Strutture Dati, Algoritmi e Complessità

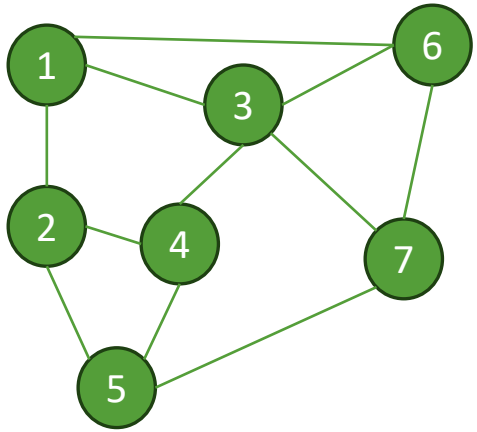
BFS E DFS

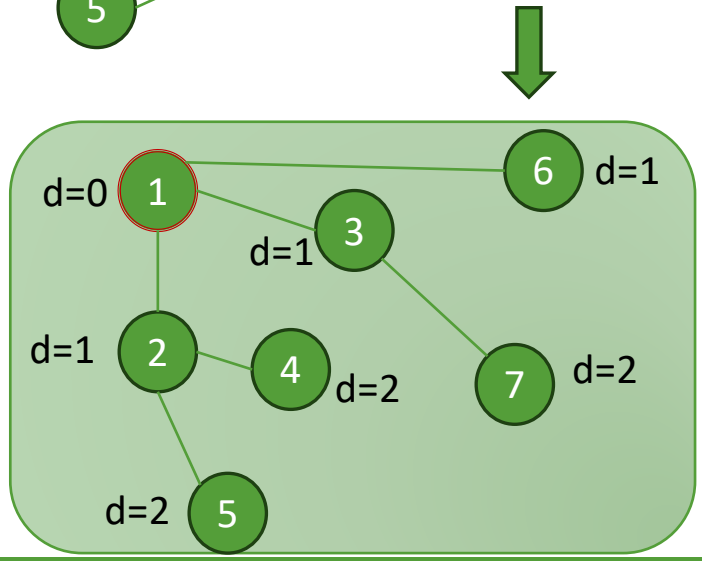
AA 2023-2024

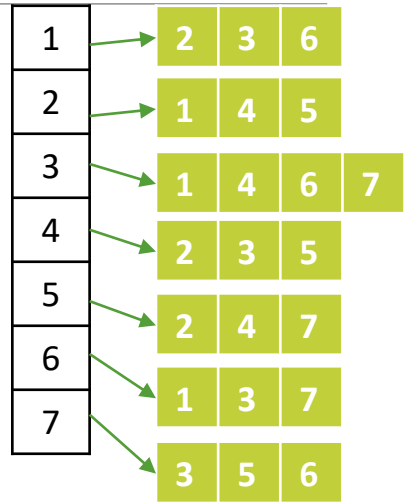
Visita in Ampiezza (BFS)

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```







BFS - Correttezza

BFS visita tutti i vertici raggiungibili da **s** e quando termina **v.d = d(s,v)** per ogni vertice **v** del grafo.

Inoltre per ogni vertice **v** $\neq$ **s** raggiungibile da **s**

- **u = v.p** $\neq$ **nil**,
- **(u,v)** $\in$ **E** e
- **d(s,v) = d(s,u) + 1**.

Per dimostrare la correttezza di BFS ci occorre prima dimostrare 3 proprietà intermedie dell'algoritmo:

- *Proprietà delle distanze*
- *Proprietà del limite superiore*
- *Proprietà della coda*

BFS - Correttezza

Proprietà delle distanze

Sia $G=(V, E)$ un grafo e $s \in V$ un vertice arbitrario (chiamato sorgente). Allora, per ogni arco $(u, v) \in E$ si ha che

$$d(s, v) \leq d(s, u)+1.$$

DIMOSTRAZIONE

Consideriamo 2 casi:

1. u è raggiungibile da s
2. u non è raggiungibile da s

BFS - Correttezza

CASO 1 – u non è raggiungibile da s

- u.d e v.d sono inizializzate a + inf in riga 4
- se u non è raggiungibile da s allora non esiste un cammino da s a u
- se non esiste un cammino vuol dire che non andrà mai nella coda e quindi u.d non verrà mai aggiornato
- OSS. se u non è raggiungibile allora non lo è neanche v quindi neanche lui sarà aggiornato e il claim segue.

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

CASO 2 – u è raggiungibile da s

- u.d e v.d sono inizializzate a + inf in riga 4
- se u è raggiungibile da s allora esiste un cammino da s a u
  - chiamiamo d(s, u) la lunghezza di questo cammino
- Se questo cammino passa già da v il claim è banalmente vero (d(s, v) < d(s, u))
- Se non passa da v, aggiungendo l'arco uv a tale cammino otteniamo un cammino di lunghezza d(s, v) = d(s,u) + 1 che congiunge s a v e il claim segue

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3      u.color ← BIANCO;
4      u.d ← ∞;
5      u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13     u ← Dequeue(Q);
14     foreach v ∈ G.adj[u] do
15         if v.color ≠ BIANCO then
16             v.color ← GRIGIO;
17             v.d ← u.d + 1;
18             v.p ← u;
19             Enqueue(Q, v);
20         end
21     end
22     u.color ← NERO;
23 end
```

BFS - Correttezza

Proprietà del limite superiore

Sia G=(V, E) un grafo e supponiamo che BFS venga eseguita da un nodo s ∈ V arbitrario (chiamato sorgente). Allora per ogni vertice v ∈ V la distanza v.d calcolata da BFS è sempre tale che

$$v.d \geq d(s, v)$$

DIMOSTRAZIONE

PASSO BASE

Dopo l'inizializzazione

- ***s.d* = 0 = *d*(*s*,*s*)** RIGA 8
- mentre per ogni vertice ***u*** ≠ ***s***
- ***u.d*** = ∞ ≥ ***d*(*s*,*u*)** RIGHE 2-6

OSS. L'unica istruzione che modifica la distanza in riga 17 che viene eseguita soltanto se esiste l'arco *uv* e solo una volta (la prima volta che si visita il nodo v).

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

PASSO INDUTTIVO

Supponiamo, per ipotesi induttiva, che la proprietà sia vera prima di eseguire riga 16.

- ossia assumiamo che u.d ≥d(s, u)

Allora dopo aver eseguito riga 17 abbiamo che

$$v.d = u.d + 1$$

$$\geq d(s,u) + 1 \qquad (\textit{ipotesi induttiva})$$

$$\geq d(s,v) \qquad (\textit{proprietà delle distanze})$$

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Proprietà della coda

Consideriamo l'esecuzione di BFS su un grafo G=(V, E) e supponiamo che la coda Q contenga i vertici $<v_1, v_2, \dots v_r>$ (con v_1 inizio e v_r fine della coda). Allora si ha che:

1. $V_r.d \leq v_1.d+1$
2. $V_i.d \leq v_{i+1}.d$ per ogni i in 1, 2, … r-1

DIMOSTRAZIONE

PASSO BASE

Dopo l'inizializzazione la proprietà è vera perché la coda contiene solo *s* **RIGA 11**

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Proprietà della coda

Consideriamo l'esecuzione di BFS su un grafo G=(V, E) e supponiamo che la coda Q contenga i vertici $<v_1, v_2, \dots v_r>$ (con v_1 inizio e v_r fine della coda). Allora si ha che:

1. $V_r.d \leq v_1.d+1$
2. $V_i.d \leq v_{i+1}.d$ per ogni i in 1, 2, … r-1

PASSO INDUTTIVO

OSS. La coda viene modificata solo facendo enqueue (riga 19) e dequeue (riga 13)

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Proprietà della coda

Consideriamo l'esecuzione di BFS su un grafo G=(V, E) e supponiamo che la coda Q contenga i vertici $<v_1, v_2, \dots v_r>$ (con v_1 inizio e v_r fine della coda). Allora si ha che:

1. $V_r.d \leq v_1.d+1$
2. $V_i.d \leq v_{i+1}.d$ per ogni i in 1, 2, … r-1

PASSO INDUTTIVO

1. Per ipotesi induttiva, assumiamo che la proprietà sia vera prima di eseguire *Dequeue*(**Q**). Allora

$$v_r.d \leq v_1.d + 1 \leq v_2.d + 1$$

e quindi la proprietà è vera anche dopo la rimozione di v_1 (non sto agendo sulle distanze con dequeue quindi se era vera prima continua a esserlo).

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

PASSO INDUTTIVO

2. Per ipotesi induttiva assumiamo che la proprietà sia vera prima di eseguire ***Enqueue*(Q,v)** e sia $v_{t'}$ l'ultimo elemento in coda prima dell'operazione.

Prima di eseguire ***Enqueue*(Q,v)** è stato posto

$$v.d = u.d + 1 \text{ (riga 17)}$$

con $u = First(Q) = v_1$ primo elemento in coda (riga 13).

Quindi dopo l'operazione

$$v_t.d = v.d = v_1.d + 1$$

Inoltre, per ipotesi induttiva:

$$v_{t'}.d \leq v_1.d + 1$$

e dunque:

$$v_{t'}.d \leq v_t.d\ .$$

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Possiamo ora dimostrare la correttezza di BFS

BFS visita tutti i vertici raggiungibili da **s** e quando termina **v.d = d(s,v)** per ogni vertice **v** del grafo.

Inoltre per ogni vertice **v** $\neq$ **s** raggiungibile da **s**

- **u = v.p** $\neq$ **nil**,
- **(u,v)** $\in$ **E** e
- **d(s,v) = d(s,u) + 1**.

BFS - Correttezza

DIMOSTRAZIONE

Se v non è raggiungibile da s allora $d(s,v) = \infty$ e $d.v = d(s,v) = \infty$ per la proprietà del limite superiore.

Quindi a $v.d$ non può venire assegnato un valore finito.

Per induzione non ci può essere un primo vertice v che viene inserito nella coda dopo aver assegnato

$$v.d = u.d + 1 = \infty .$$

Quindi i vertici non raggiungibili non vengono visitati dall'algoritmo.

BFS - Correttezza

Dimostriamo quindi che per ogni vertice $v \in V$ raggiungibile da s esiste uno ed un solo punto dell'esecuzione di ***BFS*** in cui:

a) a ***v.color*** viene assegnato il valore ***grigio***;

b) a ***v.d*** viene assegnato il valore $d(s,v)$;

c) se $v \neq s$ a ***v.p*** viene assegnato un vertice u tale che $d(s,v) = d(s,u) + 1$, mentre se $v = s$ a ***s.p*** viene assegnato ***nil***;

d) il vertice v viene aggiunto alla coda Q.

BFS - Correttezza

- Dopo l'inizializzazione nessun vertice viene più colorato di ***bianco***
- un vertice può essere colorato di ***grigio*** soltanto se era ***bianco***

ogni vertice può venir colorato di ***grigio*** una sola volta.

Tali azioni non possono quindi essere eseguite più di una volta

rimane quindi da dimostrare che esiste almeno un punto in cui vengono eseguite.

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Durante l'inizializzazione:

a) a ***s.color*** viene assegnato il valore ***grigio***;

b) a ***s.d*** viene assegnato il valore 0;

c) ad ***s.p*** viene assegnato ***nil***;

d) il vertice ***s*** viene aggiunto alla coda ***Q***.

quindi la cosa è vera per la sorgente ***s*** **(PASSO BASE)**.

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Sia $V_k = \{v \in V : d(s,v) = k\}$ l'insieme dei vertici a distanza $k < \infty$ da s.

Dimostriamo per induzione su k che per ogni vertice $v \in V_k$ esiste un punto dell'esecuzione di ***BFS*** in cui vengono eseguite le azioni:

a) a ***v.color*** viene assegnato il valore ***grigio***;

b) a ***v.d*** viene assegnato il valore $d(s,v)$;

c) se $v \neq s$ a ***v.p*** viene assegnato un vertice u tale che $d(s,v) = d(s,u) + 1$, mentre se $v = s$ a ***s.p*** viene assegnato ***nil***;

d) il vertice v viene aggiunto alla coda Q.

BFS - Correttezza

PASSO INDUTTIVO

Cominciamo ad osservare che

- durante l'esecuzione del ciclo **while** la coda ***Q*** non è mai vuota
- non appena un vertice ***v*** viene messo in coda, i valori di ***v.d*** e di ***v.p*** non possono più cambiare.

Di conseguenza, se $(s = v_0, v_1, \ldots, v_t)$ è l'ordine in cui i vertici vengono inseriti nella coda allora

$$v_i.d \leq v_{i+1}.d$$

per la proprietà della coda

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Sia ora $v \in V_k$ per un $k > 0$.

Per ipotesi induttiva tutti i vertici in V_{k-1} vengono visitati e, per l'ordine di inserimento nella coda, se il vertice v viene visitato esso viene visitato dopo tutti i vertici in V_{k-1}.

Siccome $d(s,v) = k$ esiste un cammino minimo di lunghezza k da s a v.

Esiste quindi almeno un nodo $u \in V_{k-1}$ tale che $uv \in E$.

Supponiamo che u sia il primo di tali nodi a venire visitato e inserito nella coda.

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS - Correttezza

Ad un certo punto ***u*** deve diventare il primo della coda e quindi la sua lista delle adiacenze viene percorsa e il vertice ***v*** viene scoperto.

Il vertice ***v*** non può essere scoperto prima perché non è adiacente a nessun vertice a distanza minore di ***k*-1** da ***s*** ed ***u*** è il primo vertice a distanza ***k*-1** che viene inserito in coda tra tutti quelli a cui ***v*** è adiacente.

A questo punto vengono eseguite le quattro azioni richieste:

a) a ***v.color*** viene assegnato il valore ***grigio***;

b) a ***v.d*** viene assegnato il valore ***d*(*s*,*v*) = *d*(*s*,*u*) + 1 = *k***;

c) a ***v.p*** viene assegnato $u \in V_{k-1}$;

d) il vertice ***v*** viene aggiunto alla coda ***Q***.

Algorithm 1: Visita in Ampiezza (BFS)

```
1  BFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.d ← ∞;
5  |   u.p ← nil;
6  end
7  s.color ← GRIGIO;
8  s.d ← 0;
9  u.p ← nil;
10 Q ← ∅;
11 ENQUEUE(Q, s);
12 while Q ≠ ∅ do
13 |   u ← Dequeue(Q);
14 |   foreach v ∈ G.adj[u]] do
15 |   |   if v.color ≠ BIANCO then
16 |   |   |   v.color ← GRIGIO;
17 |   |   |   v.d ← u.d + 1;
18 |   |   |   v.p ← u;
19 |   |   |   Enqueue(Q, v);
20 |   |   end
21 |   end
22 |   u.color ← NERO;
23 end
```

BFS e albero di visita

Eseguendo una BFS su un grafo G e utilizzando le informazioni ricavate in u.p è possibile costruire un **sottografo dei predecessori o albero di visita in ampiezza**

DFS

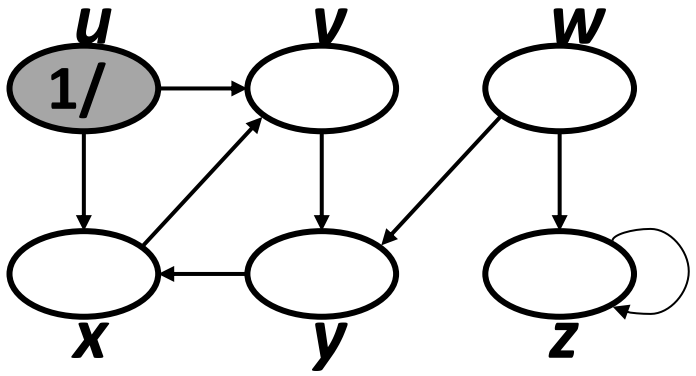
Algorithm 2: Visita in Profondità (DFS)

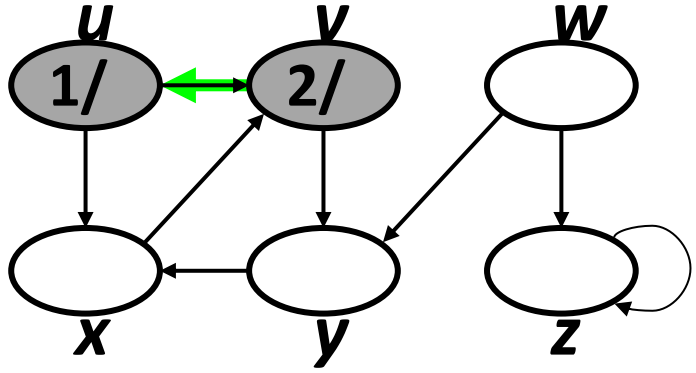
```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

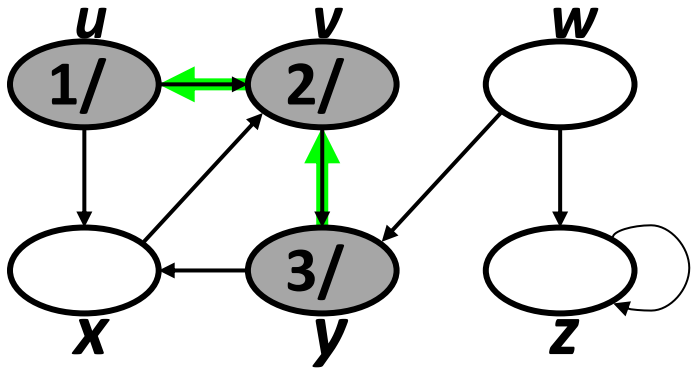


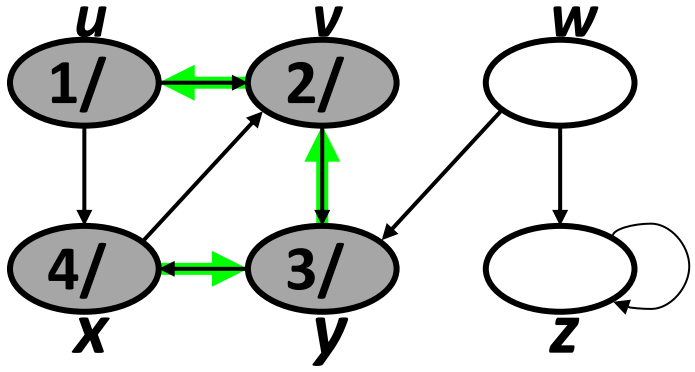


DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

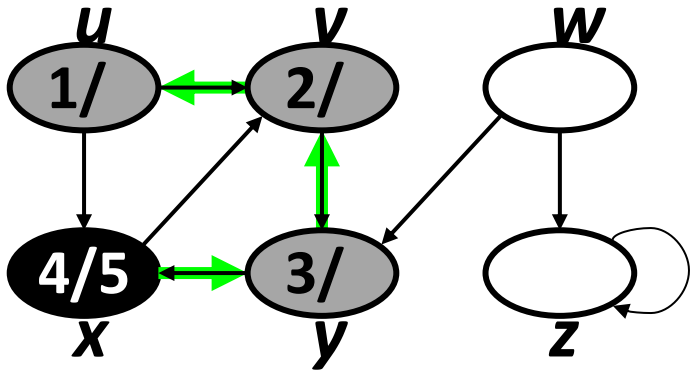


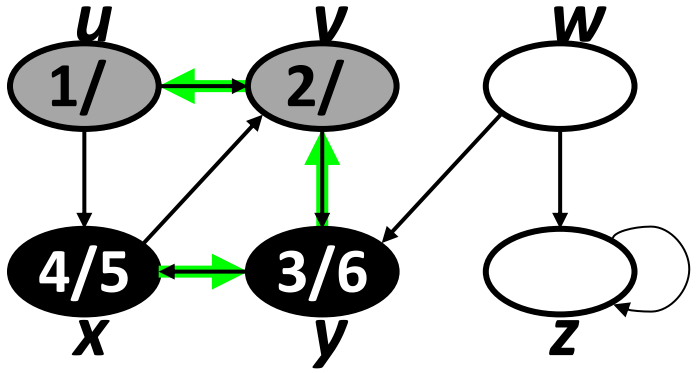


DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

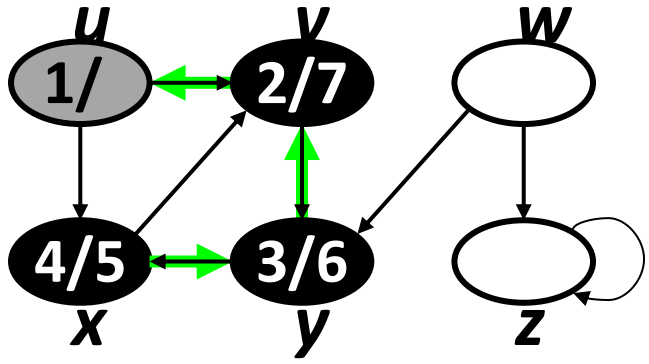


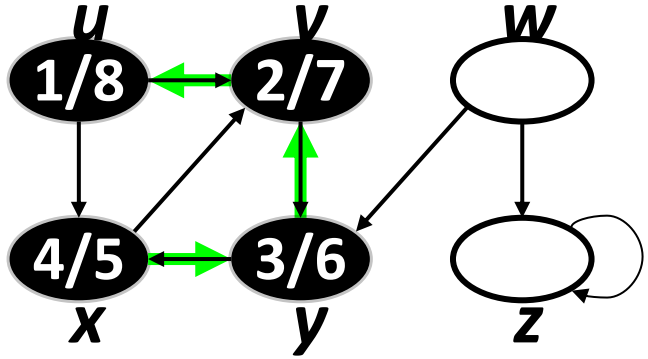


DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

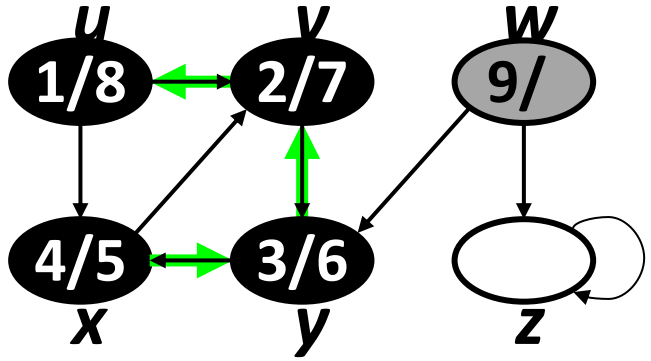


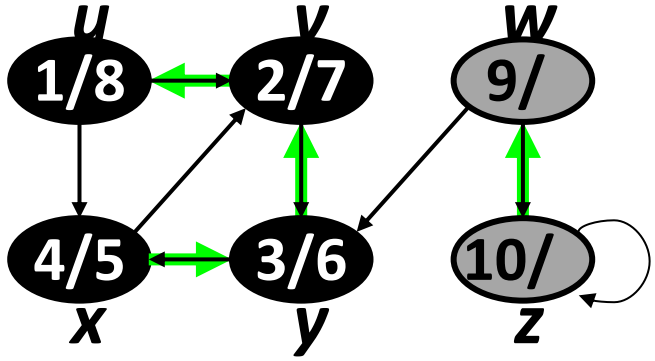


DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

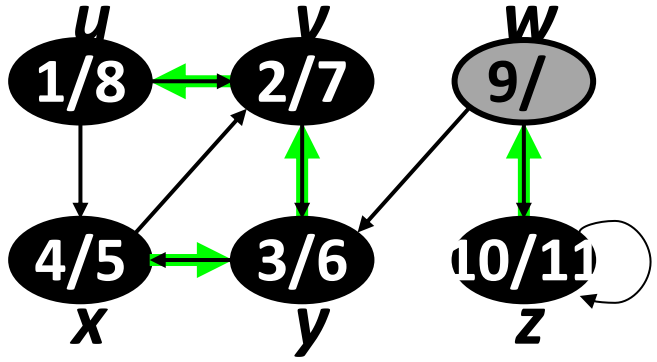


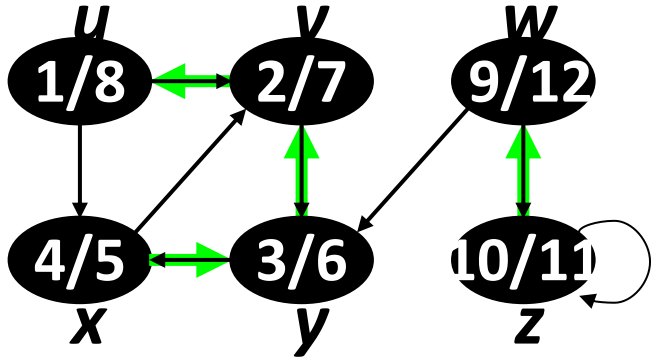


DFS

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```





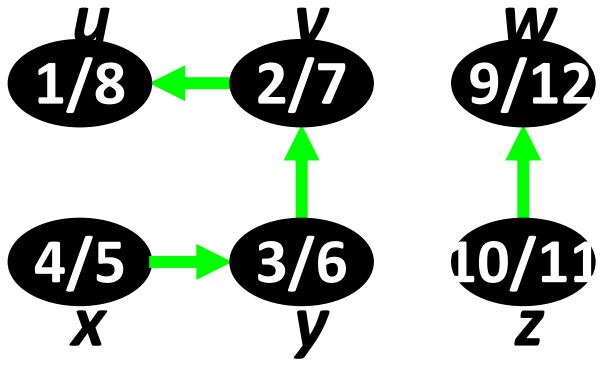
DFS

A differenza della visita in ampiezza, in cui i puntatori al predecessore definiscono un albero, nella visita in ampiezza definiscono un insieme di alberi, la *foresta di visita in profondità*.

La visita in profondità pone due *marcatempi* su ogni vertice ***u***.

- Il primo marcatempo (tempo di scoperta) ***u.d*** viene posto quando il vertice viene scoperto e colorato ***grigio***,
- Il secondo marcatempo (tempo di completamento) ***u.f*** quando il vertice è completato e viene colorato di nero.

Tali marcatempi sono utili in molti algoritmi derivati dalla visita in profondità e permettono di comprendere meglio il funzionamento della visita stessa.



DFS - Complessità

Senza le chiamate a ***DFS-Visit*** i due cicli **for** di ***DFS*** richiedono un tempo $O(|V|)$.

La funzione ***DFS-Visit*** viene richiamata solo su vertici bianchi che vengono subito colorati di grigio.

- al più una sola volta per ogni vertice.

Il ciclo interno percorre la lista delle adiacenze del vertice di invocazione.

Siccome la somma delle lunghezze di tutte le liste delle adiacenze è $\Theta(|E|)$, l'intero algoritmo ha complessità $O(|V| + |E|)$.

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

DFS – Proprietà delle Parentesi

Se si rappresenta

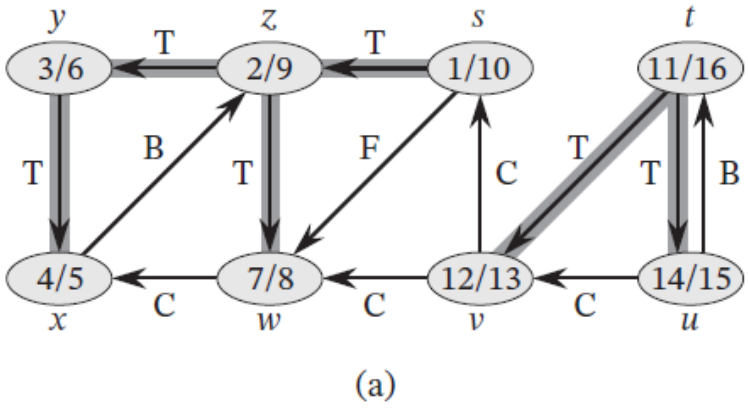
- la scoperta di ogni vertice ***u*** con una parentesi aperta $(_u$ e
- il completamento con una parentesi chiusa $_u)$

si ottiene una sequenza bilanciata di parentesi.

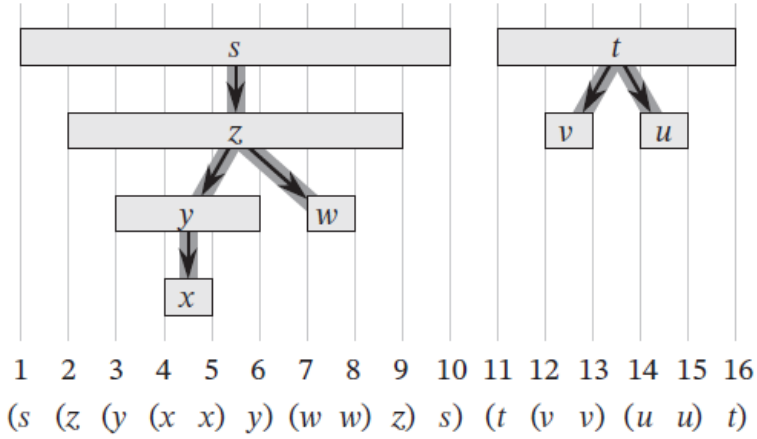
Ossia, per ogni coppia di vertici ***u*** e ***v***, ci sono quattro possibilità:

1. $(_u \ldots{}_u) \ldots (_v \ldots{}_v)$;
2. $(_v \ldots{}_v) \ldots (_u \ldots{}_u)$;
3. $(_u \ldots (_v \ldots{}_v) \ldots{}_u)$;
4. $(_v \ldots (_u \ldots{}_u) \ldots{}_v)$;

DFS – Proprietà delle Parentesi



(a)



(b)

DFS – Proprietà delle Parentesi

Teorema delle Parentesi

In una qualsiasi DFS di un grafo G = (V, E) (orientato o non orientato), per ogni coppia di nodi u, v è soddisfatta una e una sola delle seguenti tre condizioni

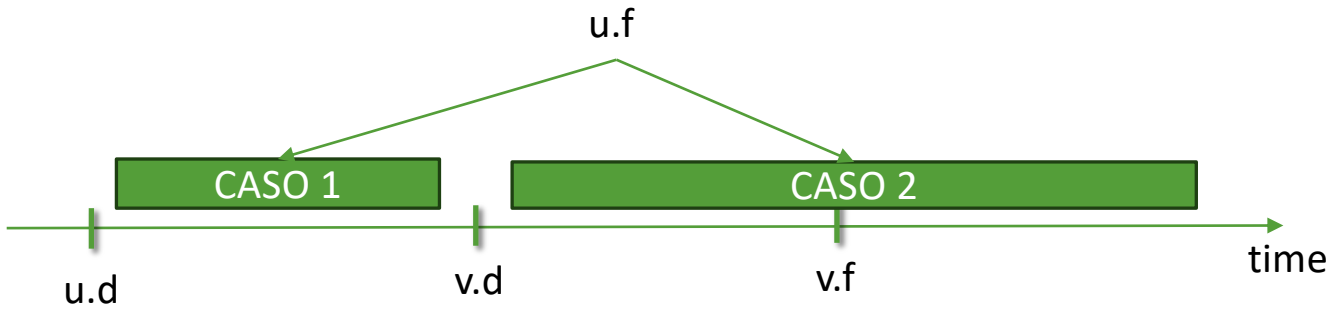
1. Gli intervalli [u.d, u.f] e [v.d, v.f] sono completamente disgiunti; u e v non sono discendenti uno dell'altro nella foresta delle visita in profondità
2. L'intervallo [u.d, u.f] è interamente contenuto nell' intervallo [v.d, v.f] e u è un discendente di v in un albero di visita in profondità
3. L'intervallo [v.d, v.f] è interamente contenuto nell' intervallo [u.d, u.f] e v è un discendente di u in un albero di visita in profondità

DFS – Proprietà delle Parentesi

DIMOSTRAZIONE

Assumiamo che ***u.d < v.d*** (il caso v.d < u.d è simmetrico e si applicano gli stessi argomenti)

- **OSS**. u.f > u.d e v.f >v.d



DFS – Proprietà delle Parentesi

Caso 1 – u.f < v.d

Considerando che

1. u.d < u.f
2. v.d < v.f

Ne segue che u.d < u.f < v.d < v.f -> gli intervalli sono disgiunti.

Se gli intervalli sono disgiunti ne segue che nessuno dei due nodi è stato scoperto mentre l'altro era grigio (i.e., v era ancora bianco quando u è diventato nero)

Ne segue che nella foresta della visita in profondità non esiste un cammino da u a v e il caso 1 è dimostrato.

DFS – Proprietà delle Parentesi

CASO 2 –u.f > v.d

- Se ***u.f > v.d***, allora quando ***v*** viene scoperto ***u*** è grigio
- poiché ***v*** è stato scoperto dopo di ***u***, la sua lista delle adiacenze verrà completamente esplorata prima di riprendere l'esplorazione di quella di ***u***.
- Quindi ***v*** viene completato prima di ***u*** e pertanto

DFS – Teorema del Cammino Bianco

Teorema del Cammino Bianco

In una foresta di visita in profondità di un grafo G = (V, E) (orientato o non orientato), il vertice v è discendente del vertice u se e solo se, al tempo u.d quando u viene scoperto il vertice v può essere raggiunto da u lungo un cammino che è formato esclusivamente da vertici bianchi

DFS – Teorema del Cammino Bianco

DIMOSTRAZIONE (⇒)

Sia ***v*** un discendente di ***u***

sia P= $x_0, \dots, x_k$ (con x_0=u e x_k=v) la sequenza dei vertici da ***u*** a ***v*** nel ramo dell'albero della foresta di visita che connette ***u*** a ***v***.

Siccome x_{i+1} viene scoperto visitando la lista delle adiacenze di x_i esiste l'arco $x_i x_{i+1}$ ed inoltre x_i viene scoperto prima di x_{i+1}.

Quindi **P** è un cammino tale che, quando $u = x_0$ viene scoperto, i vertici $x_1, \dots, x_k$ non sono ancora stati scoperti e dunque sono bianchi.

Algorithm 2: Visita in Profondità (DFS)

```
1  DFS(G, s)
2  foreach u ∈ G.V do
3  |   u.color ← BIANCO;
4  |   u.p ← nil;
5  end
6  time ← 0;
7  foreach u ∈ G.V do
8  |   if u.color = BIANCO then
9  |   |   DSF_VISIT(G, u);
10 |   end
11 end

12 DFS_VISIT(G, u)
13 time ← time + 1;
14 u.d ← time;
15 u.color ← GRIGIO;
16 foreach v ∈ G.adj[u]] do
17 |   if v.color = BIANCO then
18 |   |   v.p ← u;
19 |   |   DSF_VISIT(G, v);
20 |   end
21 end
22 time ← time + 1;
23 u.f ← time;
24 u.color ← NERO;
```

DFS – Teorema del Cammino Bianco

DIMOSTRAZIONE (⇐)

Supponiamo ora che quando ***u*** viene scoperto esista un cammino bianco **(*u* = x_0, ... , x_k = *v*)** da ***u*** a ***v***.

Poiché ***v*** viene scoperto dopo ***u***, per la proprietà delle parentesi o

$$(_u \ldots\ _u) \ldots (_v \ldots\ _v)$$

oppure

$$(_u \ldots (_v \ldots\ _v) \ldots\ _u).$$

Se $(_u \ldots (_v \ldots\ _v) \ldots\ _u)$ allora ***v*** è discendente di ***u*** per la proprietà dei discendenti.

DFS – Teorema del Cammino Bianco

DIMOSTRAZIONE (⇐)

Supponiamo per assurdo che $(_u$ .... $_u)$ .... $(_v$ .... $_v)$ e quindi che ***v*** non sia discendente di ***u***.

Possiamo anche assumere che ***v*** sia il primo vertice del cammino bianco che non è discendente di ***u***.

Sia $\boldsymbol{w = x_{k-1}}$ il vertice che precede ***v*** nel cammino bianco.

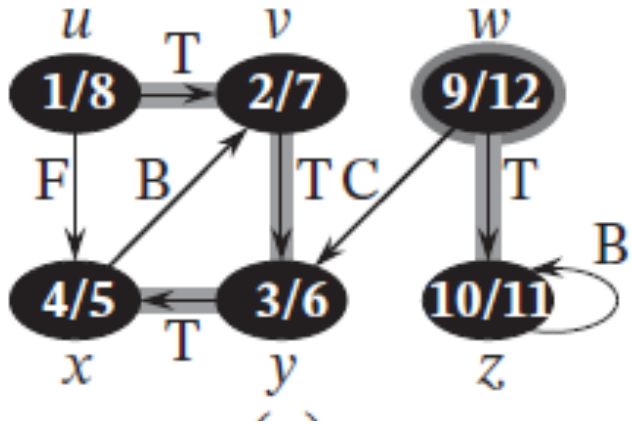
Siccome ***w*** è discendente di ***u***, per la proprietà dei discendenti si ha $(_u$ .... $(_w$ .... $_w)$ .... $_u)$ .... $(_v$ .... $_v)$.

Ma questo è assurdo, perché ***v*** è nella lista delle adiacenze di ***w*** e quindi deve essere stato scoperto prima che ***w*** sia completato.

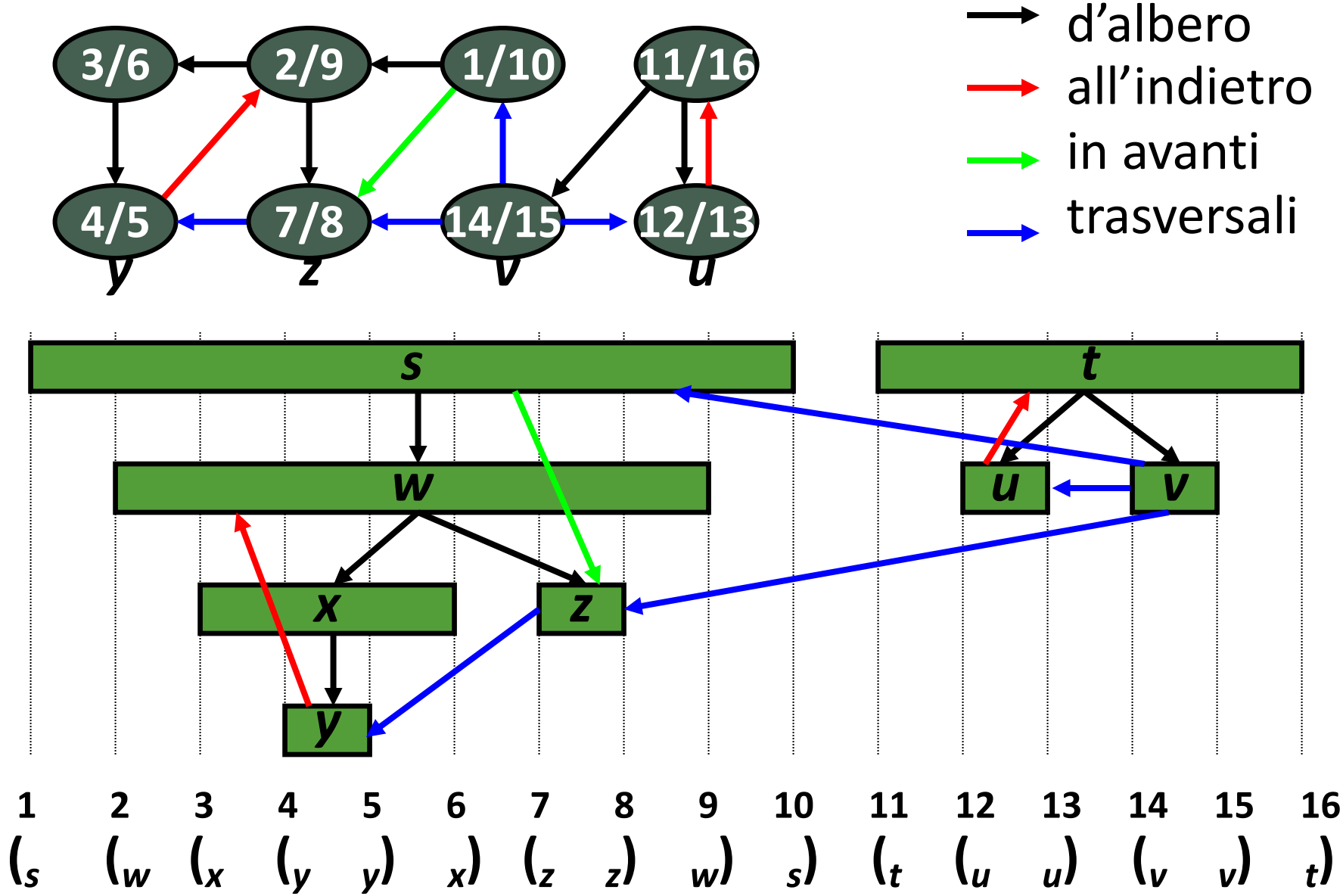
DFS

Con la visita in profondità gli archi si possono classificare in:

- *archi d'albero (T)*: archi ***uv*** con ***v*** scoperto visitando le adiacenze di ***u;***
- *archi all'indietro (B)*: archi ***uv*** con ***u = v*** oppure ***v*** ascendente di ***u*** in un albero della foresta di visita in profondità;
- *archi in avanti (F)*: archi ***uv*** con ***v*** discendente di ***u*** in un albero della foresta;
- *archi trasversali (C)*: archi ***uv*** in cui ***v*** ed ***u*** appartengono a rami o alberi diversi della foresta.



Se un arco soddisfa le condizioni per appartenere a più di una categoria esso viene classificato in quella che compare per prima nell'ordine in cui le abbiamo elencate.


3/6
2/9
1/10
11/16
4/5
7/8
14/15
12/13
y
z
v
u
d'albero
all'indietro
in avanti
trasversali
s
t
w
u
v
x
z
y
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (w (x (y y) x) (z z) w) s) (t (u u) (v v) t)

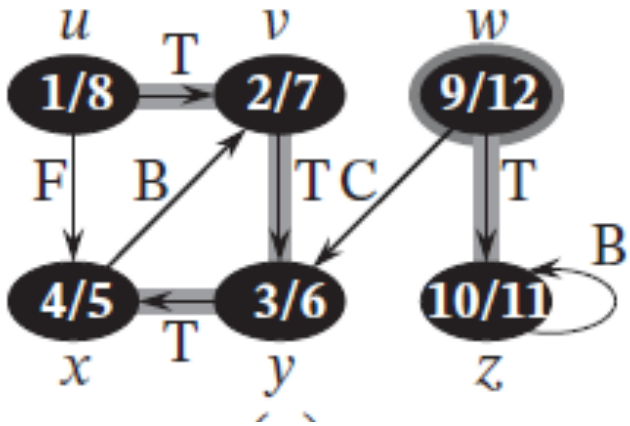
DFS – Classificazione degli archi

E' interessante osservare che l'algoritmo DFS contiene tutte le informazioni necessarie a classificare un arco quando lo analizziamo. In particoalare

- Consideriamo la prima volta che incontriamo un arco (u,v)
  - Se v è bianco allora (u, v) è un arco dell'albero (T)
  - se v è grigio allora (u, v) è un arco all'indietro (B)
  - se v è nero allora (u, v) è un arco in avanti (F) o trasversale (C)
    - se u.d > v.d allora è un arco trasversale (C)
    - se u.d < v.d allora in avanti (F)

Teorema

In una DFS di un grafo G = (V, E) non orientato, gli archi di G possono essere solo archi dell'albero (T) o archi all'indietro.



Ordinamento Topologico

DEFINIZIONE

Un *Ordinamento Topologico* di un DAG (*Directed Acyclic Graph*) G = (V, E) è un ordinamento totale di tutti i suoi vertici tale che, se G contiene un arco (u, v), allora u appare prima di v nell'ordinamento

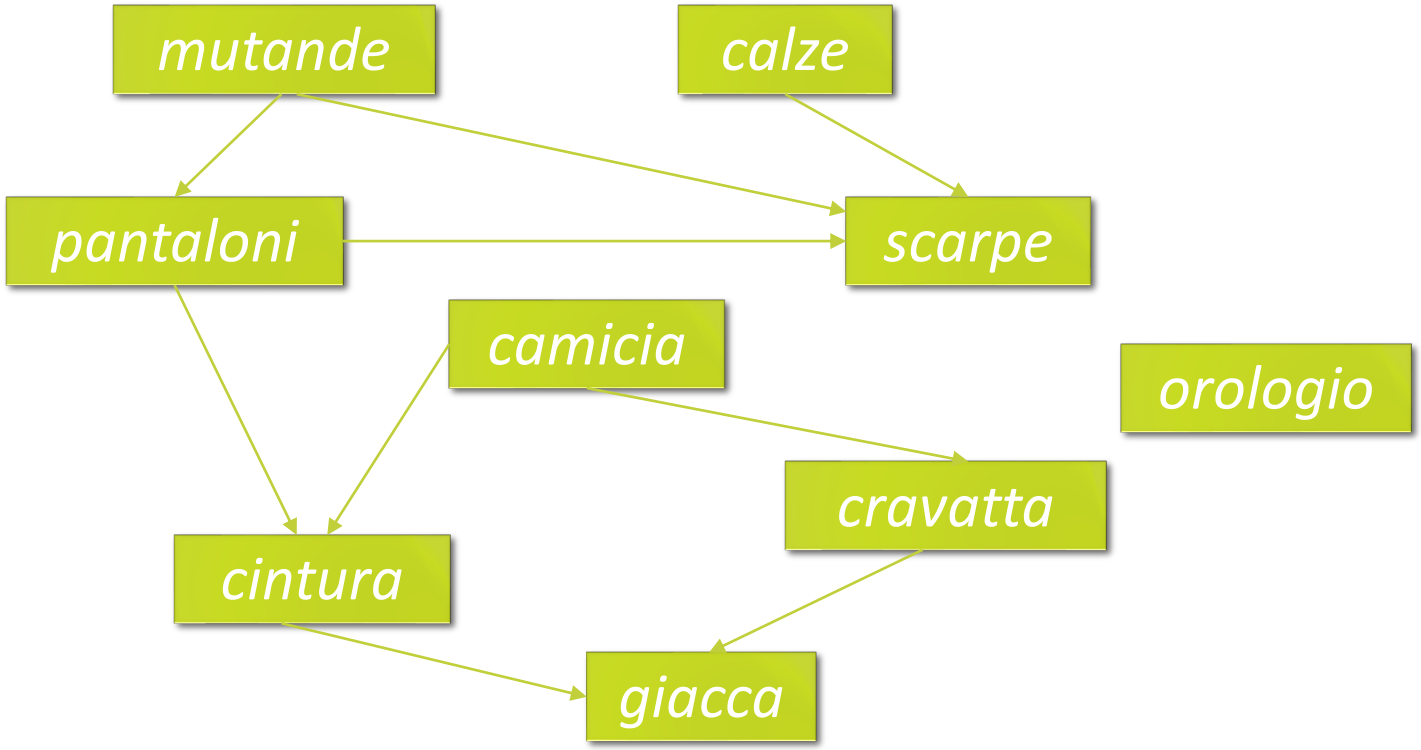
L'ordinamento topologico si usa per determinare un ordine in cui eseguire un insieme di attività in presenza di vincoli di propedeuticità.

Esempi

- l'ordine con cui indossare gli indumenti quando ci si veste,
- l'ordine con cui sostenere gli esami,
- l'ordine con cui assemblare le parti di una automobile, ecc.

Ordinamento Topologico

Esempio: Il DAG dei vestiti

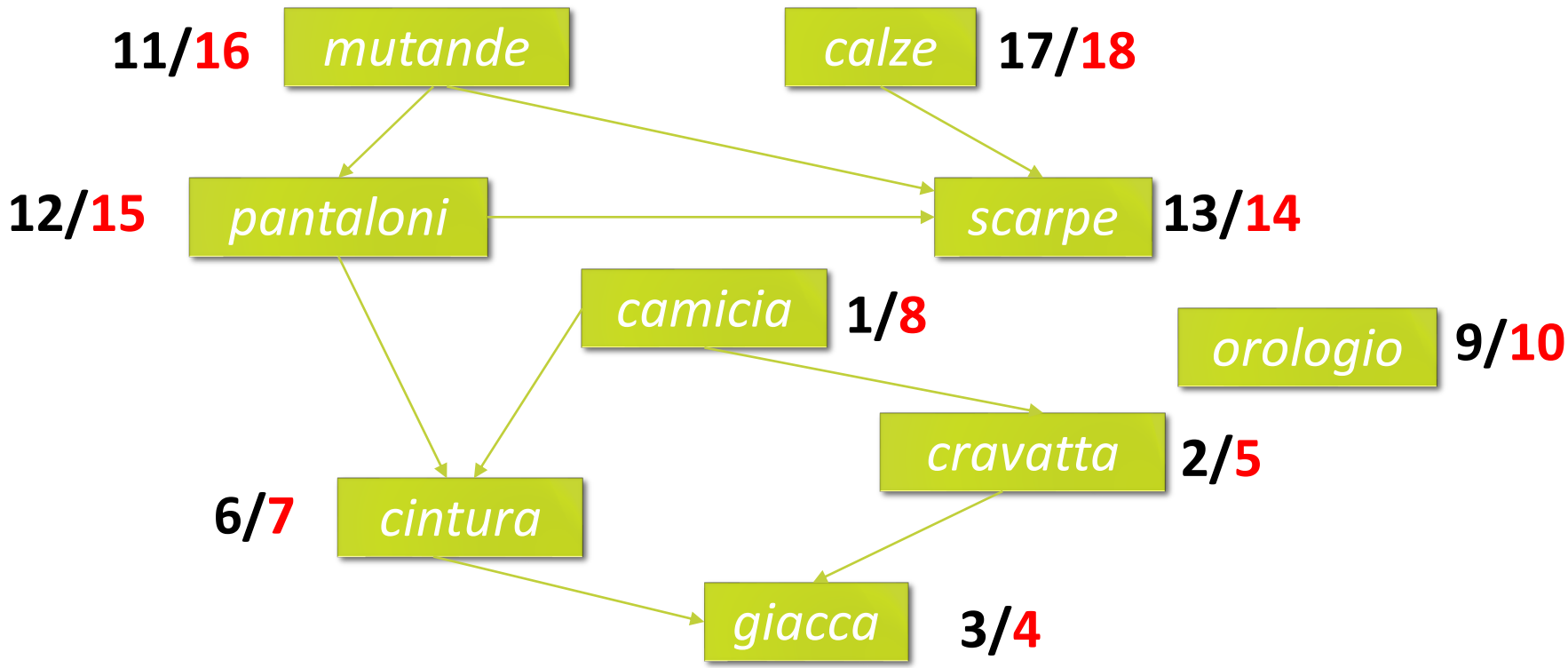


Ordinamento Topologico

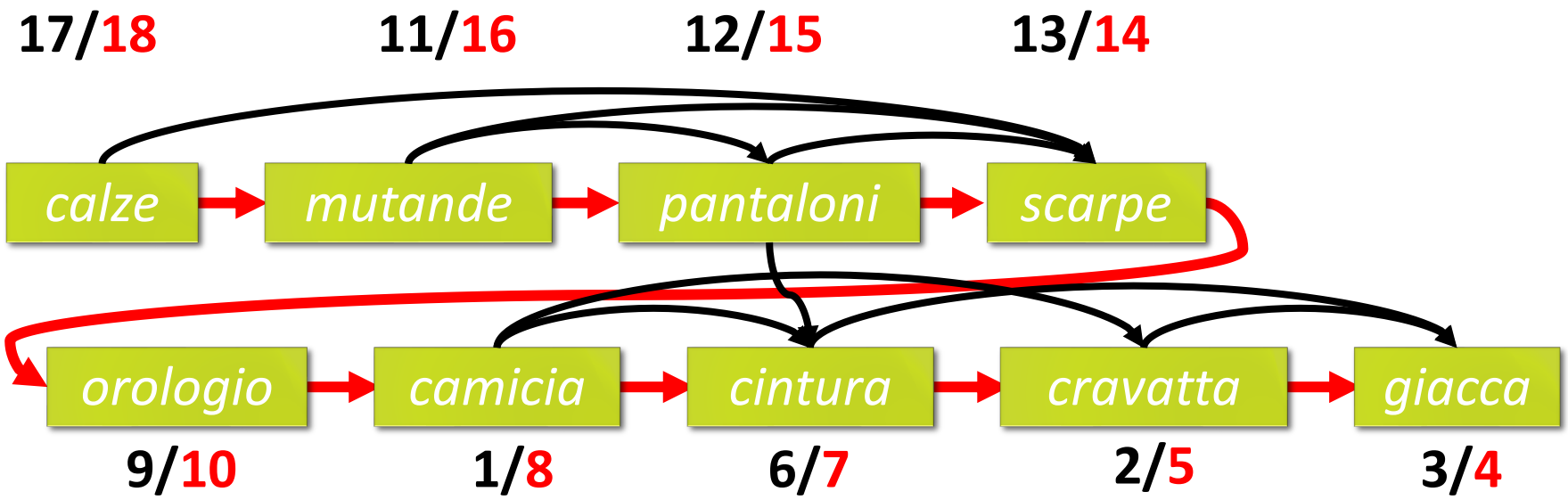
Topological-Sort(*G*)

1. Chiama DFS(*G*) per calcolare i tempi di completamento *v.f* per ciascun vertice *v*
2. Una volta completata la visita di un vertice, inserisce il vertice in testa a una lista concatenata
3. return la lista concatenata dei vertici

Ordinamento Topologico



Ordinamento Topologico



OSS. l'ordinamento topologico segue l'ordinamento di v.f dal più grande al più piccolo

Ordinamento Topologico

Teorema.

Un grafo orientato è aciclico (un ***DAG***) se, e solo se, nella visita in profondità non si trova nessun arco all'indietro.

Dimostrazione. (⇒)

Se in una visita in profondità si trova un arco all'indietro ***vu***, allora tale arco aggiunto al cammino da ***u*** a ***v*** (che esiste in quanto ***v*** è discendente di ***u***) forma un ciclo.

Ordinamento Topologico

Teorema.

TOPOLOGICAL-SORT produce un ordinamento topologico del DAG che gli viene fornito in input

DIMOSTRAZIONE

Basta dimostrare che per ogni arco ***uv*** il vertice ***v*** viene finito prima del vertice ***u***.

Quando l'arco ***uv*** viene esplorato, il vertice ***u*** è grigio mentre il vertice ***v*** non può essere grigio, altrimenti ***uv*** sarebbe un arco all'indietro.

Se ***v*** è nero, allora è gia stato completato, mentre ***u*** non lo è ancora.

Se ***v*** è bianco, allora è discendente di ***u*** per la proprietà del cammino bianco e viene completato prima di ***u*** per la proprietà delle parentesi.

Ordinamento Topologico

Teorema.

Un grafo orientato è aciclico (un ***DAG***) se, e solo se, nella visita in profondità non si trova nessun arco all'indietro.

Dimostrazione (⇐)

Viceversa, supponiamo che il grafo abbia un ciclo.

Sia ***v*** il primo vertice del ciclo ad essere scoperto e sia ***uv*** l'arco del ciclo che entra in ***v***.

Quando ***v*** viene scoperto, esiste un cammino bianco da ***v*** ad ***u*** e quindi, per la proprietà del cammino bianco, ***u*** è discendente di ***v***.

Di conseguenza ***uv*** è un arco all'indietro.

Componenti fortemente connesse

DEFINIZIONE

Una *componente fortemente connessa* (cfc) di un grafo orientato $G = (V,E)$ è un insieme massimale di vertici $U \subseteq V$ tale che, per ogni $u,v \in U$, esiste un cammino da u a v ed un cammino da v ad u.

Componenti fortemente connesse

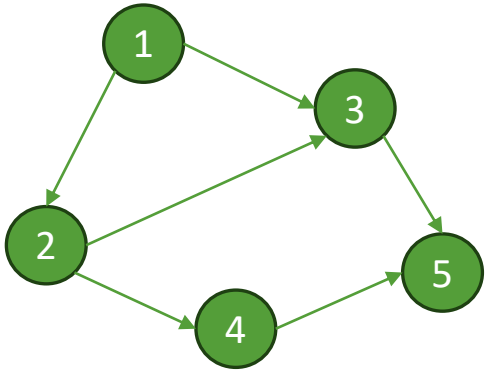
STRONGLY CONNECTED COMPONENTS(G)

1. Chiama DFS(G) per calcolare i tempi di completamento u.f per ciascun vertice u
2. Crea il grafo trasposto G^T di G
3. Chiama DFS(G^T) ma nel ciclo principale di DFS considera i vertici in ordine decrescente rispetto ai tempi u.f calcolati al punto 1
4. Ritorna i vertici di ciascun albero della foresta di visita in profondità che è stata prodotta alla riga 3 come una singola componente fortemente connessa

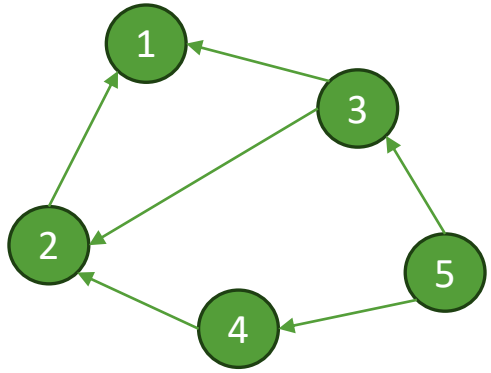
Grafo Trasposto G^T

Dato un grafo G = (V, E), il grafo trasposto G^T = (V, E^T) è il grafo avente gli stessi vertici di G e un insieme di archi definito come

- E^T ={(u, v) : (v, u) ∈ E}



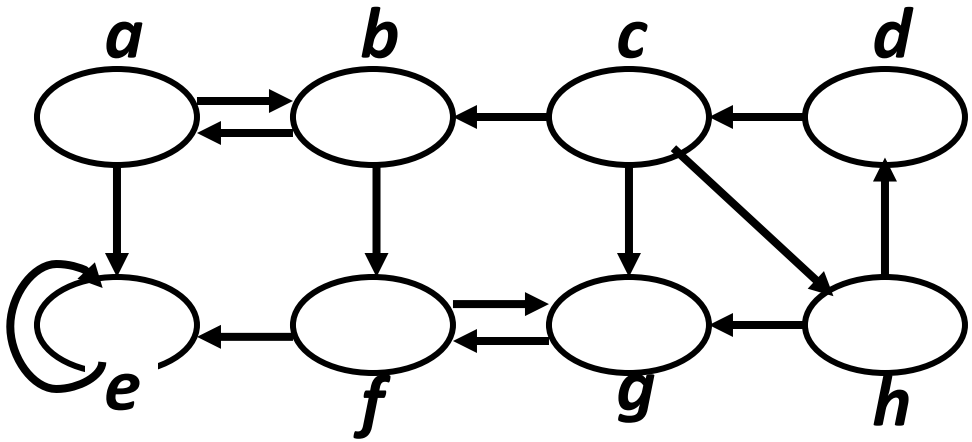
G



G^T

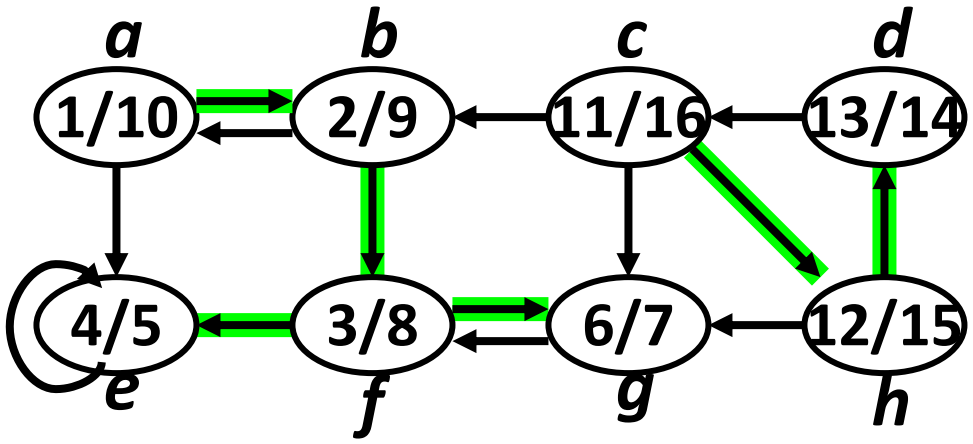
Componenti fortemente connesse

Consideriamo il grafo G riportato in figure



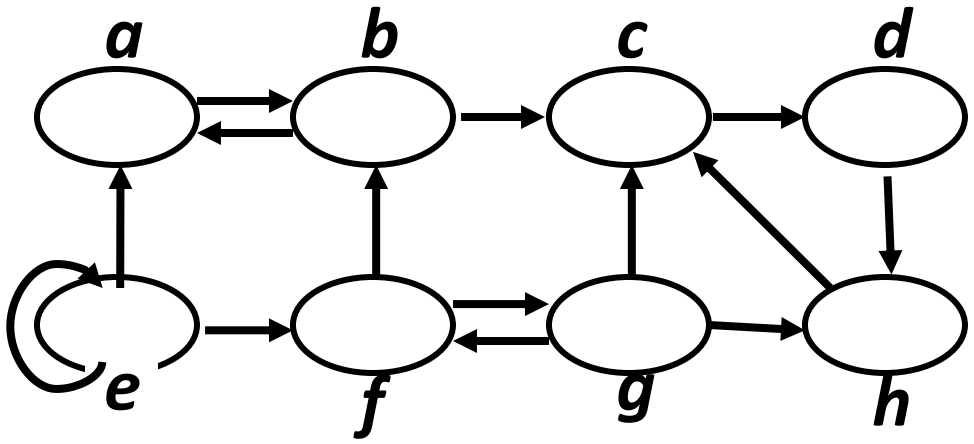
Componenti fortemente connesse

Passo 1 - Chiama DFS(G) per calcolare i tempi di completamento u.f per ciascun vertice u



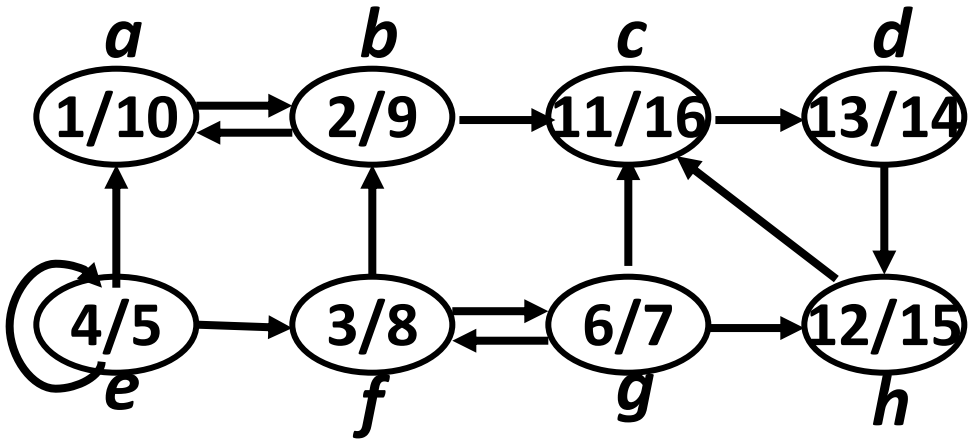
Componenti fortemente connesse

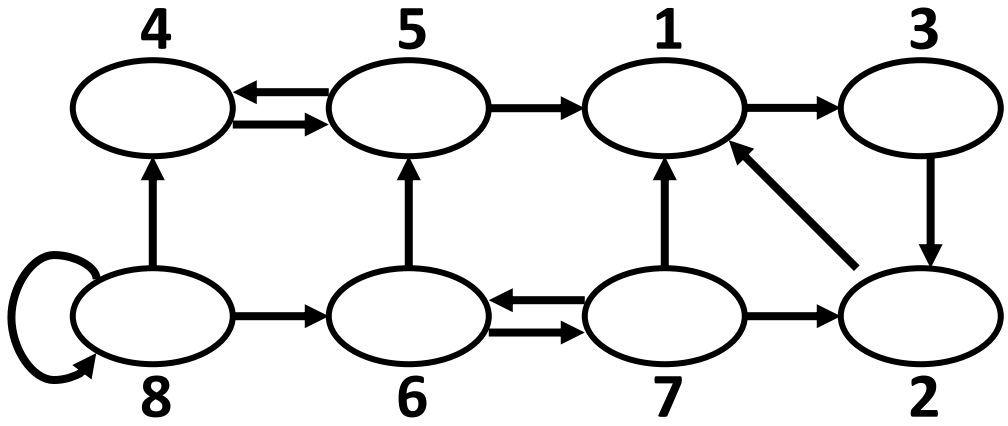
Passo 2 - Crea il grafo trasposto G^T di G



Componenti fortemente connesse

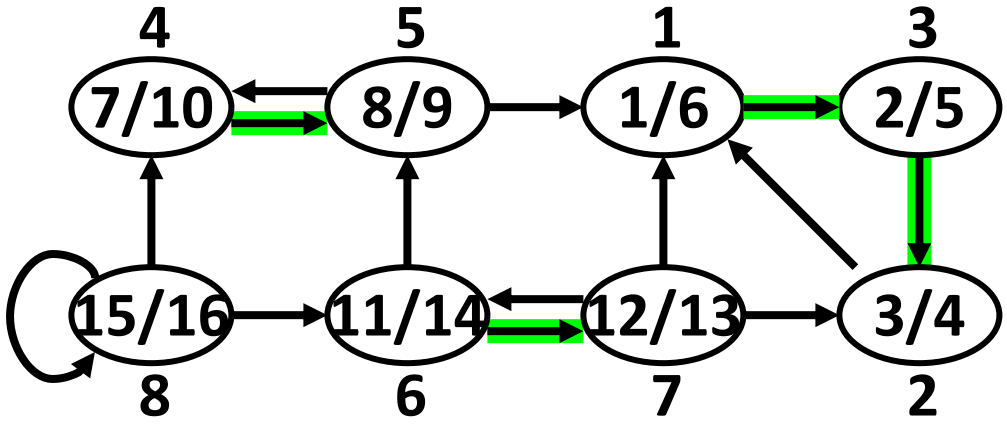
Passo 3 - Chiama DFS(G^T) ma nel ciclo principale di DFS considera i vertici in ordine decrescente rispetto ai tempi u.f calcolati al punto 1





Componenti fortemente connesse

Passo 3 - Chiama DFS(G^T) ma nel ciclo principale di DFS considera i vertici in ordine decrescente rispetto ai tempi u.f calcolati al punto 1



Componenti fortemente connesse

Passo 4 - Ritorna i vertici di ciascun albero della foresta di visita in profondità che è stata prodotta alla riga 3 come una singola componente fortemente connessa