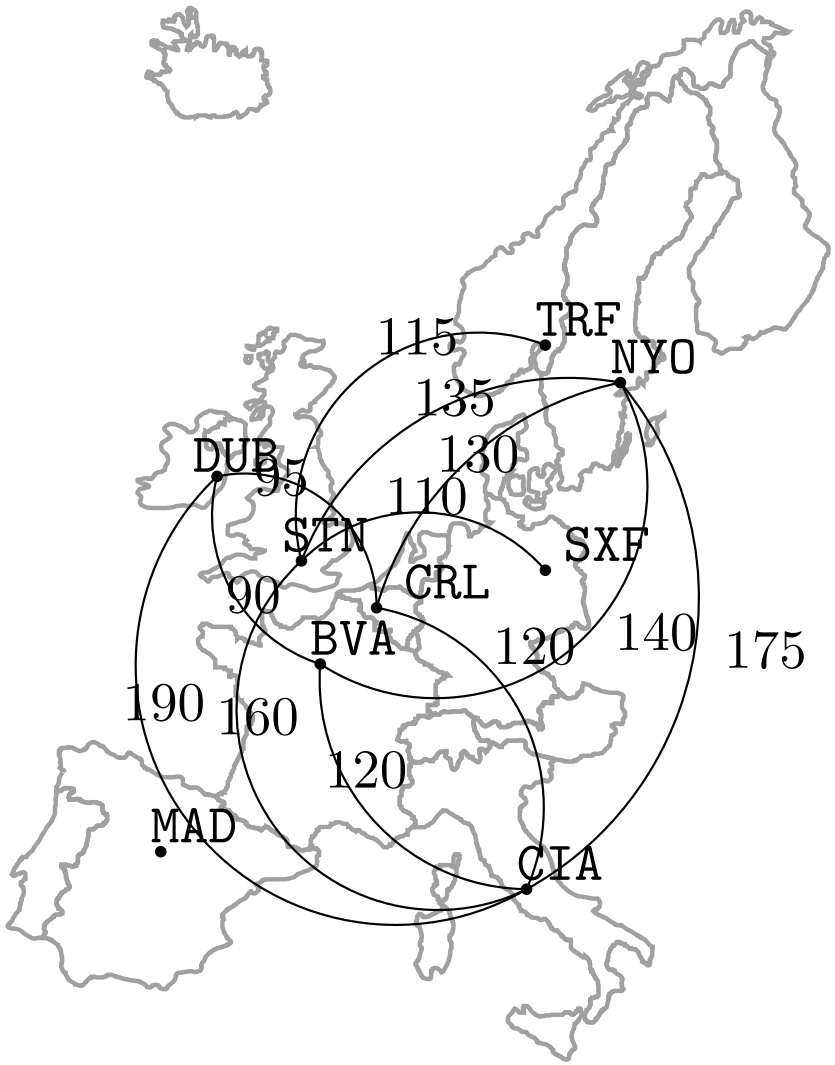
Capitolo 7
Grafi

Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

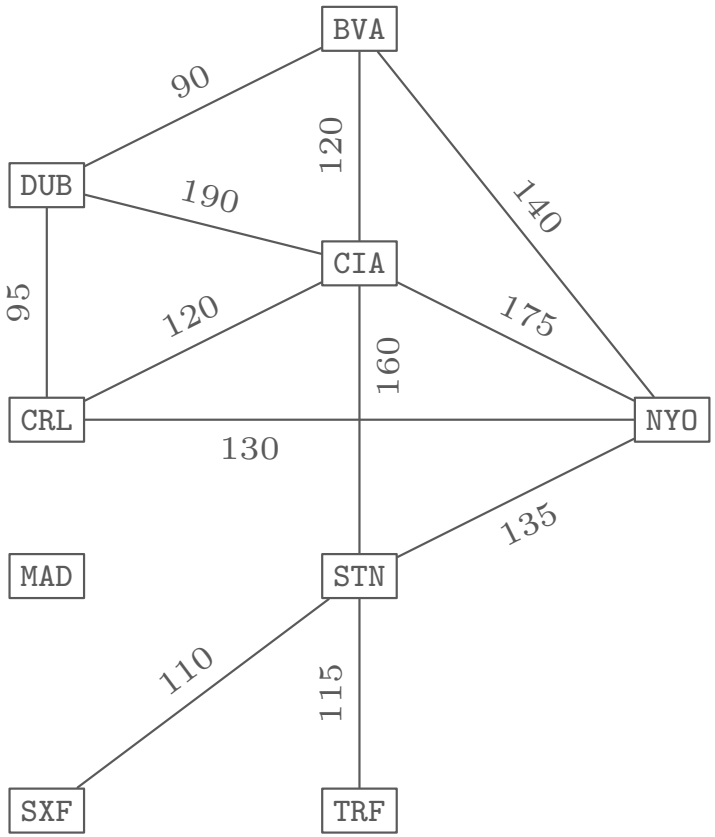
1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

Grafi

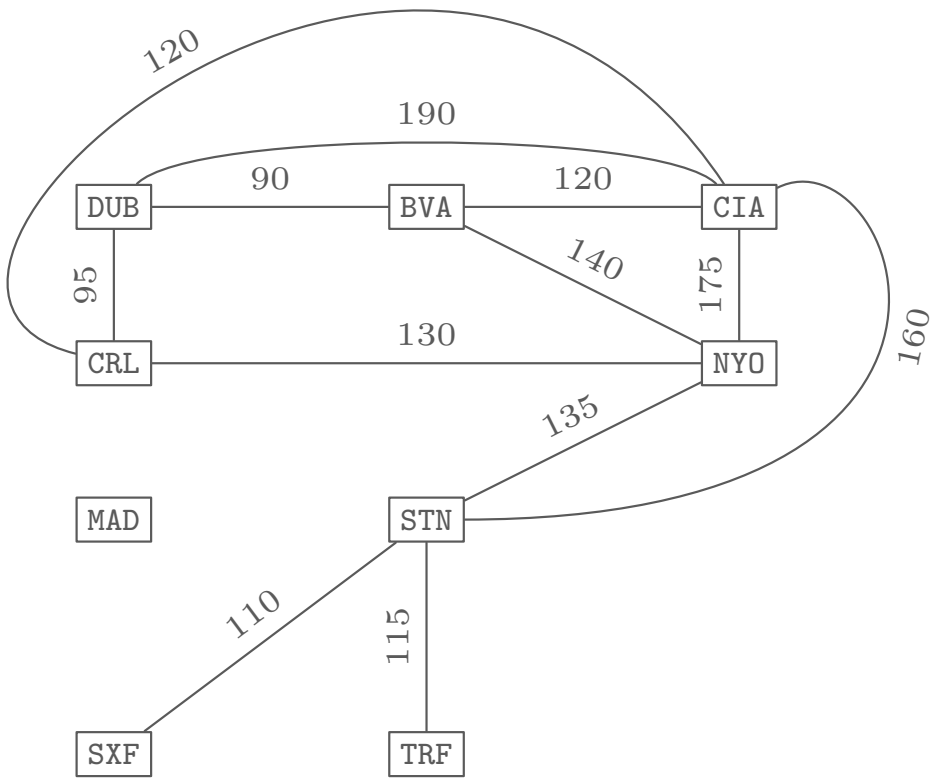
- $G = (V, E)$
- $V = [0, n-1] =$ insieme di n **vertici**
- $E \subseteq V \times V =$ insieme di m **archi**
- Opzionale $w : E \to R =$ **peso** degli archi $\Rightarrow$ $G = (V, E, W)$



Rappresentazione a grafo e tabellare



Rappresentazione a grafo e tabellare



Rappresentazione tabellare

	SXF	CRL	DUB	STN	MAD	TRF	BVA	CIA	NYO
SXF	–	–	–	110	–	–	–	–	–
CRL	–	–	95	–	–	–	–	120	130
DUB	–	95	–	–	–	–	90	190	–
STN	110	–	–	–	–	115	–	160	135
MAD	–	–	–	–	–	–	–	–	–
TRF	–	–	–	115	–	–	–	–	–
BVA	–	–	90	–	–	–	–	120	140
CIA	–	120	190	160	–	–	120	–	175
NYO	–	130	–	135	–	–	140	175	–

Grafi pesati

- Funzione $W : E \mapsto \mathbb{R}$ definita sul grafo $G = (V, E)$: assegna un valore (reale) a ogni arco del grafo
- Un grafo non pesato può essere visto come un grafo pesato con peso pari a 1 per tutti gli archi
- **Cammino** da un nodo u a un nodo z: sequenza di nodi $x_0, x_1, x_2, \ldots, x_k$ tale che $x_0 = u$, $x_k = z$ e $(x_i, x_{i+1}) \in E$ per ogni $0 \leq i < k$
- **Ciclo**: cammino per cui vale $x_0 = x_k$, ovvero che ritorna nel nodo di partenza
- Cammino (o ciclo) **semplice**: non attraversa alcun nodo più di una volta

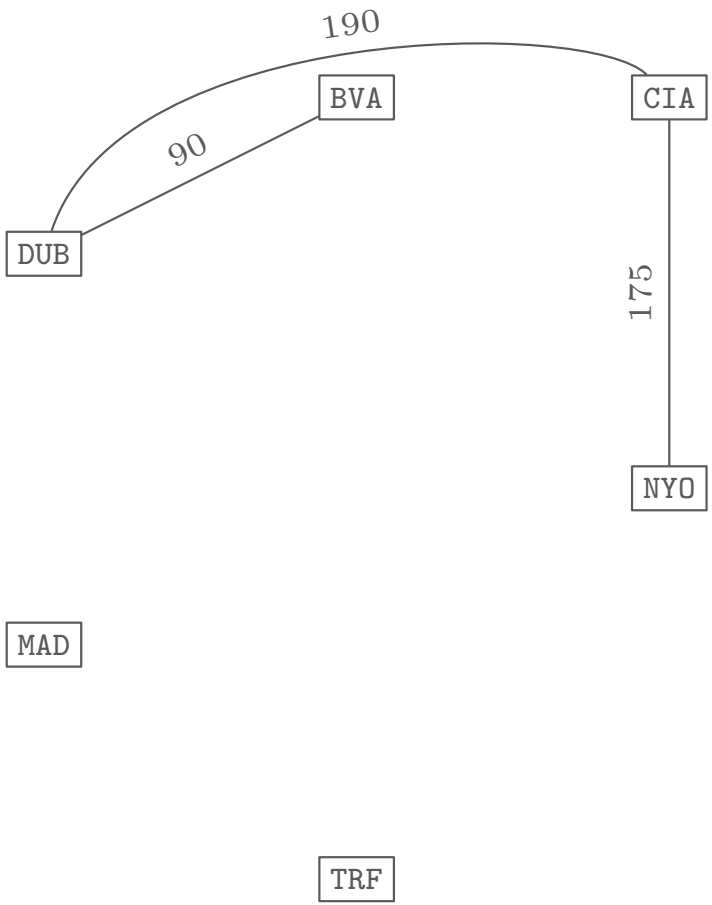
Cammini minimi

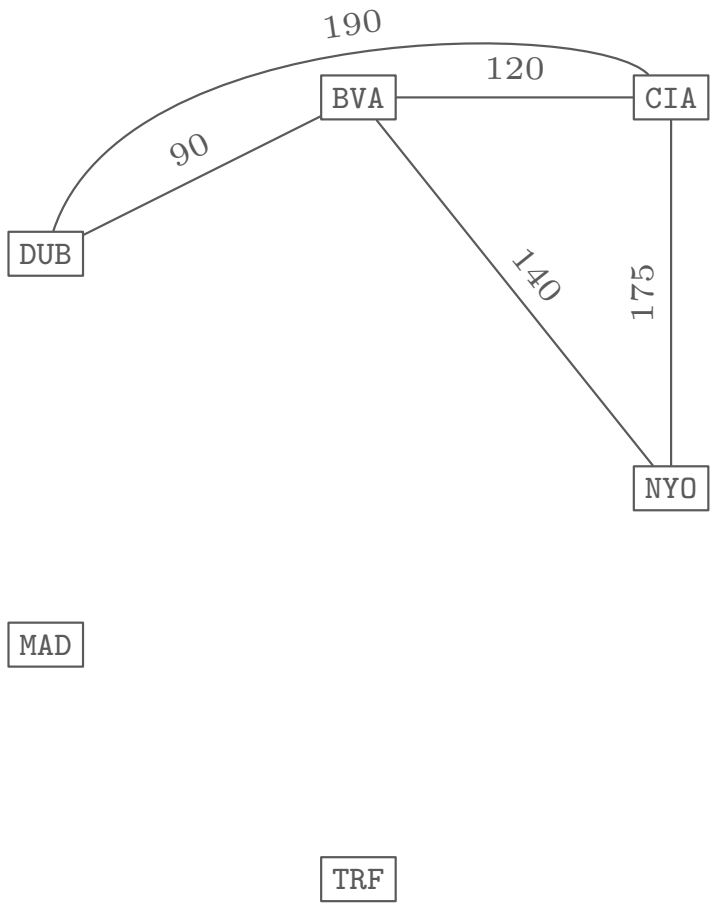
- **Lunghezza** di un cammino $x_0, x_1, x_2, \ldots, x_k$: numero k di archi attraversati
- Nel caso di grafi pesati: **peso di un cammino**, somma dei pesi degli archi attraversati , ovvero come $\sum_{i=0}^{k-1} W(x_i, x_{i+1})$
- **Cammino minimo pesato**: cammino di peso minimo tra due nodi
- **Distanza pesata** peso del cammino minimo (oppure $+\infty$ se non esiste alcun cammino tra i due nodi)

Sottografo e sottografo indotto

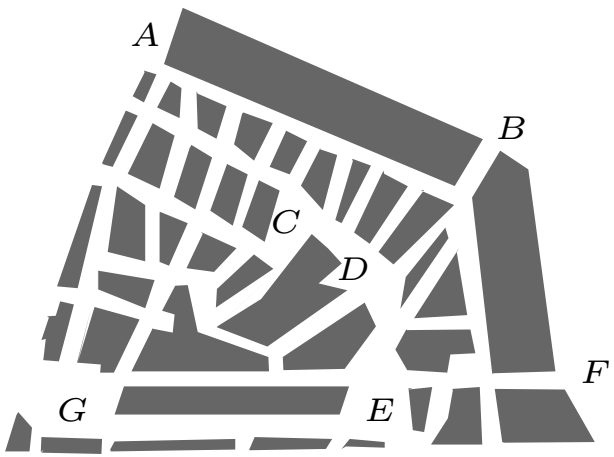
- **Sottografo** di $G = (V, E)$: grafo $G' = (V', E')$, dove $V' \subseteq V$, $E' \subseteq V' \times V'$ e, inoltre, $E' \subseteq E$
- **Sottografo indotto** di $G = (V, E)$: sottografo $G' = (V', E')$, con la condizione aggiuntiva che per ogni coppia $v_1, v_2 \in V'$, $(v_1, v_2) \in E'$ se e solo se $(v_1, v_2) \in E$
- **Componente connessa** di un grafo $G = (V, E)$: sottografo $G' = (V', E')$ di G
  - connesso: tutti i nodi di V' connessi tra loro
  - massimale: non esistono nodi in $V - V'$ connessi ai nodi di V'

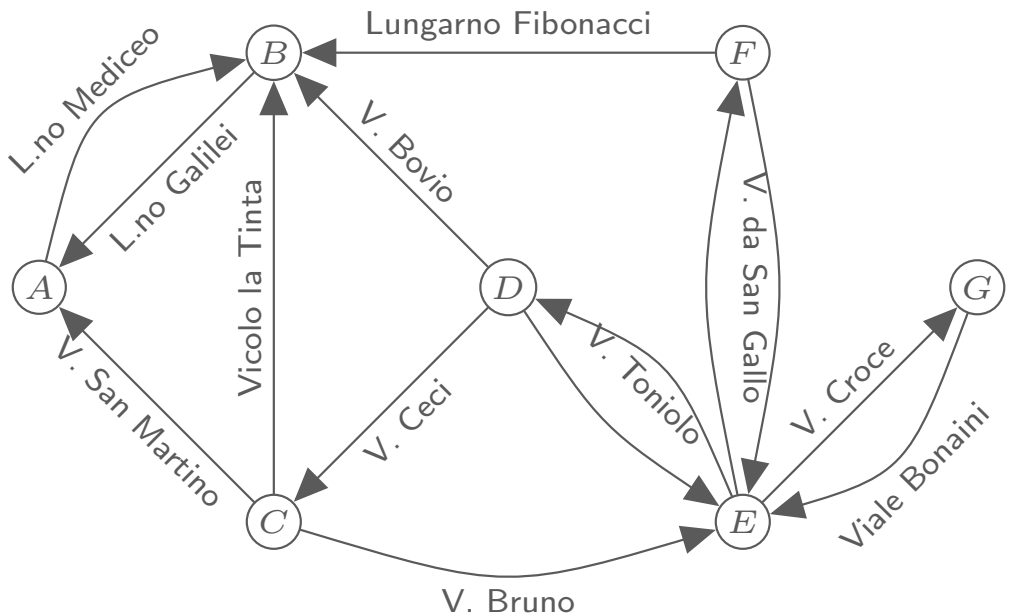
Sottografo e sottografo indotto





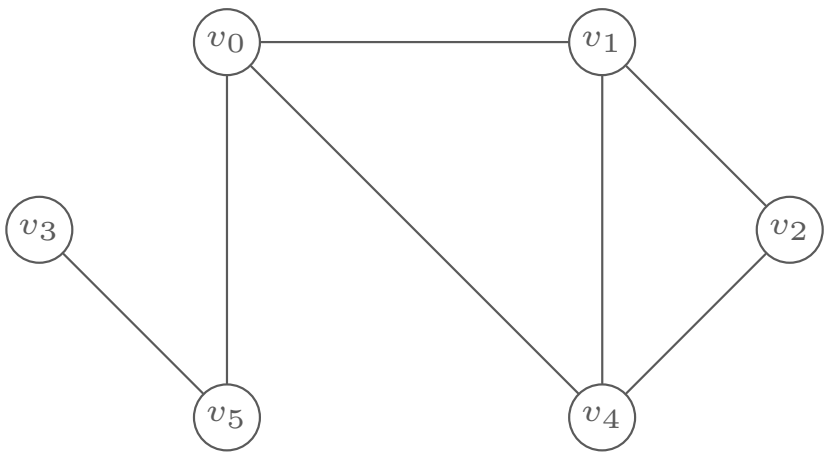
Grafo diretto





Esempi di problemi su grafi

Accoppiamento perfetto(perfect matching): Dato $G = (V, E)$, trovare un sottoinsieme $E' \subseteq E$ tale che tutti i nodi in V siano incidenti agli archi di E' e ogni nodo in V compaia soltanto in un arco di E'



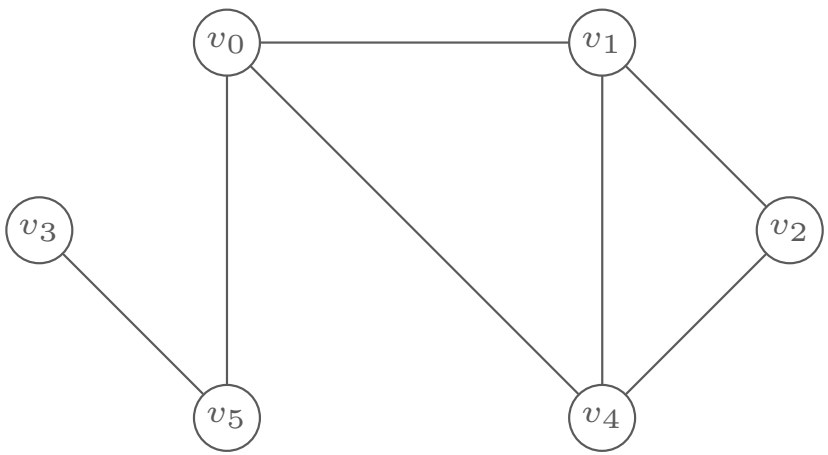
Due abbinamenti diversi:

- $\{(v_0, v_1), (v_2, v_4), (v_3, v_5)\}$
- $\{(v_0, v_4), (v_1, v_2), (v_3, v_5)\}$

Esempi di problemi su grafi

Cammino hamiltoniano: cammino che attraversa tutti i nodi una e una sola volta (permutazione $\langle \pi_0, \pi_1, \ldots, \pi_{n-1} \rangle$ dei nodi tale che $(\pi_i, \pi_{i+1}) \in E$ per ogni $0 \leq i \leq n-2$)

Ciclo hamiltoniano: cammino hamiltoniano $\langle \pi_0, \pi_1, \ldots, \pi_{n-1} \rangle$ completato da un arco $(\pi_{n-1}, \pi_0) \in E$



Quattro cammini diversi:

- $\langle v_3, v_5, v_0, v_1, v_2, v_4 \rangle$
- $\langle v_3, v_5, v_0, v_1, v_4, v_2 \rangle$
- $\langle v_3, v_5, v_0, v_4, v_2, v_1 \rangle$
- $\langle v_3, v_5, v_0, v_4, v_1, v_2 \rangle$

Esempi di problemi su grafi

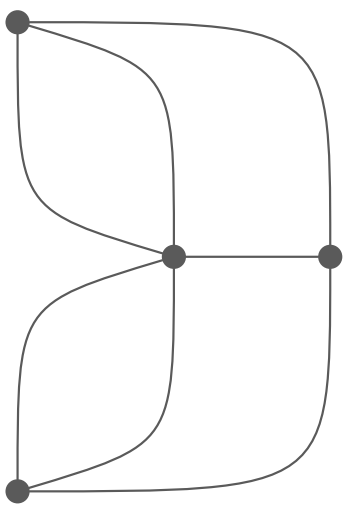
Ciclo euleriano: ciclo che attraversa tutti gli archi una e una sola volta (permutazione $\langle e_0, e_1, \ldots, e_{m-1} \rangle$ degli archi tale che e_i, e_{i+1} sono incidenti ad uno stesso nodo per ogni $0 \leq i \leq m-2$) e che nodo iniziale nodo finale coincidono

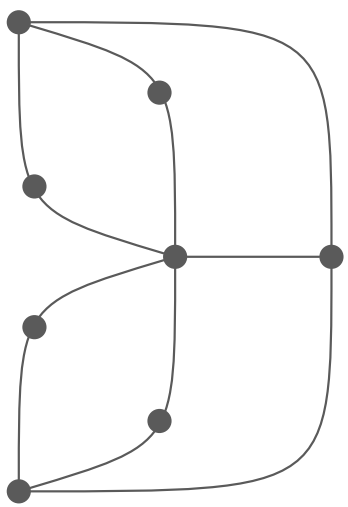
Problema dei ponti di Königsberg, di L. Euler

Esempi di problemi su grafi

Condizione necessaria e sufficiente per l'esistenza di un ciclo euleriano in G:

- G sia connesso
- i suoi nodi abbiano tutti grado pari





Il problema dei ponti di Königsberg non ha soluzione: 4 nodi di grado dispari

Rappresentazione di grafi: matrice di adiacenza

Matrice di adiacenza: dato un grafo $G = (V, E)$, matrice A di $n \times n$ elementi in $\{0, 1\}$ tale che $A[i][j] = 1$ se e solo se $(i, j) \in E$ per $0 \leq i, j \leq n - 1$

	SXF	CRL	DUB	STN	MAD	TRF	BVA	CIA	NYO
SXF	0	0	0	1	0	0	0	0	0
CRL	0	0	1	0	0	0	0	1	1
DUB	0	1	0	0	0	0	1	1	0
STN	1	0	0	0	0	1	0	1	1
MAD	0	0	0	0	0	0	0	0	0
TRF	0	0	0	1	0	0	0	0	0
BVA	0	0	1	0	0	0	0	1	1
CIA	0	1	1	1	0	0	1	0	1
NYO	0	1	0	1	0	0	1	1	0

Rappresentazione di grafi: matrice di adiacenza

- Spazio $\Theta(n^2)$
- $O(1)$ tempo per stabilire se $(i, j) \in E$
- $O(n)$ tempo per scandire gli archi incidenti a un nodo

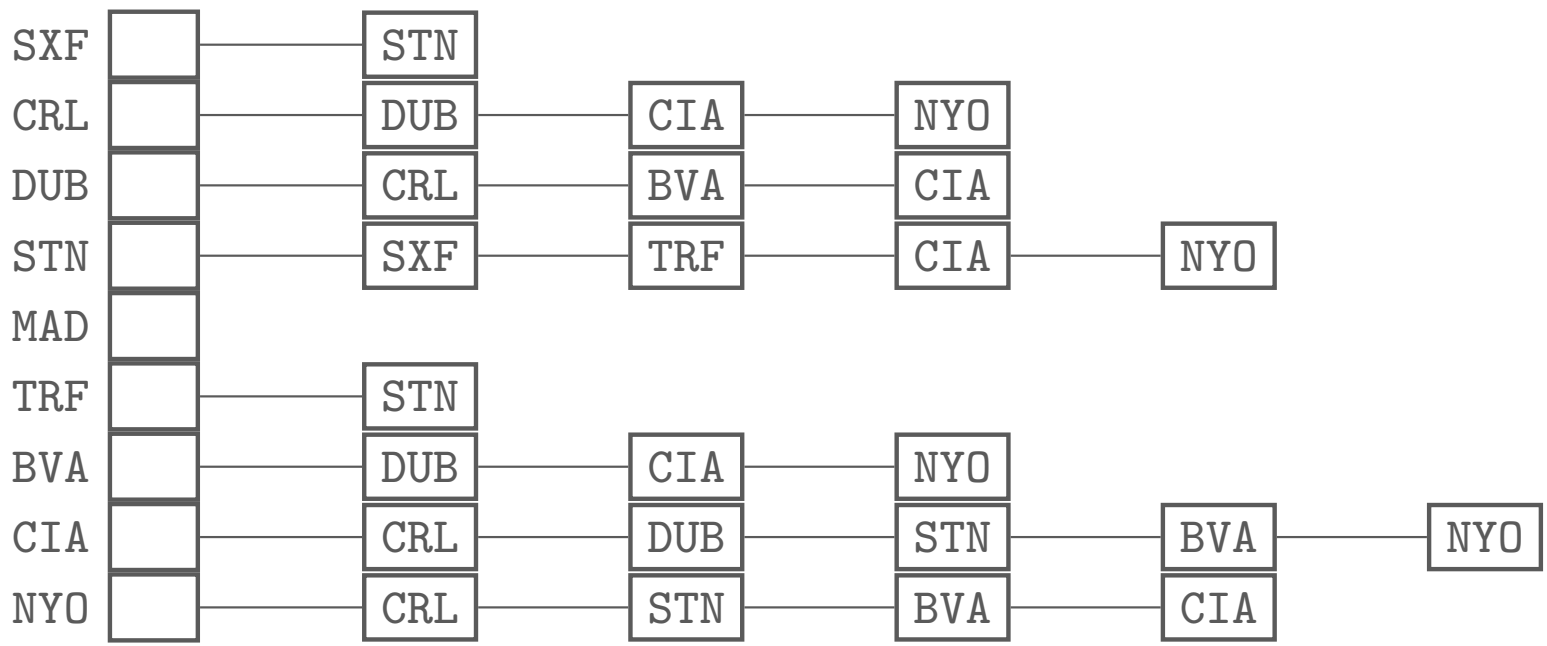
```
FOR (j = 0; j < n; j = j+1) {
  IF (A[i][j] != 0) {
    PRINT arco (i,j);
    PRINT peso P[i][j] dell'arco, se previsto;
  }
}
```

- Grafi non orientati: $A[i][j] = A[j][i]$ per ogni $0 \leq i, j \leq n-1$, e quindi A è una matrice simmetrica
- Grafo pesato: associata a A, matrice P dei pesi che rappresenta in forma tabellare la funzione W

Rappresentazione di grafi: liste di adiacenza

Liste di adiacenza: per ogni nodo i, lista `listaAdiacenza`$[i]$ dei nodi adiacenti.

- il nodo ha grado zero, la lista è vuota
- altrimenti, `listaAdiacenza`$[i]$ è una lista doppia con un riferimento sia all'elemento iniziale che a quello finale



Rappresentazione di grafi: liste di adiacenza

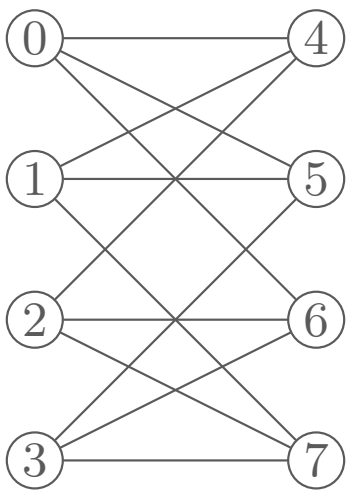
- Spazio $\Theta(n+m)$
- $O(\min\{\text{grado}(i), \text{grado}(j)\})$ tempo per stabilire se $(i,j) \in E$
- $O(\text{grado}(i))$ tempo per scandire gli archi incidenti al nodo i

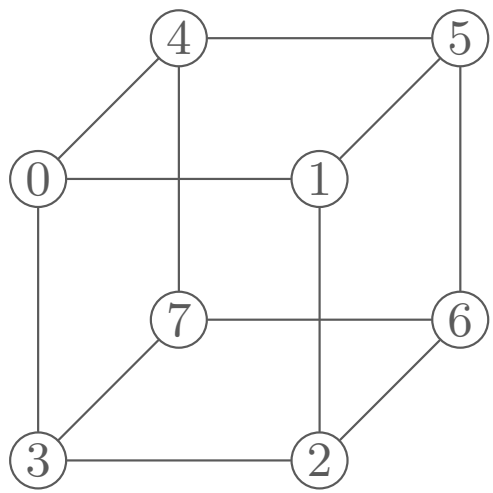
```
x = listaAdiacenza[i].inizio;
WHILE (x != null) {
  j = x.dato;
  PRINT (i,j);
  PRINT x.peso (se previsto);
  x = x.succ;
}
```

Isomorfismo (rappresentazione)

Grafi isomorfi: una semplice rinumerazione dei vertici li rende uguali

Esistono $n!$ modi per enumerare i vertici con valori distinti in $V = \{0, 1, \ldots, n-1\}$ e, quindi, altrettanti modi per rappresentare lo stesso grafo





Chiusura transitiva

$G^* = (V, E^*)$ è la **chiusura transitiva** di G se, per ogni coppia di vertici $i, j \in V$, vale $(i, j) \in E^*$ se e solo se esiste un *cammino* in G da i a j

A: matrice di adiacenza di G, *modificata* in modo che gli elementi della diagonale principale hanno tutti valori pari a 1.

- $A^2 = A \times A$ tale che $A^2[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot A[k][j]$ se e solo se esiste un cammino di lunghezza 2 da i a j
- $A^3 = A^2 \times A$ tale che $A^3[i][j] = \sum_{k=0}^{n-1} A^2[i][k] \cdot A[k][j]$ se e solo se esiste un cammino di lunghezza 3 da i a j
- $\cdots$
- $A^r = A^{r-1} \times A$ tale che $A^r[i][j] = \sum_{k=0}^{n-1} A^{r-1}[i][k] \cdot A[k][j]$ se e solo se esiste un cammino di lunghezza r da i a j

Se $A^r = A^{r-1}$ tutti cammini sono rappresentatai e $A^r = A^*$

Chiusura transitiva

$A=$

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	0	0	1	1	0	0	0	0	0
1	1	1	1	1	1	0	0	1	0	0	0
2	0	1	1	0	1	0	0	0	0	0	0
3	0	1	0	1	0	1	1	1	0	0	0
4	1	1	1	0	1	0	0	0	0	0	0
5	1	0	0	1	0	1	1	1	0	0	0
6	0	0	0	1	0	1	1	1	0	0	0
7	0	1	0	1	0	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

Chiusura transitiva

$A^2 =$

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0
2	1	1	1	1	1	0	0	1	0	0	0
3	1	1	1	1	1	1	1	1	0	0	0
4	1	1	1	1	1	1	0	1	0	0	0
5	1	1	0	1	1	1	1	1	0	0	0
6	1	1	0	1	0	1	1	1	0	0	0
7	1	1	1	1	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

Chiusura transitiva

$A^3 = A^* =$

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0
2	1	1	1	1	1	1	1	1	0	0	0
3	1	1	1	1	1	1	1	1	0	0	0
4	1	1	1	1	1	1	1	1	0	0	0
5	1	1	1	1	1	1	1	1	0	0	0
6	1	1	1	1	1	1	1	1	0	0	0
7	1	1	1	1	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

Chiusura transitiva

A^* consente di verificare in tempo costante la presenza di un cammino in G tra due nodi

Calcolo efficiente di A^* a partire da A, esemplificato dal codice seguente.

```
1  A* = A;
2  DO {
3    B = A*;
4    A* = B × B;
5  } WHILE (A* != B);
6  RETURN A*;
```

Al più $\log(n-1) = \Theta(\log n)$ iterazioni: costo totale $\Theta(n^3 \log n)$ utilizzando l'algoritmo elementare dper la moltiplicazione di matrici

Visita in ampiezza di un grafo

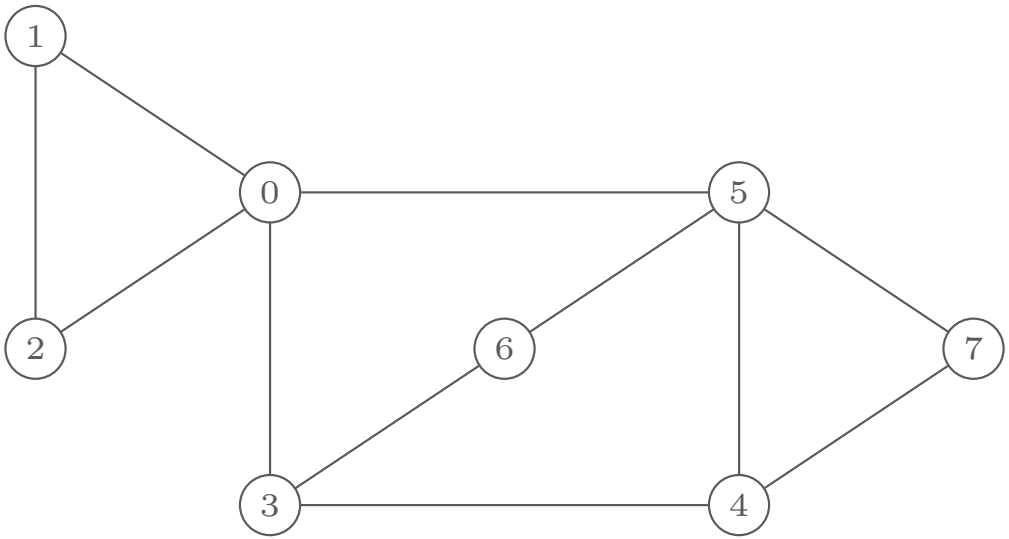
La visita in ampiezza o BFS (*Breadth-First Search*)

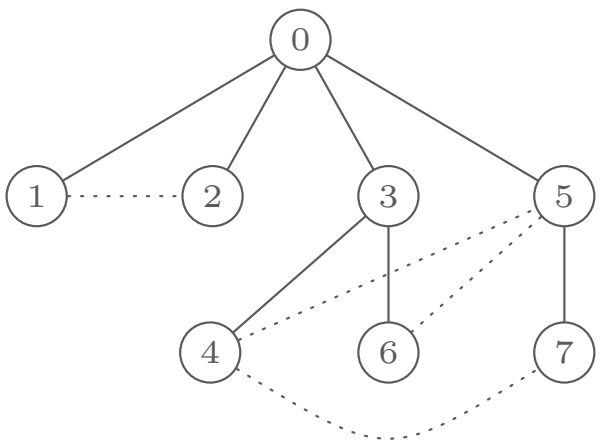
- esplora i nodi in ordine crescente di distanza da un nodo iniziale
- evita di esaminare ripetutamente gli stessi cammini

```
1  BreadthFirstSearch( s ):
2    FOR (u = 0 ; u < n; u = u + 1)
3      raggiunto[u] = FALSE;
4    Q.Enqueue( s );
5    WHILE (!Q.Empty( )) {
6      u = Q.Dequeue( );
7      IF (!raggiunto[u]) {
8        raggiunto[u] = TRUE;
9        FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
10         v = x.dato;
11         Q.Enqueue( v );
12       }
13     }
14   }
```

[alvie]

Visita in ampiezza di un grafo





Visita in ampiezza di un grafo

- Gli archi che conducono a vertici ancora non visitati, permettendone la scoperta, formano un albero detto **albero BFS**, la cui struttura dipende dall'ordine di visita
- invece di una coda `Q` di *vertici*, `Q` è una coda di *archi*
- l'albero BFS è uno **spanning tree** di G
- se G non è pesato, l'albero BFS è utile per rappresentare i cammini minimi dal vertice di partenza s verso tutti gli altri vertici
- la distanza minima di un vertice v da s nel grafo equivale alla profondità di v nell'albero BFS
- costo della BFS: $O(n + m)$

Visita in profondità di un grafo

Ottenuta sostituendo la coda `Q` della visita BFS con una pila `P`.

Analogamente alla visita in ampiezza, anche nella visita in profondità o DFS (*Depth First Search*) viene costruito un albero, detto **albero DFS**.

```
1  DepthFirstSearch( s ):
2    FOR (u = 0; u < n; u = u + 1)
3      raggiunto[u] = FALSE;
4    P.Push( s );
5    WHILE (!P.Empty( )) {
6      u = P.Pop( );
7      IF (!raggiunto[u]) {
8        raggiunto[u] = TRUE;
9        FOR (x = listaAdiacenza[u].fine; x != null; x = x.pred) {
10         v = x.dato;
11         P.Push( v );
12       }
13     }
14   }
```

[alvie]

Visita in profondità di un grafo

Versione ricorsiva della DFS

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

[alvie]

Costo della visita: $O(n+m)$

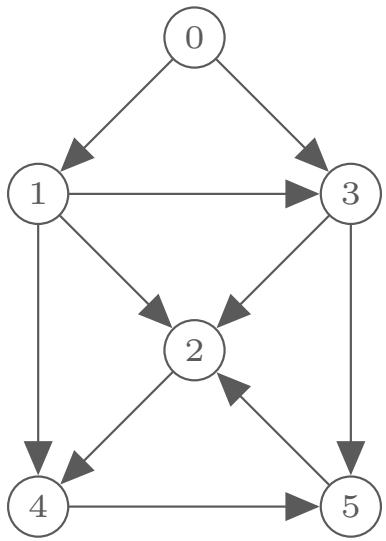
Visita in profondità di un grafo

Un arco (u, v) non appartenente all'albero DFS puèssere:

- **all'indietro** (*back*): se v è antenato di u nell'albero DFS;
- **in avanti** (*forward*): se v è discendente di u nell'albero DFS (nipote, pronipote e così via, ma non figlio);
- **trasversale** (*cross*): se v e u non sono uno antenato dell'altro.

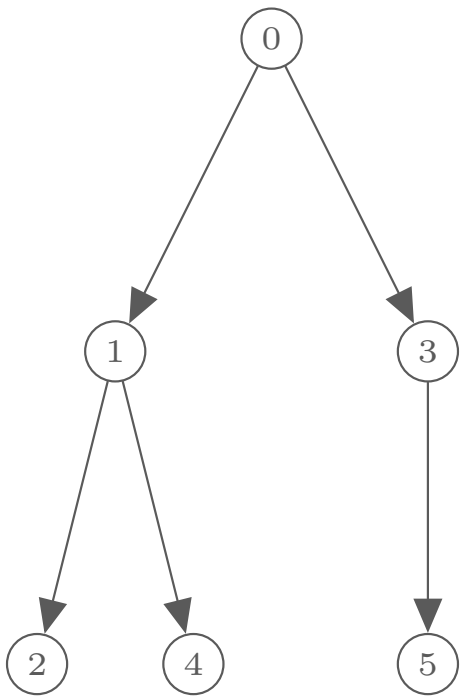
Nei grafi non orientati possono esserci solo archi back: sono gli unici a condurre a vertici già visitati

Visita in profondità di un grafo



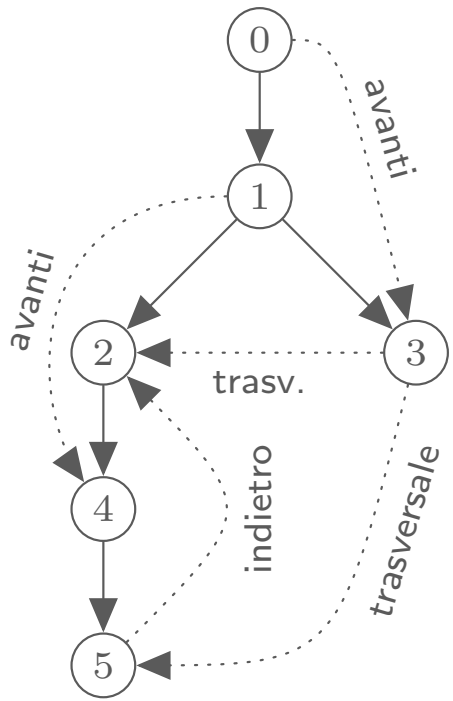
Visita in profondità di un grafo

Albero BFS del grafo precedente a partire da 0



Visita in profondità di un grafo

Albero DFS del grafo precedente a partire da 0



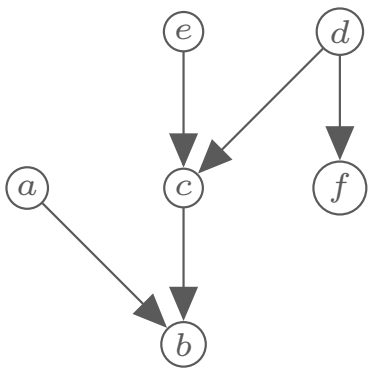
Grafi orientati aciclici

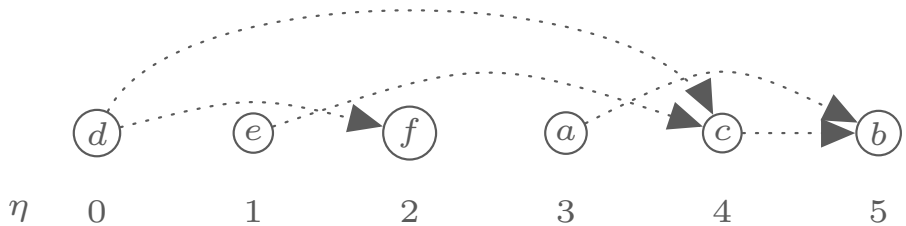
Un grafo G è **ciclico** se e solo se le visite BFS o DFS forniscono almeno un arco all'indietro

Un **grafo orientato aciciclo** è chiamato **DAG** (*Directed Acyclic Graph* e viene utilizzato in quei contesti in cui esiste una dipendenza tra oggetti espressa da una relazione d'ordine

Ordinamento topologico

Ordinamento topologico di un DAG $G = (V, E)$: numerazione $\eta : V \mapsto \{0, 1, \ldots, n-1\}$ dei suoi vertici tale che per ogni arco $(u, v) \in E$ vale $\eta(u) < \eta(v)$





Ordinamento topologico

```
1  OrdinamentoTopologico( ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    contatore = n - 1;
5    FOR (s = 0; s < n; s = s + 1) {
6      IF (!raggiunto[s]) DepthFirstSearchRicorsivaOrdina( s );
7    }
1  DepthFirstSearchRicorsivaOrdina( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsivaOrdina( v );
6    }
7    eta[u] = contatore;
8    contatore = contatore - 1;
```

[alvie]

Costo $O(n+m)$

Cammini minimi

- Grafo pesato $G = (V, E, W)$ (orientato o meno) con $W : E \mapsto \mathbb{R}$
- Si vogliono individuare i cammini di lunghezza minima tra i nodi di G
- Caratteristiche diverse in dipendenza del numero di cammini minimi da individuare
  - *all pairs shortest path*: gli $n(n-1)$ cammini minimi tra tutte le coppie di nodi
  - *single source shortest path*: gli $n-1$ cammini minimi da un nodo a tutti gli altri
- Caratteristiche diverse in dipendenza di W
  - pesi positivi
  - pesi qualunque

Cammini minimi in grafi con pesi positivi

Grafo $G = (V, E, W)$ con $W : E \mapsto \mathbb{R}^+$, nodo $s \in V$. Vi vuole

- derivare la distanza $\delta(s, v)$ da s a v, per ogni nodo $v \in V$
- ottenere, per ogni nodo, il cammino minimo: è sufficiente ottenere, per ogni nodo v, l'indicazione del nodo u che precede v nel cammino minimo da s a v

Cammini minimi in grafi con pesi positivi

Risolubile mediante algoritmo di Dijkstra:

- visita del grafo che fa uso di una coda con priorità per determinare l'ordine di visita dei nodi
- nella coda con priorità, coppie (v, p), con $v \in V$ e $p \in \mathbb{R}^+$, ordinate rispetto ai pesi p
- invariante: $p \geq \delta(s, v)$ per ogni coppia (v, p) nella coda con priorità
- quando (v, p) viene estratta dalla coda con priorità, $p = \delta(s, v)$

Ad ogni istante, il peso p associato al nodo v nella coda con priorità indica la lunghezza del cammino più breve da s a v trovato finora nel grafo

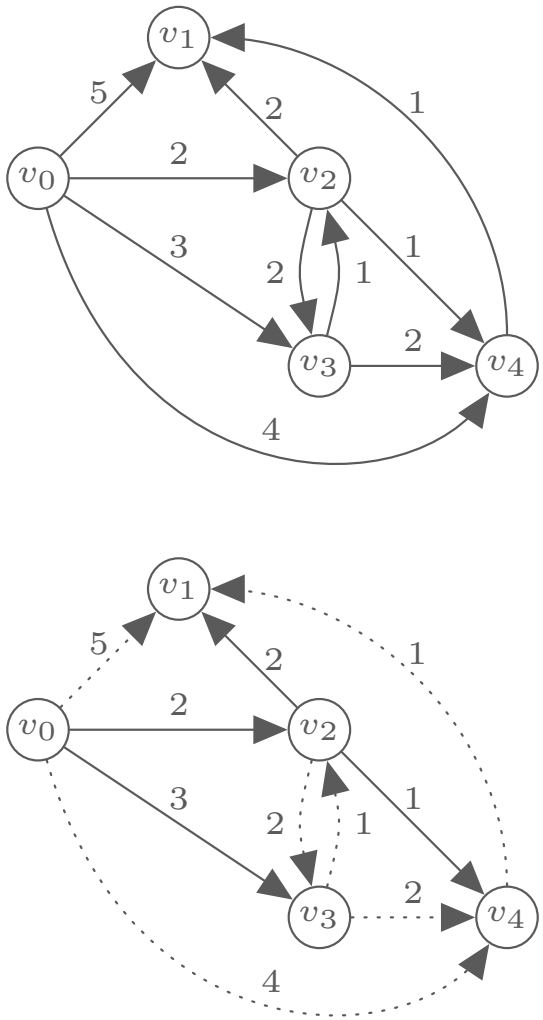
Il peso viene aggiornato ogni qual volta viene individuato un cammino più breve da s a v

Cammini minimi in grafi con pesi positivi

```
1  Dijkstra( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      pred[u] = -1;
4      dist[u] = +∞;
5    }
6    pred[s] = s;
7    dist[s] = 0;
8    FOR (u = 0; u < n; u = u + 1) {
9      elemento.peso = dist[u];
10     elemento.dato = u;
11     PQ.Enqueue( elemento );
12   }
13   WHILE (!PQ.Empty( )) {
14     e = PQ.Dequeue( );
15     v = e.dato;
16     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
17       u = x.dato;
18       IF (dist[u] > dist[v] + x.peso) {
19         dist[u] = dist[v] + x.peso;
20         pred[u] = v;
21         PQ.DecreaseKey( u, dist[u] );
22       }
23     }
24   }
```

[alvie]

Cammini minimi in grafi con pesi positivi



Cammini minimi in grafi con pesi positivi

Costo dell'algoritmo di Dijkstra $O(t_c + nt_e + mt_d)$, dipende dall'implementazione della coda con priorità

- t_c: costo di costruzione della coda
- t_e: costo di estrazione del minimo dalla coda
- t_d: costo dell'operazione `DecreaseKey`

Implementazione mediante heap: costo $O((n + m) \log n)$

Implementazione mediante lista non ordinata: costo $O(n^2 + m)$

Implementazione mediante heap di Fibonacci: costo $O(n \log n + m)$

Algoritmo di Bellman-Ford

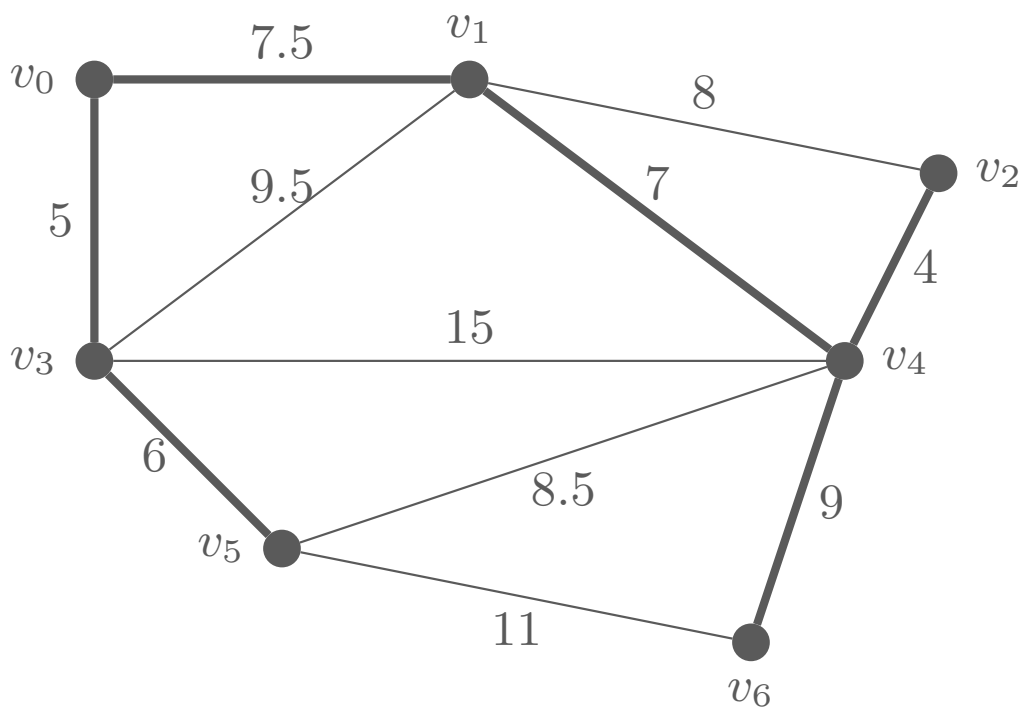
```
1  Bellman-Ford( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      pred[u] = -1;
4      dist[u] = +∞;
5    }
6    pred[s] = s;
7    dist[s] = 0;
8    FOR (i = 0; i < n; i = i + 1)
9      FOR (v = 0; v < n; v = v + 1) {
10       FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
11         u = x.dato;
12         IF (dist[u] > dist[v] + x.peso) {
13           dist[u] = dist[v] + x.peso;
14           pred[u] = v;
15         }
16       }
17     }
```

[alvie]

Minimo albero ricoprente

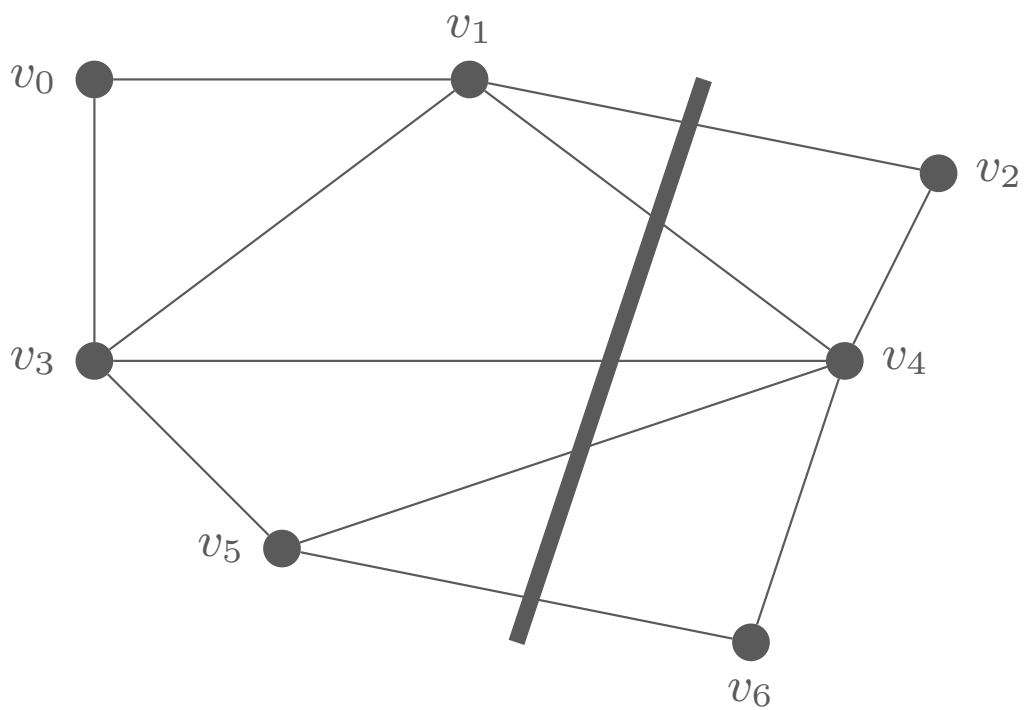
Dato un grafo non orientato e connesso $G = (V, E)$, un albero di ricoprimento di G è un albero T i cui archi sono anche archi del grafo e collegano tutti i nodi in V, ossia un albero $T = (V, E')$, dove $E' \subseteq E$.

Un albero di ricoprimento è minimo se la somma dei pesi nei suoi archi è la minima tra quelle di tutti i possibili alberi di ricoprimento



Minimo albero ricoprente

Dato un grafo $G = (V, E)$, un **taglio** (*cut*) su G è un sottoinsieme $C \subseteq E$ di archi la cui rimozione disconnette il grafo



Minimo albero ricoprente

Dato un grafo $G = (V, E, W)$ pesato sugli archi con pesi tutti distinti e un suo minimo albero ricoprente $T = (V, E')$, per ogni arco $e \in E$ abbiamo che:

Condizione di taglio $e \in E'$ se e solo se esiste un taglio in G che comprende e, per il quale e è l'arco di peso minimo.

Condizione di ciclo $e \notin E'$ se e solo se esiste un ciclo in G che comprende e, per il quale e è l'arco di peso massimo.

Algoritmo di Kruskal

Considera gli archi l'uno dopo l'altro, in ordine crescente di peso, valutando se inserire ogni arco nell'insieme E' degli archi dell'albero.
Nel considerare l'arco (u, v), possiamo avere due possibilità:

1. u e v già collegati in $G = (V, E')$, quindi l'arco (u, v) chiude un ciclo: in tal caso (u, v) è l'arco più pesante nel ciclo, e quindi non appartiene al minimo albero di ricoprimento;
2. u e v non sono già collegati in $G = (V, E')$, quindi esiste almeno un taglio che separa u da v: di tale taglio (u, v), essendo il primo arco considerato, e il più leggero, e quindi esso appartiene all'albero e va messo in E'

L'algoritmo di Kruskal opera a partire da una situazione in cui esistono n componenti connesse distinte (gli n nodi isolati), ognuna con un proprio minimo albero di ricoprimento (l'insieme vuoto degli archi)

Minimo albero ricoprente

Strutture di dati utilizzate:

1. coda con priorità `PQ` contenente l'insieme degli archi del grafo e i loro pesi;
2. struttura di dati che rappresenta una partizione dell'insieme dei nodi in modo tale da consentire di verificare se due nodi appartengono allo stesso sottoinsieme e da effettuare l'unione dei sottoinsiemi di appartenenza di due elementi;
3. array `set` che associa a ogni nodo del grafo un riferimento al corrispondente elemento nella struttura di dati precedente;
4. lista doppia `mst`, utilizzata per memorizzare gli archi nel minimo albero ricoprente, quando sono individuati dall'algoritmo.

Minimo albero ricoprente

```
1  Kruskal( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4        v = x.dato;
5        elemento.dato = <u, v>;
6        elemento.peso = x.peso;
7        PQ.Enqueue( elemento );
8      }
9      set[u] = NuovoNodo( );
10     Crea( set[u] );
11   }
12   WHILE (!PQ.Empty( )) {
13     elemento = PQ.Dequeue( );
14     <u,v> = elemento.dato;
15     IF (!Appartieni( set[u], set[v] )) {
16       Unisci( set[u], set[v] );
17       mst.InserisciFondo( <u,v> );
18     }
19   }
```

[alvie]

Costo: $O((m+n)\log n)$

Minimo albero ricoprente

Algoritmo di Jarník-Prim

- parte da un qualunque nodo s e fa crescere un minimo albero ricoprente a partire da tale nodo, aggiungendo man mano nuovi nodi e archi all'albero stesso
- se T indica la porzione di minimo albero ricoprente attualmente costruita, l'algoritmo sceglie l'arco (u, v) tale che esso è di peso minimo nel taglio tra $u \in T$ e $v \in V - T$
- v viene aggiunto a T e (u, v) all'insieme E'
- termina quando tutti i nodi sono in T

Ogni arco aggiunto a E' è il più leggero nel taglio che separa T da $V - T$ e quindi deve far parte del minimo albero ricoprente del grafo.

Al termine dell'algoritmo si ha che $\mid E' \mid = n - 1$, quindi tutti gli archi dell'albero compaiono in E'

Minimo albero ricoprente

Implementazione efficiente: punto critico = selezione efficiente dell'arco di peso minimo tra T a $V - T$

- Soluzione banale: ogni volta scansione di tutti gli m archi. Tempo di esecuzione $O(nm)$, peggiore dell'algoritmo di Kruskal
- Soluzione più efficiente: uso di coda con priorità `PQ` per mantenere l'insieme dei nodi in $V - T$. Peso di ogni nodo $v \in V - T$ è il peso dell'arco più leggero che collega v a un qualche nodo in T. Tempo di esecuzione dipende dall'implementazione di `PQ`: $O(n \log n + m)$ con heap di Fibonacci

Minimo albero ricoprente

```
1  Jarník-Prim( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      incluso[u] = FALSE;
4      pred[u] = u;
5      elemento.peso = peso[u] = +∞;
6      elemento.dato = u;
7      PQ.Enqueue( elemento );
8    }
9    WHILE (!PQ.Empty( )) {
10     elemento = PQ.Dequeue( );
11     v = elemento.dato;
12     incluso[v] = TRUE;
13     mst.InserisciFondo( <pred[v], v> );
14     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
15       u = x.dato;
16       IF (!incluso[u] && x.peso < peso[u]) {
17         pred[u] = v;
18         peso[u] = x.peso;
19         PQ.DecreaseKey( u, peso[u] );
20       }
21     }
22   }
```

[alvie]