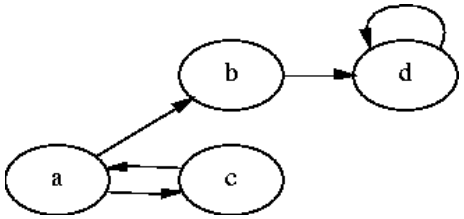
Grafo orientato (o diretto) = (V,E) V = nodi o vertici - E = archi (*edges*)

$$V = \{a, b, c, d\}$$

$$E = \{(a, b), (a, c), (c, a), (d, d), (b, d)\}$$



archi uscenti da un nodo x: (x, y)

archi incidenti su un nodo x: (y, x)

nodi adiacenti a un nodo x o **successori** di x:
$\{y : (x, y) \in E\}$. Se y è adiacente a x: $x \rightarrow y$

predecessori di un nodo x: $\{y : (y, x) \in E\}$

grado di ingresso di un nodo x: numero di archi incidenti su x.

grado di uscita di un nodo x: numero di archi uscenti da x.

sorgente nodo senza predecessori (grado di ingresso 0)

pozzo nodo senza successsori (grado di uscita 0)

Cammino da un nodo X a un nodo Y

Sequenza di nodi $(a_0, \ldots, a_n)$ tale che:

1. $a_0 = X$
2. $a_n = Y$
3. per ogni $i = 0, \ldots, n-1$: $(a_i, a_{i+1}) \in E$.

Lunghezza del cammino n (numero degli archi).

Y è raggiungibile da X se esiste un p cammino da X a Y:

$$X \overset{p}{\rightsquigarrow} Y$$

Cammino semplice: tutti i vertici sono distinti.

Esempio

$\mathbf{G}$ = (V,E);

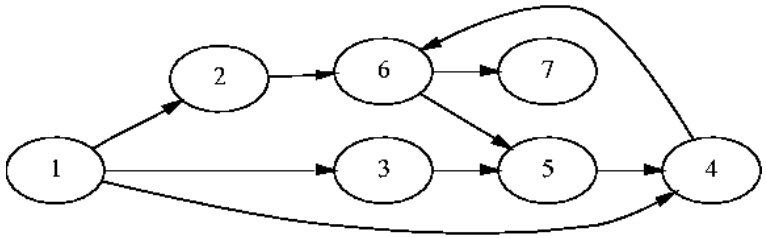
$\mathbf{V}$ = {1,2,3,4,5,6,7};

$\mathbf{E}$ = {(1,2),(1,3),(1,4),(2,6),(3,5),(4,6),(6,5),(6,7),(5,4)}

Cammini da 1 a 6:

1, 2, 6 – 1, 4, 6

1, 3, 5, 4, 6 – 1, 4, 6, 5, 4, 6

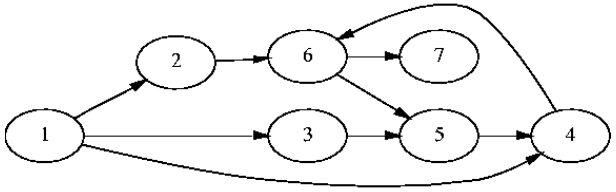


Semicammino da X a Y

Sequenza di nodi $(a_0, \ldots, a_n)$ *tale che*

1. $a_0 = X$
2. $a_n = Y$
3. per ogni $i = 0, \ldots, n-1$: $(a_i, a_{i+1}) \in E$ *oppure* $(a_{i+1}, a_i) \in E$.

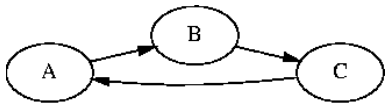
Semicammino da 1 a 6: 1, 3, 5, 6



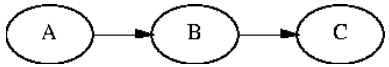
Un grafo è

fortemente connesso sse per ogni $X, Y \in V$ esiste un cammino da X a Y;

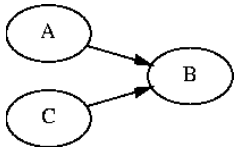
debolmente connesso sse per ogni $X, Y \in V$ esiste un semicammino da X a Y



fortemente connesso



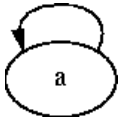
debolmente connesso

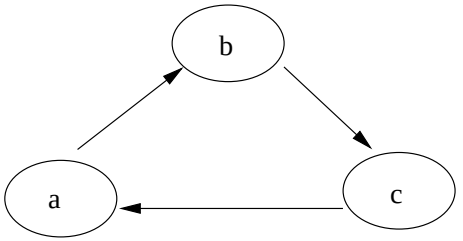


debolmente connesso

Cicli

Cammino di lunghezza ≥ 1 (almeno un arco) che inizia e termina in uno stesso nodo





Sequenza di nodi $(a_0, \ldots, a_n)$ tale che

1. $a_0 = a_n$
2. $n \geq 1$
3. per ogni $i = 0, \ldots, n-1$: $(a_i, a_{i+1}) \in E$.

Due cammini $(a_0, \ldots, a_i, \ldots, a_n)$ e $(a_i, \ldots, a_n, a_0, \ldots, a_{i-1})$ formano lo stesso ciclo

Ciclo semplice $(a_0, \ldots, a_n)$: $a_1, ..., a_n$ sono tutti distinti.

Cappio ciclo di lunghezza 1

Grafo semplice senza cappi

Grafo aciclico senza cicli

L'insieme degli archi rappresenta una relazione R simmetrica: $R(x, y) \Rightarrow R(y, x)$

L'insieme E degli archi è costituito da *coppie non ordinate* di vertici: $\{x, y\}$, con $x, y \in E$ e $x \neq y$

In un grafo non orientato i cappi sono proibiti

Usiamo ugualmente la notazione (x, y), con la convenzione che (x, y) e (y, x) denotano lo stesso arco.

Cammini e cicli in un grafo non orientato

La relazione di "raggiungibilità" mediante un cammino è una relazione di equivalenza (riflessiva, simmetrica e transitiva)

Ciclo: cammino $(a_0, \ldots, a_n)$ tale che

- $n \geq 3$; la lunghezza del cammino (numero di archi) è ≥ 3;
- $a_0 = a_n$;
- $a_1, \ldots, a_n$ sono tutti distinti.

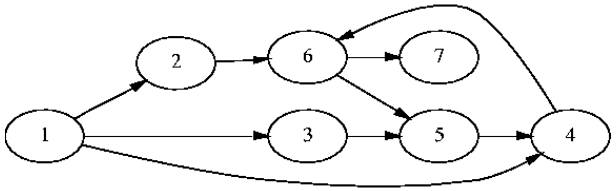
Quindi $\left.\begin{array}{l} n \\ n, n \\ n, m, n \end{array}\right\}$ non sono cicli

Componenti connesse

Grafo connesso: ogni coppia di vertici è connessa da un cammino.

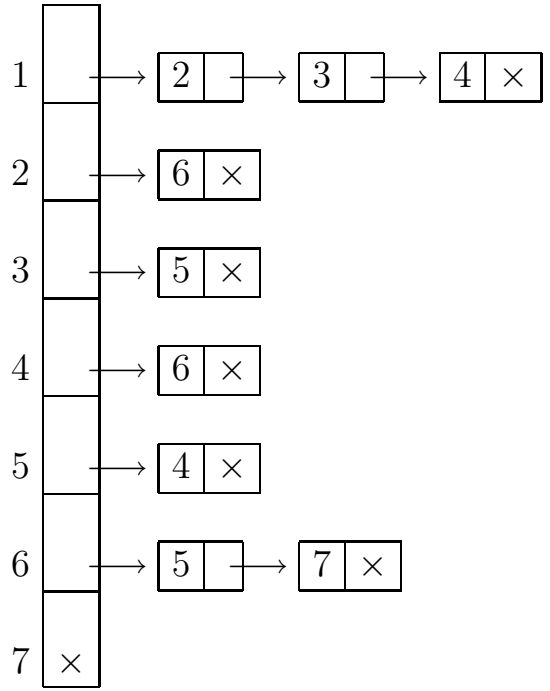
Componente connessa di un grafo: classe di equivalenza dei nodi rispetto alla relazione di "raggiungibilità".

Rappresentazione dei grafi mediante liste di adiacenza



Un array Adj di lunghezza $length[Adj] = V$

Per $u \in V$: *$Adj[v]$ è una lista con tutti i vertici u tali che $(v, u) \in E$*



È un modo compatto per la rappresentazione di *grafi sparsi:* $\mid E \mid << \mid V \mid^2$

Somma delle lunghezze di tutte le liste di adiacenza: $\mid E \mid$ grafi orientati; $2 \mid E \mid$ grafi non orientati

Memoria necessaria: $O(V + E)$

Rappresentazione dei grafi mediante matrice di adiacenza

Rappresentazione preferita per grafi densi: $\mid E \mid \sim \mid V \mid^2$

oppure

quando si vuole essere in grado di dire rapidamente se esiste un arco che collega due nodi

Matrice $A = (a_{ij})$ *di dimensione* $\mid V \mid \times \mid V \mid$

$$a_{ij} = \begin{cases} 1 & se\ (i,j) \in E \\ 0 & altrimenti \end{cases}$$

Riga i*: successori del nodo* i

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	0	0	1	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	1	0	0	0
6	0	0	0	0	1	0	1
7	0	0	0	0	0	0	0

Memoria necessaria: $\Theta(\mid V \mid^2)$

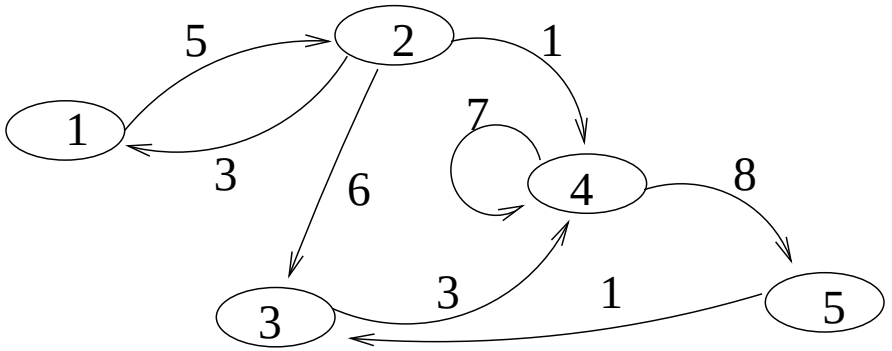
Tuttavia in un grafo non orientato la matrice di adiacenza è simmetrica. In alcune applicazioni può convenire memorizzare soltanto i dati che compaiono solo sopra la diagonale principale.

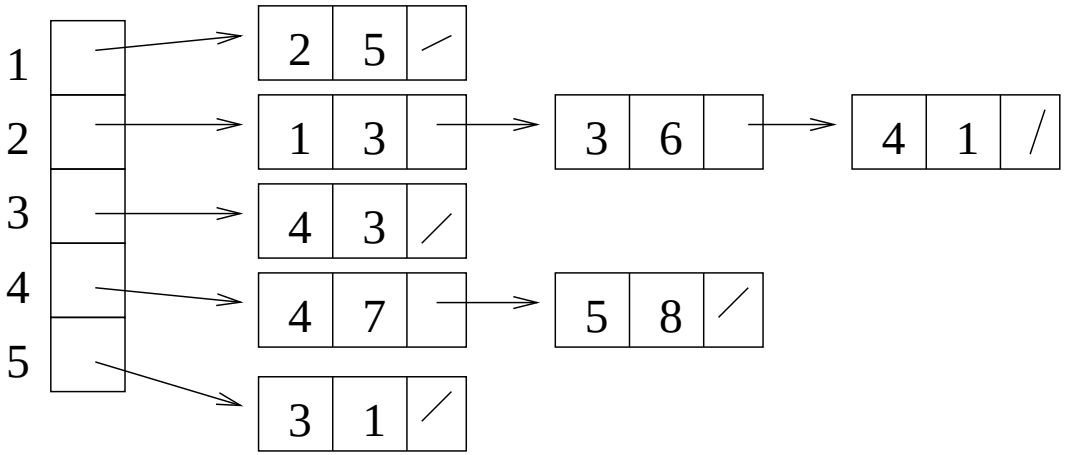
Grafi pesati

Ad ogni arco è associato un peso $w : E \longrightarrow \mathbb{R}$

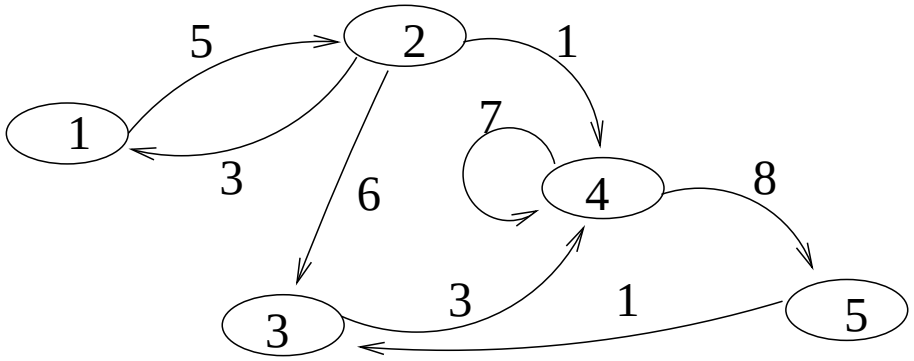
Rappresentazione

liste di adicenza il peso $w(u, v)$ dell'arco (u, v) è memorizzato insieme al nodo v nella lista di adiacenza di u





matrice di adiacenza il peso $w(u, v)$ dell'arco (u, v) è memorizzato in posizione a_{uv} (riga u, colonna v della matrice di adiacenza), al posto di 1.



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

Algoritmi di visita

A partire da un dato nodo iniziale, analizzare tutti i nodi raggiungibili dal nodo dato

Attenzione ai cicli

Visita in ampiezza : *Breadth First Search* (*BFS*):

se il nodo di partenza $start$ non è già stato visitato allora si analizza $start$, si visitano tutti i suoi successori (ricordando che $start$ è già stato considerato), poi tutti i successori dei successori di $start$, e così via.

```
- visita in ampiezza il grafo G partendo dal nodo START:
-    colora tutti i nodi di bianco
-    inserisci START nella coda e coloralo di grigio;
-    finche' la coda non e' vuota:
-        estrai il primo elemento X dalla coda
-        analizza X
-        metti nella coda tutti i successori bianchi di X e colorali di grigio
```

Visita in profondità: *Depth First Search* (*DFS*):

se il nodo di partenza $start$ non è stato già visitato, si analizza $start$ e, per ogni successore x di $start$, si visita, con lo stesso metodo, il grafo a partire da x, ricordando che $start$ è già stato visitato.

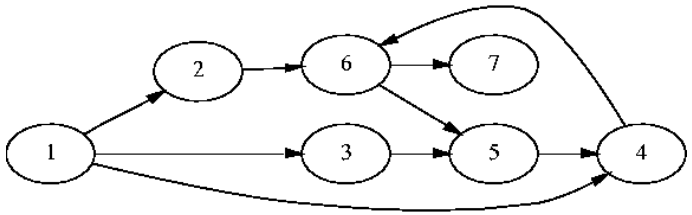
```
- visita in profondita' il grafo G partendo dal nodo START:
-    analizza START e marcalo "scoperto" (colore grigio)
-    finche' ci sono successori di START non ancora visitati (non grigi):
-        sia X il prossimo
-        visita in profondita' G a partire da X
-    colora di nero START (visita completata)
```

Visita di un grafo - Esempio

Bisogna ricordare i nodi già visitati per la possibile esistenza di cammini ciclici

Visita in ampiezza i nodi in attesa di essere visitati vengono memorizzati in una *coda*
Visita in profondità i successori di un nodo vengono visitati ad uno ad uno mediante chiamate ricorsive.
Equivalentemente, i nodi in attesa di essere visitati vengono memorizzati in una *pila*

Esempio



La "colorazione" dei nodi è il meccanismo che evita di inserire uno stesso nodo più volte nella struttura pila o coda. Di conseguenza nessun nodo viene visitato più di una volta

PROFONDITA' nodi visitati	PILA	AMPIEZZA nodi visitati	CODA
	1		1
1	2,3,4	1	2,3,4
1,2	6,3,4	1,2	3,4,6
1,2,6	7,5,3,4	1,2,3	4,6,5
1,2,6,7	5,3,4	1,2,3,4	6,5
1,2,6,7,5	3,4	1,2,3,4,6	5,7
1,2,6,7,5,3	4	1,2,3,4,6,5	7
1,2,6,7,5,3,4		1,2,3,4,6,5,7	

Operazione fondamentale:

Succ(Adj,x): riporta la lista dei successori del nodo x nel grafo rappresentato dall'array Adj.

```
SUCC(Adj,x)
1. return Adj[x]
```

Visita in ampiezza

Per "ricordare" i nodi già inseriti nella coda e per segnalare i nodi già visitati, utilizziamo un *array color*, tale che:

$$color[u] = \begin{cases} white & \text{se } u \text{ non è stato visitato né inserito nella coda} \\ gray & \text{se } u \text{ è stato inserito nella coda, ma non ancora visitato} \\ black & \text{se } u \text{ è stato visitato} \end{cases}$$

Visita in profondità

Per "ricordare" i nodi "scoperti": da cui è partita una visita in profondità e che quindi sono stati inseriti nella pila di attivazione delle chiamate ricorsive:

$$color[u] = \begin{cases} white & \text{se } u \text{ non è stato visitato} \\ gray & \text{se da } u \text{ è partita una visita in profondità} \\ black & \text{se } u \text{ è stato completamente visitato (con tutti i suoi successori)} \end{cases}$$

Visita in Ampiezza del grafo Adj a partire da s: Algoritmo

s può essere un qualunque nodo del grafo

```
-Inizializzazione di color[u]:
     colorare tutti i vertici di bianco
     eccetto s, che viene colorato di grigio
-Inizializzazione della coda Q:
     inserimento di s
-Ciclo fino a che Q non e' vuota:
     estrarre u dalla coda
     per ogni vertice v adiacente a u:
         se v e' bianco, colorarlo di grigio
                         e inserirlo nella coda
    visitare u e colorarlo di nero
```

Visita in Ampiezza del grafo rappresentato come lista di adiacenza *Adj*: Pseudocodifica

```
BFS(Adj,s)          ;; s e' il nodo di partenza
1. for ogni vertice u, eccetto s
2.    do color[u] <- white
3. color[s] <- gray
4. ;; inizializzazione della coda
5. QUEUE-EMPTY(Q)
6. ENQUEUE(Q,s)
7. while NOT(QUEUE-NULL(Q))
         ;; estrazione del primo elemento
8.    do u <- DEQUEUE(Q)
         ;; inserimento dei successori di u
9.       x <- FIRST(SUCC(Adj,u))
10.      while x != NIL
11.         do k <- KEY(x)
12.            if color[k] = white
13.               then color[k] <- gray
14.                    ENQUEUE(Q,k)
15.            x <- NEXT(x)
16.      visita u
17.      color[u] <- black
```

Visita in Ampiezza del grafo rappresentato come Matrice di Adiacenza: Pseudocodifica

Sia A la matrice di adiacenza che rappresenta il grafo G

```
BFS(A,s)          ;; s e' il nodo di partenza
1. for ogni vertice u, eccetto s
2.    do color[u] <- white
3. color[s] <- gray
4. ;; inizializzazione della coda
5. QUEUE-EMPTY(Q)
6. ENQUEUE(Q,s)
7. while NOT(QUEUE-NULL(Q))
         ;; estrazione del primo elemento
8.    do u <- DEQUEUE(Q)
         ;; inserimento dei successori di u
9.    for i=1 to n
10.     do if A[u,i]!=0 and color[i] = white
11.          then color[i] <- gray
12.               ENQUEUE(Q,i)
13.     visita u
14.   color[u] <- black
```

L'algoritmo di visita non cambia: cambia solo il modo in cui si esaminano i successori di u nel grafo

Applicazione della visita in ampiezza: calcolo dei cammini minimi

Distanza di v da u minimo numero di archi in un cammino da v a u

L'algoritmo calcola la distanza da s (nodo sorgente) di tutti i nodi u raggiungibili da s, e memorizza i cammini minimi corrispondenti

Risultati negli array d e p:

d[u] distanza di u da s (∞ se u non è raggiungibile da s);

p[u] il "padre" (predecessore) di u nel cammino minimo da s a u
(NIL se u non è raggiungibile da s o $u = s$).

Fissato il nodo padre si visita la lista dei suoi successori
Così ci si ricorda del padre per aggiornare l'array dei predecessori

Visita in ampiezza con calcolo delle distanze: Pseudocodifica

```
BFS(Adj,s)
 1. for ogni vertice u, eccetto s
 2.    do color[u] <- white
 3.       d[u] <- infinito
 4.       p[u] <- NIL
 5. color[s] <- gray
 6. d[s] <- 0
 7. p[s] <- NIL
 8. QUEUE-EMPTY(Q)
 9. ENQUEUE(Q,s)
10. while NOT(QUEUE-NULL(Q))
11.    do u <- DEQUEUE(Q)
12.       x <- FIRST(SUCC(Adj,u))
13.       while x != NIL
14.          do k <- KEY(x)
15.             if color[k] = white
16.                then color[k] <- gray
17.                     d[k] <- d[u] + 1
18.                     p[k] <- u
19.                     ENQUEUE(Q,k)
20.             x <- NEXT(x)
21.       color[u] <- black
```

Complessità della visita in ampiezza

Considerazioni

1. L'inizializzazione degli array `color`, `d` e `p` (righe 1–7) richiede un tempo $O(V)$. L'inizializzazione della coda (righe 8 e 9) richiede un tempo $O(1)$.
2. Poiché dopo l'inizializzazione nessun vertice verrà più colorato di bianco, il test della riga 15 garantisce che ogni vertice è inserito nella coda al massimo una volta, e di conseguenza che ne venga estratto al massimo una volta. Ciascun inserimento ed estrazione (righe 11 e 19) richiede tempo $O(1)$, quindi: *complessivamente il tempo dedicato alle operazioni sulla coda è* $O(V)$.
3. La lista di adiacenza di ogni vertice viene scandita solo quando il vertice è estratto dalla coda (riga 11): al massimo una volta.
   Complessivamente poiché la somma delle lunghezze delle liste di adiacenza è $\Theta(E)$*, il tempo speso per la scansione di tutte le liste di adiacenza è* $O(E)$.
4. Il tempo per colorare un nodo e per visitarlo (righe 16–18) è $O(1)$. Ciascun nodo viene visitato al massimo una volta (quando è inserito nella coda).
   Complessivamente la visita richiede tempo $O(V)$.

Il tempo totale di esecuzione dell'algoritmo BFS è $O(V + E)$

Visita in profondità: considerazioni

- Di solito si utilizza associando a ciascun vertice del grafo delle *informazioni temporali: tempi di inizio e fine visita dei vertici.*
  - $d[u]$ registra il momento in cui il vertice u viene scoperto
    corrispnde al momento della colorazione di grigio
  - $f[u]$ registra il momento in cui la visita del vertice u (e di tutti i nodi raggiungibili da v) è completata
    corrispnde al momento della colorazione di nero
- le informazioni temporali sono interi compresi tra 1 e $2 \cdot \mid V \mid$: ad ogni vertice è assegnato un tempo d'inizio e uno di fine visita
- Per ogni vertice u si ha: $d[u] < f[u]$

- Il vertice u è:
  - white prima di $d[u]$
    Inizialmente i vertici sono tutti bianchi
  - gray tra $d[u]$ e $f[u]$
    Quando un vertice u viene "scoperto", diventa grigio: inizia la visita di u
  - black dopo $f[u]$
    Quando la visita di u termina (tutti i successori sono stati visitati), u diventa nero.
- *time* è la variabile che registra i tempi

Visita in profondità generale: Algoritmo

Vengono visitati tutti i nodi del grafo

```
DFS(Adj,s)
1. for ogni vertice u
2.   do color[u] <- white
3.      p[u] <- NIL
4. time <-0
5. for ogni vertice u
6.      do if color[u] = white
7.            then DFS-visit(u)

DFS-visit(u)          ;; visita in profondita' a partire dal nodo u
1.  color[u] <- gray
2.  time <- time + 1
3.  d[u] <- time

4.  ;; visita di tutti gli archi adiacenti a u
5.  v <- FIRST(SUCC(Adj,u))
6.  while v != NIL
7.     do k <- KEY(v)
8.        if color[k] = white
9.           then p[v] <- u
10.               DFS-visit(k)
11.        v <- NEXT(v)
12. color[u] <- black
13. time <- time + 1
14. f[u] <- time
```

Complessità asintotica della visita in profondità

assumendo che la visita di un nodo richieda tempo costante

- DFS: Ogni vertice è colorato di bianco esattamente una volta (inizializzazione).
  Complessivamente il tempo è $\Theta(V)$.

- DFS-visit:

  Durante `DFS-visit(u)`, il ciclo di scansione della lista dei successori di `u` viene eseguito $\mid Adj[u] \mid$ volte.

  Il test alla riga 8 assicura che `DFS-visit` è richiamata al più una volta per ogni nodo, dato che il nodo viene subito colorato di grigio.
  Complessivamente la visita dei nodi richiede tempo $O(V)$.

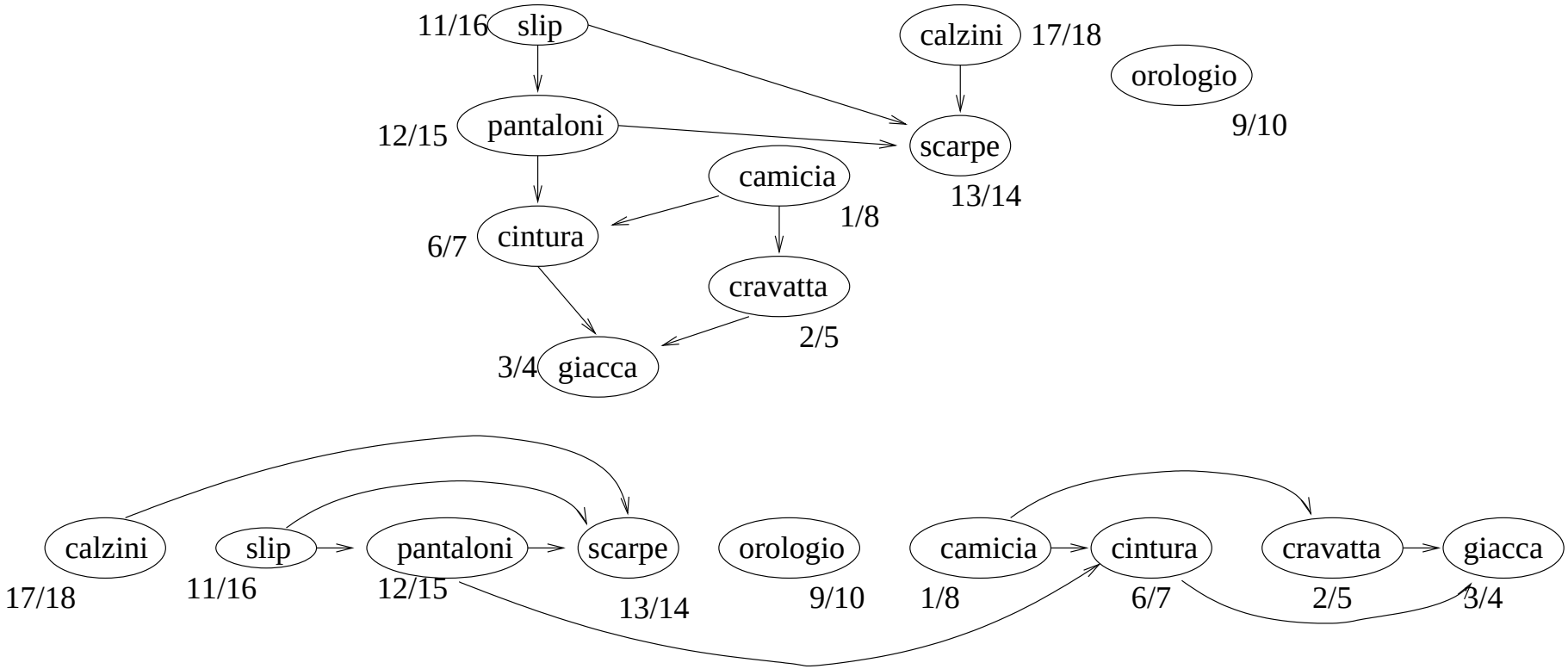
  Poiché la somma delle lunghezze delle liste di adiacenza è $\Theta(\mid E \mid)$,
  il costo complessivo per l'esecuzione delle linee 6-11 è $\Theta(\mid E \mid)$.

tutte le altre istruzioni hanno costo $\Theta(1)$

Il tempo di esecuzione di DFS è $O(V + E)$

Applicazione della visita in profondità: Ordinamento topologico di grafi orientati aciclici (DAG)

Un ordinamento topologico di un dag G è un ordinamento lineare dei vertici tale che se (u, v) è un arco in G, allora u viene prima di v.



Un algoritmo di ordinamento topologico si basa sulla visita in profondità del grafo: per ciascun nodo viene registrato il tempo di inizio e fine visita. L'ordinamento è per tempi decrescenti di fine visita. Non è necessariamente unico.

Ordinamento topologico - Pseudocodifica

La lista L contiene, alla fine della visita, un ordinamento topologico del grafo
Non appena finisce la visita di un nodo, esso viene inserito in L

DFS(G)

1. **for** ogni vertice $u \in V[G]$
2. **do** $color[u] \leftarrow white$
3. $\pi[u] \leftarrow NIL$
4. $time \leftarrow 0$
5. $L \leftarrow$LIST-EMPTY()
6. **for** ogni vertice $u \in V[G]$
7. **do if** $color[u] = white$
8. **then** DFS-visit(u, L)
9. return L

DFS-visit(u,L)

1. $color[u] \leftarrow gray$
2. $d[u] \leftarrow time \leftarrow time + 1$
3. **for** ogni vertice v adiacente a u
4. **do if** $color[v] = white$
5. **then** $\pi[v] \leftarrow u$
6. DFS-visit(v, L)
7. $color[u] \leftarrow black$
8. inserire u in testa a L
9. $f[u] \leftarrow time \leftarrow time + 1$

Esercizi

1. Scrivere un programma che, data la rappresentazione di un grafo G e un nodo n di G, verifichi se n è un pozzo.
2. Scrivere un programma che, data la rappresentazione di un grafo G e un nodo n di G, riporti il grado di ingresso di n.
3. Scrivere un programma che, data la rappresentazione di un grafo G e due nodi s e g di G, verifichi se g è raggiungibile da s, visitando il grafo a partire da s.
4. Scrivere un programma che, data la rappresentazione di un grafo G e un nodo n di G, determini se esiste in G un ciclo su n.
5. Scrivere lo pseudo-codice per l'algoritmo DFS nel caso della rappresentazione mediante matrice di adiacenza.
6. Illustrare l'algoritmo di ordinamento topologico. Descrivere l'algoritmo senza scrivere la pseudo-codifica. Dire qual è l'ordinamento che si ottiene a partire dal nodo etichettato con *a* (scrivendo su questo testo i tempi di visita).