Algoritmi su grafi

I grafi sono strutture dati che vengono usate estensivamente in informatica.

Ci sono migliaia di problemi computazionali che sono importanti per le applicazioni e che si possono modellare con i grafi.

Noi ci limiteremo a studiare i modi per rappresentare i grafi e la soluzione efficiente dei problemi basilari.

Nomenclatura dei grafi

Un *grafo* **G = (*V*,*E*)** è costituito da un *insieme di vertici* ***V*** ed un *insieme di archi* ***E*** ciascuno dei quali connette due vertici in ***V*** detti *estremi* dell'arco.

Un grafo è *orientato* quando vi è un ordine tra i due estremi degli archi. In questo caso il primo estremo si dice *coda* ed il secondo *testa*.

Un *cappio* è un arco i cui estremi coincidono.

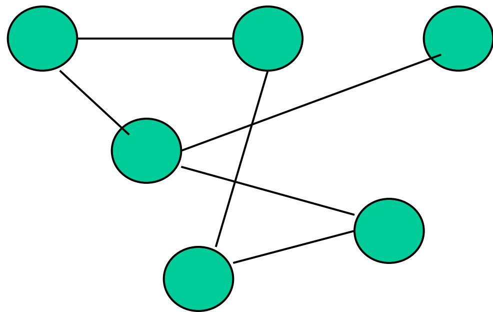
Un grafo non orientato è *semplice* se non ha cappi e non ci sono due archi con gli stessi due estremi.

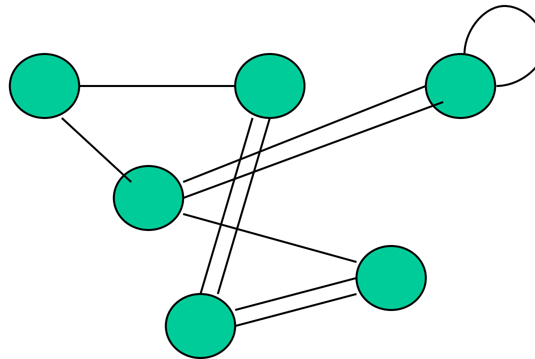
Un grafo orientato è *semplice* se non ci sono due archi con gli stessi estremi iniziale e finale.

Un grafo non semplice viene detto *multigrafo*.

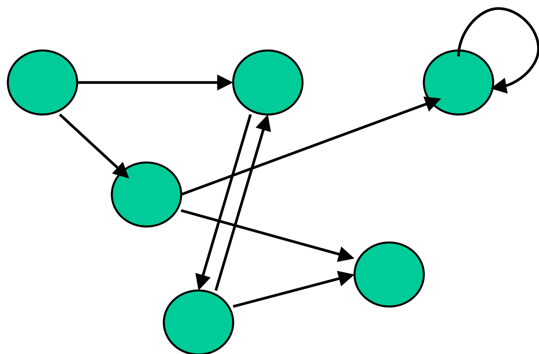
Salvo indicazione contraria noi assumeremo sempre che un grafo sia semplice.

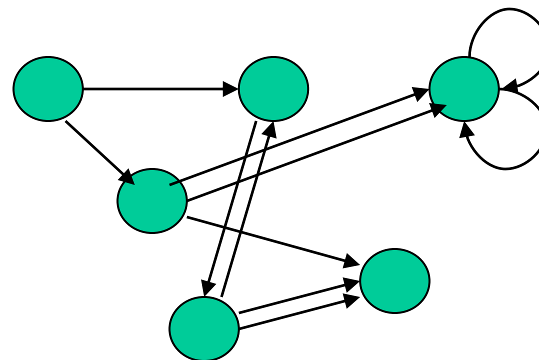
grafo semplice | multigrafo





non orientato





orientato

Se un grafo è semplice possiamo identificare un arco con la coppia dei suoi estremi: $\boldsymbol{e = uv \in E}$.

Quando $\boldsymbol{e = uv \in E}$ diciamo che l'arco $\boldsymbol{e}$ è <u>*incidente*</u> in $\boldsymbol{u}$ e in $\boldsymbol{v}$.

Se il grafo è orientato la coppia $\boldsymbol{uv}$ è ordinata.
In questo caso diciamo che l'arco $\boldsymbol{e}$ <u>*esce*</u> da $\boldsymbol{u}$ ed <u>*entra*</u> in $\boldsymbol{v}$.

Il <u>*grado*</u> $\boldsymbol{\delta}(\boldsymbol{v})$ del vertice $\boldsymbol{v}$ è il numero di archi incidenti in $\boldsymbol{v}$.

Se il grafo è orientato $\boldsymbol{\delta}(\boldsymbol{v})$ si suddivide in un <u>*grado entrante*</u> $\boldsymbol{\delta}^{-}(\boldsymbol{v})$ che è il numero di archi entranti in $\boldsymbol{v}$ ed un <u>*grado uscente*</u> $\boldsymbol{\delta}^{+}(\boldsymbol{v})$ che è il numero di archi uscenti da $\boldsymbol{v}$.

Se $\boldsymbol{uv} \in \boldsymbol{E}$ diciamo che il vertice $\boldsymbol{v}$ è <u>*adiacente*</u> al vertice $\boldsymbol{u}$. Se il grafo non è orientato la relazione di adiacenza è simmetrica e in tal caso diciamo che $\boldsymbol{u}$ e $\boldsymbol{v}$ sono adiacenti.

Un *cammino* di lunghezza $\boldsymbol{k}$ dal vertice $\boldsymbol{u}$ al vertice $\boldsymbol{v}$ in un grafo $\boldsymbol{G = (V,E)}$ è una sequenza di $\boldsymbol{k+1}$ vertici $\boldsymbol{x_0, x_1, \ldots, x_k}$ tali che $\boldsymbol{x_0 = u}$, $\boldsymbol{x_k = v}$ e $\boldsymbol{x_{i-1}x_i \in E}$ per ogni $\boldsymbol{i = 1,\ldots,k}$.

Il cammino $\boldsymbol{x_0}$ ha lunghezza $\boldsymbol{k = 0}$.
Se $\boldsymbol{k > 0}$ e $\boldsymbol{x_0 = x_k}$ diciamo che il cammino è *chiuso*.

Un *cammino semplice* è un cammino i cui vertici $\boldsymbol{x_0, x_1, \ldots, x_k}$ sono tutti distinti con la possibile eccezione di $\boldsymbol{x_0 = x_k}$ nel qual caso esso è un *ciclo*.
Un ciclo di lunghezza $\boldsymbol{k = 1}$ è un *cappio*.

Un *grafo aciclico* è un grafo che non contiene cicli.

Quando esiste almeno un cammino dal vertice ***u*** al vertice ***v*** diciamo che il vertice ***v*** è *accessibile* o *raggiungibile* da ***u***.

Un grafo non orientato si dice *connesso* se esiste almeno un cammino tra ogni coppia di vertici.

Le *componenti connesse* di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di accessibilità.

Un grafo orientato si dice *fortemente connesso* se esiste almeno un cammino da ogni vertice ***u*** ad ogni altro vertice ***v***.

Le *componenti fortemente connesse* di un grafo orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di mutua accessibilità.

Un *sottografo* del grafo $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ è un grafo $\boldsymbol{G'} = (\boldsymbol{V'},\boldsymbol{E'})$ tale che $\boldsymbol{V'} \subseteq \boldsymbol{V}$ ed

$$\boldsymbol{E'} \subseteq \{\, \boldsymbol{uv} : \boldsymbol{uv} \in \boldsymbol{E} \text{ e } \boldsymbol{u},\boldsymbol{v} \in \boldsymbol{V'} \,\}.$$

Il sottografo di $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ *indotto* da $\boldsymbol{V'} \subseteq \boldsymbol{V}$ è il grafo $\boldsymbol{G'} = (\boldsymbol{V'},\boldsymbol{E'})$ tale che

$$\boldsymbol{E'} = \{\, \boldsymbol{uv} : \boldsymbol{uv} \in \boldsymbol{E} \text{ e } \boldsymbol{u},\boldsymbol{v} \in \boldsymbol{V'} \,\}$$

Rappresentazione dei grafi

Vi sono due modi standard per rappresentare un grafo $G = (V,E)$: con le liste delle adiacenze o con la matrice delle adiacenze.

La rappresentazione di $G = (V,E)$ mediante *liste delle adiacenze* è costituita da una lista ***Adj*[*u*]** per ogni vertice $u \in V$.

La lista ***Adj*[*u*]** contiene i vertici adiacenti al vertice u (ossia tutti i vertici v tali che $uv \in E$).

Nella rappresentazione di $\boldsymbol{G = (V,E)}$ mediante <u>*matrice delle adiacenze*</u> assumiamo che i vertici siano numerati $\mathbf{1,2,\ldots,}\boldsymbol{n}$ in qualche modo arbitrario. La rappresentazione è quindi costituita da una matrice booleana $\boldsymbol{A = [a_{u,v}]}$ tale che

$$a_{u,v} = \begin{cases} 1 & \text{se}\ uv \in E \\ 0 & \text{se}\ uv \notin E \end{cases}$$

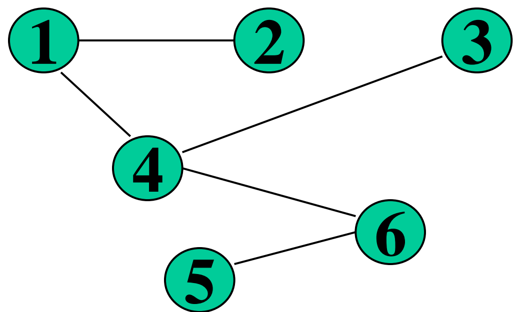
La rappresentazione di $\boldsymbol{G = (V,E)}$ mediante *liste delle adiacenze* richiede memoria per

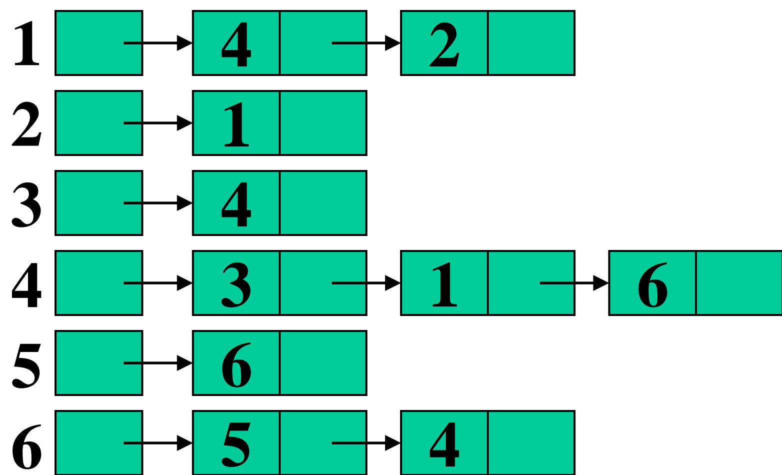
- $\boldsymbol{n = |V|}$ puntatori alle cime delle liste

- $\boldsymbol{m = |E|}$ elementi delle liste se il grafo è orientato

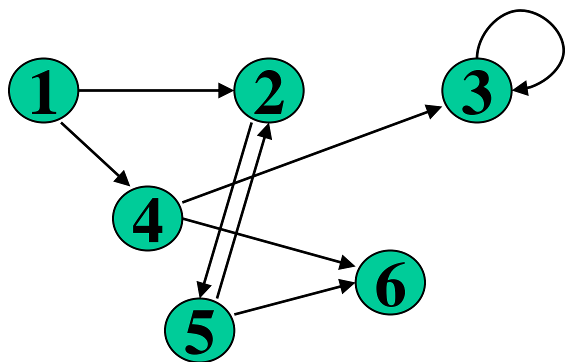
- $\boldsymbol{2m}$ elementi delle liste se non è orientato.

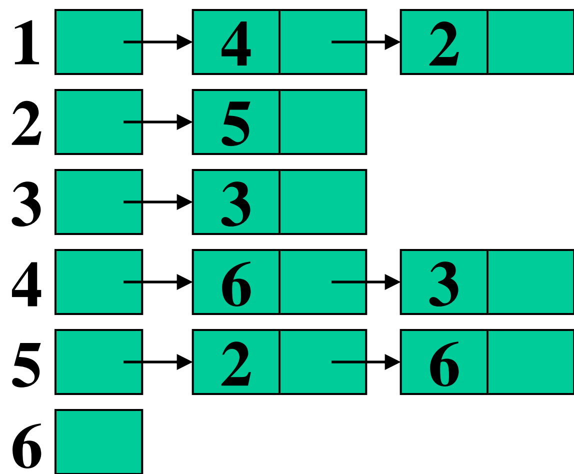
La rappresentazione di $\boldsymbol{G = (V,E)}$ mediante *matrice delle adiacenze* richiede memoria per una matrice $\boldsymbol{A}$ di $\boldsymbol{n \times n}$ valori booleani.





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	0	0
3	0	0	0	1	0	0
4	1	0	1	0	0	1
5	0	0	0	0	0	1
6	0	0	0	1	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	1	0	0	0
4	0	0	1	0	0	1
5	0	1	0	0	0	1
6	0	0	0	0	0	0

***Esercizio 1*.**

Sia $G = (V,E)$ un grafo orientato con $n = |V|$ vertici ed $m = |E|$ archi e supponiamo che puntatori e interi richiedano 32 bit.

Dire per quali valori di n ed m le liste delle adiacenze richiedono meno memoria della matrice delle adiacenze.

***Soluzione*.**

La memoria richiesta con le liste delle adiacenze è $\mathbf{32(\boldsymbol{n}+2\boldsymbol{m})}$ mentre quella richiesta con la matrice delle adiacenze è n^2.

Le liste delle adiacenze richiedono quindi meno memoria quando $\mathbf{32(\boldsymbol{n}+2\boldsymbol{m})} < n^2$ ossia

$$m < \frac{n(n-32)}{64}$$

Ecco, in funzione di n, i valori massimi di m per cui sono convenienti le liste:

n	m
1-32	**nessuno**
33	**0**
34	**1**
35	**1**
40	**5**
50	**14**
100	**106**
200	**525**
500	**3656**
1000	**15125**
$n \to \infty$	$n^2/64$

***Esercizio 2*.**

Il trasposto di un grafo orientato $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ è il grafo $\boldsymbol{G}^{\boldsymbol{T}} = (\boldsymbol{V},\boldsymbol{E}^{\boldsymbol{T}})$ in cui $\boldsymbol{E}^{\boldsymbol{T}} = \{\boldsymbol{uv} : \boldsymbol{vu} \in \boldsymbol{E}\}$.

Descrivere due algoritmi efficienti per calcolare $\boldsymbol{G}^{\boldsymbol{T}}$ con la rappresentazione con liste delle adiacenze e con la rappresentazione con matrice delle adiacenze.

***Esercizio 3*.**

Con la matrice delle adiacenze molti algoritmi richiedono tempo $\boldsymbol{O}(|\boldsymbol{V}|^2)$. Ci sono però alcune eccezioni quali il seguente *problema della celebrità*:
"Dato un grafo orientato $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ con $\boldsymbol{n} = |\boldsymbol{V}|$ vertici determinare se esiste un vertice avente grado entrante $\boldsymbol{n}$**-1** (conosciuto da tutti) e grado uscente **0** (non conosce nessuno)."
Trovare un algoritmo che risolve il problema in tempo $\boldsymbol{O}(|\boldsymbol{V}|)$ usando la matrice delle adiacenze.

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

Esercizio 4.

La chiusura transitiva e riflessiva di un grafo orientato $G = (V,E)$ è il grafo $G^* = (V,E^*)$ che si ottiene da G aggiungendo tutti gli archi uv tali che in G ci sia un cammino da u a v.

Trovare un algoritmo efficiente che calcola la chiusura transitiva e riflessiva sia usando la matrice delle adiacenze che usando le liste delle adiacenze.

```
CTR (M,M*,n)
  for u = 1 to n                    // Azzera M*
      for v = 1 to n
          M*[u,v] = 0
  for u = 1 to n                    // Chiusura riflessiva
      M*[u,u] = 1
  for u = 1 to n                    // Chiusura transitiva
      for v = 1 to n
          if M[u,v] == 1 and M* [u,v] == 0
                  // Aggiungi tutti gli archi wz con z
                  // raggiungibile da w con un cammino
                  // passante per uv
              CT (M*,u,v,n)
```

```
CT (M*,u,v,n)
    M*[u,v] = 1
    for z = 1 to n
        if M*[v,z] == 1 and M*[u,z] == 0
            CT (M*,u,z,n)
    for w = 1 to n
        if M*[w,u] == 1 and M*[w,v] == 0
            CT (M*,w,v,n)
```

Complessità: Procedura principale senza chiamate ricorsive $O(n^2)$. Numero chiamate ricorsive $O(m^*)$. Ogni chiamata ricorsiva costa $O(n)$.
Quindi $O(m^* n)$.

```
CTR (Adj, Adj*,n)
  for u = 1 to n                // Inizializza Adj* e Adj*T
      Adj*[u] = nil, Adj*T[u] = nil
  for u = 1 to n                // Chiusura riflessiva
      Push (Adj*[u],u), Push (Adj*T[u],u)
  for u = 1 to n                // Chiusura transitiva
      for v ∈ Adj[u]
          if v ∉ Adj*[u]
              // Aggiungi tutti gli archi wz con z
              // raggiungibile da w con un cammino
              // passante per uv
            CT (Adj*, Adj*T,u,v,n)
```

```
CT (Adj*, Adj*^T,u,v,n)
    Push (Adj*[u],v), Push (Adj*^T[v],u)
    for z ∈ Adj*[v]
        if z ∉ Adj*[u]
            CT (Adj*, Adj*^T,u,z,n)
    for w ∈ Adj*^T[u]
        if w ∉ Adj*^T[v]
            CT (Adj*, Adj*^T,w,v,n)
```

Complessità: Procedura principale senza chiamate ricorsive $O(m + n)$. Chiamate ricorsive $O(m^*)$.
Ogni chiamata ricorsiva $O(n)$.
Quindi $O(m^* n)$.

Esercizio 5*.**

Il problema della chiusura transitiva dinamica richiede di mantenere aggiornata la chiusura $\boldsymbol{G^* = (V,E^*)}$ di un grafo orientato $\boldsymbol{G = (V,E)}$ quando al grafo $\boldsymbol{G}$ viene aggiunto un arco.

Trovare un algoritmo efficiente che realizza l'operazione di aggiungere un arco a $\boldsymbol{G}$ aggiornando contemporaneamente la chiusura $\boldsymbol{G^*}$.

Per una sequenza di $\boldsymbol{m = O(n^2)}$ operazioni che aggiungono uno alla volta gli archi di $\boldsymbol{G}$ esso deve richiedere tempo $\boldsymbol{O(n\ m^*) = O(n^3)}$.

```
CTD (M*,u,v,n)
    M*[u,v] = 1
    for z = 1 to n
        if M*[v,z] == 1 and M*[u,z] == 0
            CTD (M*,u,z,n)
    for w = 1 to n
        if M*[w,u] == 1 and M*[w,v] == 0
            CTD (M*,w,v,n)
```

Visita in ampiezza

Dato un grafo $G = (V,E)$ ed un vertice particolare $s \in V$ la *visita in ampiezza* partendo dalla *sorgente* **s** visita sistematicamente il grafo per scoprire tutti i vertici che sono raggiungibili da **s**.

Nel contempo calcola la *distanza* di ogni vertice del grafo dalla sorgente **s** (lunghezza minima di un cammino dalla sorgente al vertice).

Esso produce inoltre l'*albero della visita in ampiezza* i cui rami sono cammini di lunghezza minima.

La visita viene detta in ampiezza perché l’algoritmo espande uniformemente la frontiera tra i vertici scoperti e quelli non ancora scoperti.

In altre parole scopre tutti i vertici a distanza ***k*** prima di scoprire quelli a distanza ***k*+1**.

Per mantenere traccia del punto a cui si è arrivati nell’esecuzione dell’algoritmo i vertici sono colorati ***bianco*** (vertici non ancora raggiunti), ***grigio*** (vertici raggiunti e che stanno sulla frontiera) e ***nero*** (vertici raggiunti ma che non stanno più sulla frontiera).

I vertici adiacenti ad un vertice nero possono essere soltanto neri o grigi mentre quelli adiacenti ad un vertice grigio possono essere anche bianchi.

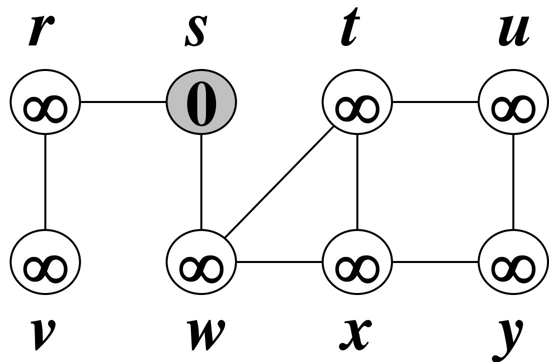
L'algoritmo costruisce un albero che all'inizio contiene soltanto la radice ***s***.

Quando viene scoperto un vertice bianco ***v*** a causa di un arco ***uv*** che lo connette ad un vertice ***u*** scoperto precedentemente il vertice ***v*** e l'arco ***uv*** vengono aggiunti all'albero.

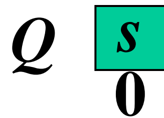
Il vertice ***u*** viene detto <u>*padre*</u> di ***v***.

La realizzazione seguente dell'algoritmo di visita in ampiezza assume che il grafo sia rappresentato con liste delle adiacenze ed usa una coda $\boldsymbol{Q}$ (una lista FIFO) di vertici in cui memorizza la frontiera.

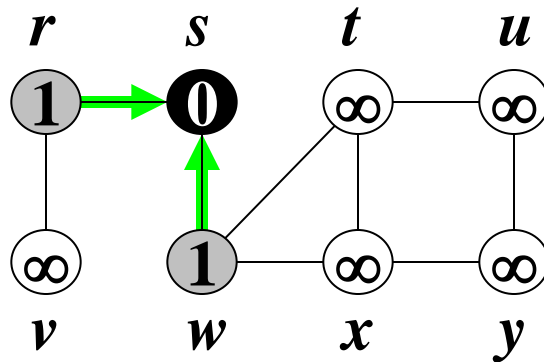
```
BFS (G, s)
  for "ogni vertice v ∈ G.V"
    v.color = bianco, v.d = ∞, v.π = nil
  s.color = grigio, s.d = 0
  Enqueue (Q, s)
  while not Empty (Q)
    u = Dequeue (Q)
    for "ogni v ∈ Adj[u]"
      if v.color == bianco
        v.color = grigio, v.d = u.d + 1, v.π = u
        Enqueue (Q,v)
    u.color = nero
```



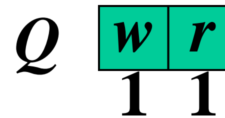
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y



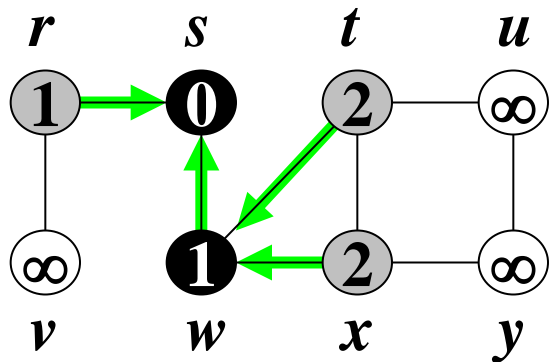
Q
s
0



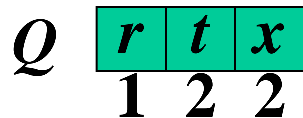
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y



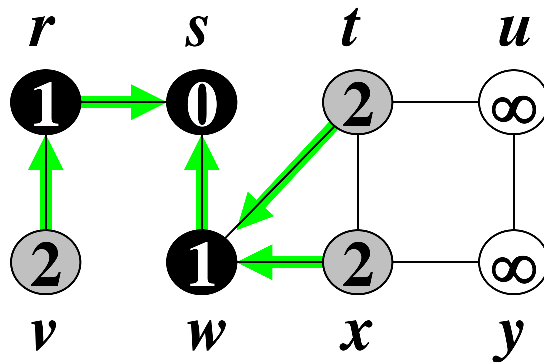
Q
w r
1 1



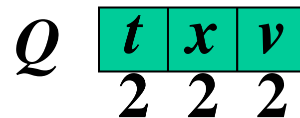
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y



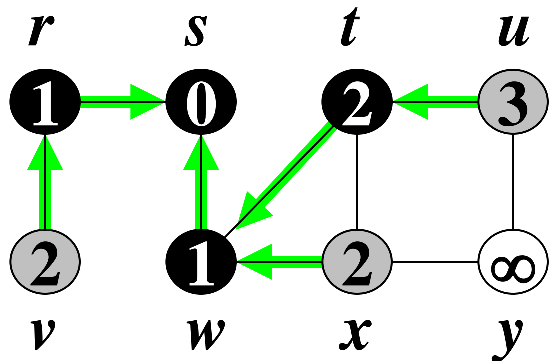
Q
r t x
1 2 2



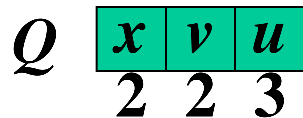
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y



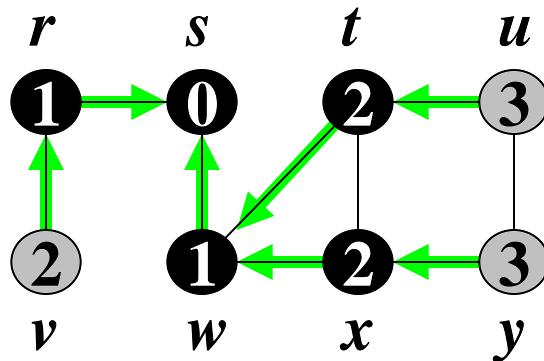
Q
t x v
2 2 2



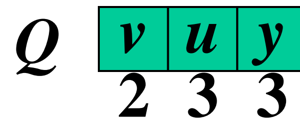
r s t u
1 0 2 3
2 1 2 ∞
v w x y



Q
x v u
2 2 3



r s t u
1 0 2 3
2 1 2 3
v w x y



Q
v u y
2 3 3

Complessità:

```
BFS (G, s)
  for "ogni vertice v ∈ G.V"
    v.color = bianco, v.d = ∞, v.π = nil
  s.color = grigio, s.d = 0
  Enqueue (Q, s)                                  } O(n)
  while not Empty (Q)
    u = Dequeue (Q)                               } O(n)
    for "ogni v ∈ Adj[u]"
      if v.color == bianco
        v.color = grigio, v.d = u.d + 1, v.π = u
        Enqueue (Q,v)                             } O(m)
    u.color = nero                                } O(n)
```

Complessità.

Valutiamo la complessità di ***BFS*** in funzione del numero ***n*** di vertici e del numero ***m*** di archi.

L'inizializzazione richiede tempo ***O*(*n*)** dovendo percorrere tutti i vertici del grafo.

Dopo l'inizializzazione nessun vertice viene più colorato ***bianco*** e questo ci assicura che ogni vertice verrà inserito nella coda al più una sola volta.

Quindi il ciclo **while** viene eseguito al più ***n*** volte.

Il tempo richiesto per il ciclo **while** è dunque $O(n)$ più il tempo richiesto per eseguire i cicli **for** interni.

I cicli **for** interni percorrono una sola volta le liste delle adiacenze di ciascun vertice visitato.

Siccome la somma delle lunghezze di tutte le liste è $O(m)$ possiamo concludere che la complessità è $O(n+m)$.

Correttezza.

Indichiamo con $\boldsymbol{\delta(u,v)}$ la *distanza* del vertice $\boldsymbol{v}$ dal vertice $\boldsymbol{u}$, ossia la lunghezza di un *cammino minimo* che congiunge $\boldsymbol{u}$ a $\boldsymbol{v}$.

Convenzionalmente, se $\boldsymbol{v}$ non è raggiungibile da $\boldsymbol{u}$ poniamo $\boldsymbol{\delta(u,v) = \infty}$.

Proprietà delle distanze.

$\boldsymbol{\delta(x,v) \leq \delta(x,u) + 1}$ per ogni $\boldsymbol{x \in V}$ e ogni $\boldsymbol{uv \in E}$.

Dimostrazione.

Se $\boldsymbol{u}$ non è raggiungibile da $\boldsymbol{x}$ allora

$\boldsymbol{\delta(x,v)} \leq \boldsymbol{\delta(x,u)} + \mathbf{1} = \infty + \mathbf{1} = \infty$.

Altrimenti esiste un cammino di lunghezza $\boldsymbol{\delta(x,u)}$ che congiunge $\boldsymbol{x}$ a $\boldsymbol{u}$.

Aggiungendo l'arco $\boldsymbol{uv}$ a tale cammino otteniamo un cammino di lunghezza $\boldsymbol{\delta(x,u)} + \mathbf{1}$ che congiunge $\boldsymbol{x}$ a $\boldsymbol{v}$.

Proprietà del limite superiore.

Per ogni vertice $\boldsymbol{u}$ e per tutta l'esecuzione di $\boldsymbol{BFS}$ vale la diseguaglianza:

$$\boldsymbol{u.d \geq \delta(s,u)}$$

Dimostrazione.

Dopo l'inizializzazione

$$\boldsymbol{s.d = 0 = \delta(s,s)}$$

mentre per ogni vertice $\boldsymbol{u \neq s}$

$$\boldsymbol{u.d = \infty \geq \delta(s,u)}$$

L'unica istruzione che modifica la distanza è

$$\boldsymbol{v.d = u.d + 1}$$

che viene eseguita soltanto se esiste l'arco $\boldsymbol{uv}$.

Supponiamo, per ipotesi induttiva, che la proprietà sia vera prima di eseguirla.

Allora dopo averla eseguita

$$\begin{aligned} \boldsymbol{v.d} &= \boldsymbol{u.d + 1} \\ &\geq \boldsymbol{\delta(s,u) + 1} && \text{(ipot. induttiva)} \\ &\geq \boldsymbol{\delta(s,v)} && \text{(propr. distanze)} \end{aligned}$$

Proprietà della coda.

Se la coda $\boldsymbol{Q}$ non è vuota e contiene i vertici $\boldsymbol{v_1, v_2, \ldots, v_r}$ allora per ogni $\boldsymbol{i = 1, \ldots, r - 1}$

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

ed inoltre

$$\boldsymbol{v_r.d \leq v_1.d + 1}$$

Dimostrazione.

Dopo l'inizializzazione la proprietà è vera perché la coda contiene solo $\boldsymbol{s}$.

Le istruzione che modificano la coda sono

***Enqueue* (*Q*,*v*)** e ***Dequeue* (*Q*)**

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

Se $\boldsymbol{r = 1}$ la coda si svuota e la proprietà rimane vera, altrimenti

$$\boldsymbol{v_r.d \le v_1.d + 1 \le v_2.d + 1}$$

e quindi la proprietà è vera anche dopo la rimozione di $\boldsymbol{v_1}$.

Consideriamo l'operazione ***Enqueue*(*Q*,*v*)**.

Quando eseguiamo ***Enqueue*(*Q*,*v*)** abbiamo già tolto dalla coda il vertice $\boldsymbol{u}$ la cui lista delle adiacenze stiamo esaminando e a tutti gli elementi di tale lista che aggiungiamo alla coda assegniamo $\boldsymbol{v.d = u.d + 1}$.

Se $\boldsymbol{u}$ era l'unico elemento la lista viene svuotata dopo di che tutti i vertici $\boldsymbol{v}$ adiacenti ad $\boldsymbol{u}$ che vengono inseriti nella lista hanno lo stesso valore $\boldsymbol{v.d = u.d + 1}$ e quindi la proprietà rimane vera.

Altrimenti sia $\boldsymbol{v_1}$ il primo e $\boldsymbol{v_r}$ l'ultimo elemento in coda prima dell'operazione. Per ipotesi induttiva tutti gli elementi che stavano nella coda prima di togliere $\boldsymbol{u}$ hanno valore compreso tra $\boldsymbol{u.d}$ e $\boldsymbol{u.d+1}$ mentre tutti quelli inseriti successivamente hanno esattamente valore $\boldsymbol{u.d+1}$.

Il nuovo elemento $\boldsymbol{v}$ inserito nella coda diventa $\boldsymbol{v_{r+1}}$ e dunque

$$\boldsymbol{v_{r+1}.d = u.d+1 \le v_1.d+1} \quad \text{e} \quad \boldsymbol{v_r.d \le u.d+1 = v_{r+1}.d}$$

e la proprietà rimane vera.

*Correttezza di **BFS***.

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

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

- $v.\pi = u \neq nil$,
- $uv \in E$ e
- uno dei cammini minimi sa s a v è costituito da un cammino minimo da s a u seguito dall'arco uv.

Dimostrazione.

Supponiamo che a qualche vertice venga assegnata una distanza $\boldsymbol{v.d \neq \delta(s,v)}$ e tra questi consideriamo quello con distanza $\boldsymbol{\delta(s,v)}$ minima. Naturalmente $\boldsymbol{v \neq s}$ e per il limite inferiore $\boldsymbol{v.d > \delta(s,v)}$.

$\boldsymbol{v}$ deve essere raggiungibile altrimenti $\boldsymbol{v.d \leq \delta(s,v) = \infty}$.

Sia $\boldsymbol{u}$ un vertice che precede $\boldsymbol{v}$ in un cammino minimo. Quindi $\boldsymbol{v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1}$.

Quando l'algoritmo toglie $\boldsymbol{u}$ dalla coda

- $\boldsymbol{v}$ non può essere bianco altrimenti $\boldsymbol{v.d} = \boldsymbol{u.d} + \mathbf{1}$
- $\boldsymbol{v}$ non può essere nero altrimenti dovrebbe essere già stato tolto e quindi $\boldsymbol{v.d} \leq \boldsymbol{u.d} < \boldsymbol{u.d} + \mathbf{1}$
- $\boldsymbol{v}$ non può essere grigio altrimenti dovrebbe essere stato aggiunto alla coda visitando un vertice $\boldsymbol{w}$ tolto dalla coda prima di $\boldsymbol{u}$ e quindi $\boldsymbol{v.d} = \boldsymbol{w.d} + \mathbf{1} \leq \boldsymbol{u.d} + \mathbf{1}$.

Quindi $\boldsymbol{v.d} = \boldsymbol{\delta(s,v)}$ è corretta per ogni vertice $\boldsymbol{v}$.

Inoltre tutti i vertici $\boldsymbol{v}$ raggiungibili vengono visitati altrimenti $\boldsymbol{v.d}$ rimarrebbe ∞ e sarebbe errata.

Quando viene eseguita l'assegnazione $\boldsymbol{v.d = u.d+1}$ esiste l'arco $\boldsymbol{uv}$ e viene posto $\boldsymbol{v.\pi = u}$.

Siccome

$$\boldsymbol{v.d = \delta(s,v) = u.d+1 = \delta(s,u)+1}$$

L'arco $\boldsymbol{uv}$ estende ogni cammino minimo da $\boldsymbol{s}$ a $\boldsymbol{u}$ in un cammino minimo da $\boldsymbol{s}$ a $\boldsymbol{v}$.

Definiamo il grafo dei predecessori $\boldsymbol{G_\pi = (V_\pi, E_\pi)}$ ponendo $\boldsymbol{V_\pi = \{s\} \cup \{v \in V : v.\pi \neq nil\}}$

ed $\boldsymbol{E_\pi = \{(v.\pi)v : v.\pi \neq nil\}}$.

$\boldsymbol{G_\pi}$ è connesso (aggiungiamo soltanto vertici connessi a un vertice precedente), i vertici $\boldsymbol{V_\pi}$ sono tutti e soli quelli raggiungibili da $\boldsymbol{s}$ e tutti i vertici, escluso $\boldsymbol{s}$, hanno un solo padre.

Dunque $\boldsymbol{G_\pi}$ è un albero con radice $\boldsymbol{s}$.

Sia $\boldsymbol{v} \in \boldsymbol{V_\pi}$ e sia $\boldsymbol{s} = \boldsymbol{x_0}, \boldsymbol{x_1}, \ldots, \boldsymbol{x_k} = \boldsymbol{v}$ l'unico cammino da $\boldsymbol{s}$ a $\boldsymbol{v}$ in $\boldsymbol{G_\pi}$. Dimostriamo per induzione su $\boldsymbol{k}$ che esso è un cammino minimo.

Se $\boldsymbol{k} = \mathbf{0}$ allora $\boldsymbol{v} = \boldsymbol{s}$ e $\boldsymbol{v.d} = \boldsymbol{0} = \boldsymbol{\delta(s,s)}$.

Se $\boldsymbol{k} > \mathbf{0}$ sia $\boldsymbol{x_{k\text{-}1}} = \boldsymbol{v.\pi}$ il vertice precedente. Per ipotesi induttiva il cammino da $\boldsymbol{s}$ a $\boldsymbol{x_{k\text{-}1}}$ è un cammino minimo di lunghezza $\boldsymbol{\delta(s,x_{k\text{-}1})} = \boldsymbol{k}\text{-}1$.

Dunque $\boldsymbol{\delta(s,v)} = \boldsymbol{v.d} = \boldsymbol{x_{k\text{-}1}.d} + \mathbf{1} = \boldsymbol{\delta(s,x_{k\text{-}1})} + \mathbf{1} = \boldsymbol{k}$.

Visita in profondità

La strategia della *visita in profondità* è quella di avanzare in profondità nella ricerca finché è possibile.

Si esplorano quindi sempre gli archi uscenti dal vertice ***u*** raggiunto per ultimo. Se viene scoperto un nuovo vertice ***v*** ci si sposta su tale vertice. Se tutti gli archi uscenti da ***u*** portano a vertici già scoperti si torna indietro e si riprende l'esplorazione degli archi uscenti dal vertice da cui ***u*** è stato scoperto.

Il procedimento continua finché si sono visitati tutti i vertici raggiungibili dal vertice iniziale scelto.

Se non sono stati visitati tutti i vertici del grafo si ripete il procedimento partendo da un vertice non ancora visitato.

Anche nella visita in profondità quando viene scoperto un nuovo vertice $\boldsymbol{v}$ esplorando la lista delle adiacenze di un vertice $\boldsymbol{u}$ si memorizza un puntatore $\boldsymbol{v.\pi}$ da $\boldsymbol{v}$ ad $\boldsymbol{u}$.

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

Anche nella visita in profondità i vertici sono colorati ***bianco*** (vertici non ancora scoperti), ***grigio*** (vertici *scoperti*) e ***nero*** (vertici *finiti* la cui lista delle adiacenze è stata completamente esplorata).

La visita in profondità pone due *marcatempi* su ogni vertice ***u***. Un primo marcatempo ***u.d*** viene posto quando il vertice viene scoperto e colorato ***grigio*** ed un secondo marcatempo ***u.f*** quando il vertice è finito e viene colorato ***nero***.

Tali marcatempi sono utili in molte applicazioni che usano la visita in profondità e sono utili per ragionare sul funzionamento della visita stessa.

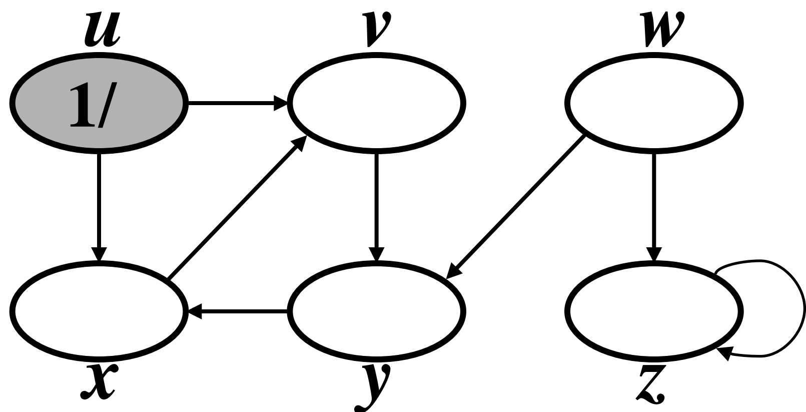
L'algoritmo di visita in profondità è:

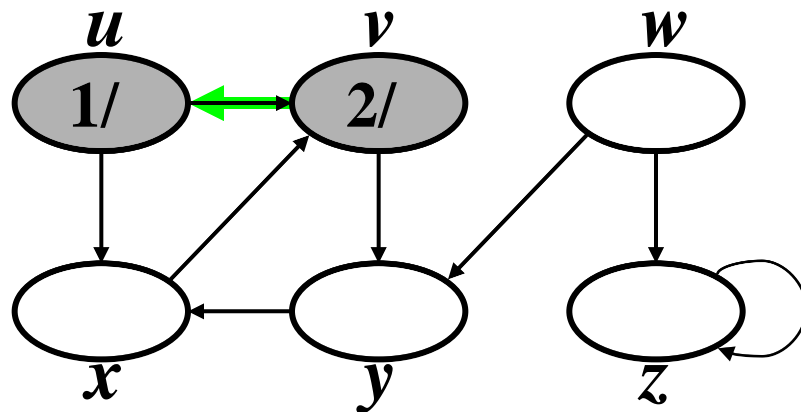
```
DFS (G)
  for "ogni v ∈ G.V"
     v.color = bianco, v.π = nil
  time = 0
  for "ogni v ∈ G.V"
    if v.color == bianco
      DFS-Visit (v)
```

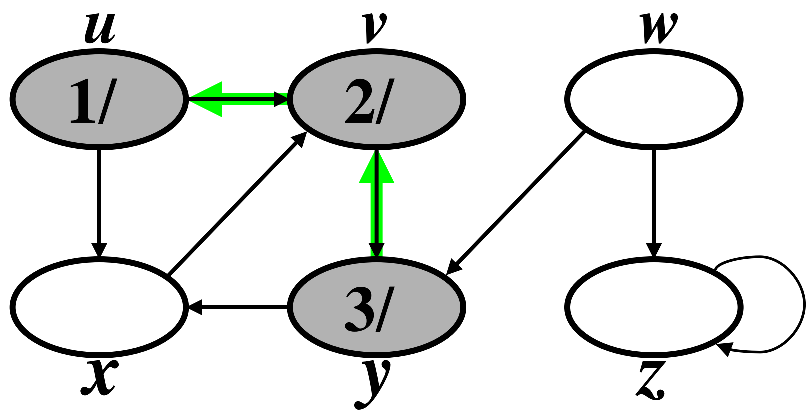
richiama la procedura ricorsiva ***DFS-Visit*(*v*)** che esplora in profondità tutti i vertici raggiungibili dal vertice ***v***.

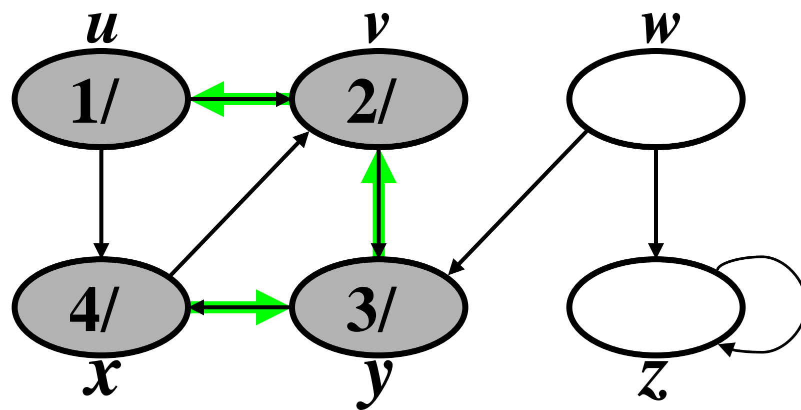
```
DFS-Visit (u)
  time = time + 1, u.d = time
  u.color = grigio
  for "ogni v ∈ Adj[u]"
      if v.color == bianco
          v.π = u
          DFS-Visit (v)
  time = time + 1, u.f = time
  u.color = nero
```

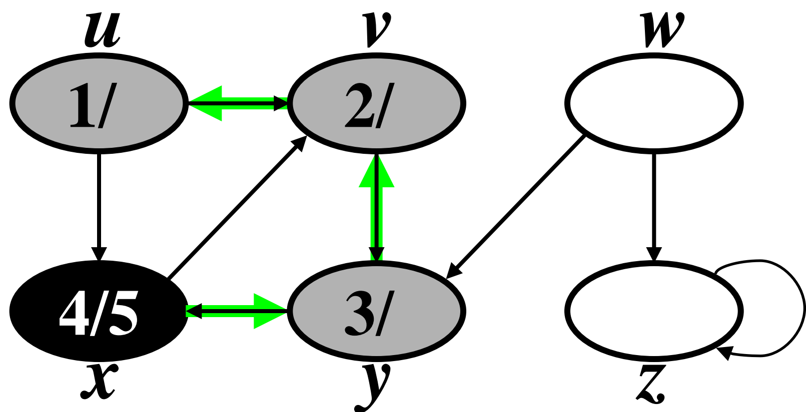
Vediamo su di un esempio come viene eseguito l'algoritmo.

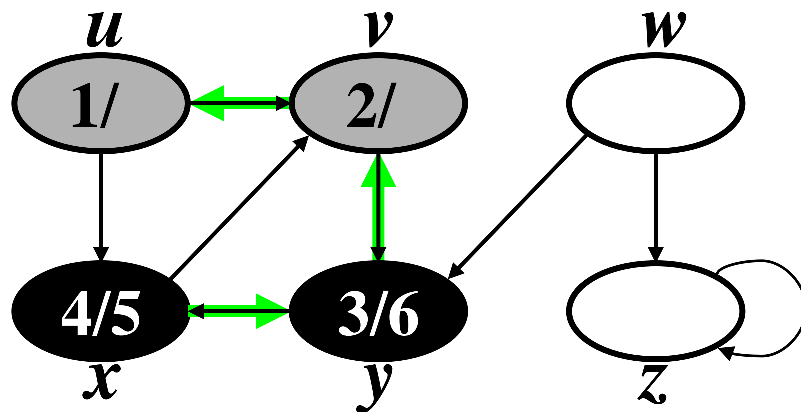

u
v
w
1/
x
y
z

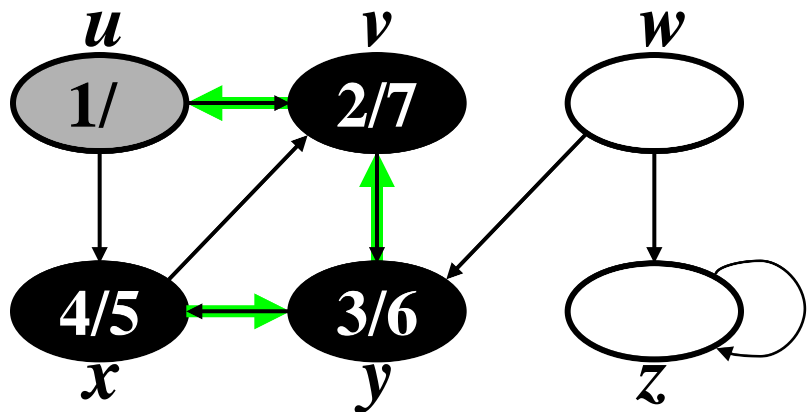

u
v
w
1/
2/
x
y
z

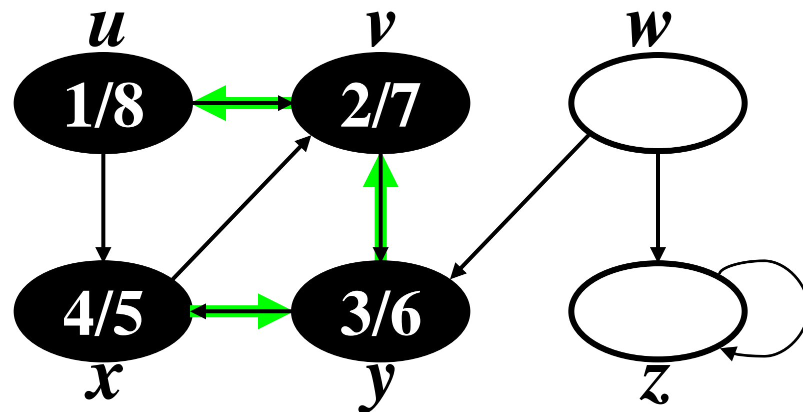

u
v
w
1/
2/
3/
x
y
z

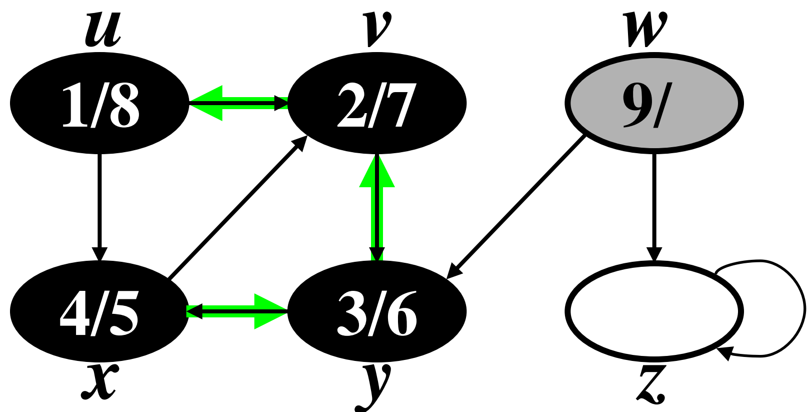

u
v
w
1/
2/
4/
3/
x
y
z

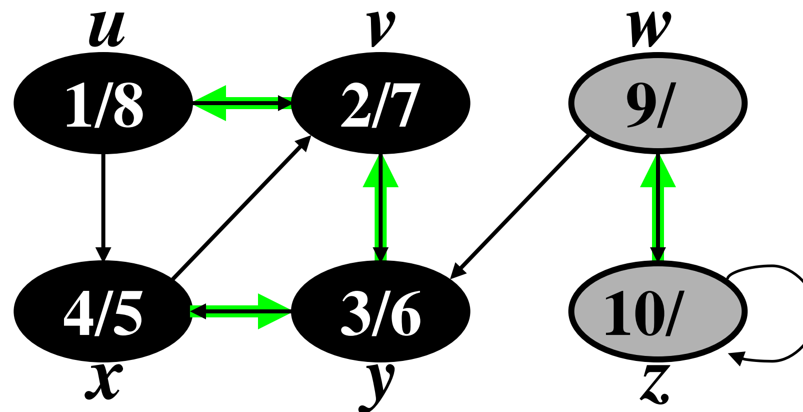

u
v
w
1/
2/
4/5
3/
x
y
z

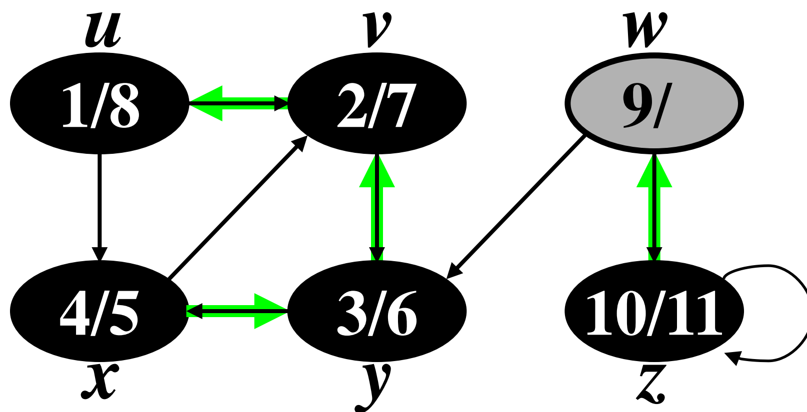

u
v
w
1/
2/
4/5
3/6
x
y
z

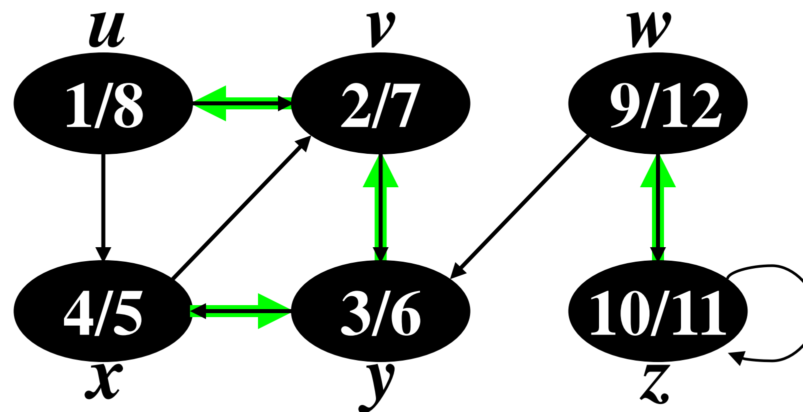

u
v
w
1/
2/7
4/5
3/6
x
y
z


u
v
w
1/8
2/7
4/5
3/6
x
y
z


u
v
w
1/8
2/7
9/
4/5
3/6
x
y
z


u
v
w
1/8
2/7
9/
4/5
3/6
10/
x
y
z


u
v
w
1/8
2/7
9/
4/5
3/6
10/11
x
y
z


u
v
w
1/8
2/7
9/12
4/5
3/6
10/11
x
y
z

```
DFS (G)
  for "ogni v ∈ G.V"
     v.color = bianco, v.π = nil
  time = 0                                  } O(n)
  for "ogni v ∈ G.V"
    if v.color == bianco
      DFS-Visit (v)
```

```
DFS-Visit (u)
  time = time + 1, u.d = time, u.color = grigio        } O(n)
  for "ogni v ∈ Adj[u]"
    if v.color == bianco                               } O(m)
      v.π = u
      DFS-Visit (v)                                    } O(n)
  time = time + 1, u.f = time , u.color = nero         } O(n)
```

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

La funzione ***DFS-Visit*** viene richiamata solo su vertici bianchi che vengono subito colorati grigio. Essa viene quindi richiamata 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 $\Theta(m)$ l'intero algoritmo ha complessità $O(n+m)$.

Proprietà delle parentesi. Se si rappresenta la scoperta di ogni vertice ***u*** con una parentesi aperta $(_u$ e la finitura con una parentesi chiusa $_u)$ si ottiene una sequenza ben formata di parentesi.

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

a) $(_u \ldots\ldots {}_u) \ldots\ldots (_v \ldots\ldots {}_v)$

b) $(_v \ldots\ldots {}_v) \ldots\ldots (_u \ldots\ldots {}_u)$

c) $(_u \ldots\ldots (_v \ldots\ldots {}_v) \ldots\ldots {}_u)$

d) $(_v \ldots\ldots (_u \ldots\ldots {}_u) \ldots\ldots {}_v)$

Dimostrazione. Assumiamo che $\boldsymbol{u.d} < \boldsymbol{v.d}$ (l'altro caso è simmetrico).

Se $\boldsymbol{u.f} < \boldsymbol{v.d}$ allora

a) $(_{u} \ldots\ldots {}_{u}) \ldots\ldots (_{v} \ldots\ldots {}_{v})$.

Se $\boldsymbol{u.f} > \boldsymbol{v.d}$ allora quando $\boldsymbol{v}$ viene scoperto $\boldsymbol{u}$ è grigio e siccome $\boldsymbol{v}$ è stato scoperto dopo di $\boldsymbol{u}$ la sua lista delle adiacenze verrà completamente esplorata prima di riprendere l'esplorazione di quella di $\boldsymbol{u}$.

Quindi $\boldsymbol{v}$ viene finito prima di $\boldsymbol{u}$ e pertanto

c) $(_{u} \ldots\ldots (_{v} \ldots\ldots {}_{v}) \ldots\ldots {}_{u})$.

Proprietà dei discendenti. Il vertice ***v*** è discendente del vertice ***u*** in un albero della foresta di ricerca in profondità se e solo se

$(_u \; .... \; (_v \; .... \; _v) \; .... \; _u)$.

Dimostrazione. Il vertice ***v*** è discendente di ***u*** se e solo se è scoperto dopo di ***u*** e prima che ***u*** sia finito e quindi, per la proprietà delle parentesi, se e solo se $(_u \; .... \; (_v \; .... \; _v) \; .... \; _u)$.

Proprietà del cammino bianco. Il vertice ***v*** è discendente del vertice ***u*** in un albero della foresta di visita in profondità se e solo se nell'istante in cui ***u*** viene scoperto esiste un cammino da ***u*** a ***v*** i cui vertici sono tutti bianchi (cammino bianco).

Dimostrazione.

Sia ***v*** discendente di ***u*** e sia $\boldsymbol{u = x_0, x_1, \ldots, x_k = v}$ la sequenza dei vertici da ***u*** a ***v*** nel ramo dell'albero della foresta di ricerca 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 $u = x_0, x_1, \ldots, x_k = v$ è un cammino tale che quando $u = x_0$ viene scoperto i vertici $x_1, \ldots, x_k = v$ non sono ancora stati scoperti e dunque sono bianchi.

Supponiamo ora che quando $\boldsymbol{u}$ viene scoperto esista un cammino bianco $\boldsymbol{u = x_0, x_1, \ldots, x_k = v}$ da $\boldsymbol{u}$ a $\boldsymbol{v}$.

Siccome $\boldsymbol{v}$ viene scoperto dopo $\boldsymbol{u}$, per la proprietà delle parentesi abbiamo una delle due possibilità

$$(_u \ldots\ldots {}_u) \ldots\ldots (_v \ldots\ldots {}_v) \text{ oppure } (_u \ldots\ldots (_v \ldots\ldots {}_v) \ldots\ldots {}_u).$$

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

Mostriamo che $(_u \ldots\ldots {}_u) \ldots\ldots (_v \ldots\ldots {}_v)$ non può accadere.

Supponiamo per assurdo $(_u \ldots\ldots {}_u) \ldots\ldots (_v \ldots\ldots {}_v)$ e quindi $\boldsymbol{v}$ non sia discendente di $\boldsymbol{u}$.

Possiamo anche assumere che $\boldsymbol{v}$ sia il primo vertice del cammino bianco $\boldsymbol{u = x_0, x_1, \ldots, x_k = v}$ che non è discendente di $\boldsymbol{u}$.

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

Siccome $\boldsymbol{w}$ è discendente di $\boldsymbol{u}$, per la proprietà dei discendenti $(_{u} \ldots (_{w} \ldots {}_{w}) \ldots {}_{u}) \ldots (_{v} \ldots {}_{v})$.

Assurdo perchè $\boldsymbol{v}$ è nella lista delle adiacenze di $\boldsymbol{w}$ e deve quindi essere stato scoperto prima che $\boldsymbol{w}$ sia finito.

Classificazione degli archi. Con la visita in profondità gli archi si possono classificare in:

a) archi d'albero: archi uv con v scoperto visitando le adiacenze di u.

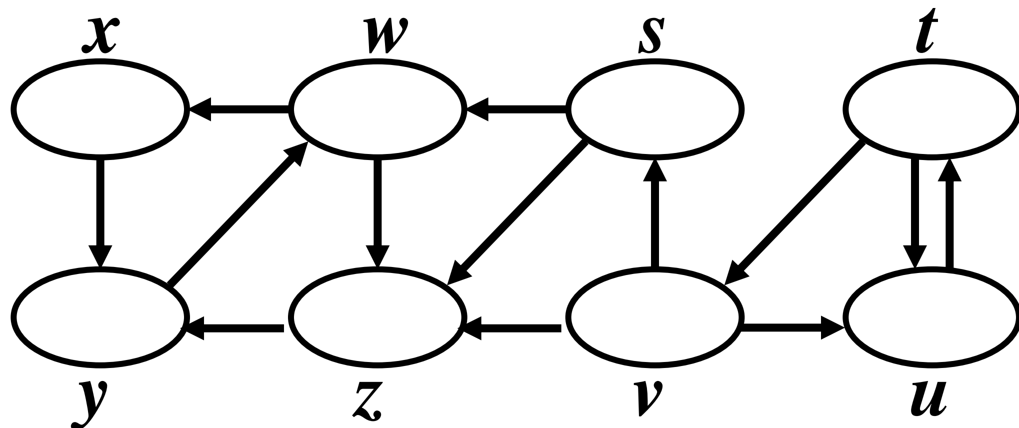
b) archi all'indietro: archi uv con $u = v$ oppure v ascendente di u in un albero della foresta di ricerca in profondità.

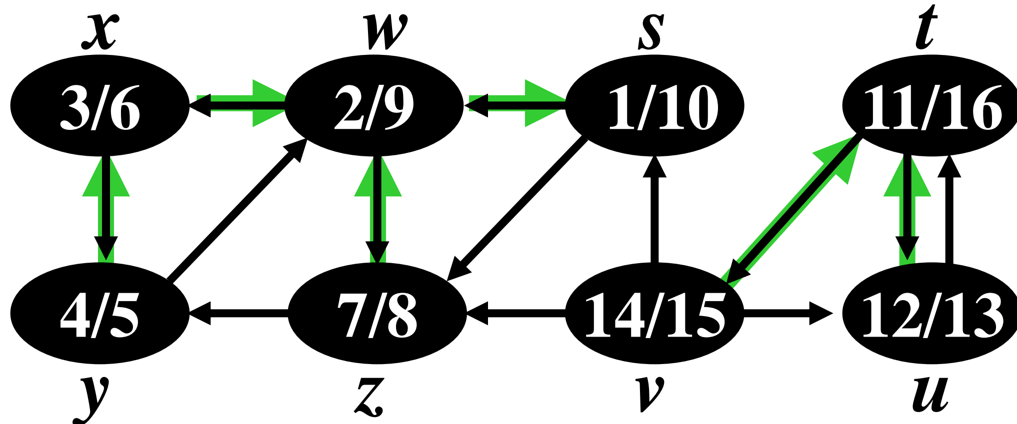
c) archi in avanti: archi uv con v discendente di u in un albero della foresta.

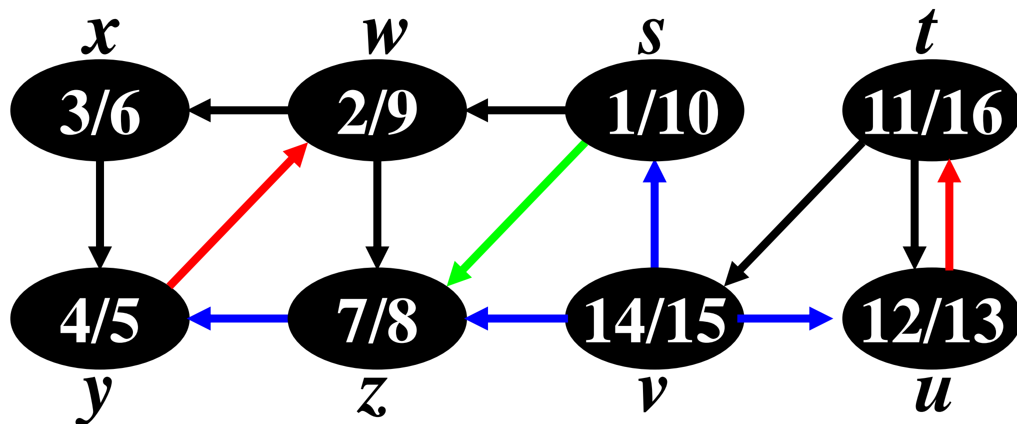
d) archi trasversali: archi uv in cui v ed u appartengono a rami o alberi distinti della foresta.

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.

Ad esempio ogni arco d'albero soddisfa anche le condizioni per essere un arco in avanti e se il grafo non è orientato ogni arco all'indietro soddisfa anche le condizioni per essere un arco in avanti.





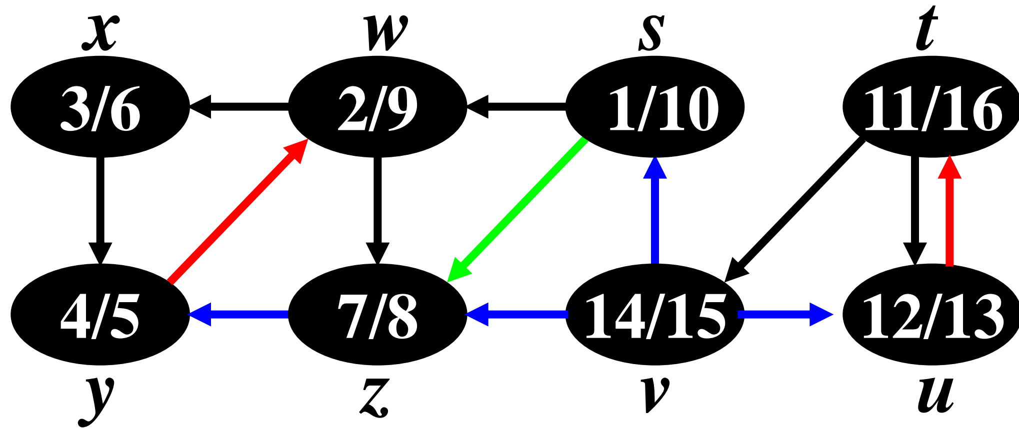


d’albero

all’indietro

in avanti

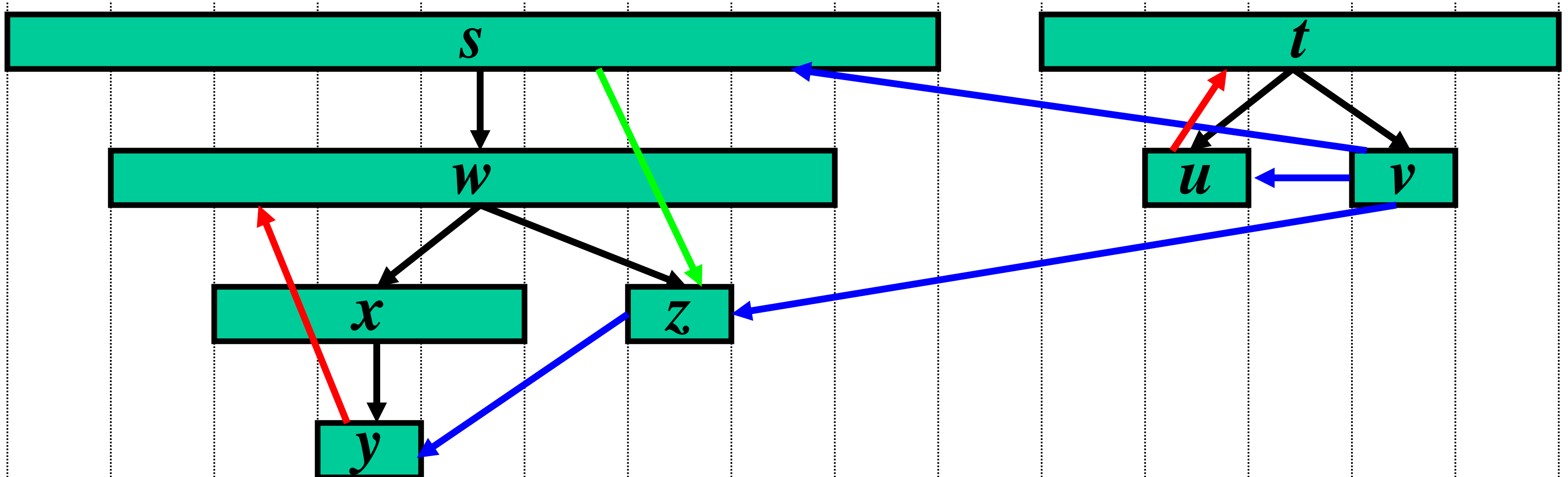
trasversali



d’albero
all’indietro
in avanti
trasversali

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

$(_s$ $(_w$ $(_x$ $(_y$ $_y)$ $_x)$ $(_z$ $_z)$ $_w)$ $_s)$ $(_t$ $(_u$ $_u)$ $(_v$ $_v)$ $_t)$



Esercizio 6.

Aggiungere alla visita in profondità su di un grafo orientato la stampa degli archi incontrati con la loro classificazione.

Esercizio 7.

Mostrare che, se il grafo su cui si effettua la visita in profondità è non orientato, vi sono soltanto archi d'albero ed archi all'indietro.

***Esercizio 8*.**

Scrivere un algoritmo che dato un grafo non orientato $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ connesso trova un cammino in $\boldsymbol{G}$ che attraversa tutti gli archi una e una sola volta in ognuna delle due direzioni.

L’algoritmo deve avere complessità $\boldsymbol{O}(|\boldsymbol{V}|+|\boldsymbol{E}|)$.

Dire come, avendo a disposizione un numero sufficiente di monetine, si possa usare tale algoritmo per cercare l’uscita in un labirinto.

Esercizio 9.

Mostrare con un controesempio che se vi è un cammino da ***u*** a ***v*** in un grafo orientato e ***u*** viene scoperto prima di ***v*** non necessariamente ***v*** è discendente di ***u*** nella foresta di visita in profondità.

Esercizio 10.

Mostrare come un vertice ***u*** di un grafo orientato possa essere l'unico vertice di un albero della foresta di visita in profondità pur avendo sia archi entranti che archi uscenti.

***Esercizio 11*.**

Mostrare che se in un grafo *non* orientato ***G*** esiste l'arco ***uv*** allora i due vertici ***u*** e ***v*** stanno in uno stesso albero della foresta di visita in profondità.

Ordinamento topologico

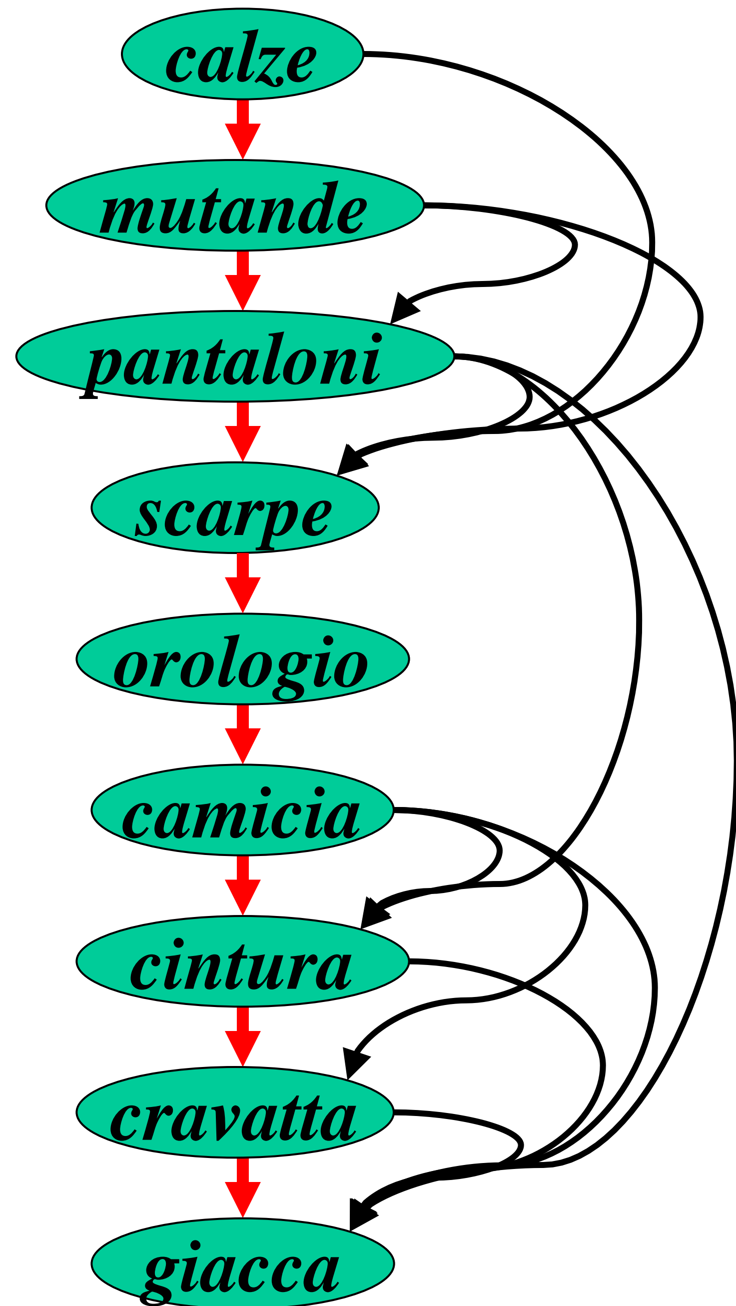
La visita in profondità si può usare per ordinare topologicamente un *grafo orientato aciclico* (detto anche ***DAG***: ***D***irected ***A***cyclic ***G***raph).

Un *ordinamento topologico* di un grafo orientato aciclico $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ è un ordinamento dei suoi vertici tale che per ogni arco $\boldsymbol{uv} \in \boldsymbol{E}$ il vertice $\boldsymbol{u}$ precede il vertice $\boldsymbol{v}$.

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

Alcuni esempi semplici sono 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.

11/16 mutande
calze 17/18
12/15 pantaloni
scarpe 13/14
camicia 1/8
cravatta 2/5
6/7 cintura
giacca 3/4
orologio 9/10



```
TS (G)
    for "ogni v ∈ G.V"
        v.color = bianco
    P = ϕ                          // P è una pila
    for "ogni v ∈ G.V"
        if v.color == bianco
            TS-Visit (G,v, P)
    return P
```

```
TS-Visit (G, u, P)
    u.color = grigio
    for "ogni v ∈ Adj[u]"
        if v.color == bianco
            TS-Visit (G,v, P)
    u.color = nero, Push (P, u)
```

Complessità. La stessa di ***DFS*** ossia $\boldsymbol{O(n+m)}$.

Caratterizzazione dei DAG. 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.

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.

Correttezza di TS.

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 esso è gia stato finito mentre ***u*** non lo è ancora.

Se ***v*** è bianco esso è discendente di ***u*** per la proprietà del cammino bianco e quindi viene finito prima di ***u***.

***Esercizio 12*.**

Scrivere un algoritmo che determina se un grafo $\boldsymbol{G} = (\boldsymbol{V},\boldsymbol{E})$ *non orientato* contiene un ciclo in tempo $\boldsymbol{O}(\boldsymbol{n})$ indipendente dal numero di archi del grafo.

Componenti fortemente connesse

La visita in profondità si può usare anche per calcolare le componenti fortemente connesse di un *grafo orientato*.

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

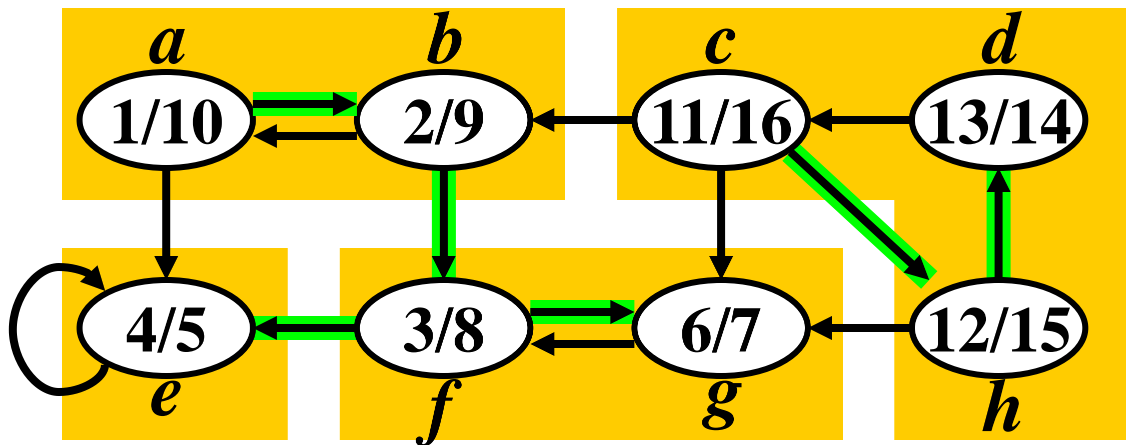
L'algoritmo per il calcolo delle componenti fortemente connesse si compone di tre fasi:

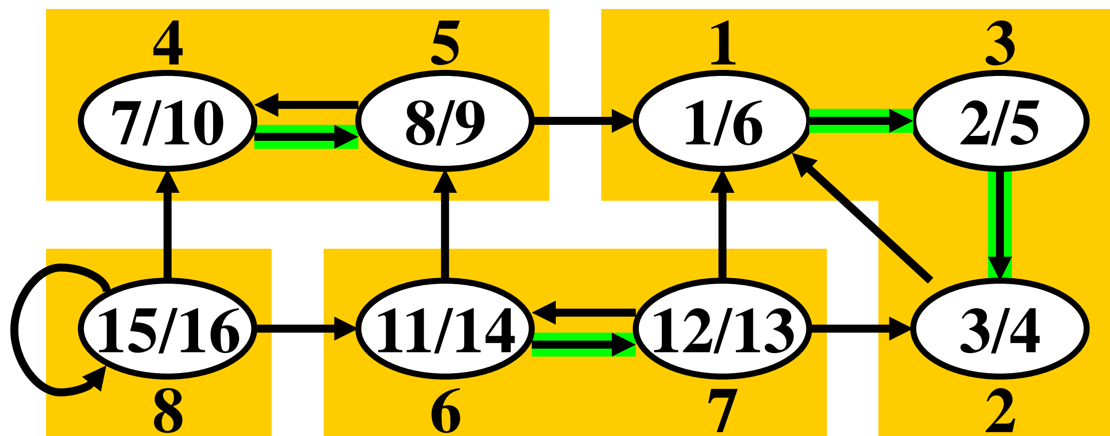
a) usa la visita in profondità in G per ordinare i vertici in ordine di tempo di fine f decrescente (come per l'ordinamento topologico);

b) calcola il grafo trasposto G^T del grafo G;

c) esegue una visita in profondità in G^T usando l'ordine dei vertici calcolato nella prima fase;

Alla fine gli alberi della visita in profondità in G^T rappresentano le componenti fortemente connesse.


a
b
c
d
1/10
2/9
11/16
13/14
4/5
3/8
6/7
12/15
e
f
g
h


4
5
1
3
7/10
8/9
1/6
2/5
15/16
11/14
12/13
3/4
8
6
7
2

Proprietà dei cammini. Siano $\boldsymbol{C}$ e $\boldsymbol{C'}$ due ***cfc*** distinte. Se esiste un cammino $\boldsymbol{P_{uu'}}$ da un vertice $\boldsymbol{u} \in \boldsymbol{C}$ ad un vertice $\boldsymbol{u'} \in \boldsymbol{C'}$ non esiste nessun cammino $\boldsymbol{P_{v'v}}$ da un vertice $\boldsymbol{v'} \in \boldsymbol{C'}$ ad un vertice $\boldsymbol{v} \in \boldsymbol{C}$.

Dimostrazione. Siccome $\boldsymbol{C}$ e $\boldsymbol{C'}$ sono ***cfc*** esiste un cammino $\boldsymbol{P_{vu}}$ da $\boldsymbol{v}$ a $\boldsymbol{u}$ ed un cammino $\boldsymbol{P_{u'v'}}$ da $\boldsymbol{u'}$ a $\boldsymbol{v'}$. Quindi esiste sia un cammino $\boldsymbol{P_{uu'}P_{u'v'}}$ da $\boldsymbol{u}$ a $\boldsymbol{v'}$ che un cammino $\boldsymbol{P_{v'v}P_{vu}}$ da $\boldsymbol{v'}$ a $\boldsymbol{u}$ contro l'ipotesi che $\boldsymbol{u}$ e $\boldsymbol{v'}$ stiano in ***cfc*** distinte.

Dato un insieme di vertici $\boldsymbol{U} \subseteq \boldsymbol{V}$ indichiamo con $\boldsymbol{d(U)}$ il tempo in cui viene scoperto il primo vertice in $\boldsymbol{U}$ e con $\boldsymbol{f(U)}$ il tempo in cui viene finito l'ultimo vertice in $\boldsymbol{U}$ <u>durante la prima visita in profondità</u>.

$$\boldsymbol{d(U) = \min_{u \in U}(u.d)} \qquad \boldsymbol{f(U) = \max_{u \in U}(u.f)}$$

<u>*Proprietà dei tempi di fine*</u>. Siano $\boldsymbol{C}$ e $\boldsymbol{C'}$ due ***cfc*** distinte. Se esiste un arco $\boldsymbol{uv}$ da $\boldsymbol{u \in C}$ a $\boldsymbol{v \in C'}$ allora $\boldsymbol{f(C) > f(C')}$.

Dimostrazione.

Se $d(C) < d(C')$, quando viene scoperto il primo vertice x di C tutti i vertici di C e di C' sono bianchi. Quindi vi è un cammino bianco da x ad ogni vertice di C e, in virtù dell'arco uv, anche un cammino bianco da x a ogni vertice di C'. Per il teorema del cammino bianco tutti i vertici di C e di C' diventano discendenti di x e quindi $x.f = f(C) > f(C')$.

Se $\boldsymbol{d}(\boldsymbol{C}) > \boldsymbol{d}(\boldsymbol{C}')$, quando viene scoperto il primo vertice $\boldsymbol{y}$ di $\boldsymbol{C}'$ tutti i vertici di $\boldsymbol{C}$ e di $\boldsymbol{C}'$ sono bianchi. Vi è un cammino bianco da $\boldsymbol{y}$ ad ogni vertice di $\boldsymbol{C}'$ e quindi $\boldsymbol{y.f} = \boldsymbol{f}(\boldsymbol{C}')$.

Siccome esiste l'arco $\boldsymbol{uv}$ non può esistere nessun cammino da un vertice di $\boldsymbol{C}'$ ad un vertice di $\boldsymbol{C}$. Quindi $\boldsymbol{C}$ non è raggiungibile da $\boldsymbol{y}$. Dunque $\boldsymbol{d}(\boldsymbol{C}) > \boldsymbol{f}(\boldsymbol{C}')$ ed a maggior ragione $\boldsymbol{f}(\boldsymbol{C}) > \boldsymbol{f}(\boldsymbol{C}')$.

Conseguenza. Siano $\boldsymbol{C}$ e $\boldsymbol{C'}$ due ***cfc*** distinte.

Se nel grafo trasposto $\boldsymbol{G^T}$ esiste un arco $\boldsymbol{uv}$ da $\boldsymbol{u} \in \boldsymbol{C}$ a $\boldsymbol{v} \in \boldsymbol{C'}$ allora $f(C) < f(C')$.

Dimostrazione.

I grafi $\boldsymbol{G}$ e $\boldsymbol{G^T}$ hanno le stesse ***cfc*** ed $\boldsymbol{uv}$ è un arco di $\boldsymbol{G^T}$ se e solo se $\boldsymbol{vu}$ è un arco di $\boldsymbol{G}$.

Correttezza dell'algoritmo.

La visita in profondità di G^T parte dal vertice x_1 terminato per ultimo nella visita in profondità di G. Sia C_1 la ***cfc*** che contiene x_1. Per ogni altra ***cfc*** C abbiamo $x_1.f = f(C_1) > f(C)$.

Dunque non esiste alcun arco vu in G^T da $v \in C_1$ a $u \in C$ e l'albero costruito partendo da x_1 contiene tutti e soli i vertici di C_1.

Dopo di che l'algoritmo riparte dal vertice $\boldsymbol{x_2}$ terminato per ultimo tra quelli che non stanno in C_1. Sia $\boldsymbol{C_2}$ la ***cfc*** che contiene $\boldsymbol{x_2}$. Per ogni altra ***cfc*** $\boldsymbol{C}$ abbiamo $\boldsymbol{x_2.f = f(C_2) > f(C)}$.

Dunque non esiste alcun arco $\boldsymbol{vu}$ in $\boldsymbol{G^T}$ da $\boldsymbol{v \in C_2}$ a $\boldsymbol{u \in C}$ e l'albero costruito partendo da $\boldsymbol{x_2}$ contiene tutti e soli i vertici di $\boldsymbol{C_2}$.

Ripetendo il ragionamento si vede che l'algoritmo costruisce esattamente un albero per ogni ***cfc***.

Esercizio 13.

Come si modifica il numero di ***cfc*** aggiungendo un arco?

Trovare un esempio in cui il numero di ***cfc*** non cambia, un esempio in cui il numero di ***cfc*** diminuisce di 1 ed un esempio in cui il numero di ***cfc*** da 10 diventa 1.

Implementazione della ricerca delle cfc.

Nella ricerca in $\boldsymbol{G}$ si usa una pila $\boldsymbol{P}$ per memorizzare i vertici in ordine di finitura decrescente mentre nella ricerca in $\boldsymbol{G^T}$ si usa una struttura per insiemi disgiunti per memorizzare le ***cfc***.

```
CFC (G)
  Passo1 (G, P)
  G^T = Trasponi (G)
  Passo2 (G^T, P)
```

```
Passo1 (G, P)
    for "ogni v ∈ G.V"
        v.color = bianco
    P = ϕ
    for "ogni v ∈ G.V"
        if v.color == bianco
            Passo1-Visit (G, v, P)
```

```
Passo1-Visit (G, u, P)
    u.color = grigio
    for "ogni v ∈ Adj[u]"
        if v.color == bianco
            Passo1-Visit (G, v, P)
    u.color = nero
    Push (P, u)
```

```
Passo2 (G^T, P)
    for "ogni v ∈ G^T.V"
        v.color = bianco
    while not Empty (P)
        v = Pop (P)
        if v.color == bianco
            Passo2-Visit (G^T, v)
```

```
Passo2-Visit (G^T, u)
    u.color = grigio, Make-Set (u)
    for "ogni v ∈ Adj^T[u]"        Adj^T[u] lista adiacenze in G^T
        if v.color == bianco
            Passo2-Visit (G^T, v)
            Union (u,v)
    u.color = nero
```

Dato un grafo orientato ***G***, *il grafo delle **cfc*** di ***G*** è il grafo orientato ***H*** avente come vertici le ***cfc*** di ***G*** e un arco ***CC'*** da ***C*** a ***C'*** se e solo se in ***G*** vi è un arco che connette un vertice di ***C*** ad un vertice di ***C'***.

Esercizio 14.

Dimostrare che il grafo delle ***cfc*** è aciclico (un DAG).

Esercizio 15*.**

Un grafo orientato è *semiconnesso* se per ogni due vertici ***u*** e ***v*** esiste o un cammino da ***u*** a ***v*** oppure un cammino da ***v*** a ***u***.

Trovare un algoritmo efficiente per verificare se un grafo è semiconnesso.

Suggerimento: Ordinare topologicamente il grafo delle componenti fortemente connesse.