# Lezione 17

Algoritmi su grafi

# Sommario

- Rappresentazione dei grafi
  - Visita in ampiezza
  - Visita in profondità
- Ordinamento topologico

## Grafi

- I grafi sono strutture dati molto diffuse in informatica
- Vengono utilizzati per rappresentare reti e organizzazioni dati complesse e articolate
- Per elaborare i grafi in genere è necessario visitarne in modo ordinato i vertici
- Vedremo a questo proposito due modi fondamentali di visita: per ampiezza e per profondità

## Nota sulla notazione asintotica

- Il tempo di esecuzione di un algoritmo su un grafo $G=(V,E)$ viene dato in funzione del numero di vertici $|V|$ e del numero di archi $|E|$
- Utilizzando la notazione asintotica adotteremo la convenzione di rappresentare $|V|$ con il simbolo $V$ e $|E|$ con $E$: quando diremo che il tempo di calcolo è $O(E+V)$ vorremo significare $O(|E|+|V|)$

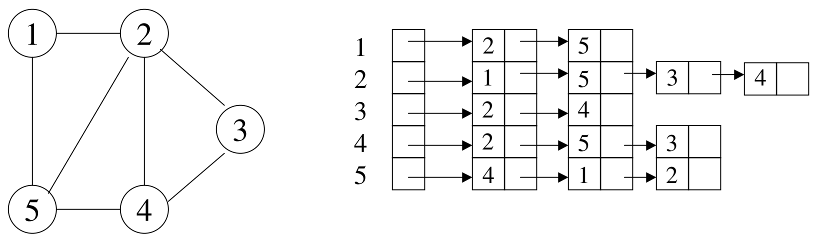
## Rappresentazione di un grafo

- Vi sono due modi per rappresentare un grafo:
  - collezione (array) di *liste di adiacenza*
  - *matrice di adiacenza*
- si preferisce la rappresentazione tramite liste di adiacenza quando il grafo è *sparso*, cioè con $|E|$ molto minore di $|V|^2$
- si preferisce la rappresentazione tramite matrice di adiacenza quando, al contrario, il grafo è *denso* o quando occorre alta efficienza nel rilevare se vi è un arco fra due vertici dati

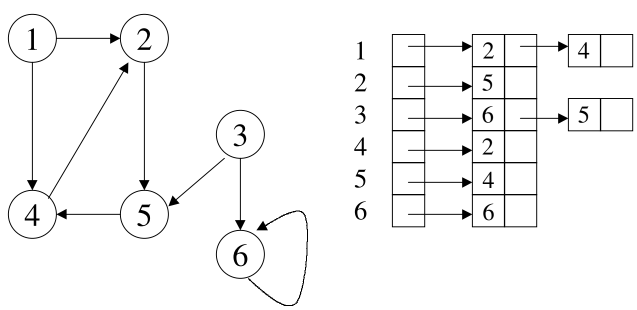
## Liste di adiacenza

- Si rappresenta un grafo G=(V,E) con un vettore Adj di liste, una lista per ogni vertice del grafo
- per ogni vertice u, Adj[u] contiene tutti i vertici v adiacenti a u, ovvero quei vertici v tali per cui esiste un arco $(u,v)\in E$
- in particolare questo insieme di vertici è memorizzato come una lista
- l'ordine dei vertici nella lista è arbitrario

## Visualizzazione: grafo non orientato con liste di adiacenza



## Visualizzazione: grafo orientato con liste di adiacenza



## Proprietà della rappresentazione con liste di adiacenza

- Se un grafo è orientato allora la somma delle lunghezze di tutte le liste di adiacenza è |E|
  - infatti per ogni arco (u,v) c'è un vertice v nella lista di posizione u
- Se un grafo non è orientato allora la somma delle lunghezze di tutte le liste di adiacenza è 2|E|
  - infatti per ogni arco (u,v) c'è un vertice v nella lista di posizione u e un vertice u nella lista di posizione v
- La quantità di memoria necessaria per memorizzare un grafo (orientato o non) è O(max(V,E)) = O(V+E)

## Grafi pesati

- In alcuni problemi si vuole poter associare una informazione (chiamata *peso*) ad ogni arco
- un grafo con archi con peso si dice *grafo pesato*
- si dice che esiste una funzione peso che associa ad un arco un valore

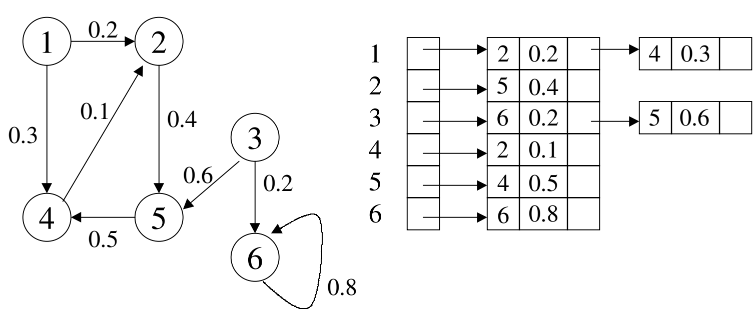
$$w : E \rightarrow \mathbf{R}$$

- ovvero un arco (u,v) ha peso w(u,v)

## Grafi pesati con liste di adiacenza

- Si memorizza il peso w(u,v) insieme al vertice v nella lista per il vertice u

## Visualizzazione: grafo orientato pesato con liste di adiacenza



## Svantaggi della rappresentazione con liste di adiacenza

- Per sapere se un arco (u,v) è presente nel grafo si deve scandire la lista degli archi di u

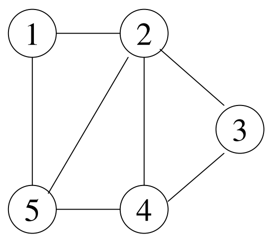
## Matrici di adiacenza

- Per la rappresentazione con matrici di adiacenza si assume che i vertici siano numerati in sequenza da 1 a |V|
- Si rappresenta un grafo G=(V,E) con una matrice $A=(a_{ij})$ di dimensione |V|x|V| tale che:

  $a_{ij}=1$ se $(i,j) \in E$

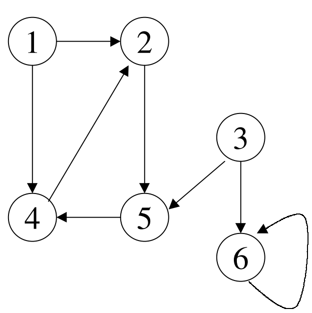
  $a_{ij}=0$ altrimenti

## Visualizzazione: grafo non orientato con matrice di adiacenza




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

## Visualizzazione: grafo orientato

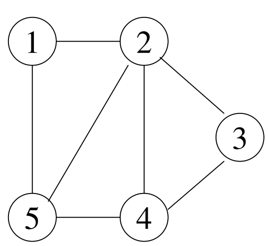



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

## Proprietà della rappresentazione con matrice di adiacenza

- La rappresentazione di un grafo G=(V,E) con matrice di adiacenza richiede memoria $\Theta(V^2)$ indipendentemente dal numero di archi
- La matrice di adiacenza di un grafo non orientato è simmetrica ovvero $a_{ij}= a_{ji}$
- Per un grafo non orientato si può allora memorizzare solo i dati sopra la diagonale (diagonale inclusa), riducendo della metà lo spazio per memorizzare la matrice

## Visualizzazione: grafo non orientato con matrice di adiacenza

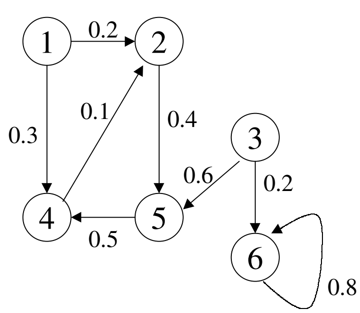



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | | 0 | 1 | 1 | 1 |
| 3 | | | 0 | 1 | 0 |
| 4 | | | | 0 | 1 |
| 5 | | | | | 0 |

## Grafi pesati con matrici di adiacenza

- Si memorizza il peso nell'elemento $a_{ij}$ invece di 1
- se l'arco non esiste si indica con 0 o ∞ o NIL a secondo del problema

## Visualizzazione: grafo orientato pesato con matrice di adiacenza




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | .2 | 0 | .3 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | .4 | 0 |
| 3 | 0 | 0 | 0 | 0 | .6 | .2 |
| 4 | 0 | .1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | .5 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | .8 |

## Vantaggi della rappresentazione con matrice di adiacenza

- la rappresentazione con matrice di adiacenza è semplice
- se il grafo è piccolo non vi è sostanziale differenza di efficienza con la rappresentazione con liste di adiacenza
- per grafi non pesati si può rappresentare ogni singolo elemento della matrice non con una parola ma con un singolo bit

## Visita in ampiezza

- La visita in ampiezza *breadth-first-search (BFS)* di un grafo dato un vertice sorgente s consiste nella esplorazione sistematica di tutti i vertici raggiungibili da s in modo tale da esplorare tutti i vertici che hanno distanza k prima di iniziare a scoprire quelli che hanno distanza k+1

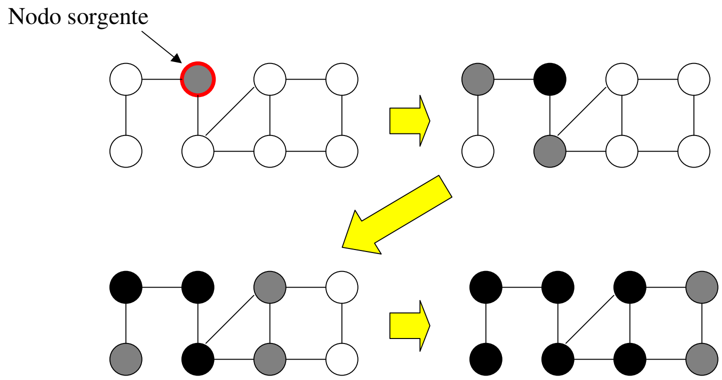
## Visita in ampiezza

- inoltre la procedura di visita in ampiezza che vedremo:
  - calcola la distanza da s ad ognuno dei vertici raggiungibili
  - produce un albero BFS che ha s come radice e che comprende tutti i vertici raggiungibili da s

## Idea intuitiva

- L'idea è quella di tenere traccia dello stato (già scoperto, appena scoperto, ancora da scoprire) di ogni vertice, "colorandolo" di un colore diverso
- i colori possibili sono:
  - bianco: vertice ancora non scoperto
  - grigio: vertice appena scoperto ed appartenente alla frontiera
  - nero: vertice per cui si è terminata la visita
- un vertice da bianco diventa grigio e poi nero
- se $(u,v) \in E$ ed u è un vertice nero, allora il vertice v è grigio, ovvero tutti i vertici adiacenti ad un vertice nero sono già stati scoperti

# Visualizzazione




# Idea intuitiva

- La visita in ampiezza costruisce un albero BFT
- Si crea un grafo T(V,E) vuoto per memorizzare il BFT
- la radice è il nodo sorgente s
- quando un vertice bianco v viene scoperto durante la scansione della lista di adiacenza di un vertice già scoperto u allora si aggiunge all'albero T il vertice v e l'arco (u,v)
- si dice che u è padre di v
- poiché un vertice viene scoperto al massimo una volta ha al massimo un padre (e quindi il grafo risultante sarà un albero)

## Strutture ausiliarie

- La procedura di visita in ampiezza assume che il grafo G=(V,E) sia rappresentato usando liste di adiacenza
- ad ogni vertice u sono associati, oltre ai vertici adiacenti, l'attributo
  - colore: **color[u]**
  - padre: **π[u]**
  - la distanza dalla sorgente s: **d[u]**
- L'algoritmo fa anche uso di una coda Q per gestire l'insieme dei vertici grigi

## Pseudocodice

```
BFS(G,s)
for ogni vertice u ∈ V[G]-{s}
do   color[u] ← WHITE
     d[u] ← ∞
     π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅
do   u ← head[Q]
     for ogni v ∈ Adj[u]
     do      if color[v]=WHITE
             then    color[v] ← GRAY
                     d[v] ← d[u]+1
                     π[v] ← u
                     Enqueue(Q,v)
     Dequeue(Q)
     color[u] ← BLACK
```

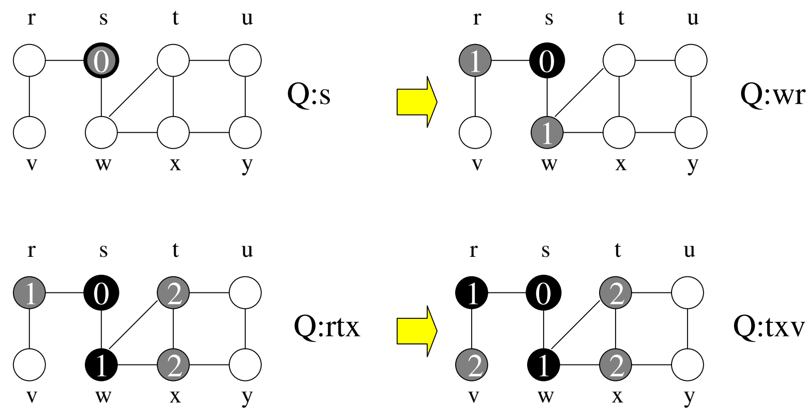
## Spiegazione del codice

- Le linee 1-4 eseguono l'inizializzazione:
  - tutti i vertici sono colorati di bianco
  - la distanza di tutti i vertici è non nota e posta a $\infty$
  - il padre di ogni vertice inizializzato a nil
- la linea 5 inizializza la sorgente a cui:
  - viene assegnato il colore grigio
  - viene assegnata distanza 0
  - viene assegnato padre nullo nil
- la linea 8 inizializza la coda Q con il vertice sorgente s

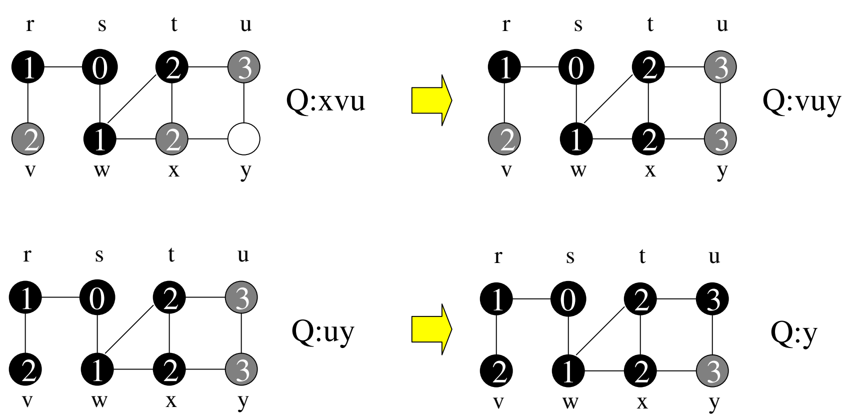
## Spiegazione del codice

- Il ciclo principale è contenuto nelle linee 9-18
- il ciclo continua fino a quando vi sono vertici grigi in Q, ovvero vertici già scoperti le cui liste di adiacenza non siano state ancora completamente esaminate
- la linea 10 preleva l'elemento in testa alla coda
- nelle linee 11-16 si esaminano tutti i vertici v adiacenti a u
- se v non è ancora stato scoperto lo si scopre
  - si colora di grigio
  - si aggiorna la sua distanza alla distanza di u +1
  - si memorizza u come suo predecesore
  - si pone in fondo alla coda
- quando tutti i vertici adiacenti a u sono stati scoperti allora si colora u di nero e lo si rimuove da Q

## Visualizzazione




## Visualizzazione



## Analisi

- Il tempo per l'inizializzazione è O(V)
- Dopo l'inizializzazione nessun vertice sarà mai colorato più di bianco
- quindi il test in 12 assicura che ogni vertice sarà inserito nella coda Q al più una volta
- le operazioni di inserimento ed eliminazione dalla coda richiedono un tempo O(1)
- il tempo dedicato alla coda nel ciclo 9-18 sarà pertanto un O(V)

## Analisi

- poiché la lista di adiacenza è scandita solo quando si estrae il vertice dalla coda allora la si scandisce solo 1 volta per vertice
- poiché il numero di archi è pari a |E| allora la somma delle lunghezze di tutte le liste è $\Theta$(E)
- allora il tempo speso per la scansione delle liste complessivamente è O(E)
- in totale si ha un tempo di O(V+E)
- quindi la procedura di visita in ampiezza richiede un tempo lineare nella rappresentazione con liste di adiacenza

## Alberi BFS

- La procedura BFS costruisce un albero BFS durante la visita del grafo
- l'informazione sull'albero è contenuta nei puntatori al padre $\pi$
- formalmente, dato G=(V,E) con sorgente s si definisce il *sottografo dei predecessori* di G come $G_\pi=(V_\pi,E_\pi)$ dove:

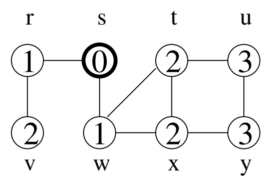
  $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$

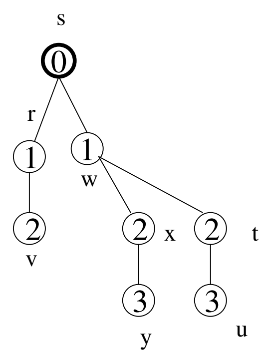
  $E=\{(\pi[v],v) \in E : v \in V_\pi - \{s\}\}$

## Alberi BFS

- $G_\pi$ è un albero BFS se
  - $V_\pi$ contiene tutti e soli i vertici raggiungibili da s
  - e se per ogni $v \in V_\pi$ vi è un unico cammino semplice da s a v in $G_\pi$ che è anche un cammino minimo da s a v in G.
- Un albero BFS è effettivamente un albero perché è connesso e $|E_\pi|=|V_\pi|-1$
- si dimostra che dopo aver eseguito la procedura BFS a partire da una sorgente s, il sottografo dei predecessori è effettivamente un albero BFS

## Visualizzazione dell’albero BFS




## Visita in profondità

- La visita in profondità *depth-first-search (DFS)* di un grafo consiste nella esplorazione sistematica di tutti i vertici andando in ogni istante il più possibile in profondità
- gli archi vengono esplorati a partire dall’ultimo vertice scoperto v che abbia ancora archi non esplorati uscenti
- quando questi sono finiti si torna indietro per esplorare gli altri archi uscenti dal vertice dal quale v era stato scoperto

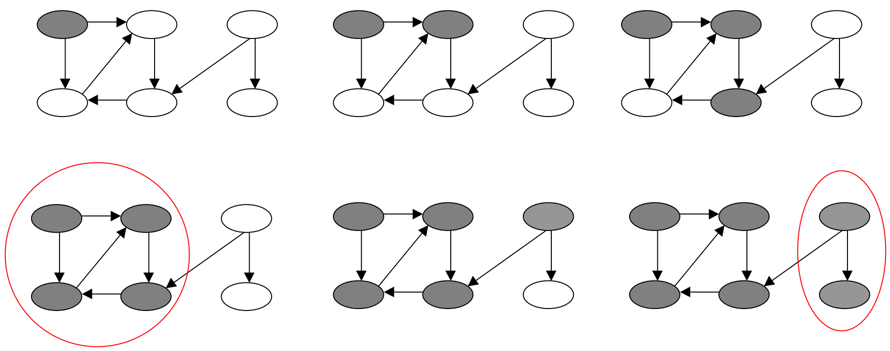
## Visita in profondità

- Il procedimento continua fino a quando non vengono scoperti tutti i vertici raggiungibili dal vertice sorgente originario
- se al termine rimane qualche vertice non scoperto uno di questi diventa una nuova sorgente e si ripete la ricerca a partire da esso
- questo fino a scoprire tutti i vertici

## Visita in profondità

- A differenza che nella visita per ampiezza il cui sottografo dei predecessori formava un albero, nel caso della visita in profondità si forma una foresta di diversi alberi DFS
- infatti si hanno più sorgenti (radici)

# Visualizzazione



# Idea intuitiva

- Come per la visita in ampiezza i vertici vengono colorati per tenere conto dello stato di visita:
  - ogni vertice è inizialmente bianco
  - è grigio quando viene scoperto
  - viene reso nero quando la visita è finita, cioè quando la sua lista di adiacenza è stata completamente esaminata

## Marcatura temporale

- Oltre al colore si associa ad ogni vertice v due informazioni temporali:
  - tempo di inizio visita d[v], cioè quando è reso grigio per la prima volta
  - tempo di fine visita f[v], cioè quando è reso nero
- il valore temporale è dato dall’ordine assoluto con cui si colorano i vari vertici del grafo
- si usa per questo una variabile globale *tempo* che viene incrementata di uno ogni volta che si esegue un inizio di visita o una fine visita

## Marcatura temporale

- il tempo è un intero compreso fra 1 e 2|V| poiché ogni vertice può essere scoperto una sola volta e la sua visita può finire una sola volta

- per ogni vertice u si ha sempre che d[u]<f[u]
- ogni vertice u è
  - WHITE prima di d[u]
  - GRAY fra d[u] e f[u]
  - BLACK dopo f[u]

## Utilità della marcatura temporale

- La marcatura temporale è usata in molti algoritmi sui grafi
- E' utile in generale per ragionare sul comportamento della visita in profondità

## Pseudocodice

```
DFS(G)
for ogni vertice u ∈ V[G]
do   color[u] ← WHITE
     π[u] ← NIL
time ← 0
for ogni vertice u ∈ V[G]
     do      if color[u]=WHITE
             then    DFS-Visit(u)

DFS-Visit(u)
color[u] ← GRAY
d[u] ← time ← time +1
for ogni v ∈ Adj[u]
do   if color[v]=WHITE
     then    π[v] ← u
             DFS-Visit(v)
color[u] ← BLACK
f[u] ← time ← time +1
```

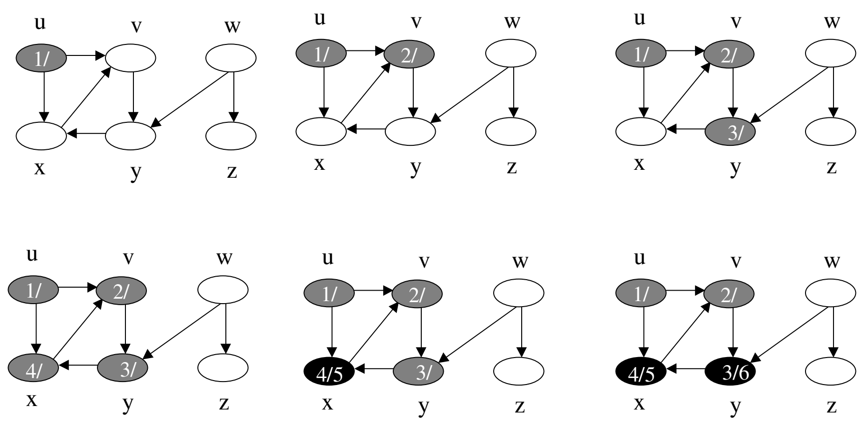
## Spiegazione dello pseudocodice

- Le righe 1-4 della procedura DFS eseguono la fase di inizializzazione colorando ogni vertice del grafo di bianco, settando il padre a NIL e impostandola variabile globale time a 0
- il ciclo 5-7 esegue la procedura DFS-Visit su ogni nodo non ancora scoperto del grafo, creando un albero DFS ogni volta che viene invocata la procedura

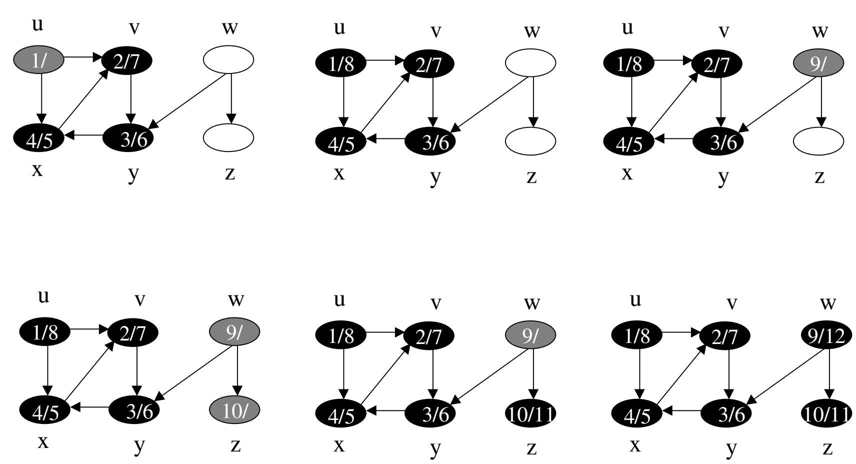
## Spiegazione dello pseudocodice

- In ogni chiamata DFS-Visit(u) il vertice u è inizialmente bianco
- viene reso grigio e viene marcato il suo tempo di inizio visita in d[u], dopo aver incrementato il contatore temporale globale time
- vengono poi esaminati tutti gli archi uscenti da u e viene invocata ricorsivamente la procedura nel caso in cui i vertici collegati non siano ancora stati esplorati
- in questo caso il loro padre viene inizializzato ad u
- dopo aver visitato tutti gli archi uscenti u viene colorato BLACK e viene registrato il tempo di fine visita in f[u]

# Visualizzazione




# Visualizzazione



## Analisi del tempo di calcolo

- Il ciclo di inizializzazione di DFS e il ciclo for 5-7 richiedono entrambi tempo $\Theta(V)$
- la procedura DFS-Visit viene chiamata solo una volta per ogni vertice (poiché viene chiamata quando il vertice è bianco e lo colora immediatamente di grigio)
- in DFS-Visit il ciclo for 3-6 viene eseguito $|Adj[v]|$ volte, e dato che la somma della lunghezza di tutte le liste di adiacenza è $\Theta(E)$, il costo è $\Theta(E)$
- il tempo totale è quindi un $\Theta(V+E)$

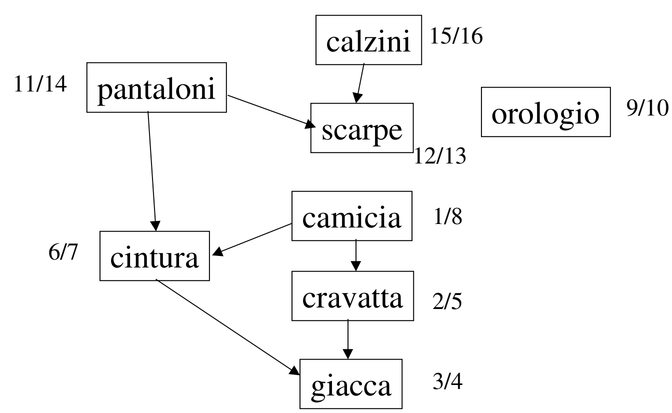
## Ordinamento topologico

- L'ordinamento topologico è un ordinamento definito su i vertici di un grafo orientato aciclico (*directed acyclic graph DAG*)
- si può pensare all'ordinamento topologico come ad un modo per ordinare i vertici di un DAG lungo una linea orizzontale in modo che tutti gli archi orientati vadano da sinistra verso destra

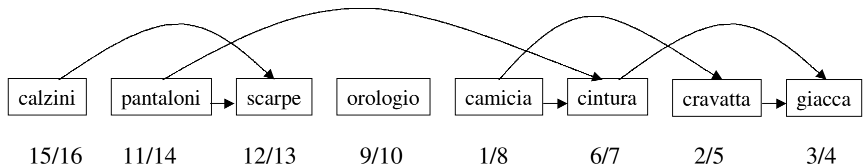
# Ordinamento topologico

- I grafi aciclici diretti sono utilizzati per modellare precedenze fra eventi
- consideriamo ad esempio le precedenze nelle operazioni del vestirsi utilizzando un DAG i cui nodi siano indumenti
- certi indumenti vanno messi prima di altri (i calzini prima delle scarpe)
- mentre altri indumenti possono essere indossati in qualsiasi ordine (calzini e pantaloni)
- un arco orientato (u,v) indica che l'indumento u deve essere indossato prima dell'indumento v
- l'ordinamento topologico del DAG fornirà dunque un ordine per vestirsi

# Visualizzazione



# Visualizzazione




# Pseudocodice

```
Topological-Sort(G)
1  chiama DFS(G) per calcolare f[v] per ogni v
2  appena la visita di un vertice è finita inseriscilo in
   testa ad una lista
3  return la lista concatenata dei vertici
```

## Analisi

- Si esegue un ordinamento topologico in tempo O(V+E) dato che:
  - la visita DFS richiede un tempo O(V+E)
  - l’inserimento di ognuno dei |V| vertici richiede ciascuno un tempo O(1)