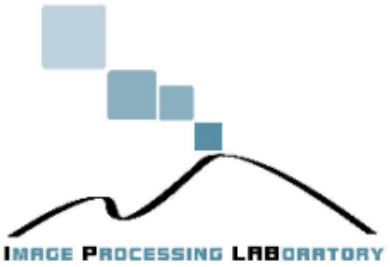
Grafi

Alessandro Ortis
Image Processing Lab - iplab.dmi.unict.it

ortis@dmi.unict.it
www.dmi.unict.it/ortis/



Grafi

- Si dice grafo un insieme di nodi legati "a due a due" da archi direzionati (o no)
- I grafi sono strutture dati di fondamentale importanza in informatica
- Vi sono centinaia di problemi computazionali ad essi legati
- Parleremo solo di alcuni algoritmi elementari sui grafi

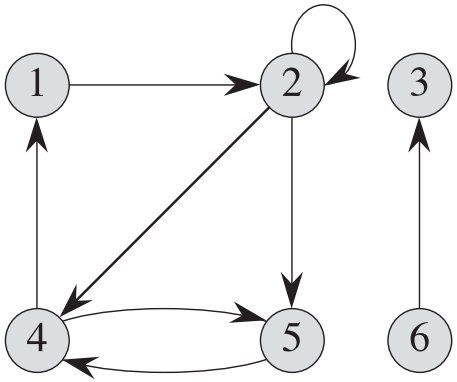
G=(V,E)

• V insieme dei nodi

• E insieme degli archi (u,v)

• Se G è direzionato l'arco (u,v) è ***uscente*** da u ed ***entrante*** in v

• Se (u,v) è in E, v è ***adiacente*** a u

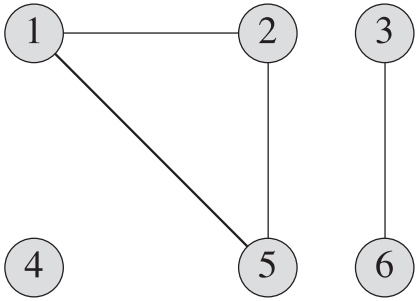


Grafi non direzionati

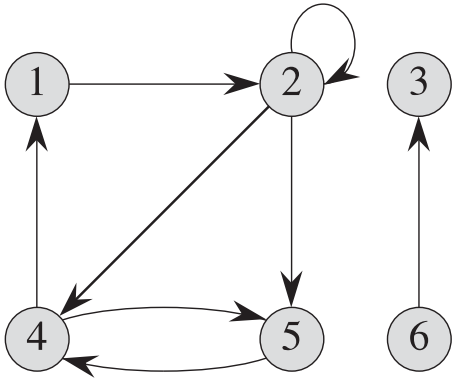
G=(V,E)

- V insieme dei nodi
- E insieme degli archi

- E consiste di coppie non ordinate di nodi
- Self-loops non ammessi
- In (u,v) u e v sono sia entranti che uscenti
- Adiacenza è simmetrica



Grafi



Grado di un nodo (caso non direzionato)

- Numero di archi entranti

Grado di un nodo (caso direzionato)

- Numero di archi entranti + numero di archi uscenti

Cammino (di lunghezza k) da u a v

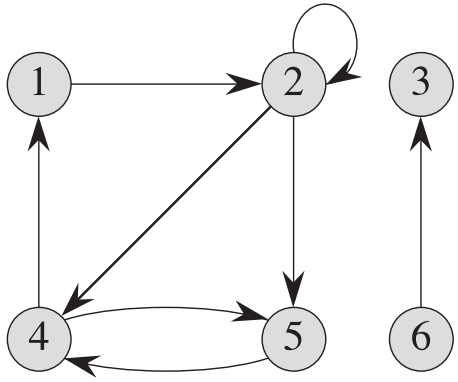
•Sequenza $v_0, \ldots, v_k$ tale che $u=v_0$ e $v=v_k$

Il cammino **contiene** i vertici $v_0,\ldots,v_k$ e gli archi $(v_0, v_1),\ldots, (v_{k-1}, v_k)$

- Un nodo v è **raggiungibile** da u se esiste un cammino da u a v

- Il cammino è **semplice** se tutti i vertici in esso contenuti sono distinti

Grafi



Cammino (di lunghezza k) da u a v

•Sequenza $v_0, \ldots, v_k$ tale che $u=v_0$ e $v=v_k$

Sottocammino: Sequenza di vertici di un cammino es: $v_i,\ldots,v_j$ per $0 \leq i \leq j \leq k$

Ciclo: Cammino $v_0, \ldots, v_k$ in cui $v_0=v_k$

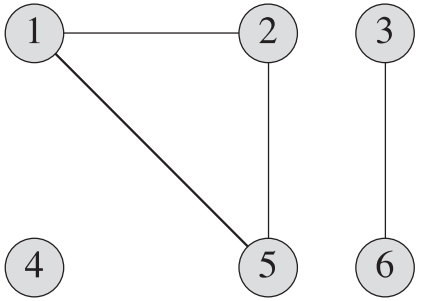
•Il ciclo è semplice se tutti i suoi nodi sono distinti.

Un grafo senza cicli è detto **aciclico**.

Grafi

- Grafo (non direzionato) **connesso**: ogni coppia di vertici è unita da un cammino.

- Componenti connesse: classi di equivalenza determinate dalla relazione “è raggiungibile da”



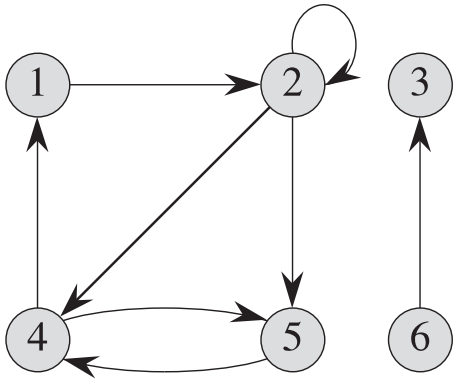
Componenti connesse: {1,2,3}, {3,6}, {4}

Un grafo non direzionato è connesso se ha 1 componente connessa

Grafi

- Grafo (direzionato) **fortemente connesso**: per ogni coppia di vertici (u,v) esiste un cammino che unisce u a v e v a u.

- Componenti **fortemente** connesse: classi di equivalenza determinate dalla relazione “sono mutualmente raggiungibili”



Componenti fortemente connesse:
{1,2,4,5}, {3}, {6}

Grafi

- G’=(V’,E’) **sottografo** di G=(V,E) se V’ sottoinsieme di V e E’ sottoinsieme di E

- Un grafo (non direzionato) è completo se ogni coppia di vertici è adiacente

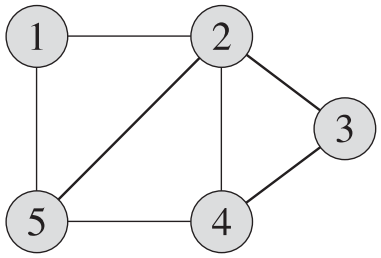
Rappresentare un grafo

Due modi fondamentali:

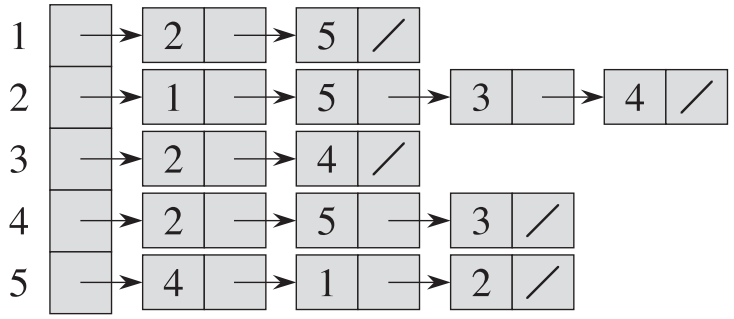
- Liste di adiacenza
  - Utile soprattutto per rappresentare grafi sparsi (con pochi archi)
  - Richiede $O(\max(|V|,|E|))=O(|V|+|E|)$ spazio
- Matrici di adiacenza
  - Richiede $O(|V|^2)$ spazio

Liste di adiacenza – Grafi non direzionati

- Array di |V| liste (una per ogni vertice)
- Adj[u] contiene (puntatori a) tutti i vertici v per i quali (u,v) è in E
- La somma delle lunghezze di tutte le liste è 2|E|



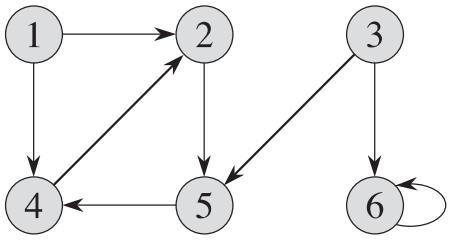
(a)



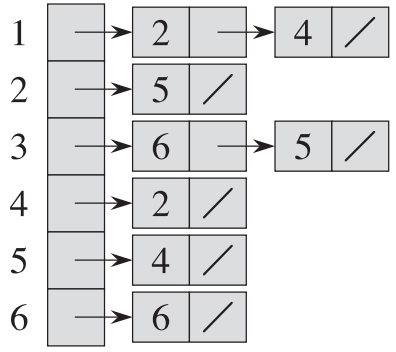
(b)

Liste di adiacenza – Grafi direzionati

- Array di |V| liste (una per ogni vertice)
- Adj[u] contiene (puntatori a) tutti i vertici v per i quali (u,v) è in E
- In tal caso, la somma delle lunghezze di tutte le liste è |E|



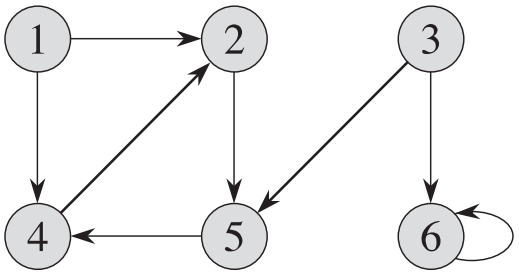
(a)



(b)

Matrici di adiacenza

- $A=[a_{ij}]$
- $a_{ij}=1$ se (i,j) è un arco in E (0 altrimenti)



	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

Ricerca in ampiezza (Breadth-First-Search)

- Dato un vertice s, “esploriamo” il grafo per scoprire ogni vertice v raggiungibile da s.
  - Calcola la distanza di ogni v da s.
  - L’algoritmo (implicitamente) produce un breadth-first-tree (BFT)
    - Il campo predecessore fa riferimento proprio a tale albero.
  - Il cammino da s a v in BFT rappresenta il cammino più breve.
- Supporremo una rappresentazione tramite liste di adiacenza.

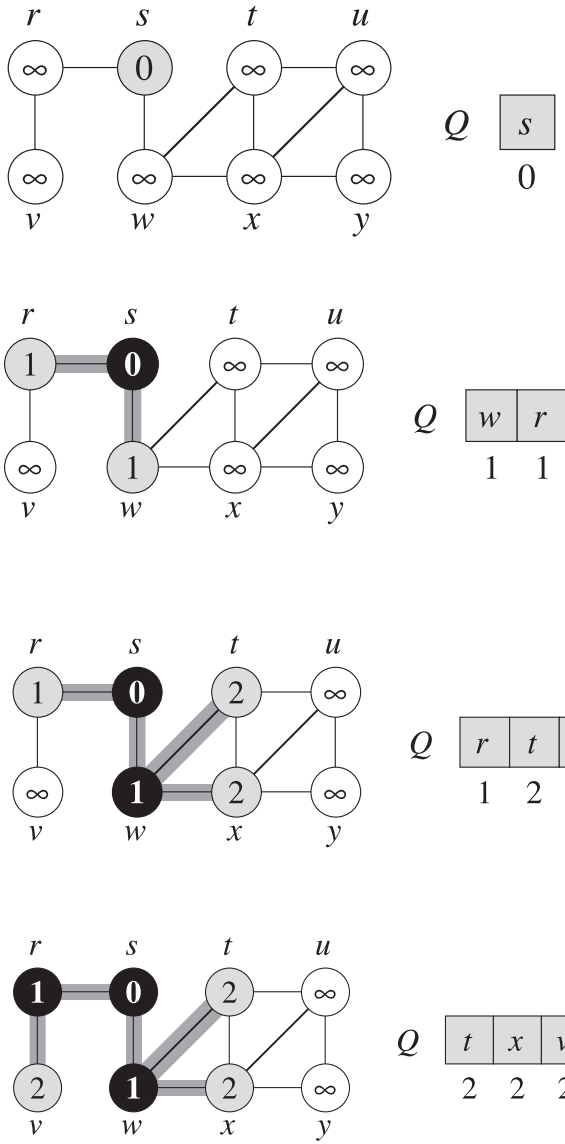
Ricerca in ampiezza -- Idee

- Inizialmente ogni nodo è colorato bianco
  - Poi i nodi diventeranno grigi o neri.
- Un nodo è **scoperto** quando è visitato la prima volta.
  - Diventa non-bianco
  - Nodi grigi: possono essere adiacenti (anche) a nodi bianchi.
    - Rappresentano la frontiera tra ciò che è già stato scoperto e ciò che non lo è ancora.
  - Nodi neri: possono essere adiacenti solo a nodi non bianchi.

Ricerca in ampiezza

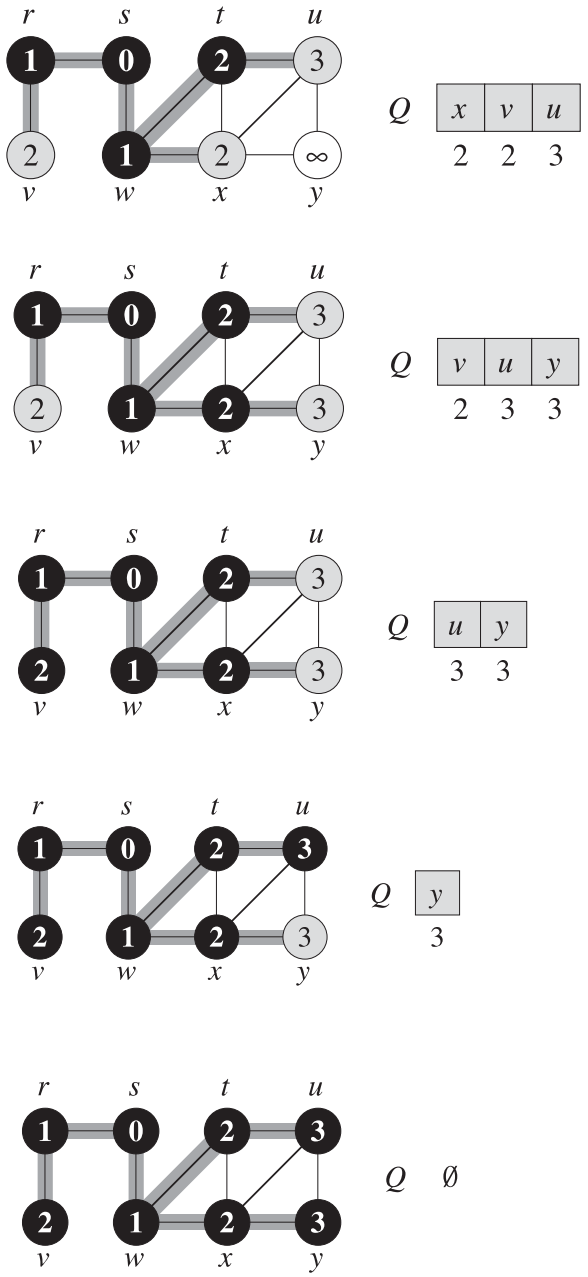
```
BFS(G,s)
1. for each vertex u in V[G] – {s}
2.     color[u]=white;
3.       d[u]=MAX;
4.       pred [u]=NULL;
5. color[s]=gray;
6. d[s]=0; pred[s]=NULL;
7. Q.Enqueue(s);
8. while (Q.NotEmpty())
9.     u=Q.Dequeue();
10.    for each v in Adj[u]
11.        if (color[v]==white)
12.            color[v]=gray;
13.            d[v]=d[u] + 1; pred[v]=u;
14.            Q.Enqueue(v);
15.    color[u]= black;
```

Invariante di ciclo: la coda Q è formata dall'insieme dei vertici grigi.



Ricerca in ampiezza

```
BFS(G,s)
1. for each vertex u in V[G] – {s}
2.     color[u]=white;
3.       d[u]=MAX;
4.       pred [u]=NULL;
5. color[s]=gray;
6. d[s]=0; pred[s]=NULL;
7. Q.Enqueue(s);
8. while (Q.NotEmpty())
9.     u=Q.Dequeue();
10.    for each v in Adj[u]
11.        if (color[v]==white)
12.            color[v]=gray;
13.            d[v]=d[u] + 1; pred[v]=u;
14.            Q.Enqueue(v);
15.    color[u]= black;
```



Ricerca in ampiezza

```
BFS(G,s)
1. for each vertex u in V[G] – {s}
2.     color[u]=white;
3.       d[u]=MAX;
4.       pred[u]=NULL;
5. color[s]=gray;
6. d[s]=0; pred[s]=NULL;
7. Q.Enqueue(s);
8. while (Q.NotEmpty())
9.     u=Q.Dequeue();
10.    for each v in Adj[u]
11.        if (color[v]==white)
12.            color[v]=gray;
13.            d[v]=d[u] + 1; pred[v]=u;
14.            Q.Enqueue(v);
15.    color[u]= black;
```

Complessità: O(n+m)
n: numero di nodi
m: numero di archi

Breadth-first Trees

- La procedura BFS(G,s) costruisce un albero (grafo dei predecessori G_p)
  - Ad ogni nodo è associato un predecessore
- V_p={v in V : p[v]≠NULL}
- E_p={(p[v],v) in E : v in V_p, v≠s}
- G_p è un albero in cui
  - C'è un unico cammino da s a v (in V_p) che è anche il cammino più breve
  - Gli archi in E_p sono chiamati **tree-edges**.

Breadth-first Trees

$V_p=\{v \text{ in } V : p[v]\neq NULL\}$ $E_p=\{(p[v],v) \text{ in } E : v \text{ in } V_p, v\neq s\}$

Print-Path

Supponiamo di aver già eseguito BFS(G,s), la seguente procedura stampa i vertici di un cammino minimo da *s* a *v*.

```
Print-Path(G,s,v)
1.if (v==s) print s
2.else if pred[v]==NULL
3.        print “No path from s to v”
4.else    Print-Path(G,s,pred[v])
5.        print v
```

Qual è la complessità di questa procedura?

Print-Path

Supponiamo di aver già eseguito BFS(G,s), la seguente procedura stampa i vertici di un cammino minimo da *s* a *v*.

```
Print-Path(G,s,v)
1.if (v==s) print s
2.else if pred[v]==NULL
3.        print “No path from s to v”
4.else    Print-Path(G,s,pred[v])
5.        print v
```

Qual è la complessità di questa procedura?

O(|V|) → ogni chiamata ricorsiva riguarda un cammino che ha un vertice in meno.

Ricerca in Profondità: DFS

- Il grafo viene visitato in profondità piuttosto che in ampiezza
- Gli archi sono esplorati a partire dal nodo v che
  - Sia stato scoperto più di recente
  - Abbia ancora archi (uscenti) non esplorati
- Quando gli archi uscenti di v terminano, si fa backtracking
  - Si esplorano eventuali altri archi uscenti dal nodo precedente a v.
- Il processo è ripetuto fin quando vi sono nodi da esplorare.

Depth first forests

- Se v è scoperto scorrendo la lista di adiacenza di u, p[v]=u
- Come per BFS si definisce un grafo dei predecessori G_p
- $V_p=V$
- $E_p=\{(p[v],v) \text{ in } E : v \text{ in } V, p[v]\neq NULL\}$
- G_p **non è** un albero (ma una foresta)
  - Depth first forest

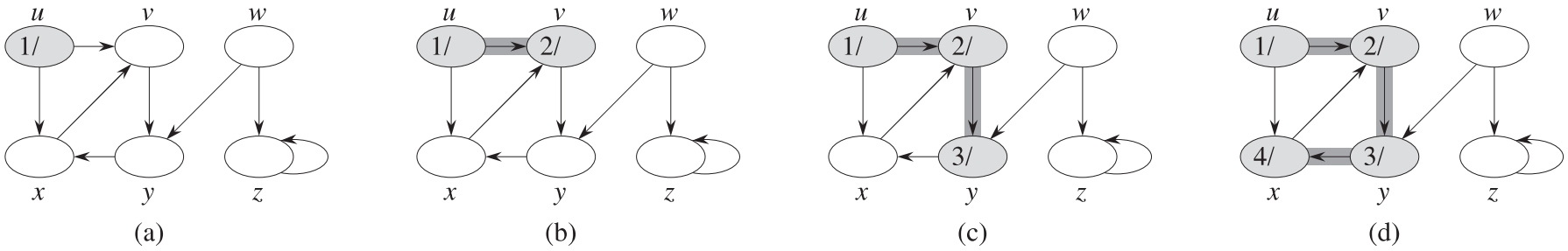
Timestamps

- DFS marca temporalmente ogni vertice visitato
  - Ogni v ha due etichette
  - La prima -- d[v] -- registra quando il nodo è stato scoperto (bianco-> grigio)
  - La seconda – f[v] – registra quando la ricerca finisce di esaminare la lista di adiacenza di v (grigio-> nero)
  - Per ogni v, d[v]<f[v]

DFS

```
DFS(G)
1.    for each u in V[G]
2.          color[u]=white:
3.          pred[u]=NULL;
4.    time = 0
5.    for each u in V[G]
6.        if (color[u]==white)
7.              DFS-Visit(u)
```

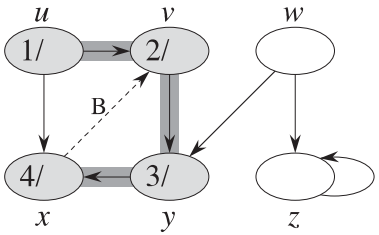
```
DFS-Visit(u)
1.color[u]=grey;   d[u]=time++;
2.for each v in Adj[u]
3.    if (color[v]==white)
4.          pred[v]=u;
5.          DFS-Visit(v);
6.color[u]=black
7.f[u]=time++;
```



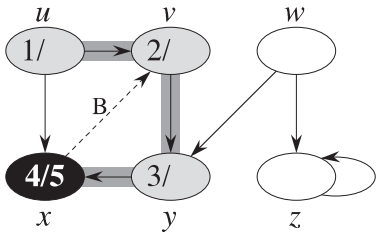
DFS

```
DFS(G)
1.    for each u in V[G]
2.          color[u]=white:
3.          pred[u]=NULL;
4.    time = 0
5.    for each u in V[G]
6.        if (color[u]==white)
7.              DFS-Visit(u)
```

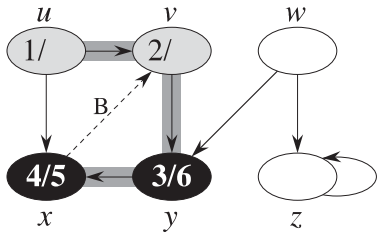
```
DFS-Visit(u)
1.color[u]=grey;   d[u]=time++;
2.for each v in Adj[u]
3.    if (color[v]==white)
4.          pred[v]=u;
5.          DFS-Visit(v);
6.color[u]=black
7.f[u]=time++;
```



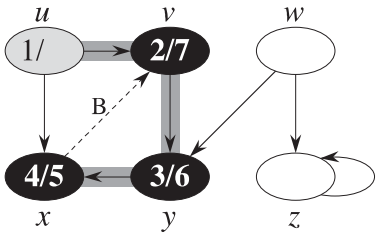
(e)



(f)



(g)

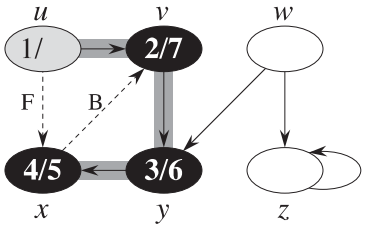


(h)

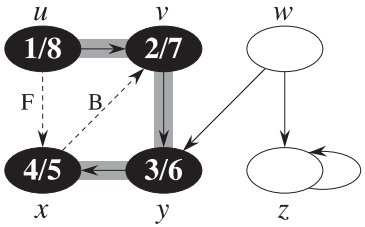
DFS

```
DFS(G)
1.    for each u in V[G]
2.          color[u]=white:
3.          pred[u]=NULL;
4.    time = 0
5.    for each u in V[G]
6.        if (color[u]==white)
7.              DFS-Visit(u)
```

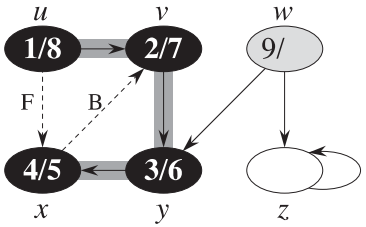
```
DFS-Visit(u)
1.color[u]=grey;   d[u]=time++;
2.for each v in Adj[u]
3.    if (color[v]==white)
4.          pred[v]=u;
5.          DFS-Visit(v);
6.color[u]=black
7.f[u]=time++;
```



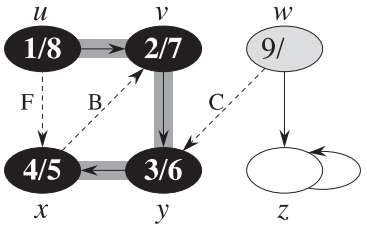
(i)



(j)



(k)



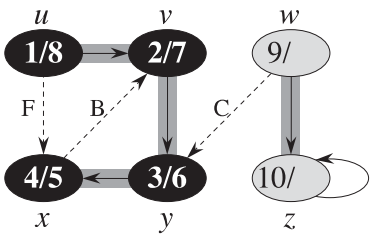
(l)

DFS

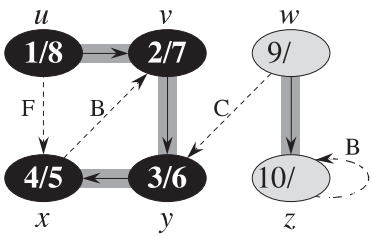
Tempo di esecuzione: Θ (V+E)

```
DFS(G)
1.    for each u in V[G]
2.              color[u]=white:
3.              pred[u]=NULL;
4.    time = 0
5.    for each u in V[G]
6.          if (color[u]==white)
7.                    DFS-Visit(u)
```

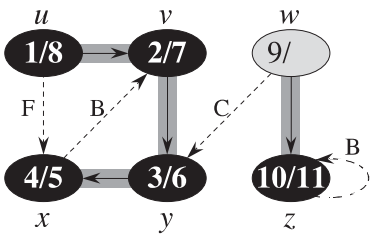
```
DFS-Visit(u)
1.color[u]=grey;   d[u]=time++;
2.for each v in Adj[u]
3.    if (color[v]==white)
4.              pred[v]=u;
5.              DFS-Visit(v);
6.color[u]=black
7.f[u]=time++;
```



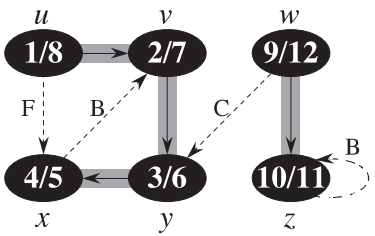
(m)



(n)



(o)



(p)

Classificazione degli archi

Tree edges

- Archi nella depth-first forest G_p
- (u,v) è un tree-edge se v è scoperto (per le prima volta) quando si è esplorato l'arco (u,v)

Back edges

- (u,v) collega u ad un antenato v nel depth-first tree

Forward edges

- (u,v) collega u ad un discendente v nel depth-first tree

Cross edges

- Tutti gli altri tipi di archi.

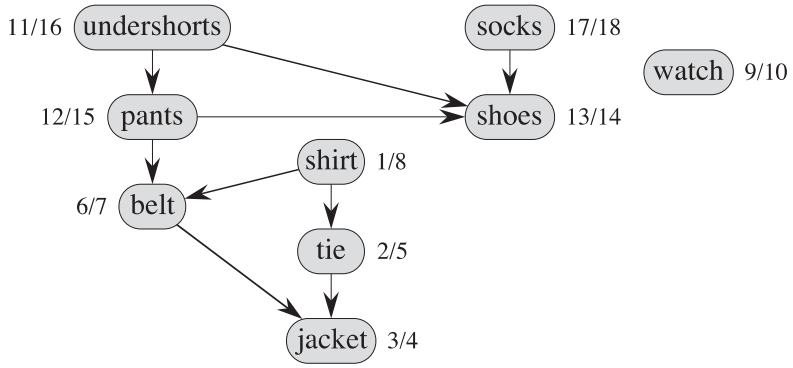
Topological Sort (Ordinamento topologico)

- DFS può essere usato per fare TS di un grafo diretto e aciclico (DAG)

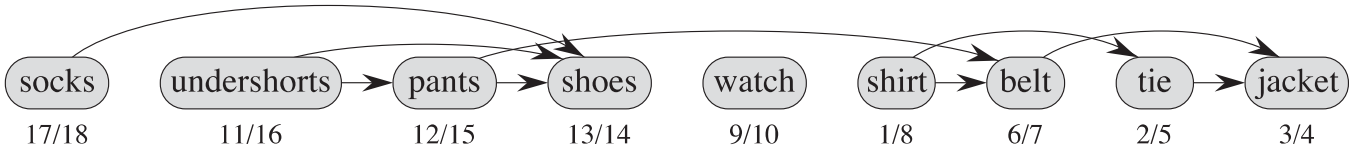
Ordinamento Topologico

- Ordinamento lineare di tutti i vertici tale che se (u,v) è in G allora u precede v nell'ordinamento.
- Può essere visto come come un ordinamento dei vertici su una linea orizzontale.

Esempio



- (u,v) indica che u deve essere indossato prima di v



- I vertici sono ordinati in base al tempo di completamento (f) in maniera decrescente

Topological Sort

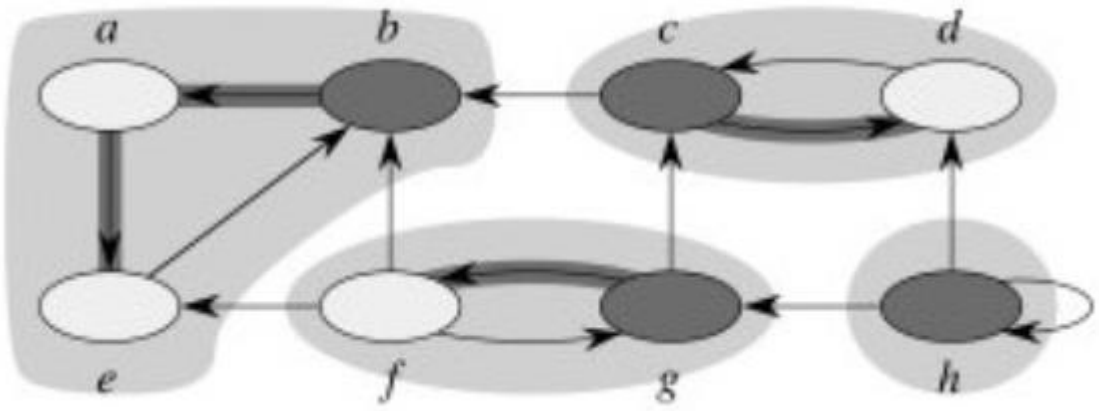
TOPOLOGICAL SORT(G)

1. DFS(G) // *Permette di calcolare f[v] per ogni v*
2. Non appena viene calcolato f[v], inserisci v (in testa) in L
3. return L

- L lista concatenata
- Complessità: O(|E|+|V|)

Componenti Fortemente Connesse (Strongly Connected Components)

- DFS permette di decomporre un grafo (diretto) nelle sue componenti fortemente connesse.
- Componente fortemente connessa: insieme massimale di vertici tale che per ogni coppia di vertici (u,v) i vertici sono raggiungibili l'uno dall'altro.



Componenti Fortemente Connesse (Strongly Connected Components)

- Utilizziamo $G^T=(V,E^T)$ del grafo originario G

$$E^T=\{(u,v): (v,u) \text{ in } E\}$$

- Tempo per creare G^T: $O(|V|+|E|)$ (usando liste di adiacenza)
- G e G^T hanno le stesse componenti (fortemente) connesse.

Stronly Connected Components

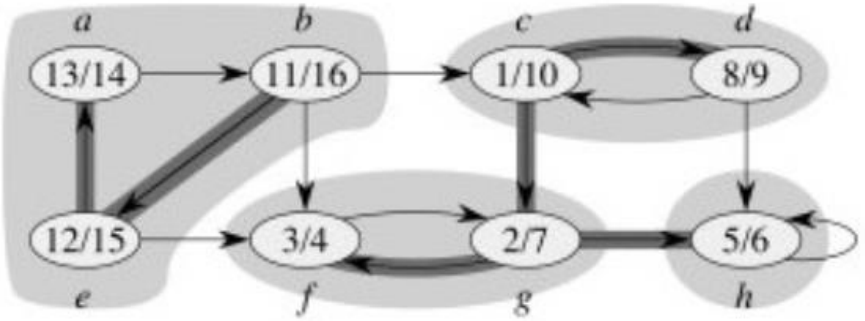
SSC(G)

1. DFS(G) // *Permette di calcolare f[v] per ogni v*
2. Calcola G^T
3. DFS(G^T) ma nel ciclo principale della procedura (scelta nodo sorgente), legge i vertici in ordine decrescente di f[v]
4. **return** i vertici di ogni albero del passo 3

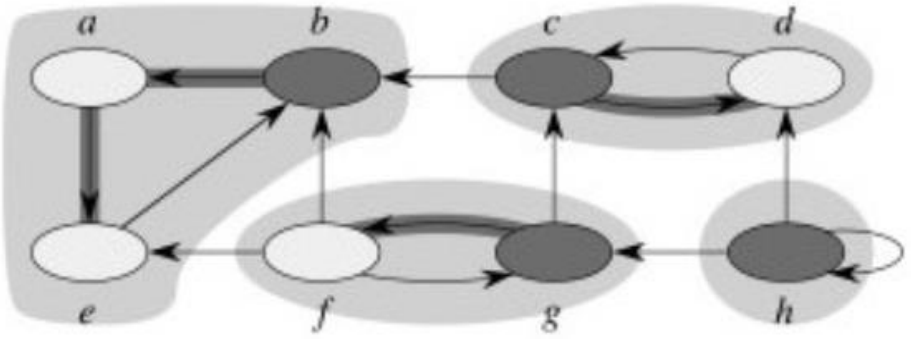
- Complessità: O(|E|+|V|)

Stronly Connected Components

Prima DFS



Seconda DFS



- Ogni DFS a partire da un nodo ‘s’ troverà tutti i nodi nella sua componente connessa
- G e G^T hanno le stesse componenti (fortemente) connesse.