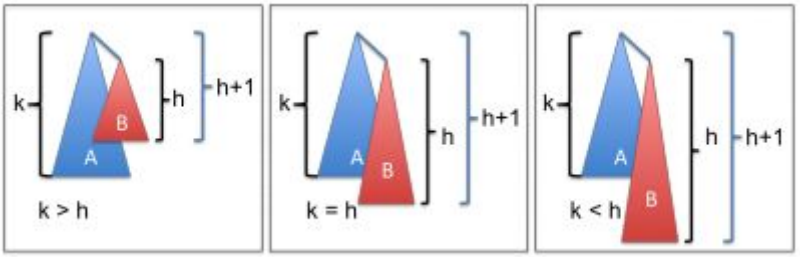
Figura B: Union(2, 3) nella collezione (3, 3, 2, 3) con strategia Quick Union, scegliendo questa volta il secondo argomento come rappresentante dell'unione. La collezione che si ottiene è rappresentata dalla sequenza (3, 3, 3, 3).

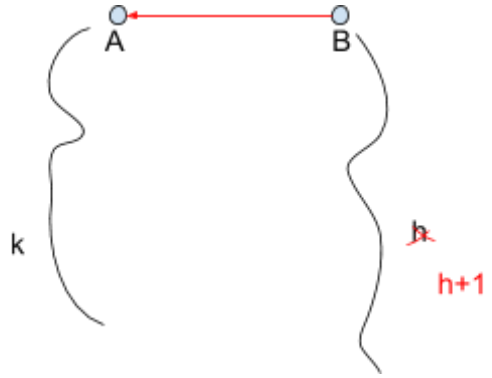


Lemma 1 (Profondità dell'albero unione)

A partire da due alberi A e B si costruisca un albero unione appendendo l'albero B come sottoalbero alla radice di A; cioè si modifica l'arco uscente dalla radice di B in modo che punti alla radice di A, mentre tutti gli altri archi dei due alberi rimangono invariati (Guarda figura).

La profondità dell'albero A U B è in funzione della profondità degli alberi A e B. La profondità di A è k e la profondità di B è h.





La radice di A è anche la radice dell'albero risultante C. Sia k la profondità di A e h la profondità di B.

1. Se k > h, la profondità di C è uguale a k.
2. Se k< h, la profondità di C è uguale a h+1
3. Se k = h, la profondità di C è pari ad h + 1 = k+1.

Dimostrazione: I rami di C sono

• tutti i rami di A, invariati: la loro lunghezza massima è per ipotesi k;

• tutti i rami dell'albero B concatenati con il nuovo arco tra le due radici: la lunghezza massima di questi rami diventa h + 1 cioè la lunghezza massima dei rami di B (h per ipotesi) incrementata di uno per l'aggiunta del nuovo arco.

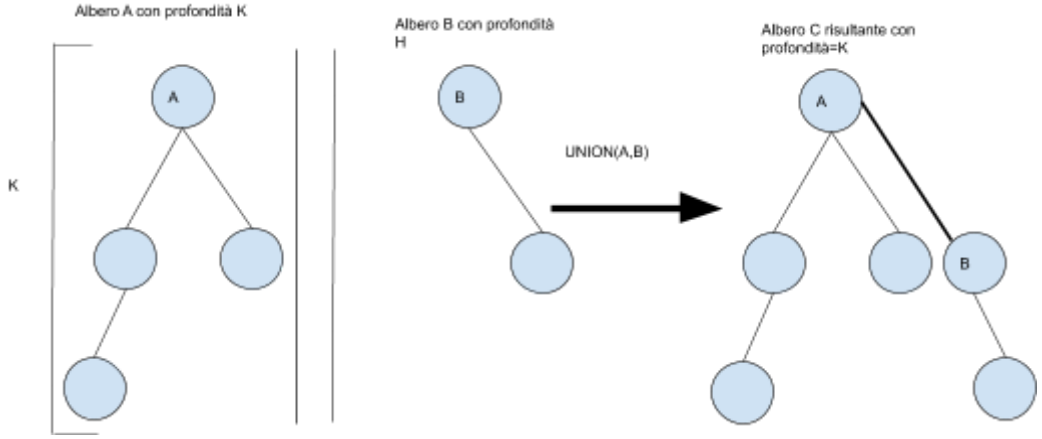
Quindi il più lungo ramo di C è lungo max(k, h + 1).

- Se k > h, h + 1 ≤ k e quindi il più lungo ramo di C ha lunghezza ancora k.
- Se invece k < h, allora k < h + 1 e quindi il più lungo ramo di C ha lunghezza h + 1.

La tesi è dimostrata.

✶ In base a questo ragionamento, se si appende sempre l'albero meno profondo a quello più profondo, si mantiene sotto controllo la profondità dell'insieme unione che aumenta solo quando i due alberi hanno la stessa profondità.

Dunque la strategia (Quick) Union by Rank prevede di mantenere l'informazione aggiuntiva della profondità di ciascun albero. A tale scopo, è sufficiente mantenere una sequenza di lunghezza pari al numero degli elementi, in cui, in posizione i è memorizzata la profondità dell'albero la cui radice è ***i*** (ovvero dell'insieme Ai il cui rappresentante è i).



Se un elemento ***e*** non è rappresentante di alcun insieme, la profondità associata a tale elemento è irrilevante e pertanto, quando un elemento cessa di essere rappresentante di un insieme, in seguito ad una unione, non è più necessario aggiornare il corrispondente valore nella sequenza delle profondità.

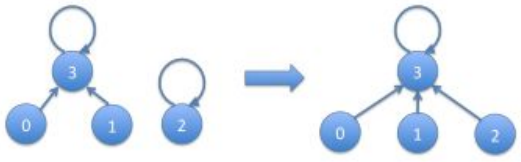
Il nome Union by Rank deriva dal fatto che la profondità degli alberi in questo caso prende il nome di rank dell'insieme e che la Union viene fatta sulla base di tale rank. Nel seguito, useremo la notazione rank(A) per indicare la profondità dell'albero A; tale valore sarà memorizzato nella posizione della sequenza delle profondità corrispondente al rappresentante dell'insieme. A questo punto, per implementare Union(a, b) bisogna determinare la profondità rank(Aa) dell'albero Aa e quella rank(Ab) dell'albero Ab. Poi si procede come segue:

• se rank(Aa) > rank(Ab), a è il rappresentante dell'insieme unione e
rank(Aa U Ab) = rank(Aa); si noti che non `e necessario aggiornare alcun valore della sequenza delle profondità, poiché il rappresentante di Aa U Ab è a.

• se rank(Aa) < rank(Ab), b è il rappresentante dell'insieme unione e è
rank(Aa U Ab) = rank(Ab); si noti che non `e necessario aggiornare alcun valore della sequenza delle profondità, poiché il rappresentante di Aa U Ab è b.

• se infine rank(Aa) = rank(Ab), il rappresentante dell'insieme unione sarà a o b a seconda dell'implementazione, e rank(Aa U Ab) = rank(Aa) + 1 = rank(Ab) + 1; nella sequenza delle profondità dovrà essere aggiornato il valore corrispondente al rappresentante dell'insieme unione.

Ad esempio, nella collezione (3, 3, 2, 3), la sequenza delle profondità è (×, ×, 0, 1), dove × in posizione e indica che il valore in quella posizione non è significativo, poiché e non è il rappresentante di alcun insieme. Dunque la Union(2, 3) verrà realizzata come in questa figura:



poiché la profondità dell'albero di radice 2 è minore della profondità dell'albero di radice 3. Quindi la collezione diventa (3, 3, 3, 3) e le profondit`a (×, ×, ×, 1) (si noti che non `e necessario modificare il valore corrispondente all'elemento 2: semplicemente quel valore non ha più alcun significato e non verrà mai più consultato nel seguito). Il costo della Union definita sui rappresentanti è quindi costante perché è necessario leggere due valori nella sequenza delle profondità, aggiornare il puntatore del rappresentante di uno dei due insiemi, e aggiornare al massimo una profondità. Per valutare il costo della Find, è necessario determinare un limite superiore della profondità degli alberi. Per fare questo, dimostriamo che la profondità degli alberi costruiti usando la strategia Union by Rank `e logaritmica nel numero degli elementi.

La complessità a questo punto è cambiata. Union costa quattro operazioni O(1) poiché costante, essendo una modifica su due array(2 operazioni in lettura e 2 operazioni in scrittura nell'array delle profondità e dei rappresentanti).
Union By Rank ha ordine del O(log n) senza considerare il ciclo while che scorre tutti gli archi perché:

- O(log n) → *find(u) ≠ find(v)* //costo operazione Find
- O(1) → Union

Infine avremo O(m log n) poichè si devono considerare tutti gli archi(ciclo while).
Quello che ci dimostra è:
Se si utilizza la strategia Union by Rank, la profondità rank(A) dell'insieme A,è legata al numero di elementi |A| dell'insieme dalla relazione**$|A| \geq 2^{rank(A)}$**
Qualunque insieme A ha cardinalità al massimo n quindi **$|A| \leq n$** → *cardinalità al massimo di n*\
Se abbiamo n elementi la cardinalità non può essere maggiore di n
Rank(A) $\leq \log_2 |A| \leq \log_2 n$
Cerchiamo di dimostrare quindi che la profondità di un albero costruito con la union by rank è al massimo logaritmica nel numero di elementi

Dimostrazione per induzione strutturale

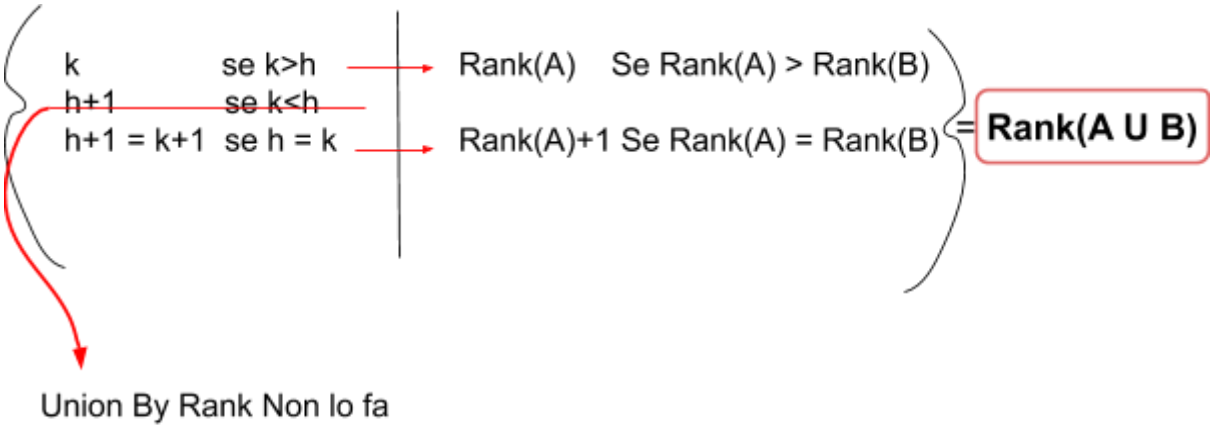
Si fa induzione sul numero di Union.

Base:
k = 0 → Insieme costruito con 0 Union
|A| = 1 Rank(A) = 0
$|A| = 2^{Rank(A)} = 2^0 = 1$
Quindi $|A| \geq 2^{rank(A)}$

Ipotesi Induttiva:
Se k = h
Se **numero di Union ≤ h** → **$|A| \geq 2^{rank(A)}$**

Tesi Induttiva:
|A U B| costruiti con h Union
$|A \cup B| \geq 2^{rank(A \cup B)}$
Se per A è vero e per B è vero, verifichiamo per A U B
|A U B| = |A| + |B|
Usiamo l'ipotesi induttiva su A e poi su B
$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)}$

Sappiamo che Union By Rank fa:



1° Caso → Rank(A) Se Rank(A) > Rank(B)

$2^{Rank(A)}+2^{Rank(B)} \geq 2^{Rank(A)} = 2^{Rank(A\ U\ B)}$

2° Caso → Rank(A)+1 Se Rank(A) = Rank(B)

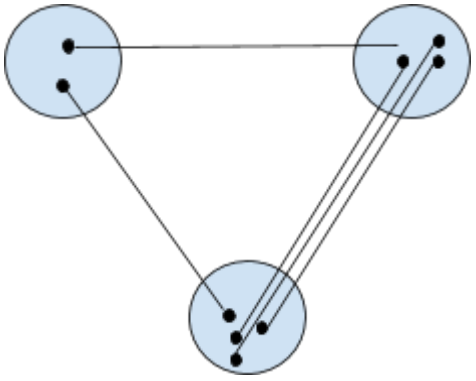
$2^{Rank(A)}+2^{Rank(B)} = 2^{Rank(A)}+2^{Rank(A)} = 2^{[Rank(A)+1]} = 2^{Rank(A\ U\ B)}$

Questo perchè Rank(A) = Rank(B) quindi possiamo sostituirlo

Abbiamo quindi dimostrato che $|A| \geq 2^{Rank(A)}$

Applicazione di Kruskal: Clustering

Si può utilizzare il minimo albero ricoprente per il problema del clustering. Data una relazione di similarità tra gli oggetti, è utile per raggruppare questi oggetti in cluster. I cluster contengono oggetti che sono in un certo senso più' simili tra loro.
Abbiamo una società che vende oggetti e associa gli utenti in base agli interessi, più un utente ha interessi diversi a un'altro e più l'arco ha peso maggiore.
Quindi due persone in gruppi diversi hanno interessi molto diversi fra loro.



Cerchiamo il minimo peso tale che min(peso(u,v))=S che è lo **SPAZIAMENTO** tale che $u \in C_1$, $v \in c \neq c_1$

Consideriamo quindi il problema: *dati n oggetti e le distanze tra ogni coppia, determinare il raggruppamento (clustering) degli oggetti in h insiemi (cluster) che abbia spaziamento massimo*.

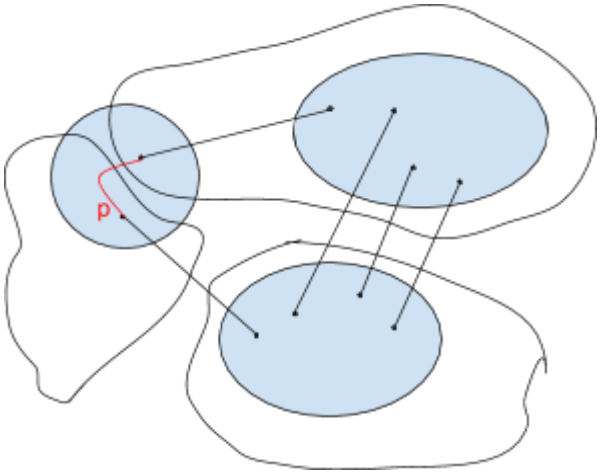
Definizione di spaziamento: lo spaziamento di un clustering è la minima distanza tra oggetti che non appartengono allo stesso cluster. Gli oggetti e le loro distanze possono essere rappresentati con un grafo non orientato, pesato, completo (vi ricordo che completo significa che per ogni coppia di nodi u,v, esiste un arco (u,v) nel grafo).

Algoritmo: si usa l'algoritmo di Kruskal, ma l'algoritmo termina non appena gli insiemi ottenuti (ovvero le componenti connesse) sono h.
Spaziamento: Poiché lo spaziamento è la minima distanza tra due oggetti che sono in due cluster diversi, lo spaziamento del clustering greedy ottenuto utilizzando l'algoritmo di Kruskal è dato dal peso del primo arco che Kruskal avrebbe selezionato dopo l'interruzione; infatti si tratta dell'arco di peso minimo tra quelli che uniscono due nodi in due cluster diversi (che è proprio la definizione di spaziamento).

Teorema di correttezza: l'algoritmo descritto calcola un clustering con spaziamento massimo.

Dimostrazione:

Consideriamo il clustering K prodotto dall'algoritmo greedy e un qualunque altro clustering A. Se i due clustering sono distinti, esiste un arco p che è nello stesso cluster di K ma in due cluster diversi di A.



Sia p l'arco che appartiene allo stesso cluster nella soluzione greedy ma in 2 cluster differenti nella soluzione non greedy.
Se consideriamo la soluzione non greedy p è un arco che collega 2 cluster differenti e tale arco è maggiore o uguale allo Spaziamento → $p \geq S(n)$
Dove S(n) è lo spaziamento della soluzione non greedy, il quale non necessariamente corrisponderà all'arco p, potrebbe esserci un'altro arco minore di p il quale appunto assumerebbe la funzione di Spaziamento tra i due cluster.
Quindi $p \geq S(n)$.
Per quanto riguarda la soluzione greedy invece noi sappiamo che l'arco p fa parte di un unico cluster, quindi vuol dire che Kruskal aveva già scelto quell'arco per la costruzione di quel cluster.

Sappiamo inoltre che Kruskal prendere come Spaziamento tra due cluster l'arco minore dopo gli archi già presi dall'algoritmo.
Quindi siamo sicuri che l'arco preso da Kruskal come spaziamento sarà >= p in quanto p era già stato preso dall'algoritmo per costruire il cluster.
Quindi sapendo che S(G) ≥ p (cioè che lo spaziamento scelto da Kruskal sarà sicuramente maggiore o uguale all'arco p) e sapendo che p ≥ S(n) come detto prima allora:
S(G) ≥ p ≥ S(n)
Quindi lo spaziamento della soluzione Greedy non è peggiore di qualsiasi altra soluzione non greedy.

Zaino Frazionario (problema NP completo)

Il problema dello Zaino Frazionario si presenta così: C'è un ladro con uno zaino di capacità C e vuole riempire lo zaino in modo che al suo interno ci sia il massimo valore possibile
Abbiamo *n* materiali: 1,.....,n che hanno un certo valore e un volume:
val_{i1},val_2,....,val_n
vol_1,vol_2,....,vol_n
Dato che lo zaino è "frazionario", di un materiale può prenderne anche solo una frazione senza dover prendere la quantità totale di quel materiale. Abbiamo quindi che per ogni materiale si potrà prendere una *dose* compresa fra 0 e 1.
$S= (d_1,....,d_n)_i$
Possiamo dire con certezza che il volume di quello che ha rubato non può superare la capacità dello zaino perciò:

$$\sum_{i=1}^{n} d_i * vol_i <= C$$

Avremmo che la somma degli oggetti rubati determinerà il valore della nostra soluzione:

$$\sum_{i=1}^{n} d_i * val_{Ti} = val(S)$$

Si vuole massimizzare quindi il val(S) per poter rubare cose del valore più alto possibile.
IDEA: Si cerca quello con rapporto migliore valore-volume:
$val_{Si}= \frac{valTi}{volTi}$
val_{Si} rappresenta quindi il valore specifico e sappiamo che $val_{Sij} > val_{Sij+1}$

Pseudo-codice Zaino Frazionario (Algortimo Greedy)

Si hanno $i_1,i_2,...,i_n$ oggetti in ordine decrescente di valore specifico → $valS_{ij} > valS_{ij+1}$

∀i dose[i] = 0; // Inizialmente di ogni dose non ha ancora rubato nulla

```
spazio = C; // Inizialmente lo zaino sarà vuoto
j = 1;
while( j ≤ n  and spazio > 0 )
{
        if (vol_ij ≤ spazio) // Se c'è spazio per l'elemento
        {
            dose[ij] = 1;
            spazio = spazio - vol_ij;
            j++;
        }
        else   //Se non ci sta l'elemento
        {
            dose[ij] = spazio/vol_ij
            spazio = 0;  // Spazio = 0, esce dal while
        }
}
```

E' un algoritmo greedy.

Dimostrazione dell'algoritmo:

Supponiamo che gli oggetti 1,.....n sono già in ordine decrescente.
$S=(d_1,\ldots\ldots,d_n)$
S_G=(1,1,1,frazione,0,0,0,0)
0<=frazione<=1
Possiamo poi avere una soluzione S' non greedy del tipo:

S'=(1,1,f',0,0,0,0,0)
0<=f'<=1

In questo caso sicuramente il valore della soluzione S'<S_G in quanto prende meno oggetti di quella greedy.

S''=(1,1,1,1,1,1,f'',0,0,0)
Questa soluzione S'' non greedy non è ammissibile perchè le somme dei volumi supererebbero la capienza C, in quanto già nella soluzione greedy veniva saturato lo spazio.

Ci restano quindi soluzioni di tipologia S''':
S'''=(.....,f',.....g,.....)

f'<1 e g>0

Utilizziamo l'Algoritmo dello scambio:
Possiamo togliere del materiale meno pregiato e sostituirlo con del materiale più pregiato a parità di volume per ottenere una soluzione migliore.
Il valore di quello che tolgo è < di quello che aggiungo.
Quindi qualsiasi soluzione non greedy è migliorabile.

La soluzione Greedy è unica nel caso in cui usiamo valori > stretti uno all'altro.
Costo algoritmo
O(n*log(n)) Che è il costo dell'ordinamento perchè il costo del resto del codice è lineare.

Zaino Non Frazionario

Si hanno n oggetti da 1,...,n in questo caso però non sono frazionari, quindi non posso spezzare gli oggetti in frazioni

$S \subseteq \{1,2,....,n\}$ $\sum_{i \in S} vol_i \leq C$ $val(S) = \sum_{i \in S} val\ i$

L'obiettivo è quello di massimizzare il valore di val(S)

Esempio.
n = 3 C = 10

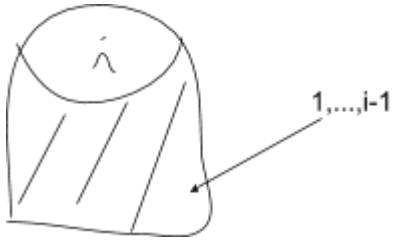
	V.S.	
$vol_1 = 6$ $val_1 = 9$	*9/6*	S{1} val(S) = 9
$vol_2 = 5$ $val_2 = 5$	*1*	S'{2,3} val(S') = 10
$vol_3 = 5$ $val_3 = 5$	*1*	

La soluzione Greedy non va bene, non basta più. Questo problema si risolve usando la ***tecnica di programmazione dinamica****.*

Tecnica di programmazione dinamica

Teorema della sottostruttura ottima:
S(1,1,1,1,1,f,0,0)
Ipotesi:
Supponiamo di avere una soluzione ottimale S per il problema $P_{c,i}$ (problema con C di capacità e i oggetti)

Partiamo da una soluzione ottimale: questa è una soluzione ottimale per $P_{c,i}$ vuol dire che ho gli elementi 1,2,...,i che possono essere dentro a questa soluzione.
In questa soluzione del nostro zaino abbiamo 2 possibilità che "i" appartenga a S oppure non appartenga ad S.
Esaminiamo quindi i 2 casi (non guardiamo l'ordine degli oggetti):
Il primo è quello dove $i \in S$:



Tesi:
Se S contiene i, $S' = S \setminus \{i\}$ è una soluzione ottimale per $P_{C-voli,i-1}$.

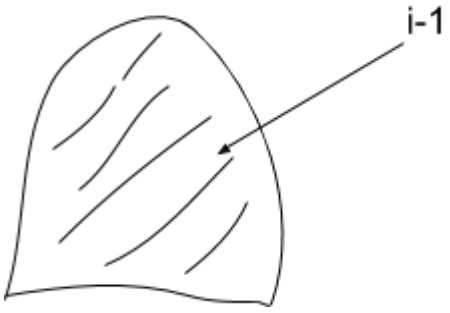
Se noi togliamo a S “i” otteniamo una soluzione S’ che è una soluzione contenente solo elementi che vanno da 1 a i-1 poi sappiamo che la somma in S dei volumi degli oggetti i è <= C.
S \ { i } = S’ sol ammissibile per $P_{C\text{-}voli, i-1}$
Noi sappiamo che:

$$\sum_{j \in S'} vol_j = \sum_{j \in S} vol_j - vol_i <= C - vol_i$$

Ora non voglio solo sostenere che S’ è ammissibile ma anche dimostrare che è ottima per questo problema.
Come lo dimostriamo? **per assurdo**
Supponiamo che S’ per assurdo non sia ottimale per $P_{C\text{-}voli, i-1}$. Ciò significa che per riempire lo zaino avrei trovato un valore diverso da S’ allora suppongo che S’’ sia migliore di S’ per questo problema dove S’ è la soluzione derivata da S.
Quindi val(S’’)>val(S’)
Adesso prendiamo S’’ e ci aggiungiamo di nuovo **i** . Questo determina che:
val(S’’U{i})>val(S) per $P_{C,i}$ //per il teorema della sottostruttura ottima

Ma è assurdo perchè abbiamo supposto che S sia ottimale. Questo significa che S’ deve essere ottimale(per il sottoproblema).

i ∉ S:



Tesi:
Se S non contiene i, S è una soluzione ammissibile e ottimale per $P_{c,i-1}$;
La seconda parte della tesi è che S è anche ottima per il problema Pc, i-1 in questo secondo caso.
Di qua il discorso è analogo, cosa succede? di qua uno dice:
Il ladro si trova gli i oggetti: stiamo supponendo che la soluzione prenda alcuni tra gli oggetti 1...i-1 ora però il ladro questa decisione l’ha presa potendo prendere anche i decidendo però di non prenderla in considerazione:
La situazione è che prima c’era anche l’oggetto i che poteva influenzare la scelta ma ora non c’è più quindi se abbiamo solo gli oggetti 1...i-1 si può fare una scelta migliore per riempire lo zaino della scelta S? se esistesse, per assurdo S* tale che val(S*)>val(S) allora questa scelta S* si poteva fare quando i esisteva perché semplicemente non lo prendevamo ma prendevamo S* .
Questo significa che S* è migliore anche per il problema con tutti gli oggetti $P_{c,i}$ ma questo è assurdo.

Della prof:

Se S non contiene i , allora abbiamo visto che S è una soluzione ammissibile anche per Pi−1; `e ottimale per Pi−1, altrimenti esisterebbe S migliore di S per Pi−1, ma S* sarebbe una soluzione migliore di S anche per P_i : assurdo!*

Ripartiamo dalla base

Se abbiamo 0 oggetti, qualunque sia la capacità del camion il valore sarà 0, dato che non abbiamo preso oggetti.

Con 0 oggetti:
for cap = 0 to C val(cap, 0) = 0;

Con n oggetti:
for i = 1 to n
  for cap = 0 to C
    if (vol_i > cap) val(cap,i) = val(cap,i-1);
    else val(cap,i) = max(val(cap - vol_i,i-1)+val_i , val(cap,i-1))

Esempio.

n = 3 C = 10

vol_1 = 6 val_1 = 9
vol_2 = 5 val_2 = 5
vol_3 = 5 val_3 = 5
vol_4 = 7 val_4 = 15

	CAPACITÀ →										
N. OGGETTI ↓		*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**0**	**0**	**9**	**9**	**9**	**9**	**9**
2	**0**	**0**	**0**	**0**	**0**	**5**	**9**	**9**	**9**	**9**	**9**
3	**0**	**0**	**0**	**0**	**0**	**5**	**9**	**9**	**9**	**9**	**10**

4	0	0	0	0	0	5	9	15	15	15	15

Complessità → O (n*C)

Si denota che la complessità dipende da un valore C che viene dato in input.
Questo algoritmo quanto costa? n * C
E' lineare sul numero di celle della tabella.
Mentre n è il numero di oggetti, qui abbiamo che l'algoritmo dipende da un valore in input.
Non è una cosa positiva. Qui la complessità è usata più in funzione di parametri specifici del problema. Per esprimere un grafo la lunghezza dell'input(che dobbiamo scrivere) è lineare al numero di nodi negli archi.
Se scriviamo la capacità C, quanto spazio occupiamo? E' log C
C lo scriviamo in binario quindi occupiamo spazio logaritmo di C. E' esponenziale nella lunghezza dell'input.
Se C è lineare o polinomiale nel numero degli oggetti siamo apposto è lineare ma se C cresce a livello esponenziale, quindi cresce velocemente, allora la complessità crescerà molto velocemente. //Se la complessità la esprimiamo in termini di lunghezza degli input e la lunghezza di quel pezzetto di //input è log C ma la complessità va con C, va come $2^{\log C}$.
Questa quindi è una complessità esponenziale.
Potrebbe andare peggio, noi abbiamo usato questo algoritmo ma abbiamo sempre una soluzione banale. Ovvero prendere tutte le soluzioni possibili e scegliere la migliore
Esempio ricerca esaustiva: MST troviamo tutti gli alberi ricoprenti e prendiamo il minimo. E' una quantità esponenziale. In questo caso avremmo potuto trovare tutte le combinazioni degli oggetti e trovare la più vantaggiosa ma sarebbe andato a livello esponenziale del numero degli oggetti(Sarebbe andato peggio).
Quindi il fattore C non è bello, questo tipo di complessità dell'algoritmo si chiama Pseudo-Polinomiale perchè non è polinomiale ma se C non cresce molto, quella complessità va bene. Ma in generale non va bene. Sarebbe stato peggio fare tutte le combinazioni poiché sarebbe stato esponenziale sul numero degli oggetti(Quindi peggio)
Questo è il primo problema che incontriamo che è molto difficile, in realtà è definito un problema NP-Completo).

Problemi di decisione

Introduzione ai problemi difficili, ma prima introducendo i problemi facili. Si parla di problemi di decisione che sono problemi che ammettono risposta sì o no. Invece i problemi trattati precedentemente sono problemi di ottimizzazione che ammettono come risposta la migliore soluzione. Sono correlati per esempio guardando il problema dello zaino, quale può essere un problema correlato? Può essere un problema in cui tutti i parametri sono gli stessi di uno di ottimizzazione ma abbiamo anche uno zaino decisione, n nodi, volumi, valori e C. In più si aggiunge un valore V.
Esiste una soluzione tale che il valore della soluzione è >= a V?(V è un input in più che viene dato)
Questo significa trasformare il problema di ottimizzazione in uno di decisione.
Se sappiamo risolvere il problema di ottimizzazione quello di decisione viene fuori gratuito poiché basta fare un confronto. Viene dato un input in più, invece di cercare una soluzione migliore mi chiedo se esiste una soluzione >= di questo valore V? Il valore diventa di decisione.

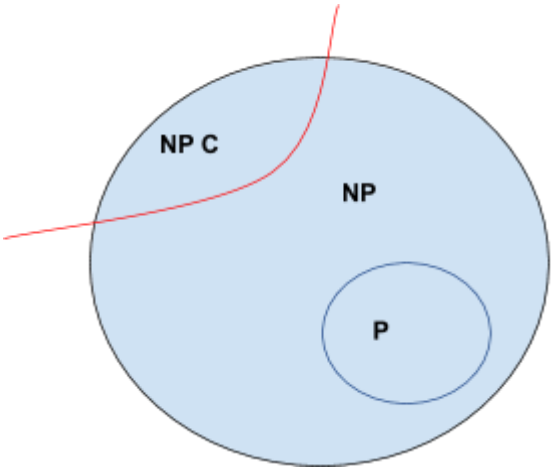
Se uno sa risolvere quello di ottimizzazione sa risolvere quello di decisione. Il problema di decisione, quindi, non può essere più difficile di quello di ottimizzazione. Se ho un algo polinomiale per un problema di ottimizzazione, ne ho uno polinomiale anche per quello di decisione collegato → basta fare un confronto sul risultato.
La classe P di problemi risolvibili in tempo polinomiale → Sono problemi di decisione
La classe NP è ancora una classe di problemi di decisione sono problemi che hanno una dimostrazione facile. Basta guardare lo zaino di decisione.

Decidere se esiste S tale che valore di S è >= a V è una cosa difficile tramite l'ottimizzazione. Però se qualcuno mi da una soluzione S. Data un'istanza x del problema [n valori , vol ecc] qualcuno mi da una soluzione Sx che è una soluzione per cui S è una soluzione >= V. Quanto ci vuole per controllare che Sx è una soluzione buona per questo problema?. E' polinomiale (bisogna controllare che ci siano dentro tutti gli oggetti , bisogna fare somma dei valori degli oggetti contenuti, i volumi degli oggetti contenuti ecc..)
Allora vedete che cercare la soluzione è difficile ma se qualcuno mi da la soluzione posso verificare che è una soluzione buona velocemente(cioè in tempo polinomiale). Se x è un'istanza per cui esiste S, Sx può essere un Sx per cui questo è vero, altrimenti Sx sarà un valore insignificante qualunque che non può verificare questa condizione.

Se prendiamo il MST qual è il problema di decisione correlato?
Esiste un MST del grafo che ha peso complessivo minore di V?
Esiste MST st tale che peso(st) < V ?

Posso esprimerlo come problema in NP, se ho istanza x che è un certo grafo, mi viene dato un albero A. Posso verificare se questo albero verifica questa condizione e posso farlo in tempo polinomiale(sommo tutti i pesi e verifico che non ci siano cicli)
Quindi il problema decisionale del MST è comunque un problema che si può vedere in NP. Però in realtà sappiamo che esistono algo polinomiali che lo risolvono, quindi se uno mi da grafo e albero per la verifica io posso ignorare l'albero, calcolare MST con Prim o Kruskal e fare la verifica.
Cosa succede? noi sappiamo che questo problema MST di decisione è un problema in P ma si può vedere anche come problema in NP. Qualunque problema in P possiamo vederlo come problema in NP. Se dato un problema P ci vengono dati aggiuntivi per la verifica, possiamo ignorare i dati aggiuntivi e fare la verifica con un algo che noi conosciamo. Quindi sto trasformando un problema di ottimizzazione in uno di decisione. Se il problema di partenza è polinomiale è banalmente vero anche che quel problema è in NP. In ogni caso ignoriamo il certificato di appartenenza, risolviamo il problema. Alla fine abbiamo che dato un input e un certificato di appartenenza riusciamo in tempo polinomiale a vedere che quella è un istanza SI o NO del problema. Solo che se il problema potrebbe essere risolto in tempo polinomiale però noi non conosciamo un algoritmo polinomiale come per lo zaino ci serve una informazione aggiuntiva. Questo ci dice che P è contenuto in NP.
$P \subseteq NP$
Ho classificato i problemi però non ho detto come li risolvo, ma dico che se ho la soluzione riesco a verificarli in modo efficiente. Chi mi da la soluzione?
I problemi in NP sono fatti in questo modo(con testimone di appartenenza o certificato) uno li può sempre risolvere tutti con una ricerca esaustiva di tutti i possibili testimoni. Prendo tutte le soluzioni dello zaino e cerco finchè non trovo una soluzione. Se uno sbatte il naso su un problema molto difficile deve riconoscerlo. Se uno trova un algoritmo polinomiale per lo zaino, molto probabilmente il suo algoritmo è sbagliato. E' inutile sbatterci la testa per un algoritmo effettivamente polinomiale perchè è molto difficile che ci riesca. E' difficile o impossibile? Non si sa. Quindi :
P =? NP //P è diverso da NP?

Se fossero uguali, vorrebbe dire che per ogni problema in NP esiste un algoritmo polinomiale
Se sono diversi, vuol dire che ci sono problemi che sono molto più difficili e non risolvibili con algoritmi polinomiali.
NP- Completo → problemi più difficili della classe NP. Sono problemi interessanti perchè se riuscissimo a risolvere uno di questi problemi in tempo polinomiale, riusciremo a risolvere tutti i problemi in NP.
Sono problemi che catturano la difficoltà della classe NP.



APPUNTI PRESI DA INFORMATICI

Problemi difficili

Si considera la complessità rispetto alla lunghezza dell'input. Normalmente, il rapporto tra lunghezza dell'input e numero di nodi (e viceversa) è sempre polinomiale, qualunque sia la rappresentazione del grafo.

Ci sono due classi di problemi:
- La classe **P di problemi di decisione** (di decisione perché ammettono risposte SI o NO) che sono risolvibili attraverso algoritmi polinomiali; (questi sono problemi "facili")

Perché "polinomiali"? Perché i polinomi permettono un costo ancora "ragionevole" rispetto a costi computazionali quadratici o cubici; perché i polinomi sono ancora gestibili, trattabili rispetto a quelli più grossi, come quelli esponenziali.

Esempio: il minimo cammino tra u e v. Questo è un problema di ottimizzazione. Il problema di decisione associato è: esiste un cammino da u e v lungo al massimo k? La risposta è SI.
Che relazione c'è tra questi problemi?
Se noi sappiamo risolvere il problema di ottimizzazione, sappiamo di conseguenza risolvere quello di decisione.
Se ho un algoritmo polinomiale per il problema di ottimizzazione, ho anche un algoritmo polinomiale per quello di decisione.

- La classe **NP di problemi di decisione** che ammettono una dimostrazione verificabile in tempo polinomiale, cioè: data una soluzione, si verifica, in tempo polinomiale, se quella soluzione è valida o no.

Prendiamo due problemi: Hamiltonian Cycle (HC): dato un grafo, un HC è un ciclo che attraversa

ciascun nodo una e una volta sola; Eulerian Cycle (EC): dato un grafo, un EC è un ciclo che passa per
ciascun arco una e una volta sola.

Caratterizzano i problemi più difficili all'interno della classe NP

– Se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale, e P = NP
– Quindi: o tutti i problemi NP-completi sono risolvibili deterministicamente in tempo polinomiale o nessuno di essi lo è

Relazione tra P e NP

P è tutta contenuta in NP, cioè i problemi risolvibili attraverso algoritmi polinomiali hanno anche dimostrazioni verificabili in tempo polinomiali. Non sappiamo se:

- P è uguale a NP, cioè il costo computazionale è uguale per entrambi?
- Oppure, P è diverso da NP?

Come si cerca una risposta ai problemi della classe NP? Tutto quello che sappiamo è che NP contiene P.
Possiamo trovare alcuni tra i problemi più difficili, quindi vederne il costo, e relazionarli.
Come si fa a relazionarli? Attraverso la riduzione.

Riduzione

Ho un algoritmo per il problema B. Applico la riduzione: mi prende un'istanza del problema A e la traduce in un'istanza del problema B.
Esempio: supponiamo che il problema A sia sapere se un numero X è pari e il problema B è sapere se
il grafo G ha un ciclo Hamiltoniano. Se alla riduzione do in input un numero pari X, la riduzione restituisce un ciclo Hamiltoniano, quindi il problema B verifica che sia effettivamente un ciclo Hamiltoniano, quindi restituisce SI. Se alla riduzione do in input un numero dispari, non restituisce un ciclo Hamiltoniano, quindi il problema B verifica che sia un ciclo Hamiltoniano (ma non lo è), quindi restituisce NO.
Tutte le volte che posso sfruttare un secondo problema per risolvere il primo, allora il primo problema è non più difficile del secondo, quindi il primo problema collassa sul secondo.

Versione 3.pd per capire problemi difficili
By ferra che non ho comunque capito

Problemi di decisione

I problemi trattati in precedenza sono *problemi di ottimizzazione* perché ammettono come risposta la miglior soluzione possibile.

Un problema di decisione è semplicemente un problema che ammette si o no come risposta non centra nulla con una S del problema di ottimizzazione.
Si puo solo dire che se abbiamo un algoritmo in tempo polinomiale per il problema di ott. allora il problema di decisione è risolvibile almeno in tempo polinomiale in quanto non è piu difficile di un problema di ottimizzazione per lo stesso problema

Nel caso in cui chiedo *se esiste una S!= da quella fornita dal problema di ottimizzazione*, sto trasformando il problema in un *problema di decisione*

- P: problemi di decisione (perché ammettono risposte si/no) che sono risolvibili attraverso algoritmi *polinomiali* (problemi " facili")

  Perché *polinomiali*?

  Perché i polinomi permettono ancora un costo ragionevole rispetto ai costi computazionali quadratici o cubici; i polinomi possono ancora essere gestibili rispetto a quelli più grossi, come gli esponenziali.

Quindi se sappiamo risolvere un
problema di ottimizzazione posso risolvere anche → *problema di decisione*
perchè devo solamente fare un confronto sul risultato.

- NP: problemi di decisione che ammettono una dimostrazione verificabile in tempo polinomiale cioè, data una S, si verifica in tempo polinomiale se quella soluzione è valida o NO

  Se esiste un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP-completi potrebbero essere risolti in tempo polinomiale, *P = NP*

  (Quindi o tutti i problemi in NP-completi sono risolvibili deterministicamente o *nessuno* di essi lo è.)

Decidere se esiste S tale che valore di S è >= a V è una cosa difficile tramite l'ottimizzazione.
Ma nel caso in cui mi venga fornita un'istanza x del problema, e una Sx per cui S>=V, per controllare quanto sia buona Sx è necessario un tempo polinomiale, poiché è solamente necessario verificare che sia accettabile.

Esempio:

Se prendiamo il MST qual'è il problema di decisione correlato?
Esiste un MST del grafo che ha preso complessivo minore di V?

Esiste MST st tale che peso(st)<V ?

Posso esprimere questo problema in NP, se ho un'istanza x che è un certo grafo e mi viene fornito un albero A, posso verificare se A (A in questo caso è Sx) verifica la condizione e posso farlo in tempo polinomiale (sommo tutti i pesi e verifico che non ci siano cicli).
Il problema di decisione del MST è un problema in NP (di conseguenza anche in P) ma in realtà sappiamo che avendo un grafo conosciamo degli algoritmi per determinare l'MST e verificare A.
Di conseguenza sto trasformando un problema di ottimizzazione in un problema di decisione. Possiamo ignorare il certificato di appartenenza e risolvere il problema.

(Alla fine abbiamo che dato un input e un certificato di appartenenza riusciamo in tempo polinomiale a vedere che quella è un istanza SI o NO del problema. Solo che se il problema potrebbe essere risolto in tempo polinomiale però noi non conosciamo un algoritmo polinomiale come per lo zaino ci serve una informazione aggiuntiva. Questo ci dice che P è contenuto in NP. $P \subseteq NP$ Ho classificato i problemi però non ho detto come li risolvo, ma dico che se ho la soluzione riesco a verificarli in modo efficiente. Chi mi da la soluzione? non lo so unlucky)

Fine versione ferra Np

LEZIONE 10/05/18

VERTEX COVER: Ricoprimento di vertici.

Dato un grafo G(V,E), R $\subseteq$ V (sottoinsieme dei nodi) è un vertex cover se per ogni arco (u,v)$\in$ E , u $\in$ R oppure v $\in$ R.

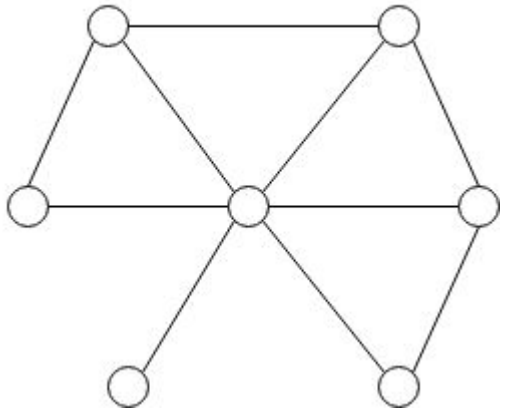
Si vuole far in modo che per ogni sala/corridoio ci sia almeno un custode che la ricopre.
Il museo vuole mettere il minimo numero di custodi possibili.

VERTEX COVER di ottimizzazione

Definire un ricoprimento R del grafo G tale che |R| sia minima.

Disegno:

Grafo VC.



Ogni nodo rappresenta una porta e ogni arco una stanza. In che porte devo piazzare i custodi per vedere più stanze possibili piazzando meno custodi possibili?

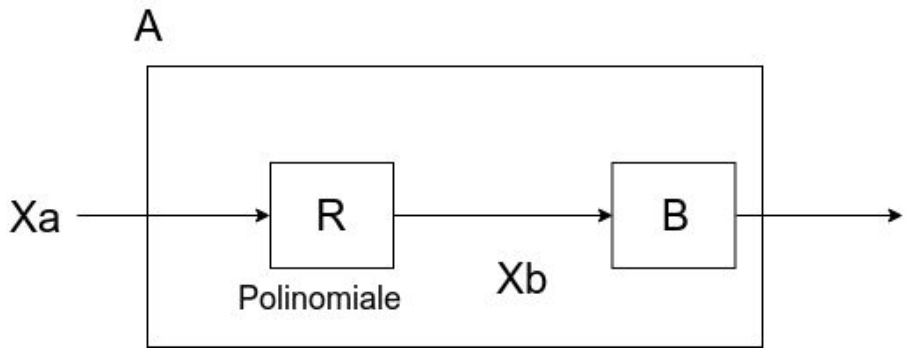
Problema di decisione:
Dato un grafo e un numero qualunque k esiste un'istanza R di G tale che |R| ≤ k??

Ci viene data un'istanza x = (G,K) e testimone dell'istanza(certificato) R_x che verifica che sia in un tempo polinomiale.
È un problema NP perchè è risolvibile in tempo polinomiale se ci viene dato il testimone.
Se conosciamo una soluzione al problema di ottimizzazione, allora la conosciamo anche per il problema di decisione associato.

(G,k) → VC -ott → R ≤ k → si/no

Ci viene data un'istanza x=(G,K) , se conosciamo la soluzione al problema di ottimizzazione si può applicare l'algoritmo di ottimizzazione evitando così il certificato e successivamente verificare che il risultato sia ≤ k, se lo è risponde "si" se no risponde "no".
Nel caso in cui si usasse l'algoritmo per il problema di ottimizzazione allora la complessità del problema di decisione è la stessa di quella di ottimizzazione.
Però si potrebbe applicare un'altro algoritmo per il problema di decisione migliore di quello per il problema di ottimizzazione per questo ne deduciamo che:
Vc-ott ≥ Vc-dec
La complessità del problema di ottimizzazione è ≥ a quella di decisione.

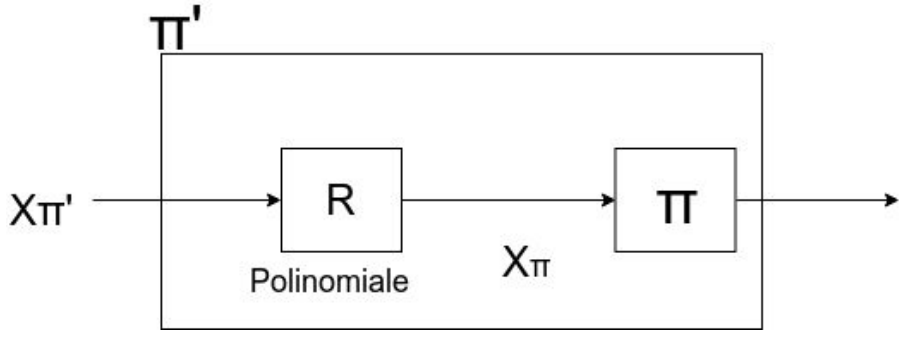
Problema A e B → si comporta allo stesso modo di Vc-ott e Vc-dec
Dati due problemi A e B, se riesco facilmente (cioè in tempo polinomiale) a risolvere il problema in A utilizzando il problema in B allora, il problema in A ha complessità polinomiale minore o uguale a B.
pA ≤ pB



Quindi se miglioriamo l'algoritmo per il problema B lo miglioramo anche per il problema in A.
Questo metodo si chiama ***riduzione polinomiale*****,** ci interessa che sia polinomiale e che conservi le istanze si.

Np-hard

È un problema per cui per qualsiasi altro problema in Np si può fare la riduzione polinomiale.
π è Np-hard se per ogni altro $\pi' \in NP$ $\mathbf{p}\pi' \leq \mathbf{p}\pi$



Quindi π è un limite superiore per indicare i problemi difficili. π ha un limite di difficoltà superiore a tutti i problemi in NP. Quindi, π può essere usato per risolvere altri problemi.
Se π è NP-hard e $\pi \in$ NP allora viene detto NP completo → quindi contiene tutta la difficoltà della classe NP.
Se trovassi una soluzione polinomiale per questo problema NP-completo lo troverei per tutti gli altri.
Esistono problemi NP-completi?
Si → Zaino, 3 - colorabilità, circuito hamiltoniano.
Mentre il problema HC (circuito hamiltoniano) è un problema NP-completo, il problema EC(circuito Euleriano) è polinomiale perchè basta vedere che ogni nodo abbia grado pari.
Come si fa quando si scopre che si sta affrontando un problema difficile?
Bisogna trovare un'altra strada.
Una strada è sempre la ricerca esaustiva.
Si può cercare di migliorare la ricerca esaustiva.
Oppure si può guardare nei sottoproblemi(se si accorge che son più facili).

Un altro metodo è la ricerca locale: Si parte da una soluzione approssimata e cerco di migliorarla il più possibile(non è detto che troveremo la soluzione ottimale).
Un'altro modo per risolvere gli NP-Completi è approssimare la soluzione, quindi se non trovo una soluzione ottimale posso accontentarmi di buona approssimazione. Come faccio a capire quanto è **buona** la nostra approssimazione?
Cerco di calcolare il **Fattore di approssimazione**.

PROBLEMA DI MINIMIZZAZIONE
Data una soluzione S, se stiamo cercando una soluzione minima abbiamo che:

val(S) -> Soluzione ottimale*
val(S) -> Soluzione approssimata

$val(S^*) \leq val(S) \leq \rho \cdot val(S^*) \quad \rho \geq 1$
min
| ρ(n) ro dipende solitamente da n → n è la grandezza del problema |

per capire quanto è cattiva (o buona) la nostra approssimazione possiamo cercare il rapporto fra le due soluzioni che sarà ≤ ρ (l'abbiamo trovato dividendo tutto per *val(S*)*)

$\frac{val(S)}{val(S*)} \leq \rho$ Quindi il rapporto tra le 2 soluzioni è minore o uguale a ro.
Otteniamo quindi un rapporto tra la soluzione ottimale e quella generica.

PROBLEMA DI MASSIMIZZAZIONE

Abbiamo poi che: val(S) ≤ val(S*) poiché è massimale, allora la limito da sotto e ottengo:

$\frac{1}{\rho} \cdot val(S^*) \leq val(S) \leq val(S^*)$

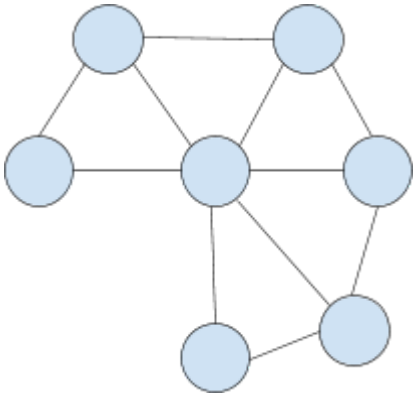
$\frac{val(S*)}{val(S)} \leq \rho$

In generale per tutti i 2 casi cioè sia per il problema di massimizzazione che per il problema di minimizzazione vale che:
$\rho \geq \max\left(\frac{val(S)}{val(S*)}, \frac{val(S*)}{val(S)}\right)$

Infatti se il caso è un problema di massimizzazione allora il secondo parametro avrà un valore ≥1 mentre il primo un valore ≤ 1, quindi prendendo il massimo si prende appunto la soluzione corrispondente al problema di massimizzazione.
Viceversa se il caso è per il problema di minimizzazione.

Vediamo due esempi di approssimazione:

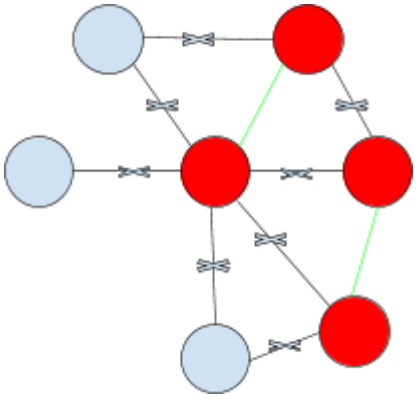
Il primo esempio è sul problema del vertex cover, sapendo che è un problema NP-Completo proviamo ad approssimare



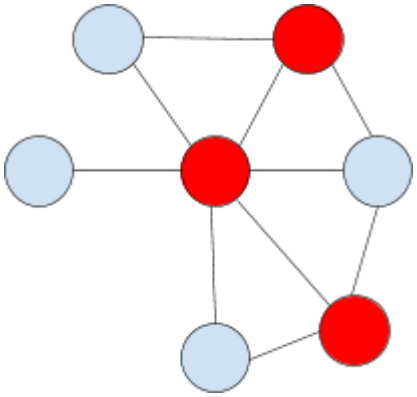
Con ϱ= 2

Abbiamo un algoritmo che elimina tutti gli archi uscenti da ‘u’ e ‘v’:
Algoritmo di approssimazione
Vertex Cover
Si cerca di prendere i nodi in modo tale da coprire tutti i corridoi(archi)

R=Ø
iterativamente sceglie (u,v) ∈ E
R= R U {u} U {v}
A = A U {u,v}
∀ arco (u,w) elimina gli archi dal grafo
∀ arco (v,t) elimina gli archi dal grafo



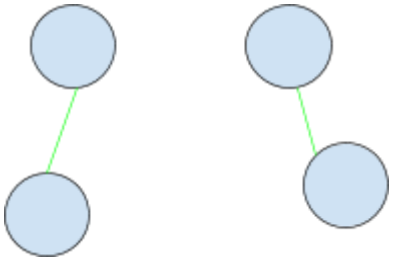
Si poteva però fare di meglio prendendo altri vertici e ottenere una soluzione migliore:



Qual è in generale la bontà di questo algoritmo?
Poiché gli archi vengono eliminati quando hanno un estremo nel ricoprimento, togliendolo sono tranquillo che questo rimanga coperto.

DIMOSTRIAMO CHE SIA AL PIU' ϱ=2 OVVERO che l'algoritmo prende al massimo il doppio del ricoprimento ottimale

Se guardiamo G_a=(R,A) ovvero:



Voglio "stimare il grafo" partendo dal sottografo:
Dato che togliamo sempre archi, avremmo tutte componenti connesse distinte.
$|R_a^*|$ è il ricoprimento del grafo G_a trovato dopo la esecuzione dell'algoritmo.
$|R|$ è il ricoprimento del grafo iniziale calcolato dall'algoritmo.
$|R_G^*|$ è ricoprimento ottimale del grafo iniziale.

$|R_A^*| = ½ |R|$
Diciamo che il ricoprimento ottimale del grafo calcolato dopo l'algoritmo è uguale alla metà dei nodi presenti dopo l'esecuzione dell'algoritmo.(in quanto sono tutte componenti connesse distinte)
$|R_A^*|$ è ottimale per il grafo G_A trovato dopo l'algoritmo.

Torniamo al ricoprimento di tutto il grafo, se vogliamo coprire tutto, sarà sicuramente ≥ R_A
$|R_G^*| \geq |R_A^*| = ½ |R|$ → Possiamo vederlo così: $|R_G^*| \geq ½ |R| \rightarrow 2 |R_G^*| \geq |R|$

perciò sapendo che il ricoprimento del grafo iniziale calcolato dall'algoritmo è ≥ del ricoprimento ottimale iniziale, possiamo dire che, grazie alla relazione di prima otteniamo

$|R_G^*| \leq |R| \leq 2|R_G^*|$
Quindi 2 è il fattore di approssimazione ro, come avevamo già detto inizialmente.

IL PROBLEMA DEL COMMESSO VIAGGIATORE o TSP (travelling salesman problem)

Un commesso deve girare tutte le città e tornare indietro. Il problema di ottimizzazione è minimizzare il cammino che deve percorrere.

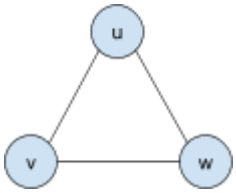
Date n città, ∀ arco u,v c'è una distanza minima d(u,v)
Il grafo è non orientato, completo e pesato con pesi ≥ 0.
Si vuole trovare il tour di peso minimo di G → ovvero si vuole percorrere tutti i nodi con somma dei pesi minima

Tour è il cammino chiuso che passa per tutti i nodi.

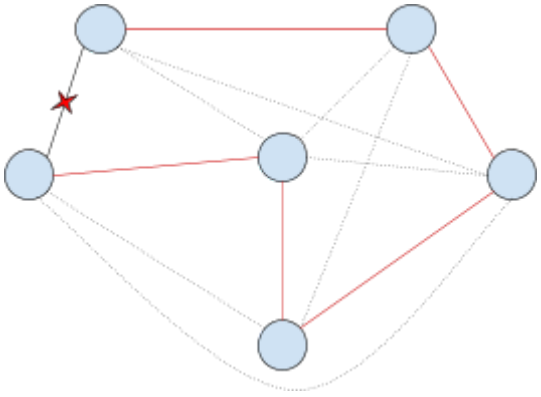
Questo è un problema NP-completo → è un problema difficile

Non ammette algoritmi di approssimazione per nessuna R, quindi si può escogitare un'ipotesi aggiuntiva in modo da riuscire ad ottenere una soluzione vicina a quella ottimale.

Ipotesi aggiuntiva → disuguaglianza triangolare
d(u, v) + d(v, w) ≥ d(u, w)



Supponiamo di avere un grafo del genere:



Supponiamo che questo grafo sia derivato da una soluzione S*

Se tolgo l'arco crocettato dalla "x"(o un arco in generale), diventa un albero ricoprente, quindi non ha cicli.

c(S*) è il costo ottimale che vorremmo raggiungere, ma togliendo l'arco otteniamo una soluzione
T = S\{arco}*
Quindi il costo della soluzione S* è maggiore o uguale al costo della soluzione T poiché gli archi hanno tutti pesi positivi e togliendone uno il peso totale è minore o uguale(perchè potrei togliere un arco di peso 0), quindi:
c(S) ≥ c(T)*
Se calcoliamo l'MST A con prim possiamo dedurre che:
c(S) ≥ c(T) ≥ c(A)* → questo perchè A è il MST quindi c(T) ≥ c(A) per ipotesi.

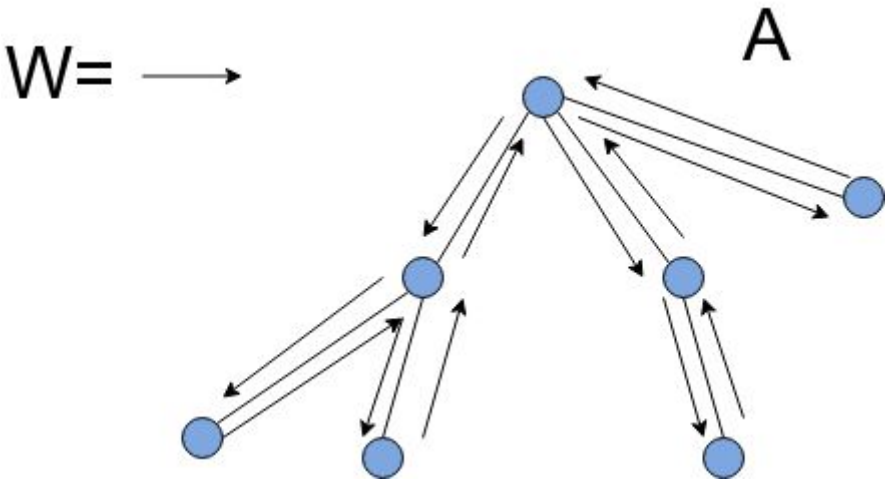
1. **Prim → MST A**

c(W) = 2 c(A)
Questo perché il viaggiatore percorre due volte gli archi dell'albero per fare il tour.
A questo punto sorge un dubbio:
Al commesso viaggiatore conviene tornare su e scendere di nuovo o prendere un arco in modo da permettergli di tagliare il percorso e ottenere una scorciatoia?
(Ricordo che è possibile tagliare e prendere un'altro arco perché il grafo iniziale è completo.)

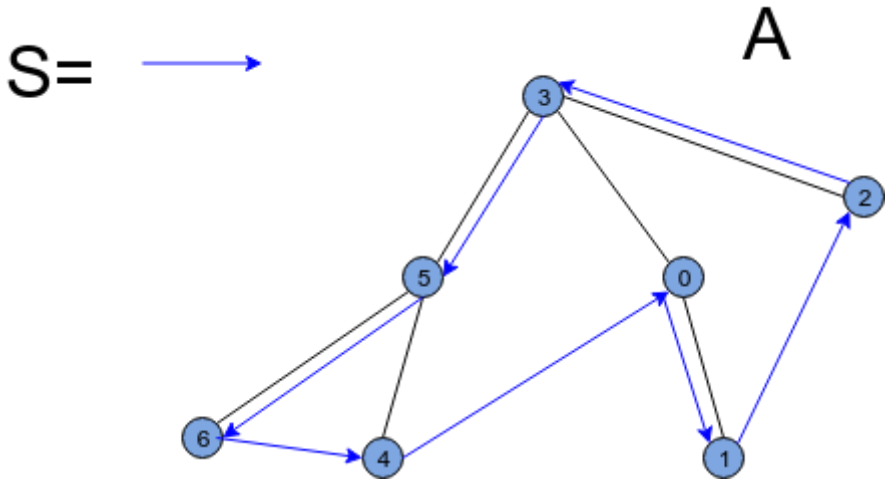
Conviene una scorciatoia!
c(W') ≤ c(W) → secondo la disuguaglianza triangolare

Come facciamo a fargli prendere sempre delle scorciatoie?
Algoritmicamente con una visita DFS.

2. **Visita DFS → per ottenere l'ordine dei nodi scoperti**



Si ottiene, attraverso una visita DFS, questo ordine di visita : 3,5,6,4,0,1,2
Grazie a questo ordine di visita so come tagliare i percorsi:



si ottiene che:
c(S) ≤ c(W) = 2 c(A)

Se combiniamo con questa relazione ⇒ *c(S*) ≥ c(T) ≥ c(A)*
si ottiene una nuova relazione:
c(S) ≤ c(W) = 2 c(A) ***≤ 2 c(S*)***

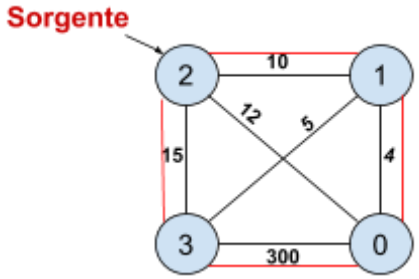
c(S) con le scorciatoie, sicuramente non è peggiore di due volte c(S).*

In definitiva si ottiene che

c(S) ≤ 2 c(S)*
Sappiamo quindi che la soluzione S non è peggiore di due volte la soluzione (ottimale) S*.
In questo caso il fattore di approssimazione è uguale a 2 → *ϱ = 2*

Potrebbe comunque esserci un percorso migliore di quello calcolato in questo modo, perchè l'MST non ha nulla a che fare con i cammini minimi.
Non riesco a dimostrare che questa sia la soluzione ottimale.

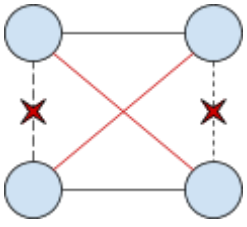
Esempio.
Il problema del commesso viaggiatore dove non vale la disuguaglianza triangolare



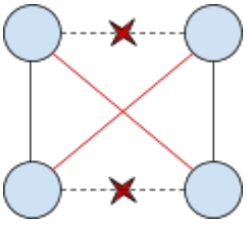
Proviamo a scegliere un percorso che ottenga un ciclo e che spenda meno senza tornare indietro.
Usando una tecnica Greedy(prim modificato):
Si sceglie a seconda della posizione del viaggiatore perchè voglio creare un circuito.
Complessità O(n + m)
La soluzione greedy darebbe un peso di 300 o più → p(300 +)
Una soluzione ottimale darebbe un peso di 36 → $p_g(36)$
Quindi l'idea greedy è disastrosa.

Si può cercare di scambiare una coppia di archi con altra coppia mantenedo il ciclo.
Per esempio si potrebbe togliere arco di peso 15 e 4 con i due interni di peso minore cioè 12 e 5.
Oppure scambiare gli archi di peso 10 e 300 con gli archi interni di peso 12 e 5.

Otterremo quindi due soluzioni differenti:



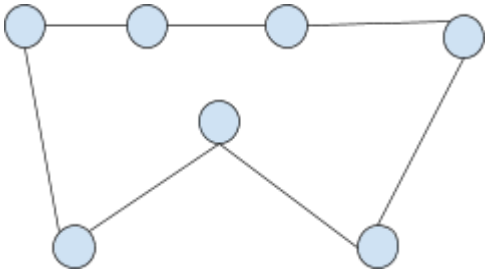
(a)



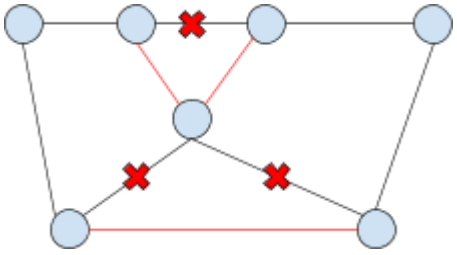
(b)

Sceglie tra tutte le possibilità che ha. Quella ottimale è la soluzione **(b)**.
È evidente che se i nodi sono tanti ci sono tante possibilità tra cui scegliere.

Avendo un grafo G=(V,E) dove il peso degli archi è proporzionale alla lunghezza:



Non è detto che, modificando solo due archi, ci si avvicini alla soluzione ottimale per G. Devo incrementare il numero di archi se voglio avvicinarmi sempre più alla soluzione ottimale,ovvero facendo una RICERCA LOCALE: nel nostro caso, prendo tre archi invece di due:



Notare che, a causa delle maggiori combinazioni possibili, il costo, togliendo tre archi, aumenta.

Per avvicinarci alla soluzione ottimale, dobbiamo fare delle scelte drastiche che possono portare ad un costo eccessivo. Nel nostro caso, aumentando il numero di archi ci si avvicina sempre più alla soluzione ottimale, ma nel contempo i costi aumentano.

ESONERO 22/05/12 FINO A QUI ESCLUSA PROGRAMMAZIONE DINAMICA(zaino non frazionario).

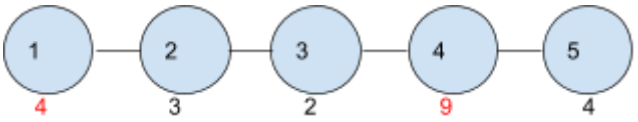
Lezione 17/06/2018

Programmazione dinamica

MASSIMO SOTTOINSIEME INDIPENDENTE (su grafo lineare)

Grafo lineare

Un grafo lineare ha tutti i nodi uno in fila all'altro. Qui non sono gli archi ad avere i pesi ma sono i nodi ad essere pesati.



Il problema è trovare un sottoinsieme di V ($S \subseteq V$) t.c. $(u,v) \in E$, $u \notin S$ oppure $v \notin S$
Ovvero non posso mai prendere 2 nodi adiacenti, la somma dei pesi dei nodi 'i' con 'i' $\in S$ deve essere massimale $\Sigma_{i \in S} p(i)$ sia massimale → le somme dei pesi dei nodi sia massimale

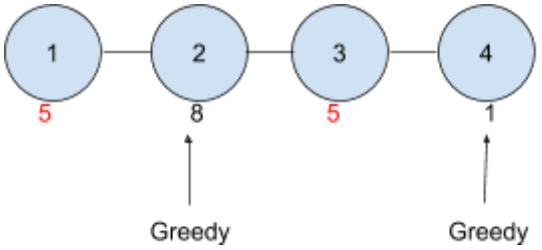
Esempio:
Rossi affitta una sala conferenza ma la può affittare solo un giorno sì e uno no (perché,ad esempio, il giorno dopo dovrà preparare la sala e pulirla etc.).Ogni richiesta per affittare la sala conferenze ha un prezzo (peso del nodo) e bisogna quindi decidere come massimizzare i guadagni prenotandola nei giorni giusti.

ESEMPIO BASATO SULL'INSIEME PRECEDENTE:
Se io prendo i nodi:
S' = 2,4 → 12
S''= 1,3,5 → 10
S''' = 1,4 → 13

Come ci siamo arrivati?
Abbiamo fatto una ricerca esaustiva che è esponenziale e quindi questo non ci piace.

Possiamo utilizzare un algoritmo greedy?

Facciamo finta di partire da quello di peso maggiore: Il nodo 4 che ha peso 9, poi prendo il prossimo maggiore che è 1 ed ha peso 4. In questo caso funziona ma non è sempre così. Guardiamo un controesempio:



Qui l'algoritmo greedy non va bene perchè prende il nodo 2 e poi il nodo 4 si avrebbe come somma 9. In realtà, prendendo 1 e 3 avrei avuto una soluzione migliore! (10>9)

Proviamo con la tecnica divide et impera:
Divido a metà il grafo, calcolo i massimi nei due sottografi e poi li unisco, ma non funziona comunque nell'esempio appena fatto.
Infatti se i nodi di peso maggiore sono situati entrambi vicino al taglio non funziona.
Risolveremo quindi attraverso la PROGRAMMAZIONE DINAMICA

Si cerca un teorema di sottostruttura ottima.

Teorema di sottostruttura ottima:

Prima di tutto bisogna decidere quali saranno i sottoproblemi su cui lavorare.
Per esempio qui possiamo lavorare su:

P_i= problema con 'i' nodi.
S ottimale con 'i' nodi

IDEA: Proviamo a capire se l'i-esimo nodi appartiene alla soluzione i $\in$ S e i $\notin$ S

- ***Soluzione 1 : i*** $\in$ ***S***

se i $\in$ S implica che i-1 non appartiene alla soluzione
S' = S - {i} è ammissibile per P_{i-2} ($\rightarrow$ poichè non prende l'adiacente) ed è indipendente perché è derivata da S.
È massimale per P_{i-2}?
La risposta è "sì" e si dimostra per assurdo:
Ipotizziamo per assurdo che S' non è ottimale per P_{i-2}, se S' non è ottimale allora implica che $\exists$ S'' tale che val(S'') > val(S') per P_{i-2} allora S'' U {i} è una soluzione ammissibile per P_i.
Quindi val(S'' U {i}) > val(S) ed è ASSURDO perché abbiamo supposto S ottimale.

- ***Soluzione 2: i*** $\notin$ ***S***

Se S $\subseteq$ {1,.....,i-1} Siccome è ottimale è un sottoinsieme indipendente.
S allora è ammissibile per P_{i-1}, è ottimale? Si perchè se per assurdo S non ottimale $\Leftrightarrow$ $\exists$ S' ammissibile per P_{i-1} e vale la relazione: val(S')>val(S) Per P_{i-1}
Se S' è ammissibile per il problema P_{i-1} allora è ammissibile anche per il problema P_i il che è assurdo perchè significa che Val(S')>val(S) per P_i.In conclusione S è ottimale anche per P_{i-1}
Distinguiamo tra Enunciato e Dimostrazione:

Enunciato:
Ipotesi:*Supposto S ottimale per* P_i
Se i ∈ S
Tesi:S' = S - {i} ammissibile e ottimale per P_{i-2}
……..

Se i ∉ S
Tesi:S = ammissibile e ottimale per P_{i-1}
……..

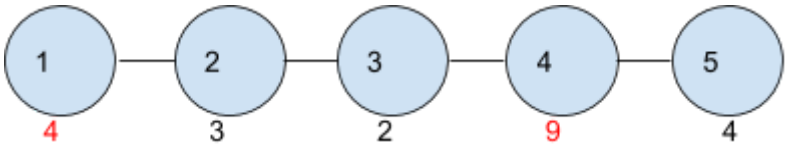
Come lo usiamo per progettare un algoritmo?
Se vogliamo sapere il valore ottimale per il problema i

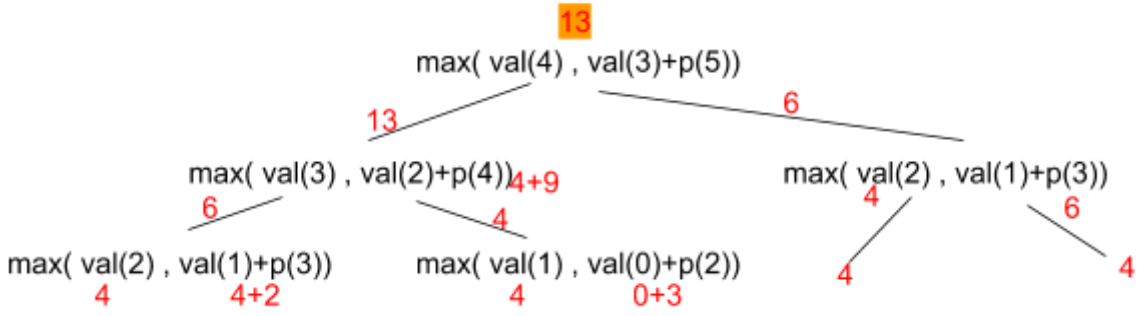
val(i) = max(val(i-2)+p(i), val(i-1))

Dimostrazione:
Se noi sappiamo che val(i-2)+p(i) è ottimale oppure val(i-1) è ottimale allora val(i) è ottimale per P_i perchè se non lo fosse, siccome vale il teorema di sottostruttura ottima si potrebbe scomporre e ottenere 2 soluzioni migliori per i sottoproblemi il che è assurdo.

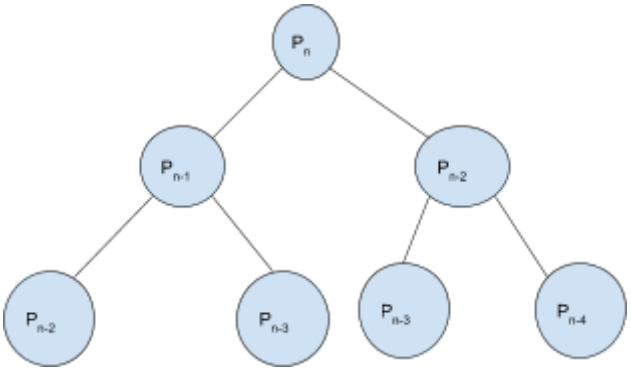
```
val(i) {
        if( i==0) return 0;
        else if(i==1) return peso(1); //oppure peso(i), è indifferente.
        else return max( val(i-2)+p(i), val(i-1) );
}
```

vediamolo funzionare su val(5) = 13;





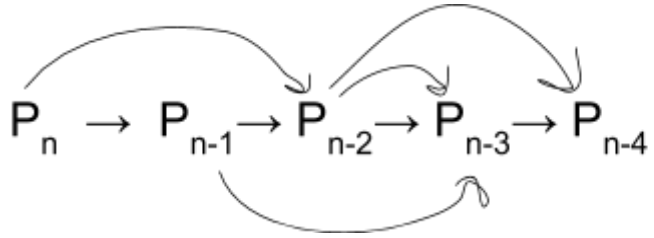
Se l'albero è profondo n ci sono 2^n richiami, quindi avremo una complessità di ordine $O(2^n)$.



Si ha in questo modo un sacco di lavoro ripetuto.
Si usa una tecnica di **Memoization** che consiste nel ricordare ciò che abbiamo fatto.
Quanti sono i sottoproblemi?
n+1 Perchè partiamo da 0 {P_0,P_1,.....P_n}
Pseudocodice con tecnica di Memoization:

```
∀ i A[i]=null;
A[0]=0;
A[1]=P(1);
val(i){
        if(A[i]==null)
        {
                A[i]=max(val(i-1),val(i-2)+p(i));
        }
        return A[i];
}
```

E' lineare in quanto risolviamo un sottoproblema una volta sola e non più volte come avveniva senza la tecnica di **Memoization.**



I problemi vengono risolti una volta sola.

Si può anche sviluppare lo pseudocodice in modo iterativo:

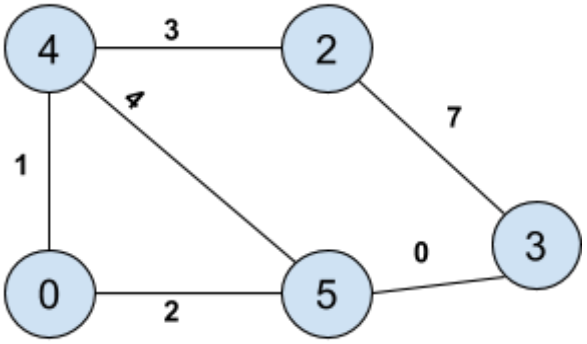
```
A[0]=0;
A[1]=p(1);
for(i=2 to n)
        A[i]=max(A[i-1],A[i-2] + p(i))
```

Come mai questo approccio NON è greedy?
Perchè modifica costantemente i valori.
Mentre sceglie il massimo, la decisione di prendere o non prendere un oggetto la rivede continuamente.

L’approccio Greedy è un caso particolare della programmazione dinamica, nel quale c’è un solo sottoproblema quindi basta migliorare questa soluzione.
Nel caso del massimo sottoinsieme indipendente dobbiamo scegliere tra 2 sottoproblemi, infatti non è Greedy.

Divide et Impera:Altra tecnica di risoluzione dei sottoproblemi.
Il divide et Impera utilizza sottoproblemi disgiunti , e non capita mai che risolvi 2 volte lo stesso problema,a differenza del massimo sottoinsieme indipendente.

Non è vero che qualsiasi problema di ottimizzazione ha una sottostruttura ottima.
Prendiamo il problema del cammino massimo:



Il cammino massimo non ha proprietà di sottostruttura ottima.
Tra il nodo 0 e il nodo 3 il massimo è questo: 0,5,4,2,3 (2+4+3+7)

Se questo cammino fosse minimo, qualunque cammino intermedio sarebbe minimo, in questo caso se noi spezziamo il cammino massimo da 0 a 5, quello non è il cammino massimo!

Tra 0 e 5 non è :0,5
Tra 0 e 5 il massimo è:0,4,2,3,5.
Quindi non ha una proprietà di sottostruttura ottima.

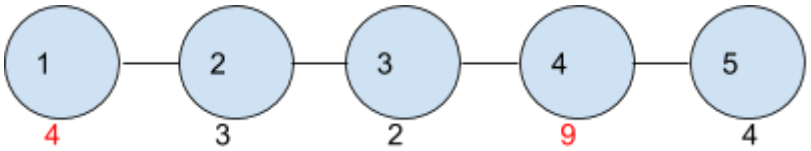
In sostanza la proprietà di sottostruttura ottima dipende dal problema specifico.

Come si ricostruisce la soluzione ottimale?
Partendo da “n” sappiamo che A[i] è stato costruito tra il max di A[i-1] e A[i-2]+p(i).

```
i=n;
S={};

while(i>=1){
        if(A[i]>A[i-1])             // allora il nodo ‘i’ l’avevamo preso
        {
                S=S U{i};
```

```
            i=i-2;          // se c'è l'oggetto 'i' allora 'i-1' non c'è sicuro
        }
        else{
            i=i-1;
        }
}
```

Per costruire la soluzione si opera all'indietro perchè in avanti continuano a cambiare i valori.



s

esecuzione della soluzione:
A:

Peso ottimale	0	4	4	6	13	13
Nodi	0	1	2	3	4	5
Op per calcolo soluzione ott	Caso base val=0;	Caso base val=peso(1)	Max(4,0+3)	Max(4,4+2)	Max(6,4+9)	max(13,6+4)

Ora ricostruendo la soluzione ottimale seguendo lo pseudocodice per la ricostruzione della soluzione S avrà valore :
S={4,1}.

È la restrizione di un problema molto difficile.
Infatti in questo problema viene utilizzato un grafo lineare, se cercassimo di risolvere il problema per un grafo non lineare diventerebbe probabilmente un problema Np completo.

Ricostruzione della soluzione per il problema dello zaino.

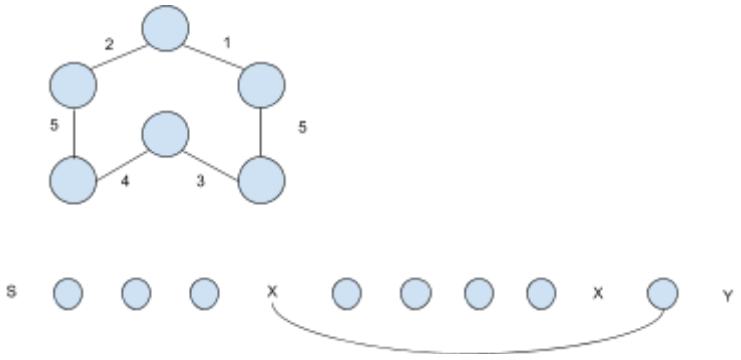
Dato che noi sappiamo che Z(C,i)=max(Z(C,i-1),Z(C-vol(i),i-1)+val(i))
Ricostruiamo la soluzione in questo modo:

```
i=n; //numero oggetti
While(i>0 and C>0){
        if(Z(C,i)>Z(C,i-1)){
                S=S U{i};
                i=i-1;
                C=C-vol_i;
        }
        else{i=i-1;}
}
```

Lezione 24/05/2018

Cammini minimi da una sorgente con archi di peso negativo

Il nostro obiettivo è trovare i cammini minimi da una sorgente con pesi anche negativi.
Se abbiamo un grafo con pesi tutti positivi un cammino minimo può avere cicli??
No. Si potrebbe sempre evitare il ciclo e ottenere un costo minore.



Quindi un cammino minimo sarà sicuramente un cammino semplice.

Ciclo negativo :
Un ciclo negativo è un ciclo la cui somma dei pesi degli archi è negativa.

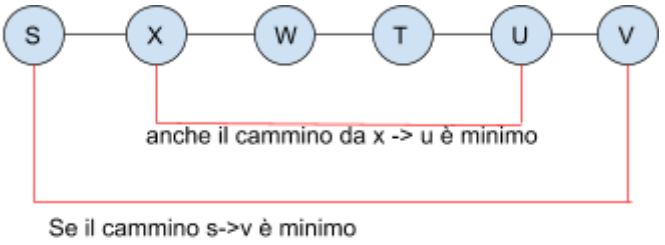
Con pesi negativi:
Se la somma del ciclo è negativa sembrerebbe conveniente prenderlo. **In realtà no!**
Se ci sono cicli negativi non esiste il cammino minimo poiché si può continuare a percorrere il ciclo fino a raggiungere un peso di $-\infty$. Si può rendere più' piccolo infinite volte qualunque cammino che riesce a raggiungere il ciclo.
Quindi il problema di trovare cammini minimi non è ben posto se vi sono cicli negativi.

Quindi non vogliamo che ci siano cicli negativi.
In assenza di cicli negativi, un cammino minimo per essere semplice in un grafo con grado |V| ha al massimo n nodi e quindi n-1 archi.
In definitiva:

- In assenza di cicli negativi il cammino minimo è per forza semplice
- In assenza di cicli negativi il cammino minimo ha al massimo n nodi, quindi n-1 archi

Come progettare l'algoritmo?
Si userà un algoritmo di programmazione dinamica e si cercherà di capire come deve essere fatta la soluzione ottimale, quindi come scomporla in problemi più semplici.

Teorema di sottostruttura ottima: Se prendiamo una qualunque coppia di nodi di un cammino minimo, sicuramente il cammino tra quella coppia di nodi è minimo.



Bellman-Ford

Questo algoritmo è un algoritmo di **programmazione dinamica**, utilizzato nel ROUTING.
Essendo di programmazione dinamica, in quali sottoproblemi posso scomporre il problema?

Il problema è quello di arrivare al nodo 'v' attraversando al massimo 'i' archi.
Il problema $P_{V,i}$ → cammino minimo dalla sorgente a v, attraversando al massimo 'i' archi.
Come possiamo esprimere il problema?
$\forall$ v $\in$ V risolvo $P_{v,n-1}$ quindi un problema con massimo n-1 archi.

Ora guardiamo come sono fatte le soluzioni ottimali dei problemi.
Ipotesi:
Supponiamo di avere S come soluzione ottimale per il problema $P_{v,i}$.
Sappiamo che questa soluzione è un cammino con massimo "i" archi.

In quali sottoproblemi riduciamo i problemi?
L'idea è di diminuire il numero degli archi passando ai sottoproblemi successivi. Quando arriveremo ad avere 0 archi il problema è risolto (quindi la base sarà 0 archi).

Quindi, abbiamo 2 tipi di casi possibili:

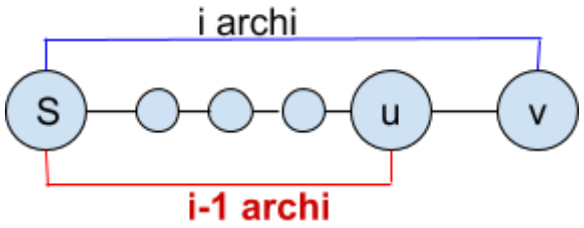
- "i" esimo arco ∉ alla soluzione (S ha <= i-1 archi) *//i-1 poichè il cammino ottimale potrebbe essere un passo solo*
- "i" esimo arco ∈ alla soluzione (S ha i archi).

DIMOSTRAZIONE:

- **1° caso: i∉S**
  S è ammissibile e ottimale per il problema $P_{v,i-1}$:
  Supposto per assurdo che S non sia ottimale, esiste un S' ottimale per $P_{v,i-1}$ tale per cui L(S')<L(S) -- *dove L è lunghezza*--, perciò S' sarà ammissibile anche per il problema

$P_{v,i}$. Quindi, sarà anche migliore di S. Questo però è un assurdo perché S si suppone ottimale per $P_{v,i}$.

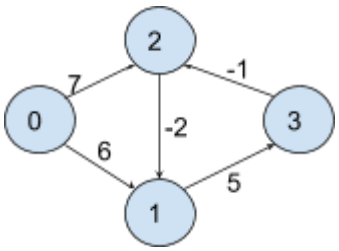
- **2° caso: i∈S**
  Supposto che S ha "i" archi: S \ {v} è soluzione ammissibile e ottimale per $P_{u,i-1}$.



Dove u è il nodo che precede v.
E' ottimale per il teorema della sottostruttura ottima (dei cammini minimi/sotto cammini). Quindi il teorema è dimostrato.

Algoritmo di Bellman Ford

```
Val(sorg,0)=0;
∀ v!=sorg val(v,0)=∞;   // non va impostato a -1 perchè gli archi possono essere negativi!
for(i=1 to n-1) //numero di archi in quanto abbiamo detto che un ciclo ha al max n-1 archi
        for(v=0 to n-1) //numero di nodi
        {
                val(v,i)=min(val(v,i-1),min ∀ (u,v) ∈ E [val(u,i-1)+peso(u,v)];
                //min ∀ (u,v) ∈ E → significa che il minimo lo sto facendo in ogni arco entrante in v
                Padre[v] =
                {
                        padre[v] se val(v,i-1) è il MINIMO
                        u se val(u,i-1)+peso(u,v) è MINIMO
                }
        }

return riga i=n-1;
```



Complessità: O(n*m) dove n sono gli archi del ciclo esterno e m invece corrispondono alla ricerca degli archi all'interno del ciclo.

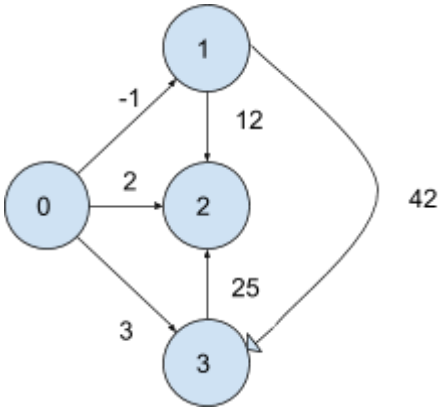
Il numero di archi nel cammino è n-1. Nel ciclo interno ogni volta andiamo a guardare tutti gli archi entranti nel nodo. Non è n*n*m Per risolvere questi cicli qui tutti insieme dovremmo andare a guardare tutti i singoli archi del grafo. Quindi il costo del ciclo for interno è m. Se grafo è connesso ha almeno n-1 archi, però il numero di archi potrebbe essere di più e arrivare nell'ordine di n^2. Per ogni iterazione dobbiamo guardare tutti gli archi entranti in un nodo. Però alla fine per ognuno dei cicli esterni guardiamo una volta sola ogni arco del grafo perchè ogni arco è entrante in un solo nodo(ogni arco lo guarderemo solo una volta) . Quindi, complessivamente i cicli sono n ma guardano m archi quindi siccome m è più grande di n allora metteremo il costo del ciclo come m.

Quanto è grande questa tabella??

	NODI			
ARCHI	**0**	**1**	**2**	**3**
0	0	∞	∞	∞
1	0	6	7	∞
2	0	5	7	11
3	0	5	7	10

In spazio costa O(n*n):
Perchè non è O(n*m)?
Perchè anche se il nostro grafo di partenza è completo e quindi con n^2-n archi, dato che il nostro problema riguarda i cammini minimi semplici, avremo al massimo n-1 archi].
Si potrebbe fare con 2 righe, dove una dipende dalla precedente.
Con una sola riga si potrebbe fare e potrebbe risultare più efficiente a costo di perdere qualcosa in chiarezza.

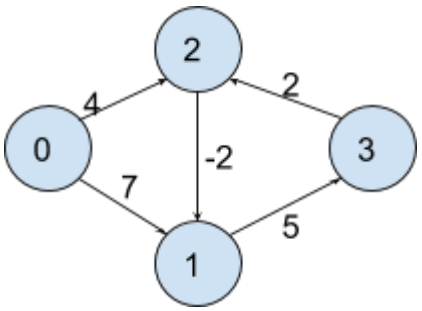
Viste le complessità se i pesi fossero tutti positivi conviene usare Dijkstra poichè Bellman-Ford è più pesante come algoritmo.
Inoltre, se il grafo fosse non orientato con pesi negativi non si potrebbe usare Bellman-Ford in quanto sono presenti i cicli banali (è come se ci fossero dei cicli negativi).
Il problema è quindi posto male e non avrebbe senso con questa impostazione.
In un grafo non orientati i pesi negativi creano cicli semplici negativi banali.



	NODI			
ARCHI	**0**	**1**	**2**	**3**
0	0	∞	∞	∞
1	0	-1	2	3
2	0	-1	2	3
3	0	-1	2	3

Se vediamo che una riga è uguale a quella precedente allora sicuramente non cambierà più nulla, potremmo risparmiare del tempo fermandoci li.
Il motivo per cui si può fare questa affermazione è: dato che una riga viene calcolata sul valore della riga precedente se si trovano 2 righe uguali allora quei valori non cambieranno più.
Se un grafo ha cicli negativi?
Dato che se in 1 riga abbiamo già raggiunto il minimo quella riga non cambierà più.
Se deve cambiare cambierebbe entro la riga n-1.
Con cicli negativi cambierebbe anche dopo n-1 iterazioni.
Se cambia vuol dire che è presente un ciclo negativo e quindi non è possibile trovare il cammino minimo.

Come ricostruiamo la soluzione?
IDEA: Utilizziamo l'array dei padri il quale però cambierebbe nel tempo in quanto si sovrascriverebbe in continuazione.



1° ITERAZIONE (1 arco)

0	1	2	3
-1	0	0	-1

2° ITERAZIONE (2 archi)

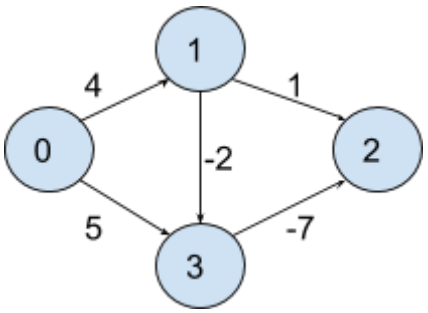
0	1	2	3
-1	2	0	1

3° ITERAZIONE (3 archi)

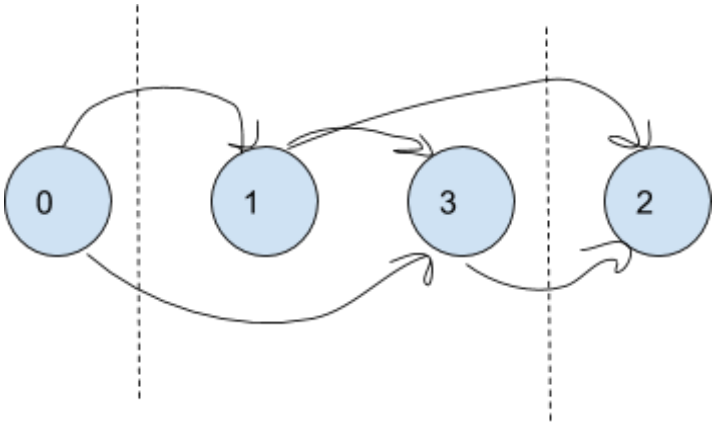
0	1	2	3
-1	2	0	1

Padre[v] = |
| **padre[v]** se val(v,i-1) è il MINIMO
| **u** se val(u,i-1)+peso(u,v) è MINIMO
|

Grafo DAG:

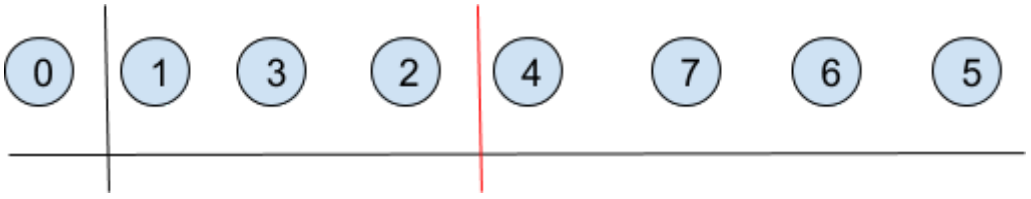


Un’altra situazione vantaggiosa si verifica quando il grafo è un DAG: in questo caso può seguire l’ordine topologico.



Bellman-ford farebbe n iterazioni in tutto.
Se usiamo la minimizzazione in ordine topologico, quando guardiamo un nodo, tutti gli altri che sono antecedenti hanno già il valore minimo assegnato. E’ sufficiente fare un’unica passata e il costo diventa O(n)(ordine di n). Dimostriamolo per induzione.

Dato il seguente ordine topologico:



Dimostrazione dell’audio

Supponiamo che ad un certo punto abbiamo sistemato tutti i nodi precedenti e abbiamo calcolato la lunghezza del cammino minimo. Arriviamo al prossimo nodo, naturalmente, la lunghezza dipenderà da tutto ciò che c’era prima e per il teorema della sottostruttura ottima dei cammini minimi avremo già i cammini minimi dalla sorgente a tutti i nodi dai quali si può raggiungere 4 (l’idea è quella di Dijkstra) . Seguendo l'ordine topologico ci allontaniamo sempre più dalla sorgente e anche se ci sono pesi negativi non possiamo avere sorprese perché non può succedere che girando da un'altra parte otteniamo un cammino migliore. Tutti i possibili cammini per arrivare a questo nodo vengono già trattati. Quindi questo nodo si può sistemare semplicemente guardando i nodi precedenti. Se per ipotesi induttiva i nodi precedenti sono apposto, dopo questo passo sarà apposto anche 4.
Il passo induttivo,quindi, è dimostrato ed è vero per tutti i nodi.

E’ possibile farlo poiché i nodi sono ordinati in un certo modo. L’ordine topologico garantisce che tutto quello che può influenzare 4 è già stato discusso e sistemato, e quindi quello che succederà dopo non cambierà più 4.

In conclusione: Se un grafo ammette un ordine topologico, anche se ha pesi negativi, si può trasformare Bellman-Ford in un algoritmo molto efficiente.

Dimostrazione per induzione:

Base:
Per il nodo 0 è corretto in quanto viene messo a 0 dall’algoritmo.
Ipotesi induttiva:
Con k nodi:
Per tutti i nodi fino a k abbiamo calcolato il cammino minimo ed è corretto.

Tesi induttiva:
Per k+1 nodi, se sappiamo che fino a k nodi la distanza è calcolata correttamente allora dato che il k+1 esimo nodo è raggiungibile solo dai nodi precedenti siamo sicuri che il cammino è il minimo tra quelli presenti prima del k+1 esimo nodo più l’arco che porta al k+1 esimo nodo. (Viene utilizzato una sorta di Dijkstra).

1) Determinare l’ordine topologico Ot(n)

2) for(i=1 to n)
val(ot[i]) = min $\forall$ (u,v) $\in$ E(val(u)+peso(u,v));

Complessità: **O(n+m)**
ha costo O(n+m) in quanto n sono i cicli e m sono gli archi della ricerca interna.
Lezione 31/05/2018

Cammini minimi tra ogni coppia di nodi

Vogliamo calcolare dato un grafo pesato orientato con pesi anche negativi il cammino minimo tra ogni coppia di nodi.

Analisi della complessità

Con Dijkstra

Se supponiamo che ci siano solo pesi positivi, si potrebbe usare **Dijkstra** che ha costo O(m*log n) dato che dobbiamo calcolare il cammino minimo per ogni coppia di nodi il costo diventerà O(n*m (log n)) poichè dobbiamo iterare una volta per nodo.
Abbiamo 2 casi che il grafo sia sparso oppure denso.

- Se è Sparso il costo diventerà → O(n^2 log n).

- Se è Denso il costo diventerà → $O(n^3 \log n)$).

Con Bellman-Ford
Un'altro algoritmo che abbiamo visto è **Bellman-Ford** il quale ha un costo O(n*m) dato che bisogna considerarlo per ogni coppia di nodi il costo diventerà $O(n^2*m)$ e anche qui abbiamo 2 possibilità o il grafo è sparso oppure è denso.
- Se è Sparso costo → $O(n^3)$
- Se è Denso costo → $O(n^4)$

Ricordiamo che Bellman-Ford a differenza di Dijkstra funziona anche con pesi negativi.

Algoritmo di Floyd Warshall

Floyd-Warshall viene utilizzato per grafi orientati, pesati e **senza cicli negativi.**
Questo algoritmo è stato ideato per calcolare il cammino minimo tra ogni coppia di nodi anche se i pesi sono negativi con costo $O(n^3)$.
Se avessimo un grafo sparso con pesi positivi ci converrebbe iterare n volte Dijkstra.
Invece, guardando le complessità di Dijkstra e di Bellman-Ford può valere la pena utilizzare Floyd Warshall in tutti i casi in cui il grafo sia denso.
L'idea di Floyd-Warshall mentre Bellman-Ford guarda la lunghezza del cammino minimo, Floyd-Warshall limita i nodi che possono stare in mezzo tra i 2.

E' un algoritmo di programmazione dinamica
L'idea di Floyd-Warshall limita la lunghezza del cammino in modo indiretto, ovvero che limita i nodi in mezzo tra i due presi in considerazione.

V_k={1,...,k} è un sottoinsieme dei nodi fino a k. → $V_k \subseteq V$
Si ha il problema $P_{i,j,k}$ nel quale dobbiamo determinare il cammino minimo da i a j con nodi intermedi appartenenti a V_k. Quindi, i nodi intermedi devono essere tra 1 e k.
Questa non è una limitazione per i nodi i e j ma solo per i nodi intermedi, idealmente i e j potrebbero essere anche k+1 e k+2.

Programmazione dinamica

Ipotesi: S soluzione ottimale per $P_{i,j,k}$.
Il nostro interesse è quello di ridurre in sottoproblemi in cui k diminuisce.

Abbiamo 2 possibilità:
1) k $\notin$ *nodi intermedi* → *k non è tra i nodi intermedi*
2) k $\in$ *nodi intermedi* → *k è tra i nodi intermedi*

1) k $\notin$ *nodi intermedi*
Avremo un percorso i → j che attraversa solo nodi presenti in V_{k-1}.

Tesi 1:Se k $\notin$ nodi intermedi S è ammissibile e ottimale per $P_{i,j,k-1}$.

È ammissibile perchè k $\notin$ *ai nodi intermedi quindi S è ammissibile per* $P_{i,j,k-1}$.
Per dimostrare che è ottimale lo si dimostra per assurdo come al solito.
Se per assurdo S non è ottimale per $P_{i,j,k-1}$ *allora esisterà una soluzione S* tale che L(S*)<L(S) per il problema* $P_{i,j,k-1}$ *questa soluzione è ammissibile anche per il problema* $P_{i,j,k}$ *e sarà anche migliore di S quindi è assurdo.*

2) k $\in$ *nodi intermedi*
Avremo un percorso: i → k → j
Lo possiamo dividere in 2 soluzioni:
S1: i → k
S2: k → j
Tesi 2:

1) **S1 è ammissibile e ottimale per** $\mathbf{P_{i,k,k-1}}$.
2) **S2 è ammissibile e ottimale per** $\mathbf{P_{k,j,k-1}}$.

1)Vuol dire che nella soluzione S1 i nodi intermedi sono nodi da 1 a k-1, k non può essere presente tra i nodi intermedi perchè sennò vorrebbe dire che c'è un ciclo e dato che stiamo parlando di cammini minimi non può essere presente un ciclo(dato che si possono considerare solo cicli positivi, se fosse presente un ciclo esso porterebbe a un costo maggiore della soluzione). S1 è soluzione ammissibile per il problema $\mathbf{P_{i,k,k-1}}$.
E' ammissibile poichè il cammino che va da i a j contiene all'interno solo nodi tra 1 e k-1.

2)Vuol dire che nella soluzione S2 i nodi intermedi sono nodi fino a k-1, k non può essere presente tra i nodi intermedi perchè sennò vorrebbe dire che c'è un ciclo e dato che stiamo parlando di cammini minimi non può essere presente un ciclo(dato che si possono considerare solo cicli positivi, se fosse presente un ciclo esso porterebbe a un costo maggiore della soluzione). S2 è soluzione ammissibile per il problema $\mathbf{P_{k,j,k-1}}$.

Quindi queste 2 porzioni di cammino sono ammissibili e sono **ottimali per il teorema della sottostruttura ottima dei cammini minimi.**
**
***Teorema di sottostruttura ottima dei cammini minimi**:*
Se prendiamo una qualunque coppia di nodi di un cammino minimo, sicuramente il cammino tra quella coppia di nodi è minimo.
**

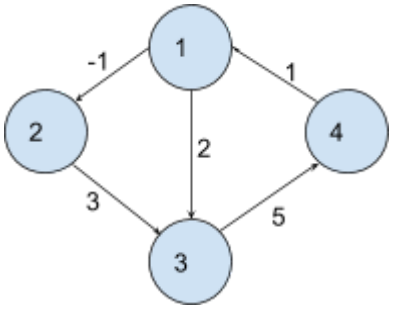
Pseudo-codice Algoritmo di Floyd-Warshall

```
// inizializzazione per ogni coppia i,j
if(i=j) val(i,j,0) = 0;
if( (i,j) ∈ E ) val(i,j,0) = peso(i,j);      //Se esiste l'arco (i,j)
if( (i,j) ∉ E ) val(i,j,0) = ∞;              //Se non esiste l'arco (i,j)

// floyd-warshall
```

```
for(k=1 to n) //O(n)
        for(i=1 to n) // O(n)
                for(j=1 to n) //O(n)
                        val(i,j,k) = min( val(i,j,k-1) ,val(i,k,k-1)+val(k,j,k-1)); //O(1)
```

Costo totale è $O(n^3)$

Esempio di applicazione dell'algoritmo



/* K=n indica che si può fare "tappa" solo nei nodi n, ad esempio: K=2 si può fare scalo solo in 1 e 2 */

K=0	1	2	3	4
1	0	-1	2	∞
2	∞	0	3	∞
3	∞	∞	0	5
4	1	∞	∞	0

K=1 V={1}	1	2	3	4
1	0	-1	2	∞
2	∞	0	3	∞
3	∞	∞	0	5
4	1	*0*	*3*	0

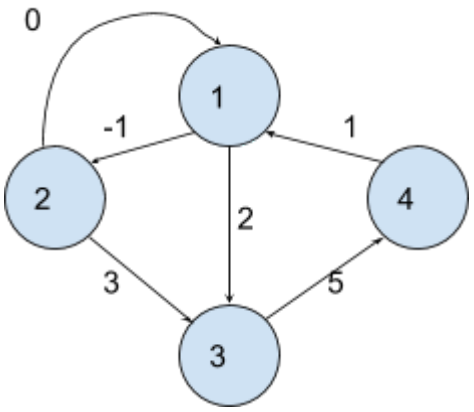
K=2 V={1,2}	1	2	3	4
1	0	-1	2	∞
2	∞	0	3	∞
3	∞	∞	0	5
4	1	0	3	0

K=3 V={1,2,3}	1	2	3	4
1	0	-1	2	*7*
2	∞	0	3	*8*
3	∞	∞	0	5
4	1	0	3	0

K=4 V={1,2,3,4}	1	2	3	4
1	0	-1	2	7
2	*9*	0	3	8
3	*6*	*5*	0	5
4	1	0	3	0

Servono almeno 2 tabelle nel nostro caso una vecchia e una nuova.
Quindi avremo un costo in spazio dato da $O(2*n^2)$.

Potremmo usarne solo una sovrascrivendo sempre perdendo qualcosa in chiarezza.
Nel caso in cui un ciclo negativo sia presente, questo ciclo si chiuderà in almeno un nodo quindi sulla diagonale a un certo punto potremmo trovarci invece di uno 0 un valore negativo.



Ci potremmo fermare monitorando la diagonale e verificando se si presenta un valore negativo oppure no.

Problema Massima Sottosequenza Comune

Vogliamo confrontare due stringhe di caratteri per stabilire quanto differiscono una dall'altra (massima sottosequenza comune): La LCS **(Longest Common Subsequence)** non tiene solo conto di quali caratteri compaiono nelle due stringhe ma anche dell'ordine in cui essi compaiono.

Sequenza di caratteri: caratteri messi in fila seguendo un certo ordine

Definiamo il concetto di **sottosequenza**:
Una sottosequenza è un insieme di elementi che compare nella sequenza nello stesso ordine.
Una stringa *w* è sottosequenza di un'altra stringa *x,* se *w* è ottenibile da *x* cancellando zero o più caratteri di x.

Es: "DICIOTTO" - "DCTT" è una sottosequenza di DICIOTTO ottenuta eliminando "I" e "O".

Se prendiamo due parole come "DICIOTTO" e "CIAO" la LCS sarà "CIO"

DIMOSTRAZIONE:

Il problema da affrontare è quello della massima sottosequenza comune tra due stringhe.

Poniamo:
Una sequenza lunga m → $X_m=x_1,x_2,...,x_m$
e una lunga n → $Y_n=y_1,y_2,....,y_n$

La massima sottosequenza comune sia → $Z_k= Z_1,Z_2,....,Z_K$ è LCS(X_m,Y_n) per il problema $P_{m,n}$.
Supponiamo Z_k soluzione ottimale per il problema $P_{m,n}$, guardiamo l'ultimo elemento di Z, ovvero Z_k.

1) $Z_k = Y_n = X_m \rightarrow Z_{k-1}$ è soluzione ammissibile e ottimale per $P_{m-1,n-1}$
Se Z_{k-1} per assurdo non fosse ottimale per il problema $P_{m-1,n-1}$ allora esiste un'altra soluzione Z* migliore di Z_{k-1} per il problema $P_{m-1,n-1}$ allora se aggiungiamo alla soluzione il k-esimo carattere verrebbe fuori che Z* U {k} migliore di Z_k per il problema $P_{m,n}$ il che è assurdo.

2) $Z_k \neq X_m \rightarrow Z_k$ è soluzione ammissibile e ottimale per $P_{m-1,n}$.
Se per assurdo Z_k non ottimale per $P_{m-1,n}$ allora esisterebbe una soluzione Z* migliore di Z_k per il problema $P_{m-1,n}$. Quindi, questa scelta la potevo fare anche quando l'm-esimo carattere c'era e dato che Z* è ammissibile per il problema $P_{m,n}$ sarebbe anche migliore di Z_k per il medesimo problema il che è assurdo.

3) $Z_k \neq Y_n \rightarrow Z_k$ ammissibile e ottimale $P_{m,n-1}$
Questa dimostrazione è uguale alla 2) solo che si poteva fare la stessa scelta quando n-esimo carattere c'era.

Quindi abbiamo 2 casi che X_i sia uguale a y_j oppure che X_i sia diverso Y_j

if (X_i=Y_j) L(i,j) = L(i-1,j-1)+1;

if ($X_i \neq Y_j$) L(i,j) = max(L(i,j-1), L(i-1,j);

Pseudo-codice Algoritmo

```
//Come si inizializza? Tutte a 0
L(0,j) = 0 per ogni j
L(i,0) = 0 per ogni i
for i=1 to m
        for j=1 to n
                if (Xi=Yj) L(i,j) = L(i-1,j-1)+1;
                if (Xi≠Yj) L(i,j) = max(L(i,j-1), L(i-1,j);
```

Complessità: O(n*m) in tempo
O(n*m) in spazio

Esempio di applicazione dell'algoritmo

	ε	D	I	C	I	O	T	T	O
ε	0	0	0	0	0	0	0	0	0
C	0	0	0	1	1	1	1	1	1
I	0	0	1	1	**2**	2	2	2	2
A	0	0	1	1	2	2	2	2	2
O	0	0	1	1	2	3	3	3	3

Si può notare che quando incontriamo 2 caratteri uguali tutta la riga precedente a quel valore non mi serve più.
Come nell'esempio nella posizione della matrice rispettiva a i,i dalla costruzione della matrice notiamo che tutti i valori della riga prima di 2 non ci serviranno più in quanto 2 è stato calcolato guardando solo in posizione l(i-1,j-1).Quindi si può pensare ad utilizzare un approccio ricorsivo per evitare di fare del lavoro in più.
Se utilizzassi un approccio Top Down ricorsivamente risparmierei dei valori calcolati.
Esempio con 2 stringhe uguali
X1=CIAO
Y1=CIAO

	C	I	A	O
C	1			
I		2		
A			3	
O				4

In questo caso diventerebbe lineare.
Questo è un caso estremo ,in questo caso utilizzando l'approccio ricorsivo risparmierei un sacco di lavoro.
Anche nel caso generale risparmierei comunque del lavoro.

Come ricostruisco la soluzione?
Si opera all'indietro come negli altri problemi di prog dinamica.

Credits: *Simone Caggese, Edoardo Favorido,* ***Andrea Ierardi***
collaboratori: *Marco Rizzi, Andrea Ferrari, Marco CaDrega,* ***Franco Minchia***

Wall of fame