Ricapitoliamo - la volta scorsa abbiamo visto...

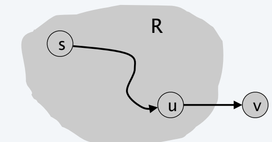
Nozioni di base di teoria dei grafi

- G = (V, E) vertici e archi
  - grafi non orientati (oggi parleremo anche di grafi orientati)
  - |V| = n, |E| = m
  - il grado di un vertice è il numero di vicini (vertici collegati da un arco)
  - somma dei gradi = 2 m
- rappresentazione di grafi
  - liste di adiacenza (preferita) spazio O(m+n)
  - matrice di adiacenza, spazio O(n^2)
- connettività:
  - quali vertici connessi da un cammino ad un dato vertice
  - componente connessa
  - distanza minima in numero di vertici da attraversare
- visita BFS - ricerca di tutti i vertici raggiungibili da un vertice
  - complessità O(m+n)
  - struttura dati: coda

Componente connessa - un algoritmo generale

Componente connessa. Trova tutti i nodi raggiungibili da s.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



possiamo aggiungere v

Teorema. Alla terminazione, R è la componente connessa di s.

- BFS = esplora in ordine di distanza da s.

Usando BFS, possiamo trovare **tutte** le componenti connesse in tempo *O(n+m)*

- *inizializza un array con tutti i nodi non visitati*
- *per ogni nodo* ***s*** *non visitato esegui BFS(s)*

Esercizio: Scrivere l'intero algoritmo con la complessità O(n+m)

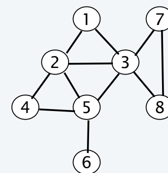
Visita in profondità (Depth first search)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- visita il primo nodo s
- per ogni vicino x di s non visitato
  - visita ricorsivamente a partire da x

prima di visitare un altro vicino di s può allontanarsi molto da s, lungo un qualche cammino che parte da s

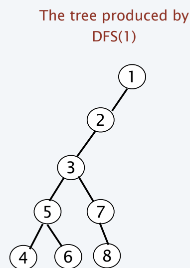
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

The tree produced by DFS(1)



Visita in profondità (Depth-first search)

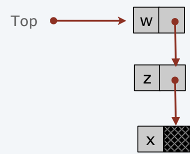
Struttura dati: Stack o Pila

Una pila P è una lista ordinata di oggetti, dinamica, in cui ogni nuovo oggetto viene aggiunto all'inizio (al top) della lista ed ogni oggetto che lascia la pila viene preso dall'inizio (top) della lista.

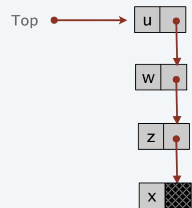
- politica LIFO (last in, first out)
- implementabile da una lista a puntatori,
- le operazioni di inserimento (Stack-in(P,u)) e di estrazione (Stack-out(P)) costano O(1)

Strutture dati: Stack (Pila)

La pila P contiene gli elementi: x, z, w (entrati in quest'ordine)
L'elemento w è al top

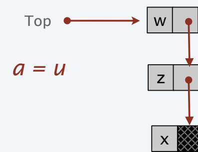


Aggiungere un elemento: Stack-in(P, u) - La pila diventa



Estrarre/Rimuovere un elemento: a = Stack-out(P)
dalla pila eliminiamo u, e ad a viene assegnato il valore u

$a = u$



Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x

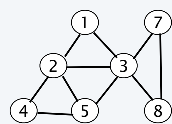
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

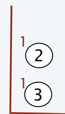
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



Pila P

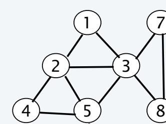
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

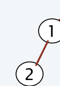
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x

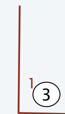


The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)





Pila P

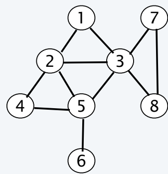
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

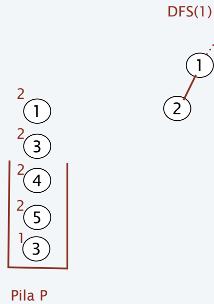
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7


Pila P

DFS(1)

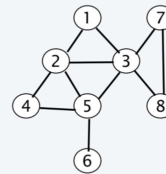
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
     execute DFS on the extracted
```


Visita in profondità (Depth first search) usando una pila (o stack)

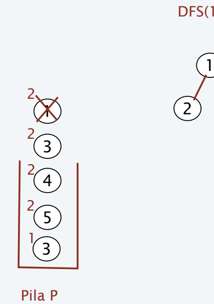
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



Pila P

DFS(1)

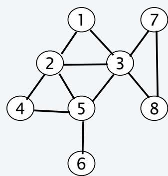
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
     execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

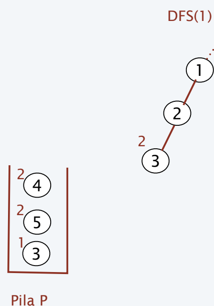
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



Pila P

DFS(1)

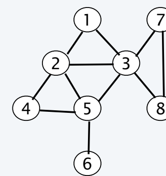
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
     execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

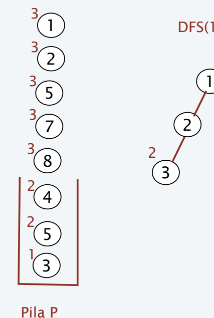
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



Pila P

DFS(1)

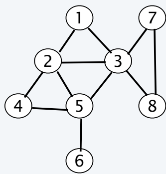
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
     execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

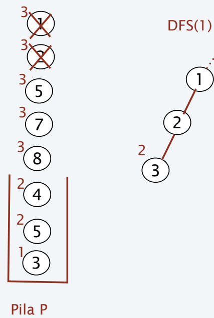
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



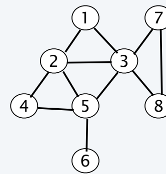

```
DFS(v)   // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
     execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

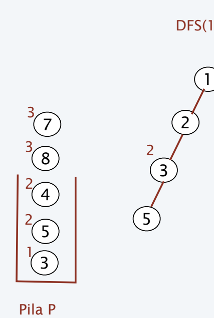
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



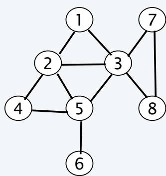
DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

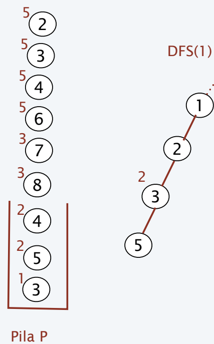
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



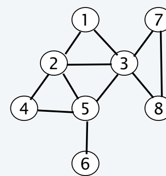
DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

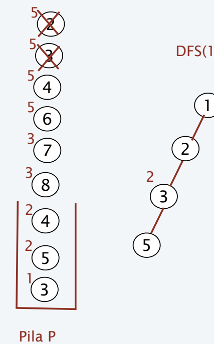
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



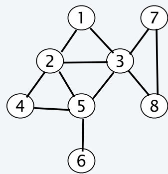
DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

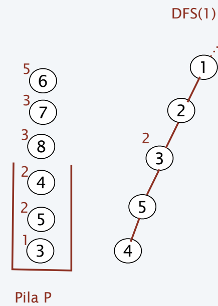
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



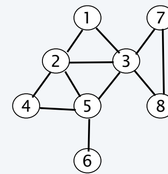

```
DFS(v)   // versione ricorsiva
Set visited[v] = true
   for each u in Adj[v]
      if (visited[u] = false)
         add edge (v,u) to the BFS tree T
         DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
   if not visited yet
      link to its parent in DFS tree
       execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

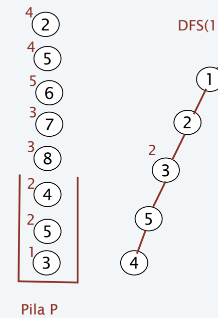
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



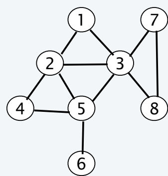
DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

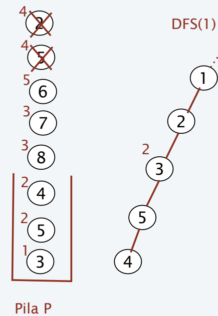
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7



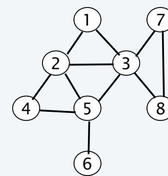
DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

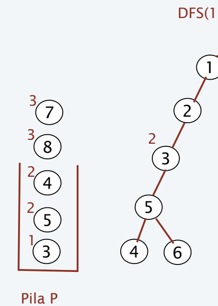
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

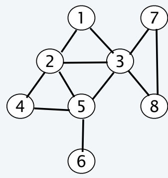


DFS(v)   // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

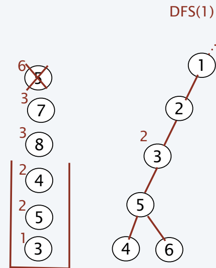
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



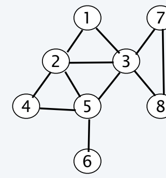
Pila P

```
DFS(v)   // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

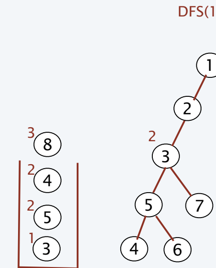
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



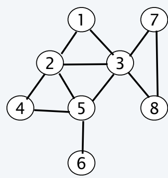
Pila P

DFS(v) // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

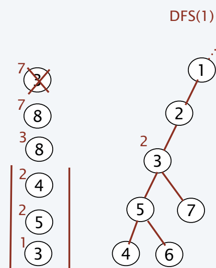
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



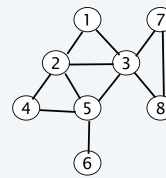
Pila P

DFS(v) // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

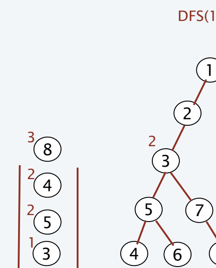
DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



Pila P

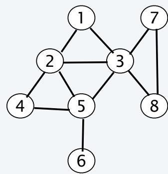
DFS(v) // versione ricorsiva
Set visited[v] = true
for each u in Adj[v]
if (visited[u] = false)
add edge (v,u) to the BFS tree T
DFS(u)

Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
if not visited yet
link to its parent in DFS tree
execute DFS on the extracted

Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

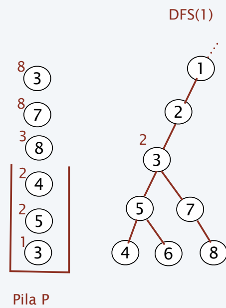
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



Pila P

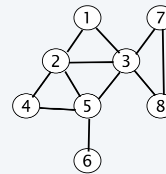
```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```


Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

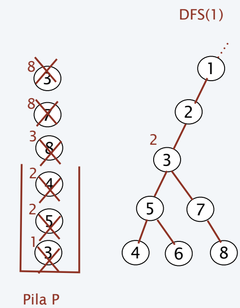
- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x



The graph G

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

DFS(1)



Pila P

```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
Insert all neighbors of v in a stack
remember their parent is v
extract them one at a time
  if not visited yet
    link to its parent in DFS tree
    execute DFS on the extracted
```

Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x

```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```


```
DFS(v)  //  utilizzando uno stack
Set visited[u] = false  for each vertex u
Stack-in(P, v); parent[v] = nil
while (P is not empty) {
  v ⟵ Stack-out(P)
  if (visited[v] = false)
    visited[v] = true
    add edge (v,parent[v]) to the DFS tree T
    for each u in Adj[v] //in reverse order
      Stack-in(P,u); parent[u] = v
```


Visita in profondità (Depth first search) usando una pila (o stack)

DFS(s) visita gli stessi nodi di BFS(s) ma in ordine (anche molto) diverso

- per ogni vicino x di s non visitato
- visita ricorsivamente a partire da x

- si noti che l'implementazione mediante la pila è “identica” alla DFS tranne che per il fatto che usiamo una pila al posto di una coda
  - il ciclo for è eseguito al più una volta per nodo


```
DFS(v)  // versione ricorsiva
Set visited[v] = true
  for each u in Adj[v]
    if (visited[u] = false)
      add edge (v,u) to the BFS tree T
      DFS(u)
```

```
DFS(v)  //  utilizzando uno stack
Set visited[u] = false  for each vertex u
Stack-in(P, v); parent[v] = nil
while (P is not empty) {
  v ⟵ Stack-out(P)
  if (visited[v] = false)
    visited[v] = true
    add edge (v,parent[v]) to the DFS tree T
    for each u in Adj[v] //in reverse order
      Stack-in(P,u); parent[u] = v
```

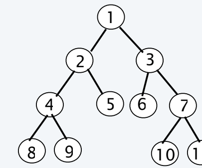
Visita in profondità (Depth-first search): analysis

Teorema. La DFS può essere implementata in tempo $O(m + n)$ se il grafo è rappresentato dalle liste delle adiacenze.

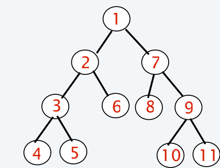
Dim.
- ogni nodo può entrare nella pila P più volte
  - diventa visited la prima volta che ne esce
  - il nodo viene aggiunto nell'albero al più una volta
  - il ciclo for viene eseguito al più una volta per ogni nodo
- per un nodo *v* il for viene eseguito $d(v)$ volte (il grado del nodo *v*)
- spendiamo in tutto tempo
  $\Sigma_{v \in V} d(v) = 2m = O(m)$
  nelle iterazioni del while
- all'inizio spendiamo $O(n)$ per settare `visited[]=false` per tutti
  - quindi $O(n+m)$

```
DFS(v)  //  utilizzando uno stack
Set visited[u] = false  for each vertex u
Stack-in(P, v); parent[v] = nil
while (P is not empty) {
  v ← Stack-out(P)
  if (visited[v] = false)
    visited[v] = true
    add edge (v,parent[v]) to the DFS tree T
    for each u in Adj[v] //in reverse order
      Stack-in(P,u); parent[u] = v
```

Un confronto tra BFS e DFS (il grafo in questo caso è un albero)



Ordine di visita BFS



Ordine di visita DFS

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

```
DFS(v)  //  utilizzando uno stack
Set visited[u] = false  for each vertex u
Stack-in(P, v); parent[v] = nil
while (P is not empty) {
  v ← Stack-out(P)
  if (visited[v] = false)
    visited[v] = true
    add edge (v,parent[v]) to the DFS tree T
    for each u in Adj[v] //in reverse order
      Stack-in(P,u); parent[u] = v
```

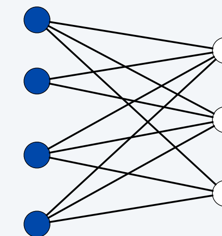
Applicazioni delle visite BFS e DFS

- testare se un grafo è bipartito
- connettività in grafi orientati
- DAG e ordinamento topologico

Grafi Bipartiti

Def. Un grafo (non orientato) G = (V, E) è bipartito se i nodi possono essere colorati di **blu** o bianco in modo tale che ogni arco unisca un nodo bianco con un nodo blu.

Applicazioni.
- Stable matching/marriage: men = blu, women = bianco.
- Scheduling: machines = blu, jobs = bianco.



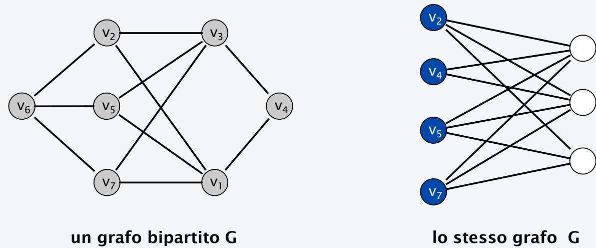
un grafo bipartito

Testare se un grafo è bipartito

Molti problemi su grafi diventano:

- più facili se il grafo è bipartito (matching).
  - trattabili (polinomiali) se il grafo in input è bipartito, mentre solo algoritmi esponenziali sono noti in generale (es. independent set).

E' utile avere una procedura che può dirci in maniera efficiente se un grafo è bipartito

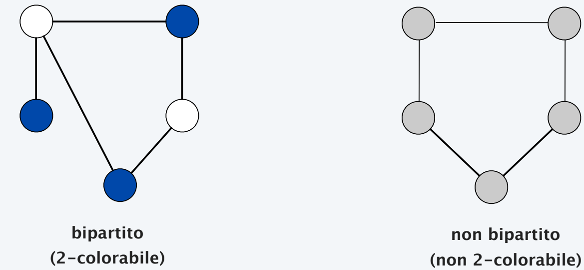


un grafo bipartito G — **lo stesso grafo G**

Un ostacolo all'essere bipartito

Lemma. Se un grafo G è bipartito, non può contenere un ciclo di lunghezza dispari.

Dim. Non è possibile avere una bicolorazione di un ciclo dispari, quindi a maggior ragione non esiste una bi-colorazione di G.



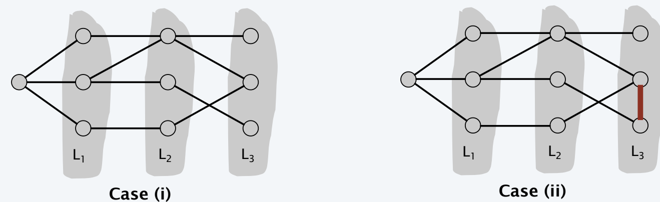
bipartito (2-colorabile) — **non bipartito (non 2-colorabile)**

Grafi bipartiti

Lemma. Sia G un grafo connesso, e siano $L_0, \ldots, L_k$ i livelli prodotti da una BFS con nodo di partenza s. Solo una delle seguenti affermazioni è vera:

(i) Nessun arco di G unisce due nodi dello stesso livello, e G è bipartito.

(ii) Un arco di G unisce due nodi dello stesso livello, e G contiene un ciclo di lunghezza dispari (e quindi NON è bipartito).



Case (i) — **Case (ii)**

Grafi bipartiti

Lemma. Sia G un grafo connesso, e siano $L_0, \ldots, L_k$ i livelli prodotti da una BFS con nodo di partenza s. Solo una delle seguenti affermazioni è vera:

(i) Nessun arco di G unisce due nodi dello stesso livello, e G è bipartito.

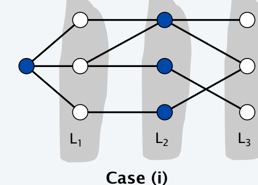
(ii) Un arco di G unisce due nodi dello stesso livello, e G contiene un ciclo di lunghezza dispari (e quindi NON è bipartito).

Dim. (i)

- Supponiamo che nessun arco unisca nodi dello stesso livello.

  Avevamo provato che la BFS riorganizza il grafo in modo da avere tutti gli archi tra nodi di livelli adiacenti o tra nodi sullo stesso livello

- quindi, tutti gli archi sono tra nodi di livelli adiacenti.
- Bipartizione: bianchi = nodi di livello dispari, blu = nodi su livelli pari.



Case (i)

Grafi bipartiti

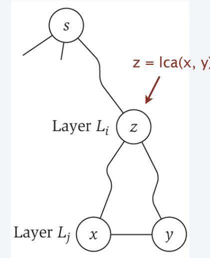
Lemma. Sia G un grafo connesso, e siano $L_0, \ldots, L_k$ i livelli prodotti da una BFS con nodo di partenza s. Solo una delle seguenti affermazioni è vera:

(i) Nessun arco di G unisce due nodi dello stesso livello, e G è bipartito.

(ii) Un arco di G unisce due nodi dello stesso livello, e G contiene un ciclo di lunghezza dispari (e quindi NON è bipartito).

Dim. (ii)

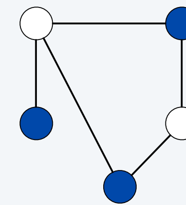
- Supponiamo (x, y) sia un arco con x, y nel livello L_j.
- Sia $z = lca(x, y)$ = il più vicino antenato comune di x,y.
- Sia L_i il livello di z.
- Consideriamo il ciclo C costituito da: (i) l'arco tra x e y, (ii) il cammino da y a z, (iii) il cammino da z a x.
- la lunghezza di C è $\underbrace{1}_{(x,y)} + \underbrace{(j-i)}_{\text{cammino da } y \text{ a } z} + \underbrace{(j-i)}_{\text{cammino da } z \text{ a } x} = 2(j-i)+1$, quindi **dispari**. ▪



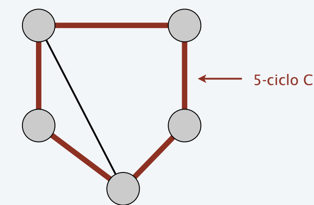
L'unico ostacolo all'essere bipartito

Corollario. Un grafo G è bipartito se e solo se NON contiene un ciclo di lunghezza dispari.

- l'abbiamo provato (al contrario) facendo vedere che:
  - un grafo **non** è bipartito **se e solo se contiene** un ciclo dispari



bipartito
(2-colorabile)



non bipartito
(non 2-colorabile)

Esercizio: Scrivere l'intero pseudocodice per testare se un grafo è bipartito
Modificare BFS, ottenendo un algoritmo con la complessità O(n+m)

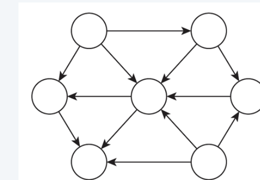
Applicazioni delle visite BFS e DFS

- testare se un grafo è bipartito
- connettività in grafi orientati
- DAG e ordinamento topologico

Grafi orientati

Notazione. $G = (V, E)$.

- Arco (u, v) esce dal nodo u ed entra nel nodo v.

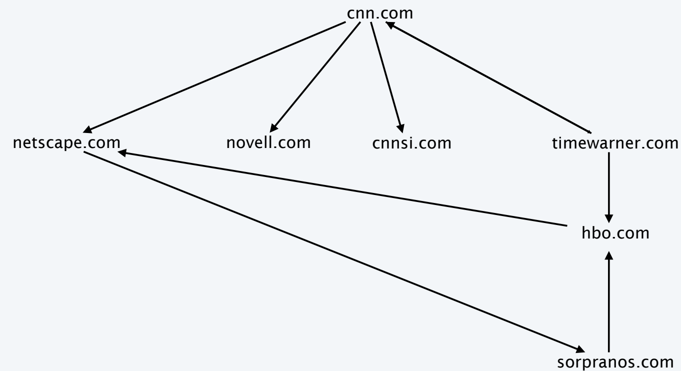


Es. Web: gli hyperlink creano collegamenti ***da*** una pagina web ***ad*** un'altra.

- L'orientamento degli archi è cruciale.
- I motori di ricerca sfruttano la struttura degli hyperlink per classificare le pagine (determinare quelle più importanti)

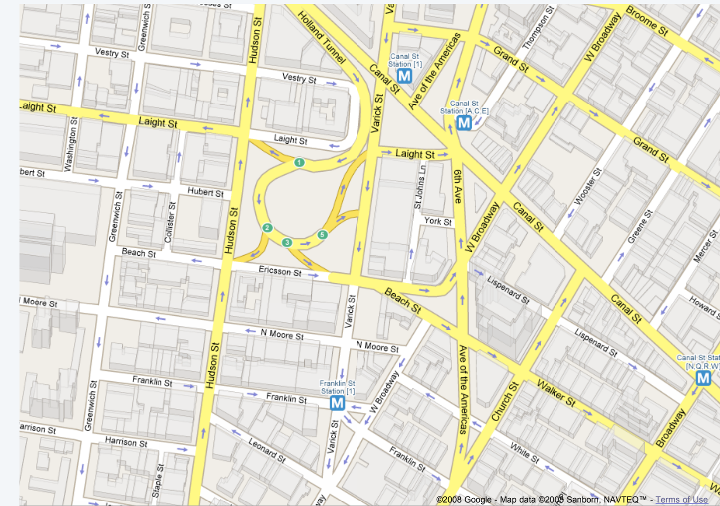
World wide web

Il Grafo del Web
- Nodi: pagine web.
- Archi: hyperlink da una pagina ad un'altra.
  - la struttura degli hyperlink è sfruttata dai motori di ricerca per determinare l'importanza delle pagine web.



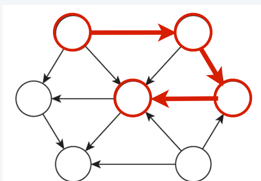
La rete stradale

Vertici = incroci; archi = strade a senso unico.

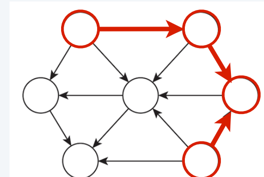


Cammini e connettività in grafi orientati

Def. Un cammino in un grafo orientato $G = (V, E)$ è una sequenza di nodi v_1, v_2, …, v_k con la proprietà che ogni coppia di nodi consecutivi v_{i-1}, v_i risulta connessa da un arco (orientato) in E.



Un cammino (orientato)



Questo NON è un cammino
(nel grafo orientato)

Ricerca in un grafo (orientato)

Raggiungibilità. Dato un nodo s, trova tutti i nodi raggiungibili **da** s.

Il problema del cammino minimo (orientato). Dati due nodi s e t, qual è la lunghezza del cammino minimo **da** s **a** t?

Osservazione. La visita BFS si può estendere al caso di grafi orientati.

Web crawler. Cominciano da una pagina s. Trovano tutte le pagine web collegate (attraverso una sequenza di hyperlink) ad s (sia direttamente che indirettamente).

Domande:

1. Possiamo risolvere il problema della raggiungibilità usando la visita DFS ?

2. Possiamo risolvere il problema del cammino minimo orientato, usando la DFS?

Connettività forte (in grafi orientati)

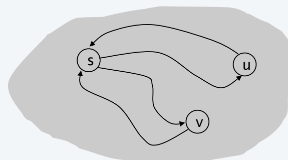
Def. Nodi u e v sono mutuamente raggiungibili se esiste sia un cammino (orientato) da u a v che un cammino (orientato) da v ad u.

Def. Un grafo è fortemente connesso se ogni coppia di nodi è mutuamente raggiungibile.

Lemma. Sia s un nodo. G è fortemente connesso sse ogni nodo è raggiungibile da s, ed s è raggiungibile da ogni nodo.

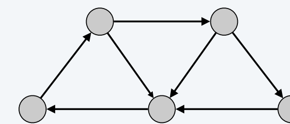
Dim. $\Rightarrow$ segue dalla definizione.
Dim. $\Leftarrow$ Cammino da u a v: concatena il cammino $u \to s$ col cammino $s \to v$.
Cammino da v a u: concatena $v \to s$ con $s \to u$ path. ▪

ok se i cammini si sovrappongono

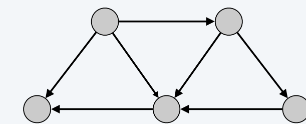


Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.
Dim.
- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪



fortemente connesso

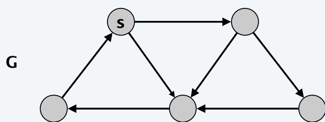


non fortemente connesso

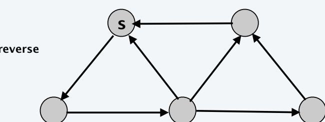
Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.
Dim.
- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪

G



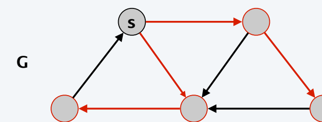
$G^{reverse}$



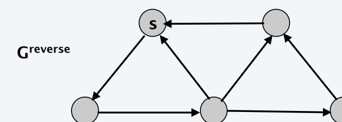
Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.
Dim.
- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪

G



$G^{reverse}$

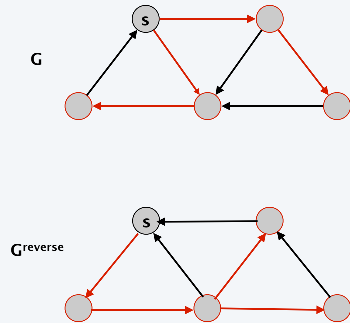


Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m+n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G.
- Eseguiamo BFS da s in $G^{reverse}$. (Ottenuto invertendo la direzione di ogni arco di G)
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪

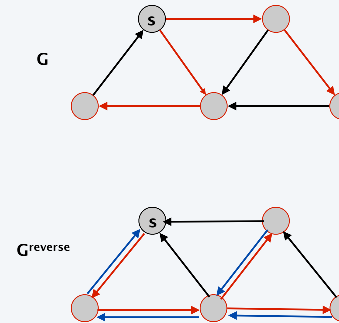


Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m+n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G.
- Eseguiamo BFS da s in $G^{reverse}$. (Ottenuto invertendo la direzione di ogni arco di G)
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪

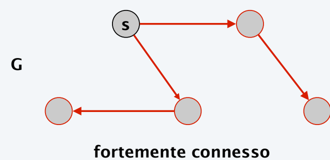


Connettività forte: algoritmo

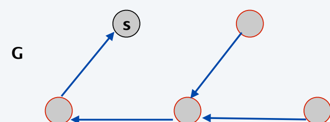
Teorema. è possibile determinare se G è fortemente connesso in $O(m+n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G.
- Eseguiamo BFS da s in $G^{reverse}$. (Ottenuto invertendo la direzione di ogni arco di G)
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪



fortemente connesso

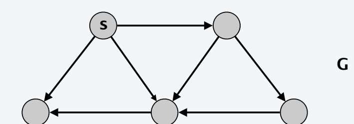


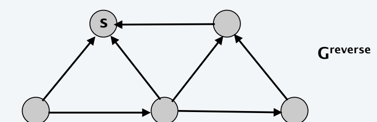
Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m+n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G.
- Eseguiamo BFS da s in $G^{reverse}$. (Ottenuto invertendo la direzione di ogni arco di G)
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪



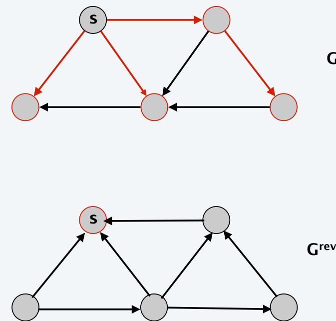


Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪

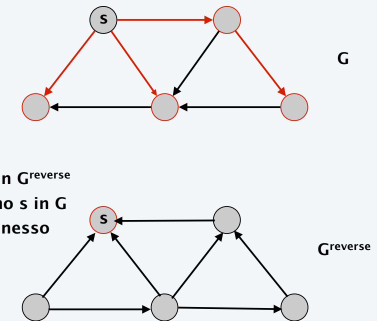


Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.

Dim.

- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪



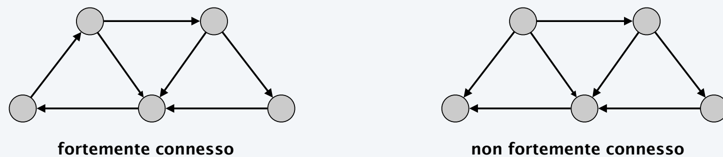
da s non raggiungiamo nodi in $G^{reverse}$
da nessun nodo raggiungiamo s in G
-> G non è fortemente connesso

Connettività forte: algoritmo

Teorema. è possibile determinare se G è fortemente connesso in $O(m + n)$.

Dim.

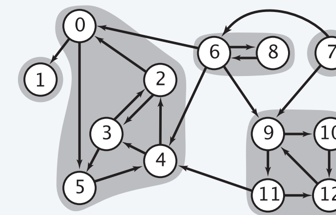
- Scegliamo arbitrariamente un nodo s.
- Eseguiamo BFS da s in G. Ottenuto invertendo la direzione di ogni arco di G
- Eseguiamo BFS da s in $G^{reverse}$.
- Ritorna "true" sse tutti i nodi risultano visitati da entrambe le BFS.
- La correttezza è una diretta conseguenza del lemma precedente. ▪



fortemente connesso **non fortemente connesso**

Componenti fortemente connesse

Def. Una componente fortemente connessa è un sottinsieme di nodi massimale rispetto alla mutua raggiungibilità.



Teorema. [Tarjan 1972] Tutte le componenti fortemente connesse possono essere trovate in tempo $O(m + n)$.

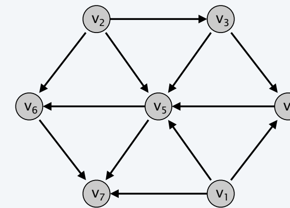
Applicazioni delle visite BFS e DFS

- testare se un grafo è bipartito
- connettività in grafi orientati
- DAG e ordinamento topologico

Grafi orientati aciclici (Directed acyclic graphs, DAG)

Def. Un DAG è un grafo orientato che non contiene un ciclo orientato.

Def. Un ordinamento topologico di un grafo orientato $G = (V, E)$ è un ordinamento dei suoi nodi $v_1, v_2, \ldots, v_n$ tale che per ogni arco (v_i, v_j) vale $i < j$.

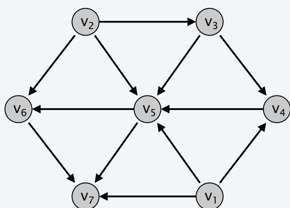


un DAG G

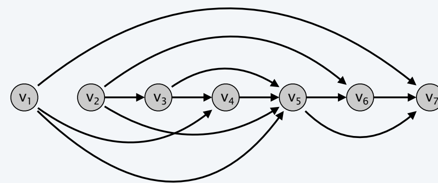
Grafi orientati aciclici (Directed acyclic graphs, DAG)

Def. Un DAG è un grafo orientato che non contiene un ciclo orientato.

Def. Un ordinamento topologico di un grafo orientato $G = (V, E)$ è un ordinamento dei suoi nodi $v_1, v_2, \ldots, v_n$ tale che per ogni arco (v_i, v_j) vale $i < j$.



un DAG G

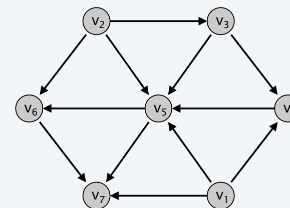


un ordinamento tolopogico di G

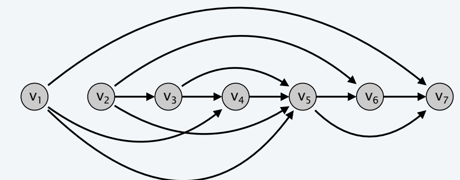
DAG e Vincoli di precedenza

Vincoli di precedenza. L'arco (v_i, v_j) indica che v_i deve apparire prima di v_j.

Applicazioni.
- Grafo dei prerequisiti tra corsi: il corso v_i deve essere superato prima di v_j.
- Compilazione: modulo v_i deve essere compilato prima di v_j.
- Gene networks: nodi = proteine o geni; archi = interazioni (attivazioni, inibizioni, specificità di binding); orientamento (relazione di precursione)
- ordinamento topologico = risoluzione di vincoli di precedenza



un DAG G



un ordinamento tolopogico di G

Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

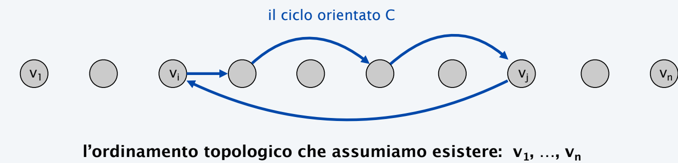
Dim. [per assurdo]

Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

Dim. [per assurdo]

- Supponiamo che G abbia un ordinamento topologico $v_1, v_2, \ldots, v_n$ e che in G ci sia anche un ciclo orientato C. Vediamo cosa succede.

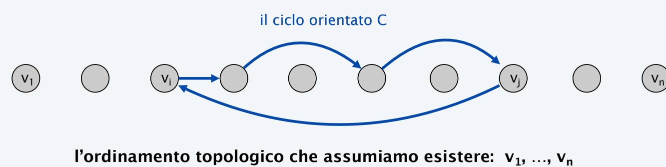


Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

Dim. [per assurdo]

- Supponiamo che G abbia un ordinamento topologico $v_1, v_2, \ldots, v_n$ e che in G ci sia anche un ciclo orientato C. Vediamo cosa succede.
- Sia v_i il nodo in C di indice minimo nell'ordinamento, e sia v_j il nodo che lo precede nel ciclo; quindi (per il ciclo) (v_j, v_i) è un arco (orientato).

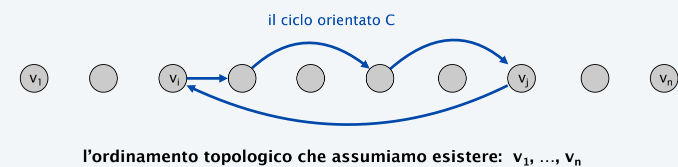


Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

Dim. [per assurdo]

- Supponiamo che G abbia un ordinamento topologico $v_1, v_2, \ldots, v_n$ e che in G ci sia anche un ciclo orientato C. Vediamo cosa succede.
- Sia v_i il nodo in C di indice minimo nell'ordinamento, e sia v_j il nodo che lo precede nel ciclo; quindi (per il ciclo) (v_j, v_i) è un arco (orientato).
- Per la scelta di i (il minimo indice di un nodo nel ciclo) deve valere $i < j$.

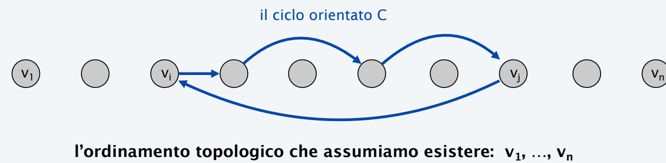


Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

Dim. [per assurdo]

- Supponiamo che G abbia un ordinamento topologico $v_1, v_2, \ldots, v_n$ e che in G ci sia anche un ciclo orientato C. Vediamo cosa succede.
- Sia v_i il nodo in C di indice minimo nell'ordinamento, e sia v_j il nodo che lo precede nel ciclo; quindi (per il ciclo) (v_j, v_i) è un arco (orientato).
- Per la scelta di i (il minimo indice di un nodo nel ciclo) deve valere $i < j$.
- Tuttavia, poiché (v_j, v_i) è un arco e $v_1, v_2, \ldots, v_n$ è un ordinamento topologico, deve valere $j < i$, quindi abbiamo una contraddizione. ▪



Grafi orientati aciclici

Lemma. Se G ha un ordinamento topologico, allora G è un DAG.

Q. e' anche vero che ogni DAG ha un ordinamento topologico?

Q. Se sì, come possiamo trovarlo?

Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?

Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

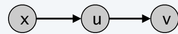
- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?
- Prendiamo un nodo v, e seguiamo gli archi all'indietro cominciando da v. Poiché v ha almeno un arco entrante (u, v) possiamo andare ad u.

Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?
- Prendiamo un nodo v, e seguiamo gli archi all'indietro cominciando da v. Poiché v ha almeno un arco entrante (u, v) possiamo andare ad u.
- Quindi, poiché u ha almeno un arco entrante (x, u), possiamo passare al nodo x.

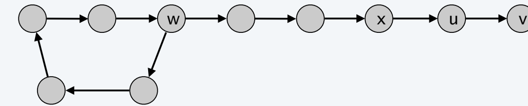


Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?
- Prendiamo un nodo v, e seguiamo gli archi all'indietro cominciando da v. Poiché v ha almeno un arco entrante (u, v) possiamo andare ad u.
- Quindi, poiché u ha almeno un arco entrante (x, u), possiamo passare al nodo x.
- Possiamo continuare così fino a quando visitiamo un nodo, diciamo w, per la seconda volta (deve capitare dopo al più n passi)

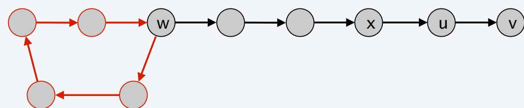


Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?
- Prendiamo un nodo v, e seguiamo gli archi all'indietro cominciando da v. Poiché v ha almeno un arco entrante (u, v) possiamo andare ad u.
- Quindi, poiché u ha almeno un arco entrante (x, u), possiamo passare al nodo x.
- Possiamo continuare così fino a quando visitiamo un nodo, diciamo w, per la seconda volta (deve capitare dopo al più n passi)
- Sia C la sequenza di nodi visitati tra le due visite di w. C è un ciclo.

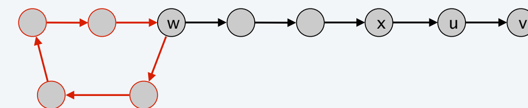


Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti.

Dim. [per assurdo]

- Supponiamo che G sia un DAG e ogni nodo abbia un arco entrante. Cosa succede?
- Prendiamo un nodo v, e seguiamo gli archi all'indietro cominciando da v. Poiché v ha almeno un arco entrante (u, v) possiamo andare ad u.
- Quindi, poiché u ha almeno un arco entrante (x, u), possiamo passare al nodo x.
- Possiamo continuare così fino a quando visitiamo un nodo, diciamo w, per la seconda volta (deve capitare dopo al più n passi)
- Sia C la sequenza di nodi visitati tra le due visite di w. C è un ciclo.
- Poiché c'è un ciclo (orientato) C, allora G non può essere un DAG▪



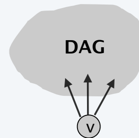
Grafi orientati aciclici

Lemma. Se G è un DAG, allora G ha un nodo senza archi entranti. (Provato!)
Lemma. Se G è un DAG, allora G ha un ordinamento topologico.

Dim. [per induzione su n]

- Base di induzione: vero se $n = 1$.
- Dato un DAG su $n > 1$ nodi, troviamo un nodo v senza archi entranti.
- $G - \{ v \}$ è un DAG, in quanto eliminando v non creiamo cicli.
- Per ipotesi induttiva, $G - \{ v \}$ ha un ordinamento topologico.
- Se ora creiamo la sequenza che comincia con v e continua con l'ordinamento topologico di $G - \{ v \}$
- otteniamo un ordinamento topologico per G. Nota che v non ha archi entranti . ▪

```
Ordinamento-topologico(G)
Trova un nodo v senza archi entranti
  dichiara v il primo nodo nell'ordinamento topologico di G
Sia H = G - v il grafo ottenuto da G eliminando v
Trova, ricorsivamente, un ordinamento topologico S di H
La sequenza ottenuta mettendo prima v e poi S
    è un ordinamento topologico di G
```



Algoritmo di ordinamento topologico: running time

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.

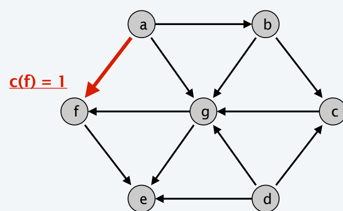
Dim.

- Manteniamo le seguenti informazioni:
  - $c(w)$ = numero di archi entranti rimanenti
  - S = insieme di nodi rimanenti senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w;
    e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.
Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w;
  e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪

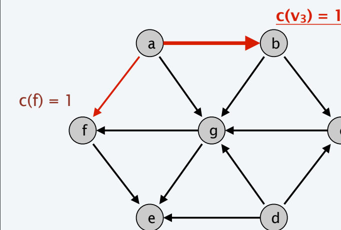


un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.
Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w;
  e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪



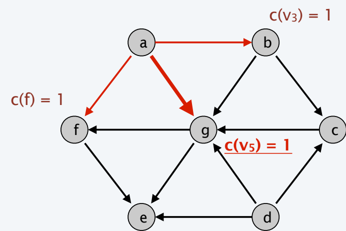
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪



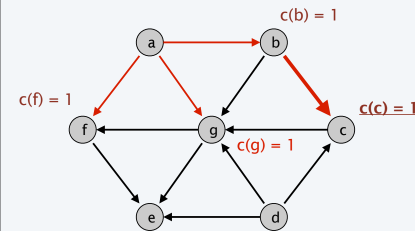
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪



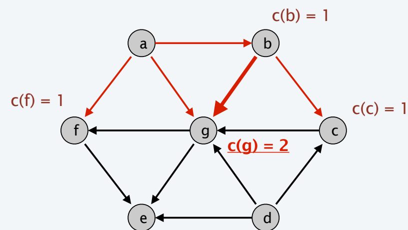
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪



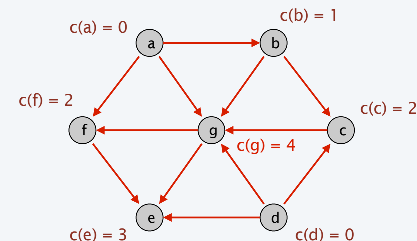
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
- per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
- poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
- rimuoviamo v da S
- decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
- questo richiede tempo $O(1)$ per ogni arco ▪



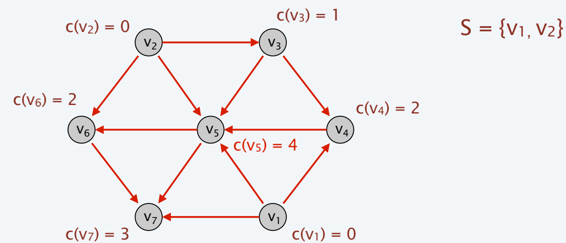
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



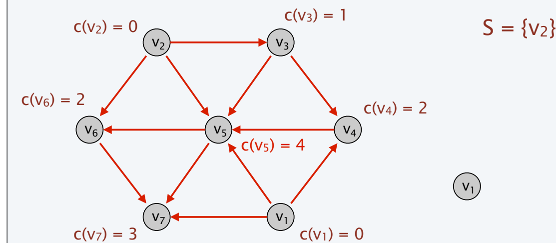
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



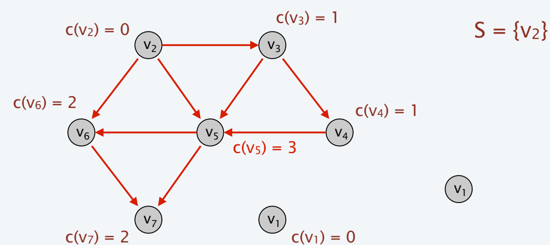
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



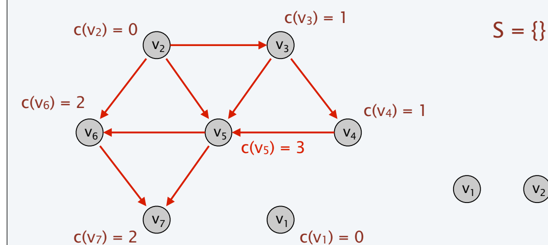
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



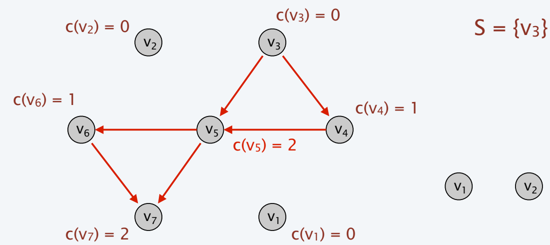
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



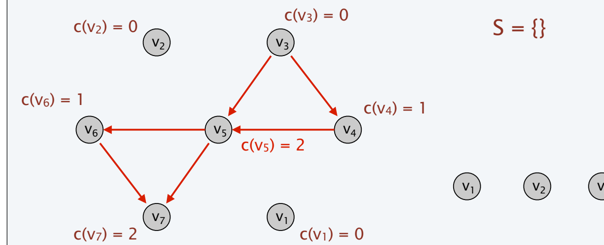
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



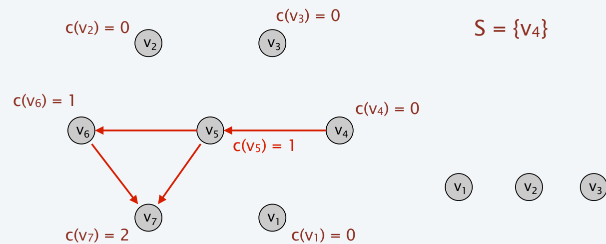
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



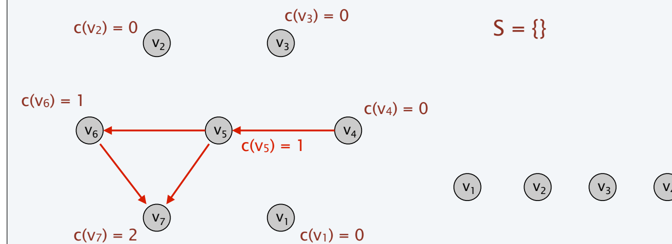
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



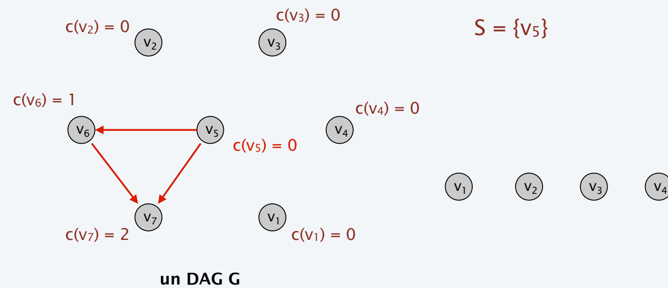
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪

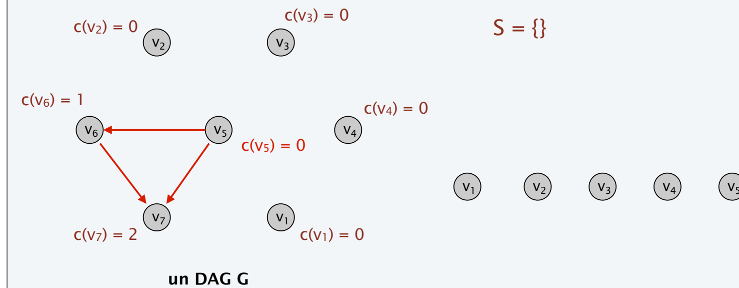


Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪

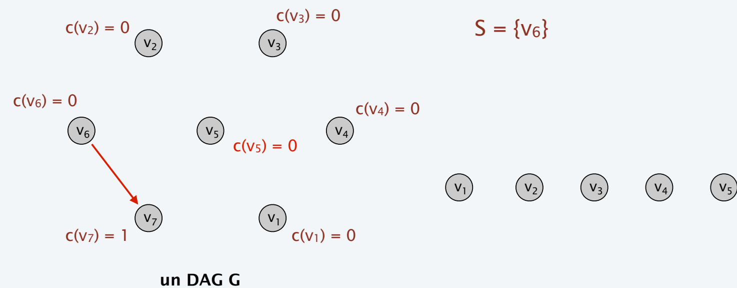


Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪

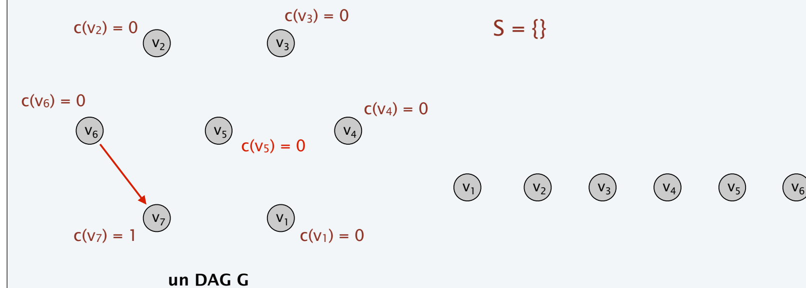


Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m+n)$.

Dim.

- $c(w)$ = # archi entranti rimanenti; S = nodi senza archi entranti
- Inizializzazione: $O(m+n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪

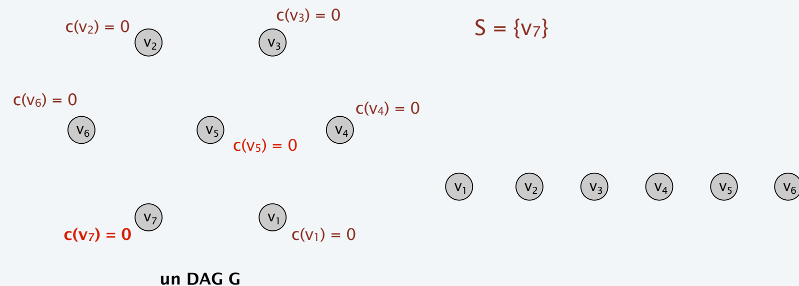


Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.

Dim.

- $c(w) =$ # archi entranti rimanenti; $S =$ nodi senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



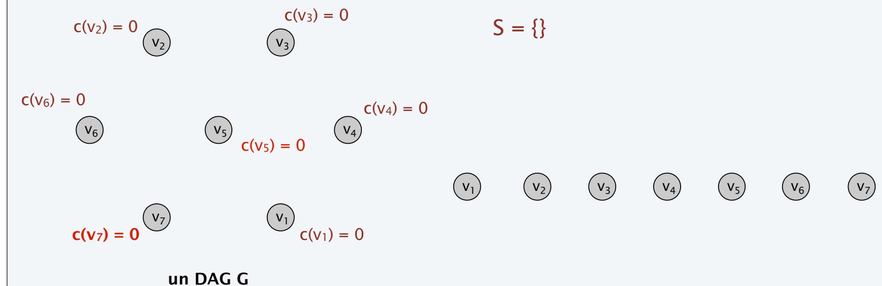
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.

Dim.

- $c(w) =$ # archi entranti rimanenti; $S =$ nodi senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



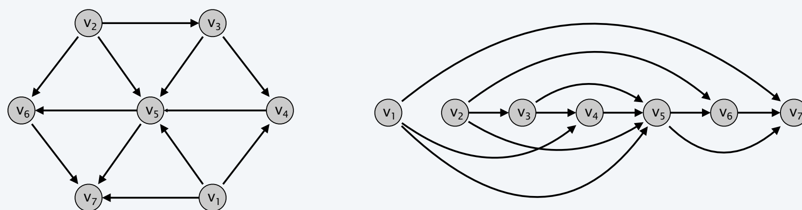
un DAG G

Ordinamento Topologico in O(n+m)

Teorema. L'algoritmo trova un ordinamento topologico in tempo $O(m + n)$.

Dim.

- $c(w) =$ # archi entranti rimanenti; $S =$ nodi senza archi entranti
- Inizializzazione: $O(m + n)$ - mentre leggiamo l'input.
  - per ogni arco (v,w) che troviamo, incrementiamo $c(w)$;
  - poi controlliamo per ogni w se $c(w)$=0, se sì mettiamo w in S
- Aggiornamento: per eliminare un nodo v
  - rimuoviamo v da S
  - decrementiamo $c(w)$ per ogni arco da v a w; e aggiungiamo w ad S se $c(w)$ diventa 0
  - questo richiede tempo $O(1)$ per ogni arco ▪



un DAG G

un ordinamento tolopogico di G