

Algoritmi e Strutture Dati

Capitolo 9 - Grafi

Alberto Montresor
Università di Trento

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Sommario

1 Introduzione

- Esempi
- Definizioni
- Specifica
- Memorizzazione

2 Visite dei grafi

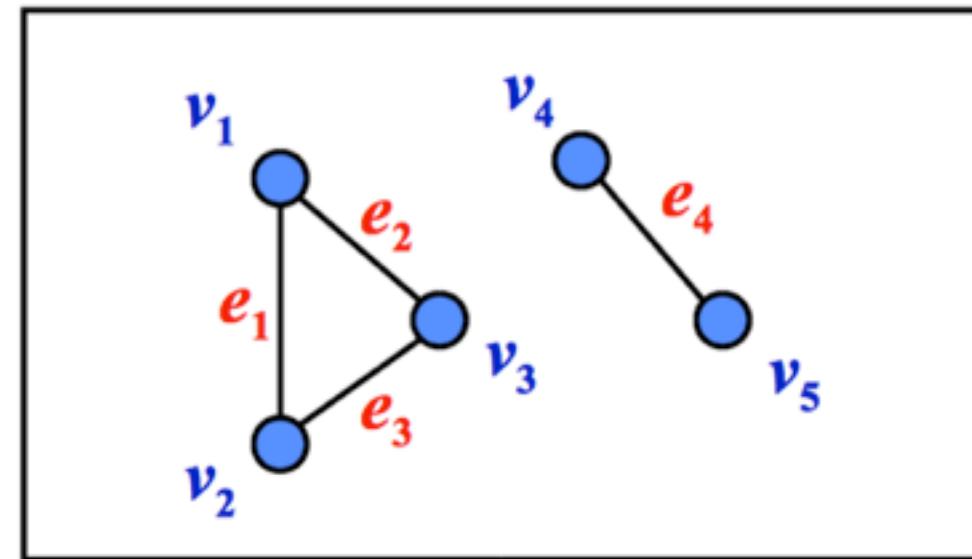
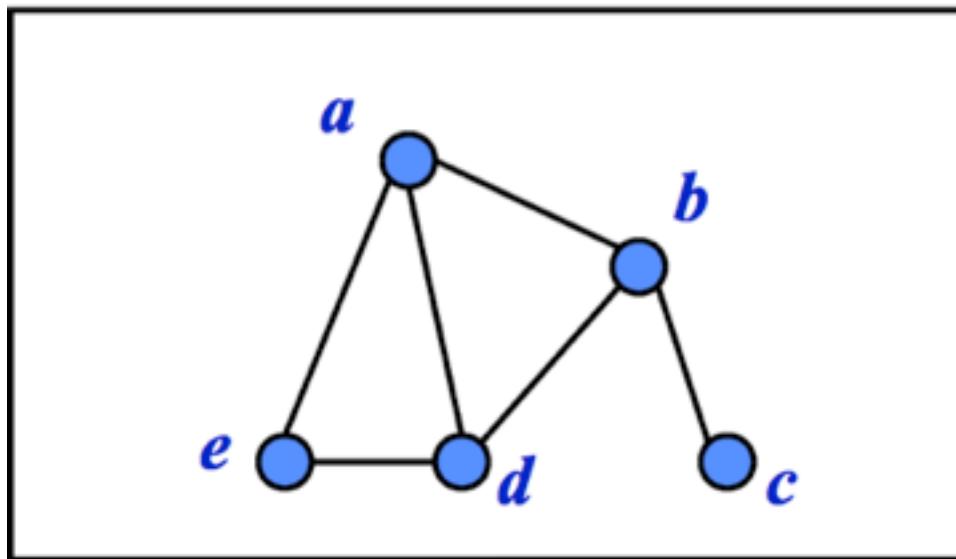
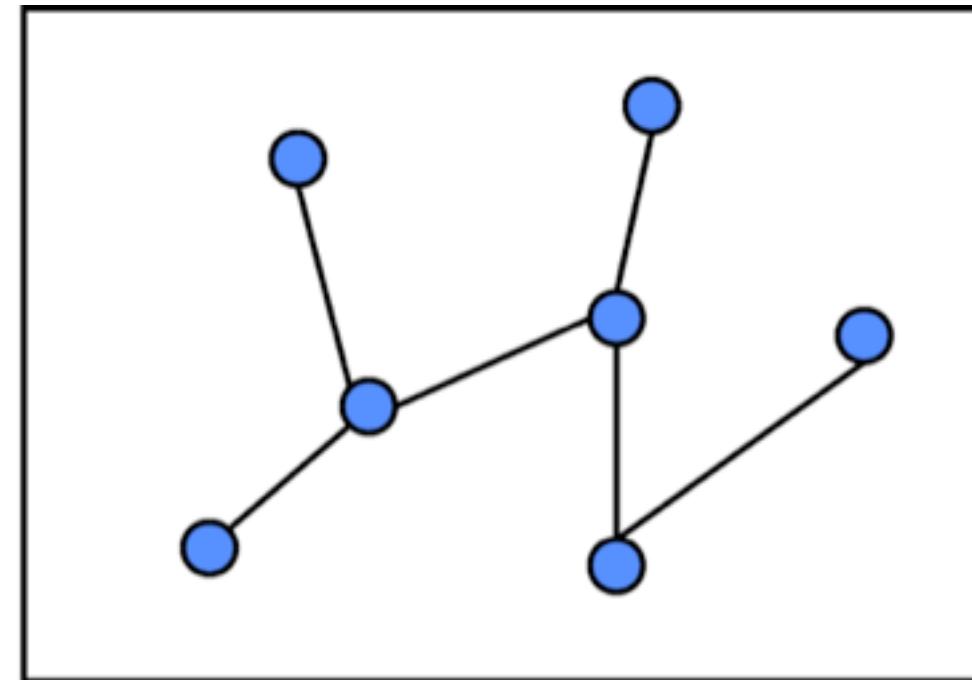
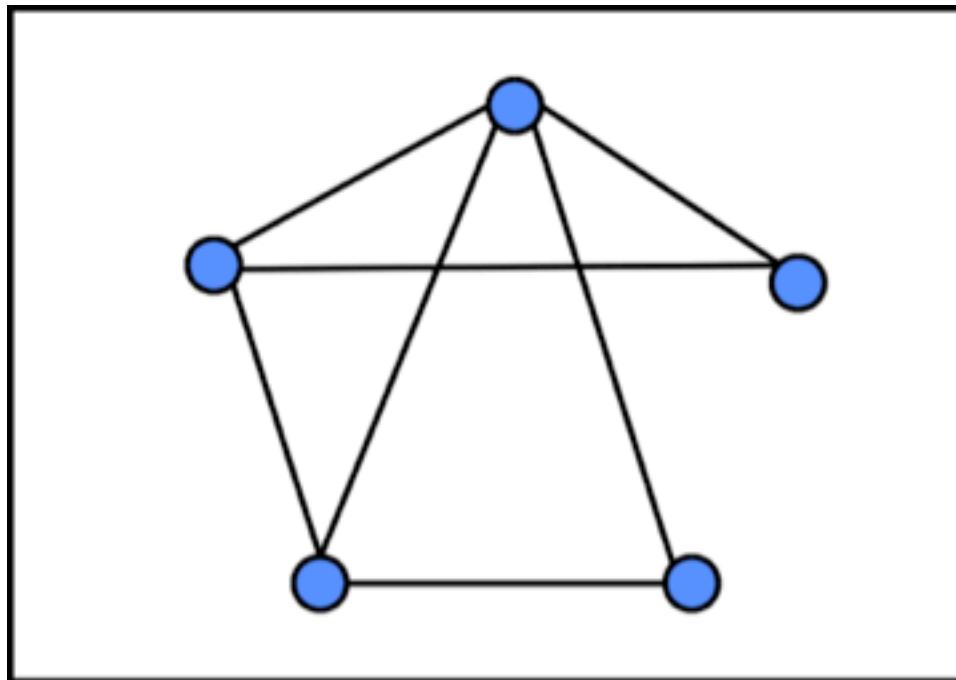
3 BFS

- Cammini più brevi

4 DFS

- Componenti connesse
- Grafi aciclici non orientati
- Classificazione degli archi
- Grafi aciclici orientati
- Ordinamento topologico
- Componenti fortemente connesse

Esempi di grafi



Problemi relativi ai grafi

Problemi in grafi non pesati

- Ricerca del cammino più breve (misurato in numero di archi)
- Componenti (fortemente) connesse, verifica ciclicità, ordinamento topologico

Problemi in grafi pesati

- Cammini di peso minimo
- Alberi di copertura di peso minimo
- Flusso massimo

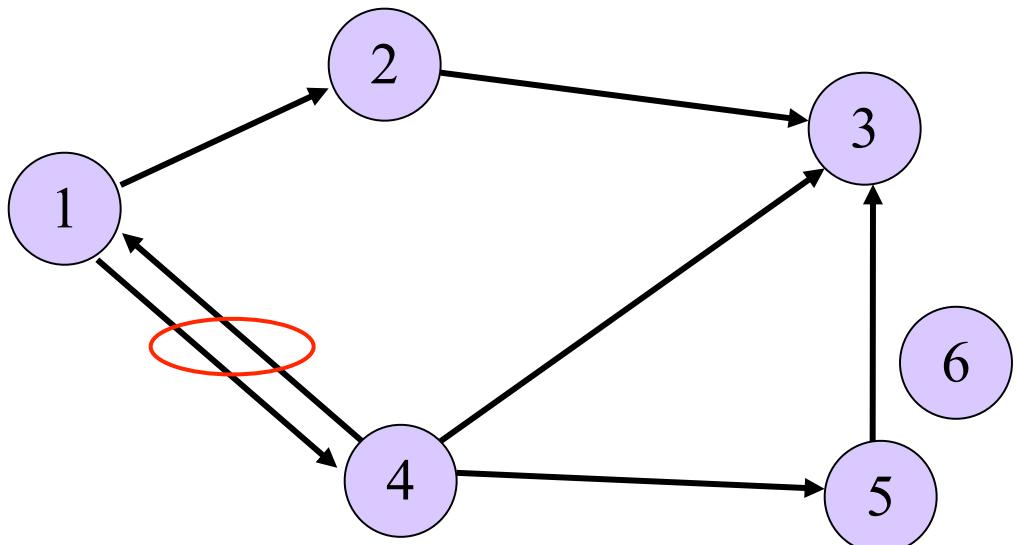
Problemi sui grafi

- ♦ **Visite**
 - ♦ Visite in ampiezza (numero di Erdös)
 - ♦ Visite in profondità (ordinamento topologico, componenti (fortemente) connesse)
- ♦ **Cammini minimi**
 - ♦ Da singola sorgente
 - ♦ Fra tutte le coppie di vertici
- ♦ **Alberi di connessione minimi**
- ♦ **Problemi di flusso**
- ♦

Grafi orientati e non orientati: definizione

- ♦ Un **grafo orientato** G è una coppia (V, E) dove:

- ♦ Insieme finito dei vertici V
- ♦ Insieme degli archi E : relazione binaria tra vertici

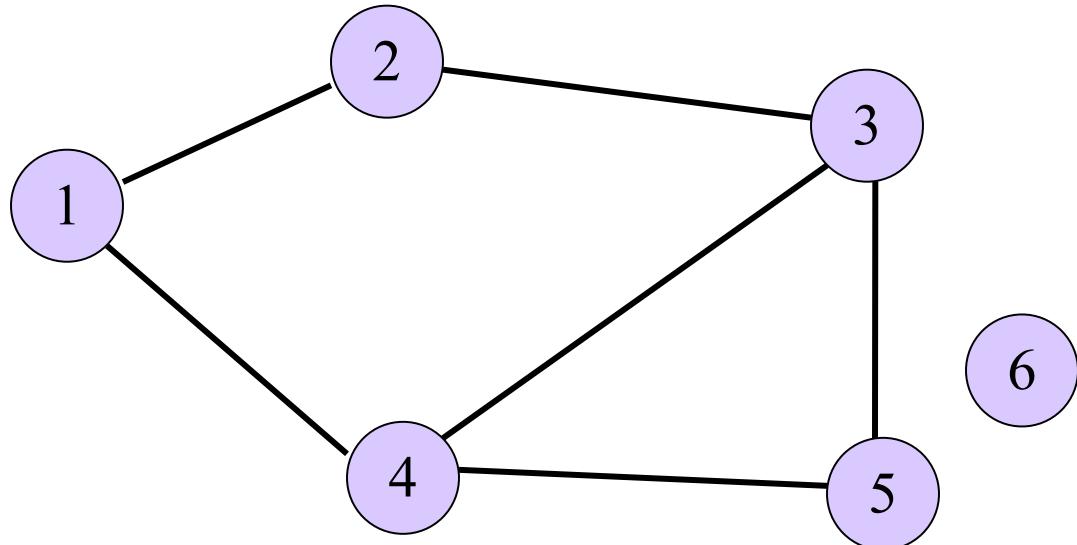


$$V = \{1, 2, 3, 4, 5, 6\}$$

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

- ♦ Un **grafo non orientato** G è una coppia (V, E) dove:

- ♦ Insieme finito dei vertici V
- ♦ Insieme degli archi E : coppie non ordinate



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

$$E = \{[1,2], [1,4], [2,3], [3,4], [3,5], [4,5]\}$$

Definizioni: incidenza e adiacenza

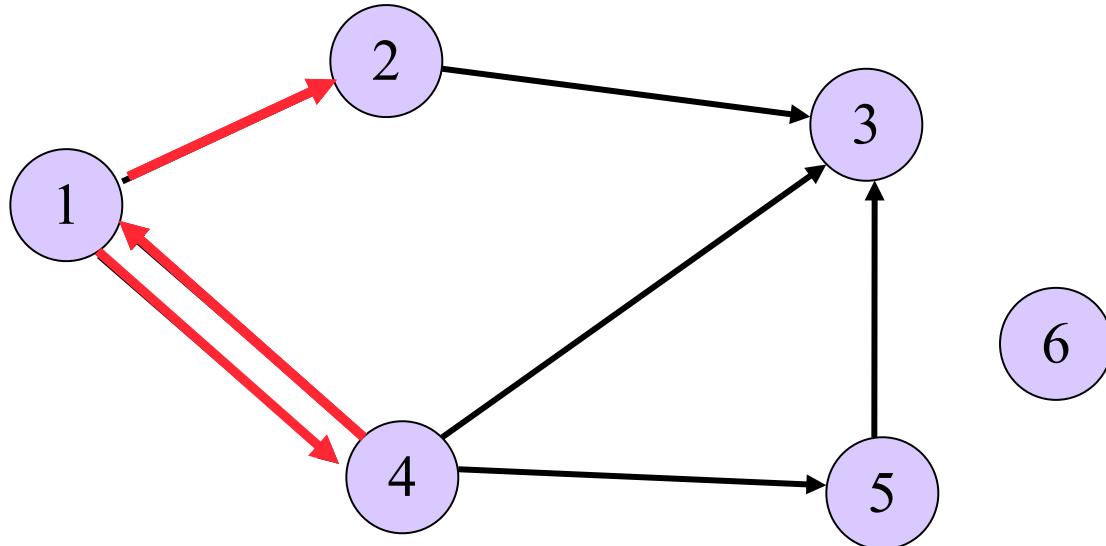
- ♦ **In un grafo orientato**

- ♦ un arco (u,v) si dice *incidente* da u in v

- ♦ **In un grafo non orientato**

- ♦ la relazione di adiacenza tra vertici è simmetrica

- ♦ **Un vertice v si dice *adiacente* a u se e solo se $(u, v) \in E$**



(1,2) è incidente da 1 a 2
(1,4) è incidente da 1 a 4
(4,1) è incidente da 4 a 1

2 è adiacente ad 1
3 è adiacente a 2, 4, 5
1 è adiacente a 4 e viceversa
2 non è adiacente a 3,4
6 non è adiacente ad alcun vertice

Dimensioni del grafo

Definizioni

- $n = |V|$: numero di nodi
- $m = |E|$: numero di archi

Alcune relazioni fra n e m

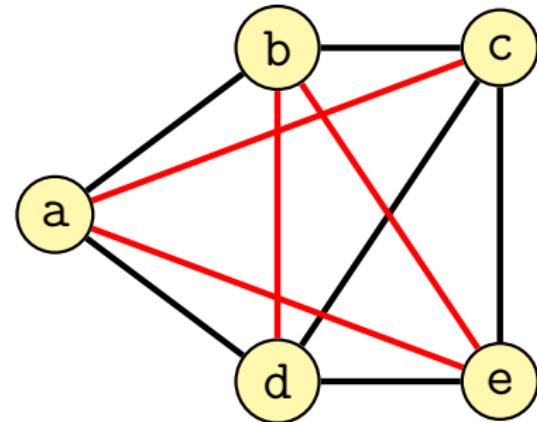
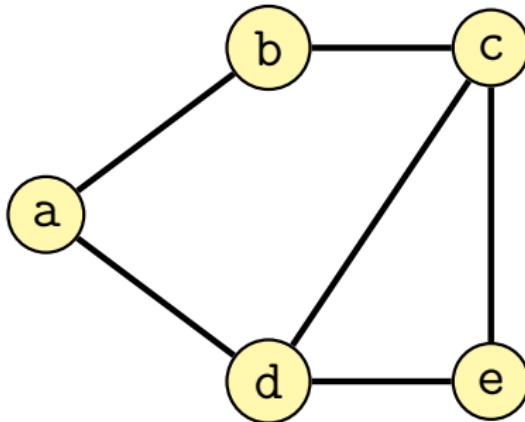
- In grafo non orientato, $m \leq \frac{n(n-1)}{2} = O(n^2)$
- In grafo orientato, $m \leq n^2 - n = O(n^2)$

Complessità di algoritmi su grafi

- La complessità è espressa in termini sia di n che di m
(ad es. $O(n + m)$)

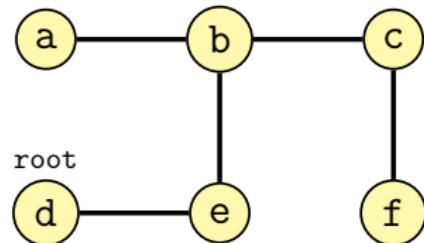
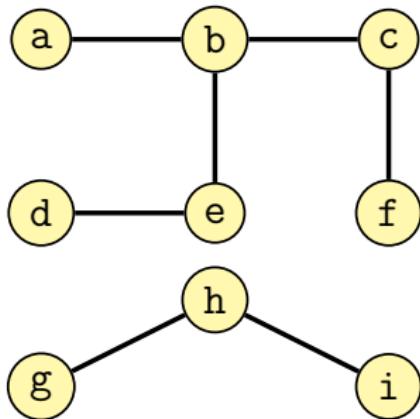
Alcuni casi speciali

- Un grafo con un arco fra tutte le coppie di nodi è detto **completo**
- Informalmente (non c'è accordo sulla definizione)
 - Un grafo si dice **sparso** se ha "pochi archi"; grafi con $m = O(n)$, $m = O(n \log n)$ sono considerati sparsi
 - Un grafo si dice **denso** se ha "tanti archi"; e.g., $m = \Omega(n^2)$



Alcuni casi speciali

- Un **albero libero** (free tree) è un grafo connesso con $m = n - 1$
- Un **albero radicato** (rooted tree) è un grafo connesso con $m = n - 1$ nel quale uno dei nodi è designato come radice.
- Un insieme di alberi è un grafo detto **foresta**



- ♦ **Poniamo**

- ♦ $n = |V|$ numero di nodi
- ♦ $m = |E|$ numero di archi

- ♦ **Matrice di adiacenza**

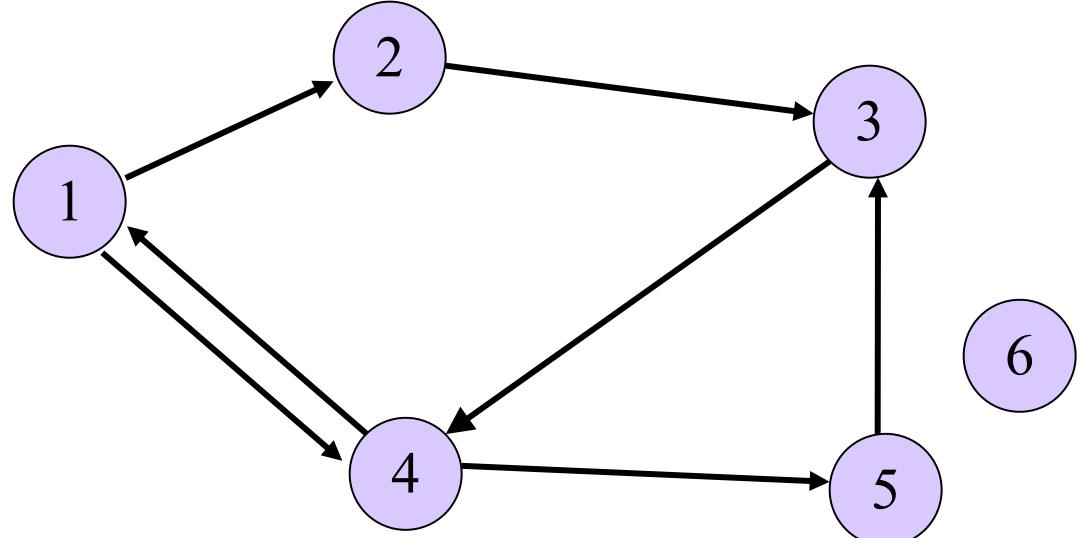
- ♦ Spazio richiesto $O(n^2)$
- ♦ Verificare se il vertice u è adiacente a v richiede tempo $O(1)$
- ♦ Elencare tutti gli archi costa $O(n^2)$

- ♦ **Liste di adiacenza**

- ♦ Spazio richiesto $O(n+m)$
- ♦ Verificare se il vertice u è adiacente a v richiede tempo $O(n)$
- ♦ Elencare tutti gli archi costa $O(n+m)$

Matrice di adiacenza: grafo orientato o non orientato

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



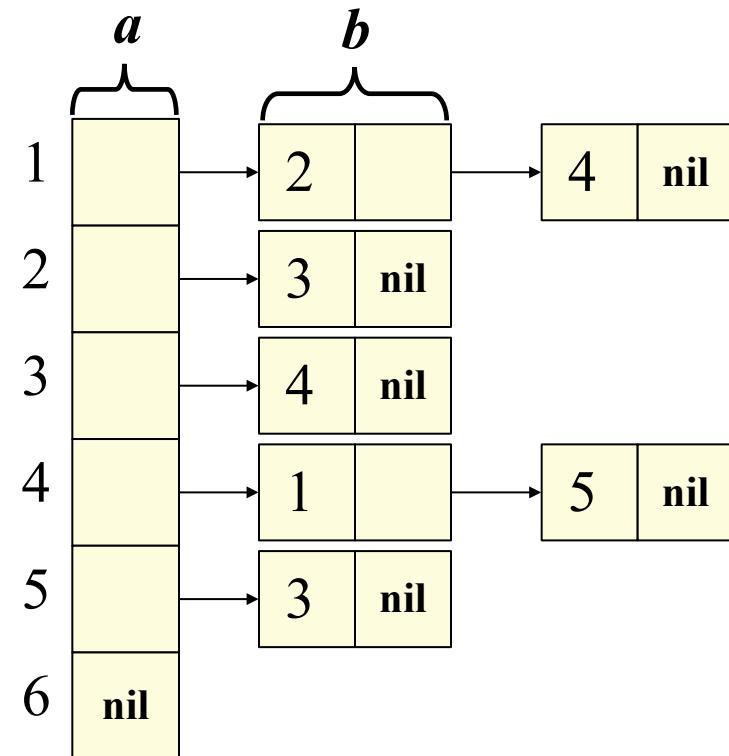
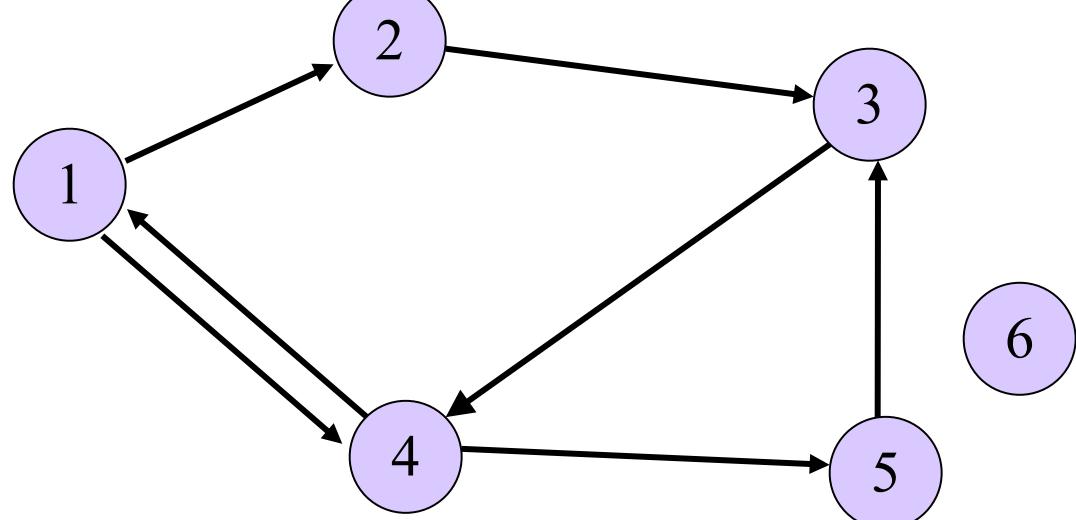
Spazio: n^2

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

Lista di adiacenza: grafo orientato

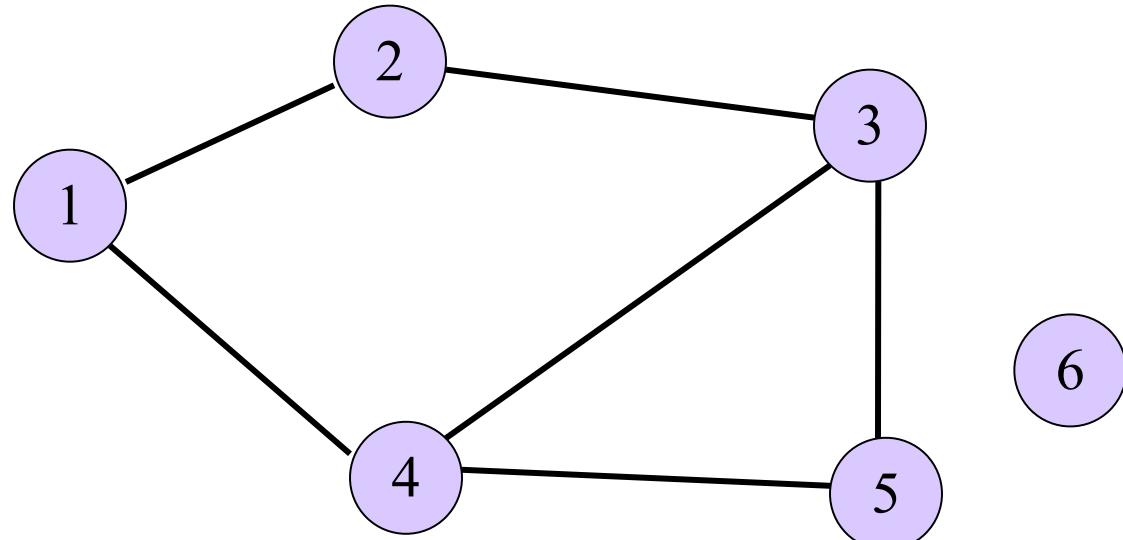
$$G.adj(u) = \{ v \mid (u,v) \in E \}$$

Spazio: $a \cdot n + b \cdot m$



Matrice di adiacenza: grafo non orientato

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



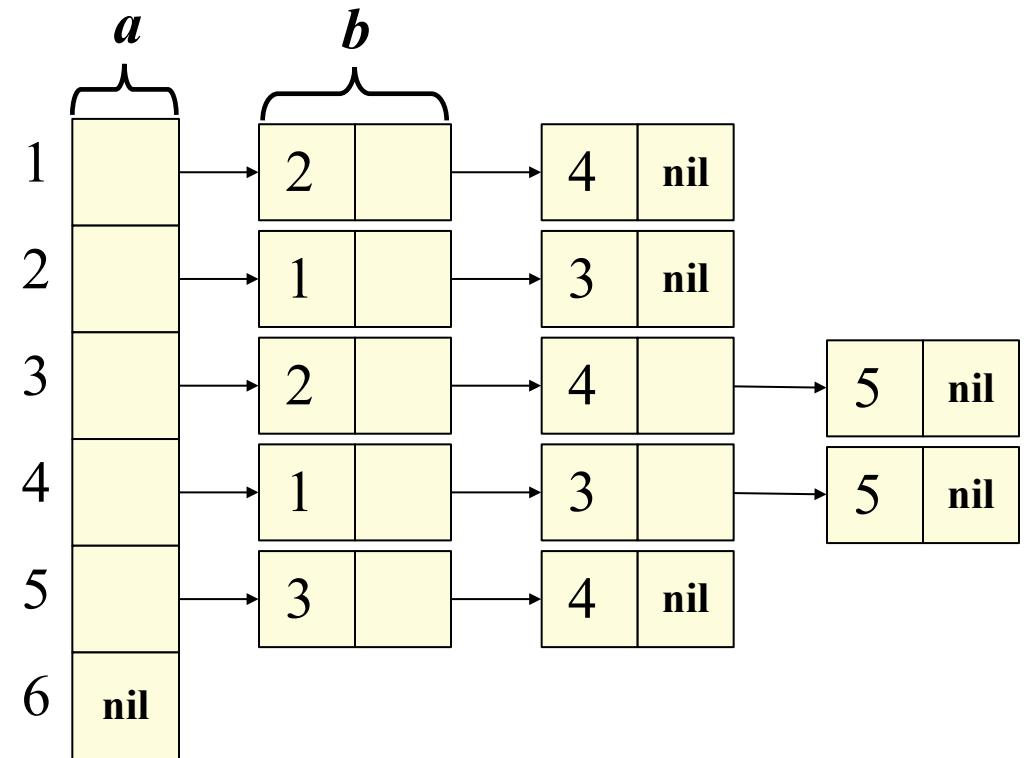
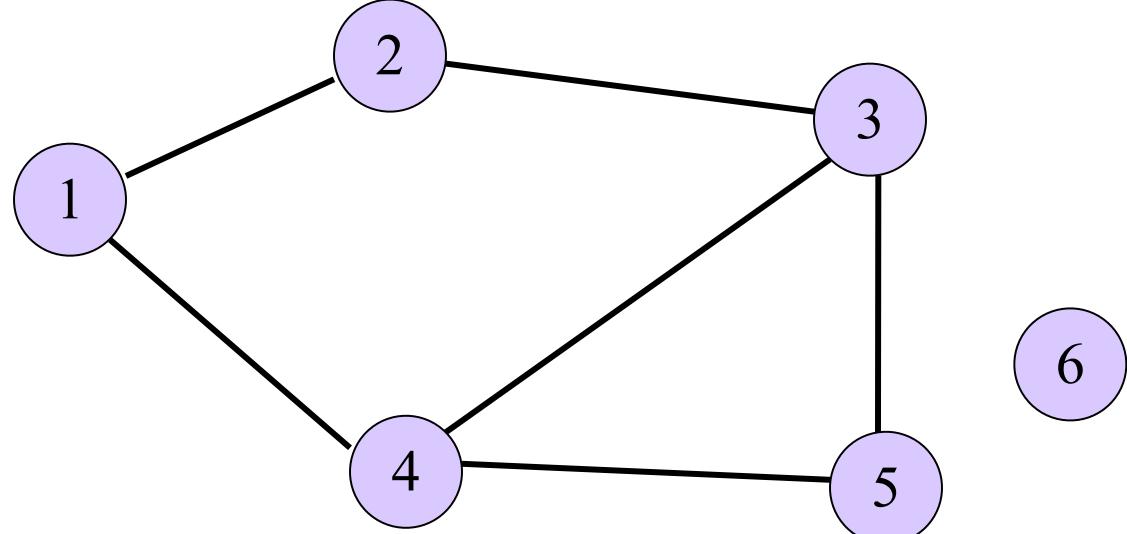
Spazio: n^2 o $n(n+1)/2$

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

Liste di adiacenza: grafo non orientato

$$G.adj(u) = \{ v \mid [u,v] \in E \text{ or } [v,u] \in E\}$$

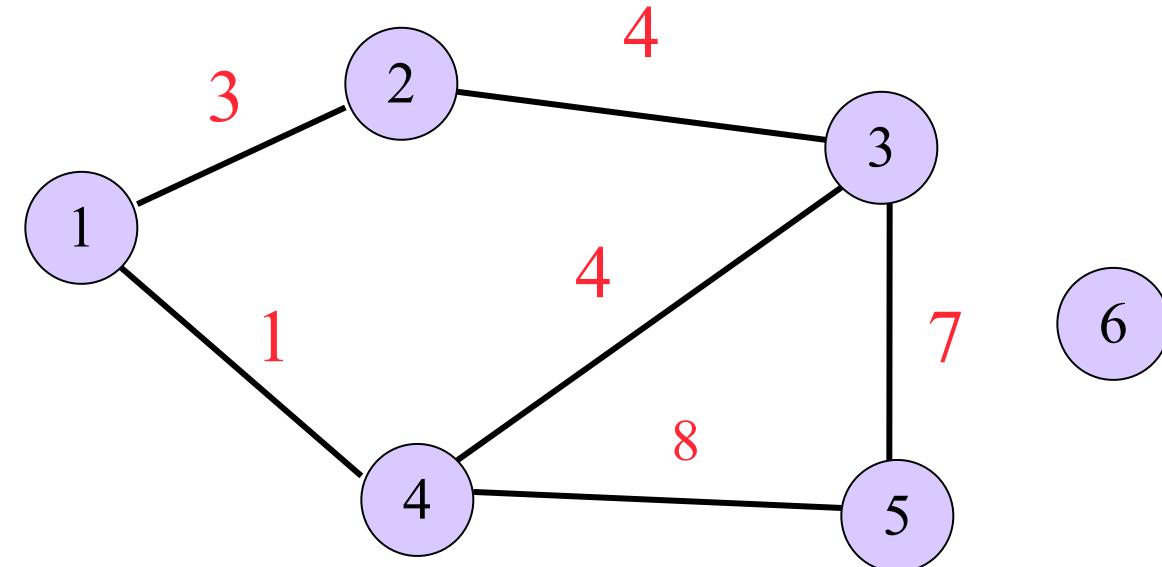
Spazio: $a \cdot n + 2 \cdot b \cdot m$



Grafi pesati

- ♦ In alcuni casi ogni arco ha un **peso (costo, guadagno)** associato
 - ♦ Il peso può essere determinato tramite una funzione di costo $p: V \times V \rightarrow \mathbf{R}$, dove \mathbf{R} è l'insieme dei numeri reali
 - ♦ Quando tra due vertici non esiste un arco, il peso è infinito

$$m_{uv} = \begin{cases} p_{uv}, & \text{se } (u, v) \in E, \\ +\infty \text{ (oppure } -\infty) & \text{se } (u, v) \notin E. \end{cases}$$



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

Liste di adiacenza - variazioni sul tema

Sia il concetto di *lista di adiacenza* che il concetto di *lista dei nodi* possono essere declinati in molti modi:

Struttura	Java	C++	Python
Lista collegata	LinkedList	list	
Vettore statico	[]	[]	[]
Vettore dinamico	ArrayList	vector	list
Insieme	HashSet TreeSet	set	set
Dizionario	HashMap TreeMap	map	dict

Specifica

GRAPH

Graph()	% Crea un grafo vuoto
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u , NODE v)	% Aggiunge l'arco (u, v) al grafo
deleteNode(NODE u)	% Rimuove il nodo u dal grafo
deleteEdge(NODE u , NODE v)	% Rimuove l'arco (u, v) nel grafo
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti ad u
SET V()	% Restituisce l'insieme di tutti i nodi

```
foreach  $u \in G.V()$  do
    foreach  $v \in G.adj(u)$  do
        fai un'operazione sull'arco  $(u, v)$ 
```

♦ Complessità

- ♦ $O(n+m)$ liste di adiacenza
- ♦ $O(n^2)$ matrice di adiacenza
- ♦ $O(m)$ “operazioni”

Riassumendo

Matrici di adiacenza

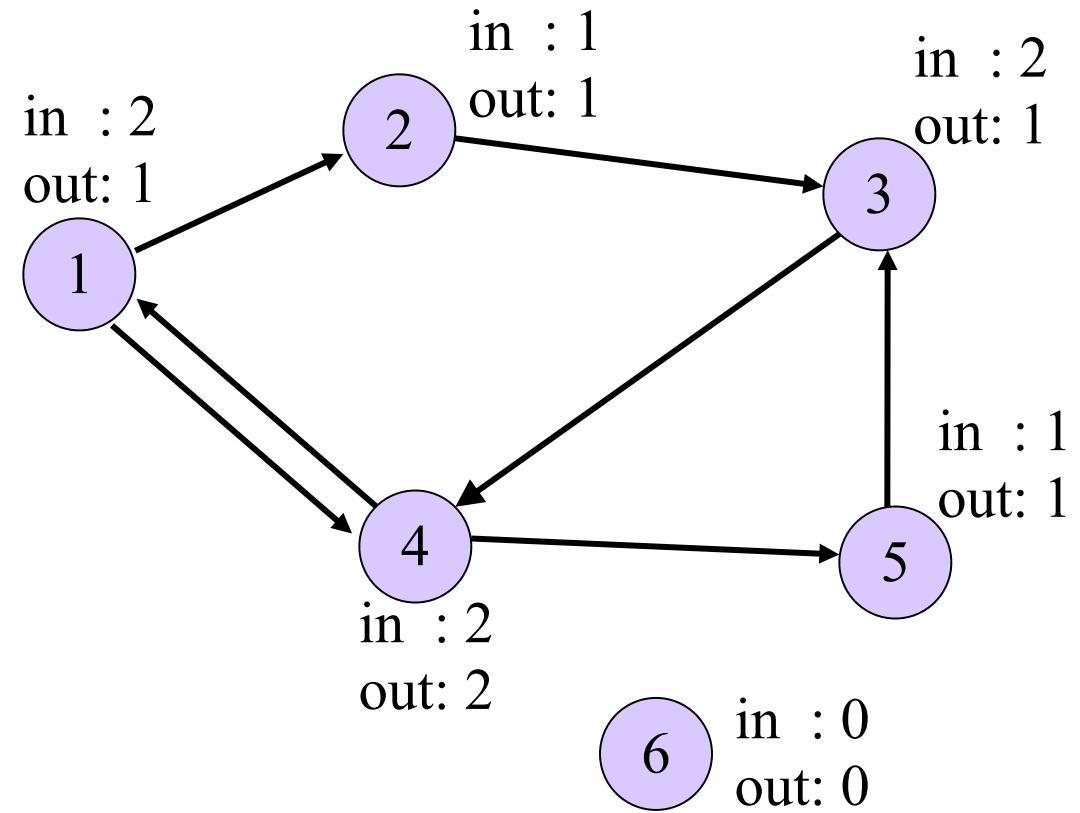
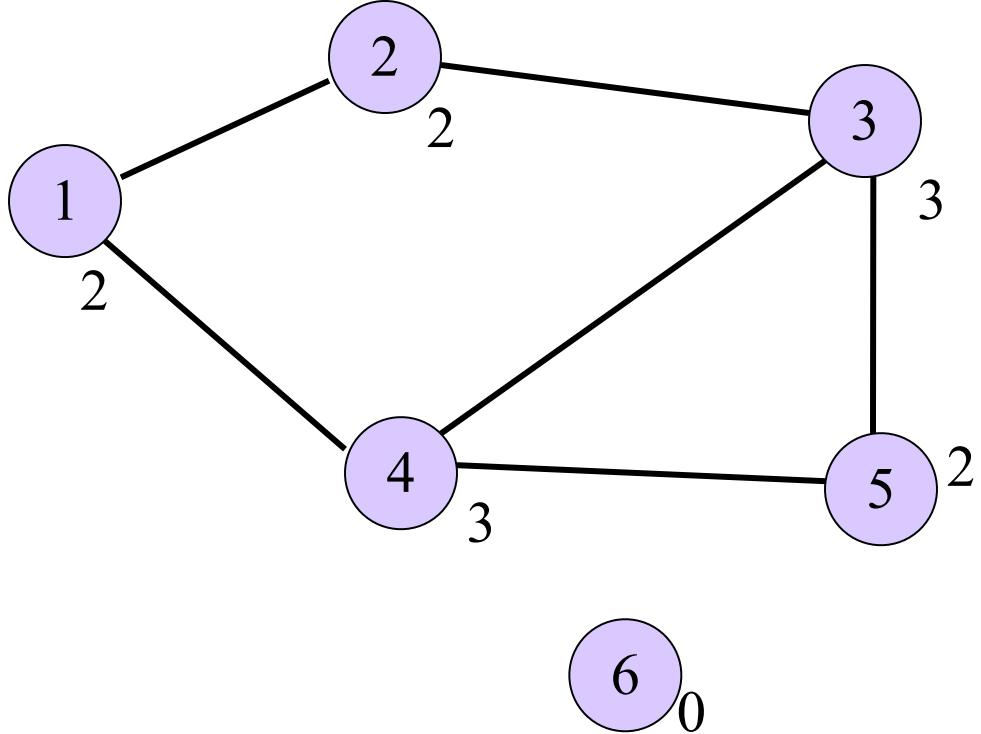
- Spazio richiesto $O(n^2)$
- Verificare se u è adiacente a v richiede tempo $O(1)$
- Iterare su tutti gli archi richiede tempo $O(n^2)$
- Ideale per grafi densi

Liste di adiacenza

- Spazio richiesto $O(n + m)$
- Verificare se u è adiacente a v richiede tempo $O(n)$
- Iterare su tutti gli archi richiede tempo $O(n + m)$
- Ideale per grafi sparsi

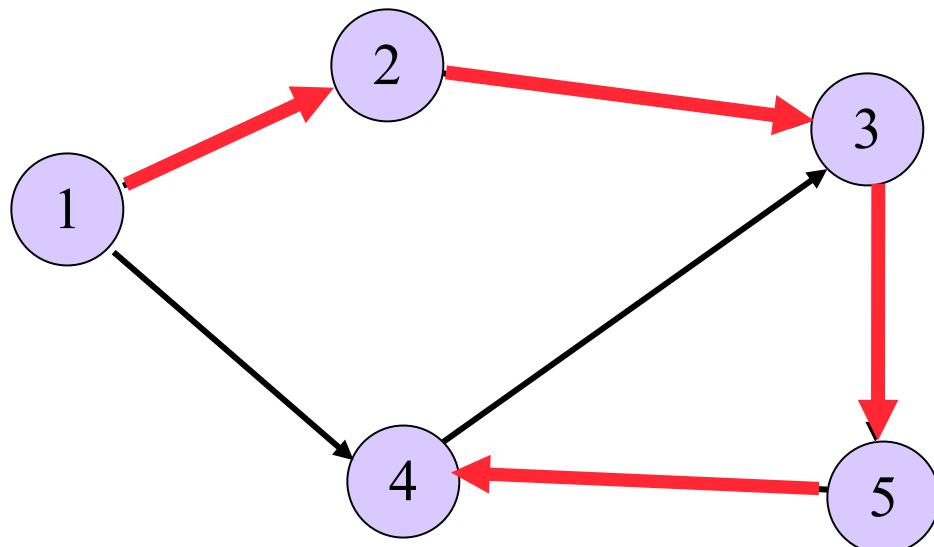
Definizioni: Grado

- ♦ In un grafo non orientato
 - ♦ il *grado* di un vertice è il numero di archi che partono da esso
- ♦ In un grafo orientato
 - ♦ il *grado entrante (uscente)* di un vertice è il numero di archi incidenti in (da) esso



Definizioni: Cammini

- ♦ In un grafo orientato $G=(V,E)$
 - ♦ un *cammino* di lunghezza k è una sequenza di vertici u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$
- ♦ In un grafo non orientato $G=(V,E)$
 - ♦ una *catena* di lunghezza k è una sequenza di vertici u_0, u_1, \dots, u_k tale che $[u_i, u_{i+1}] \in E$ per $0 \leq i \leq k-1$

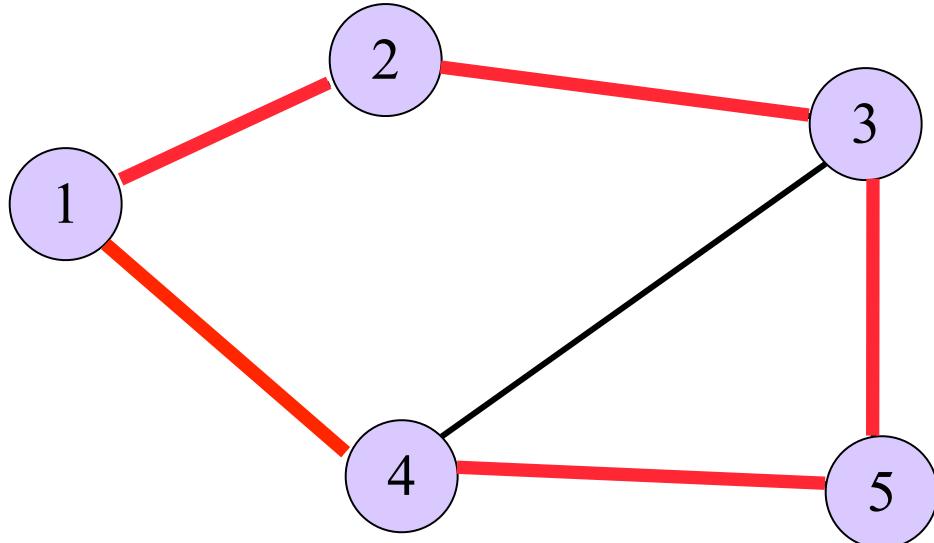


Esempio: **1, 2, 3, 5, 4** è una catena nel grafo con lunghezza 4

Un cammino (catena) si dice *semplice* se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza)

Definizioni: Cicli

- ♦ In un grafo orientato $G=(V,E)$
 - ♦ un *ciclo* di lunghezza k è un cammino u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$, $u_0 = u_k$, e $k > 2$
- ♦ In un grafo non orientato $G=(V,E)$
 - ♦ un *circuito* di lunghezza k è una catena u_0, u_1, \dots, u_k tale che $[u_i, u_{i+1}] \in E$ per $0 \leq i \leq k-1$, $u_0 = u_k$, e $k > 2$

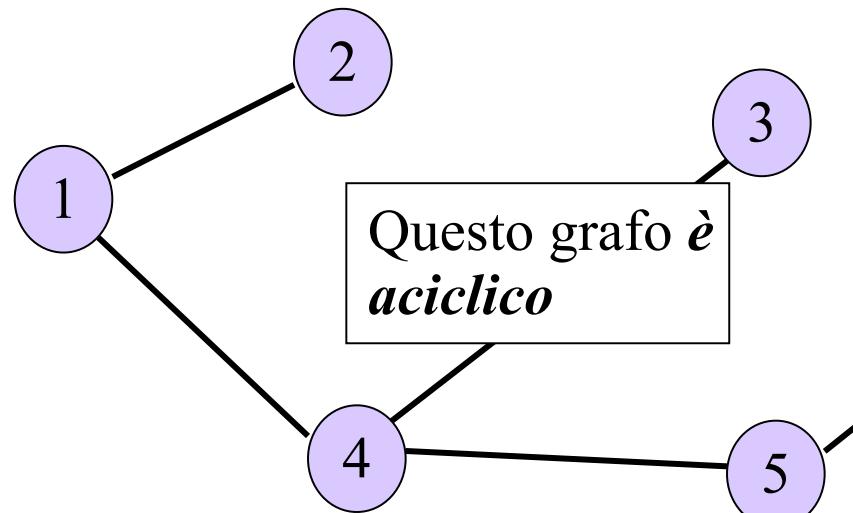


Esempio: **1, 2, 3, 5, 4, 1** è un circuito con lunghezza 5

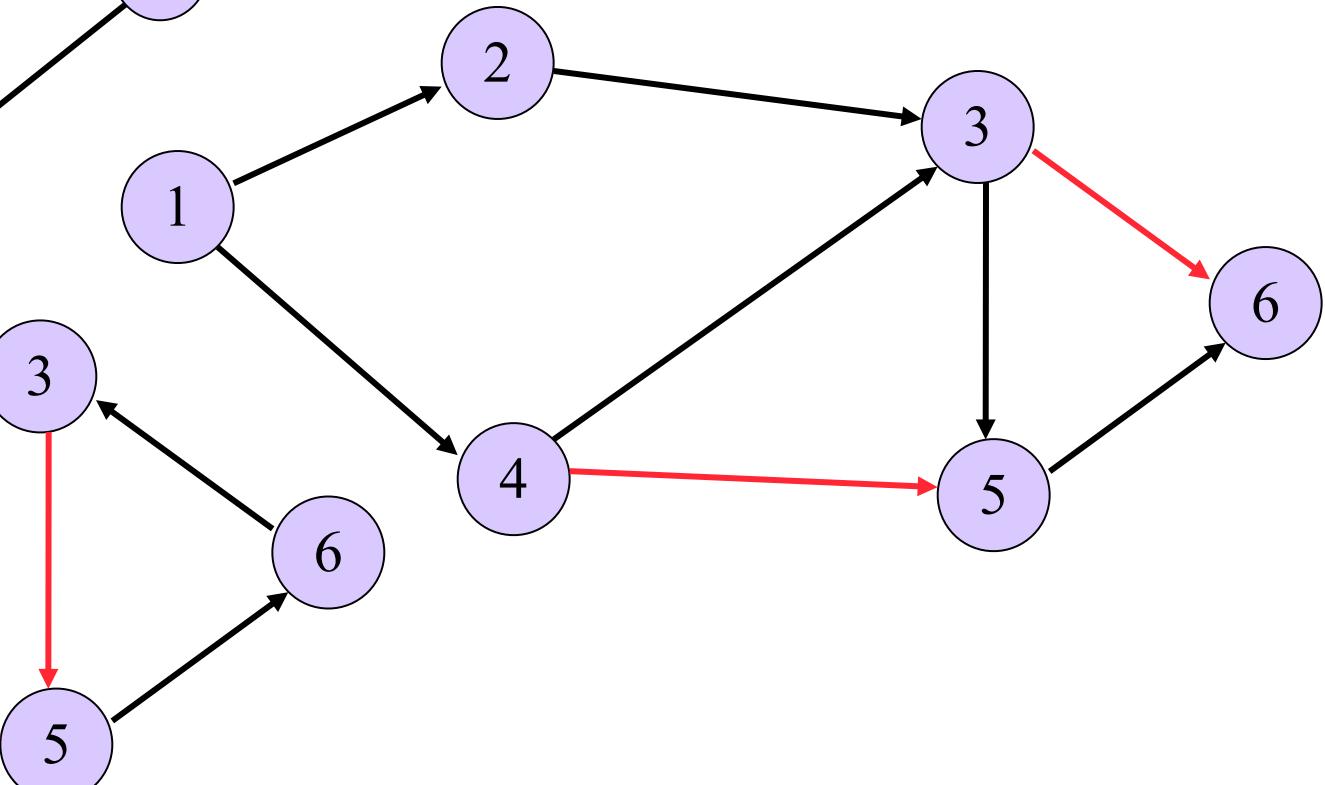
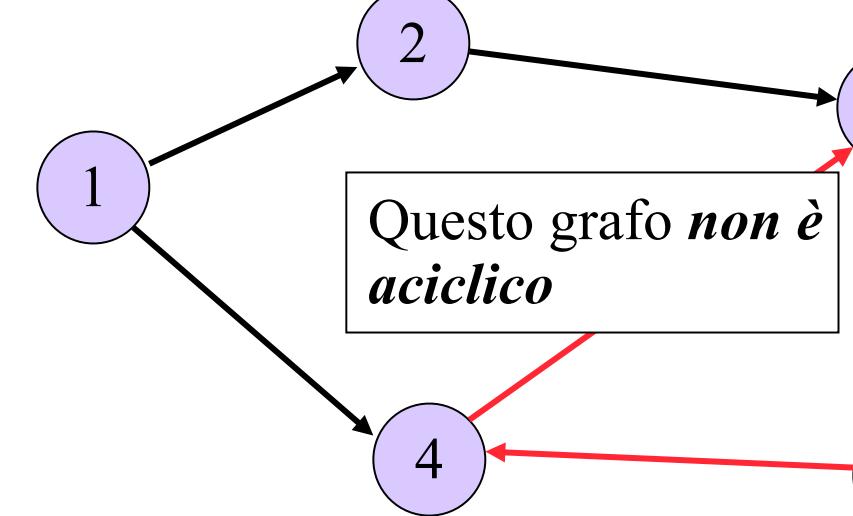
Un ciclo (circuito) si dice *semplice* se tutti i suoi vertici sono distinti (tranne ovviamente l'ultimo)

Definizioni: Grafi aciclici

- Un grafo senza cicli è detto aciclico



Un grafo orientato aciclico è chiamato DAG (*Directed Acyclic Graph*)

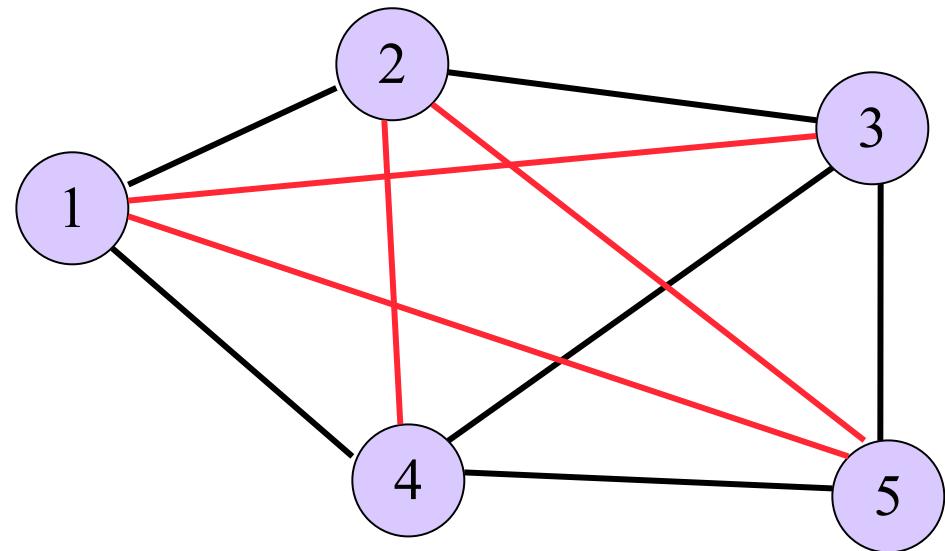
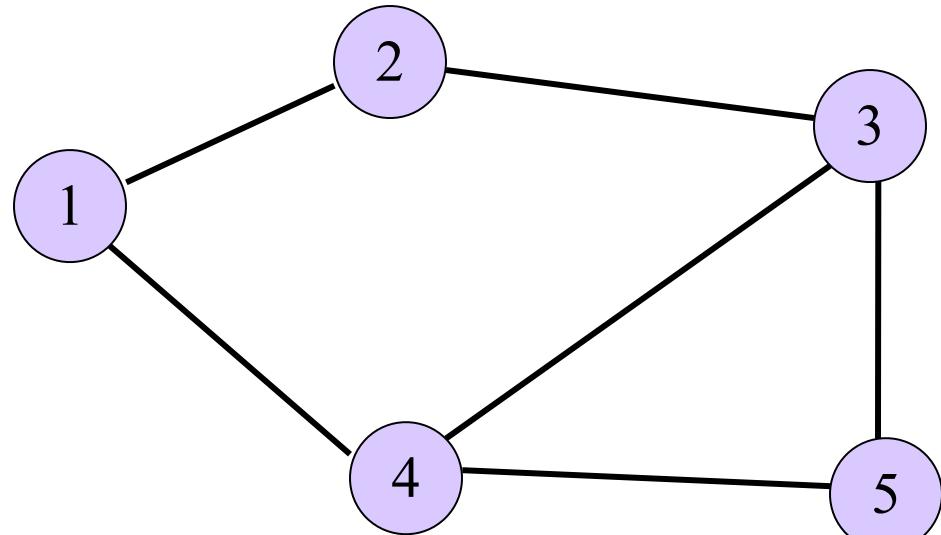


Definizioni: Grafo completo

- Un *grafo completo* è un grafo che ha un arco tra ogni coppia di vertici.

Questo grafo *è completo*

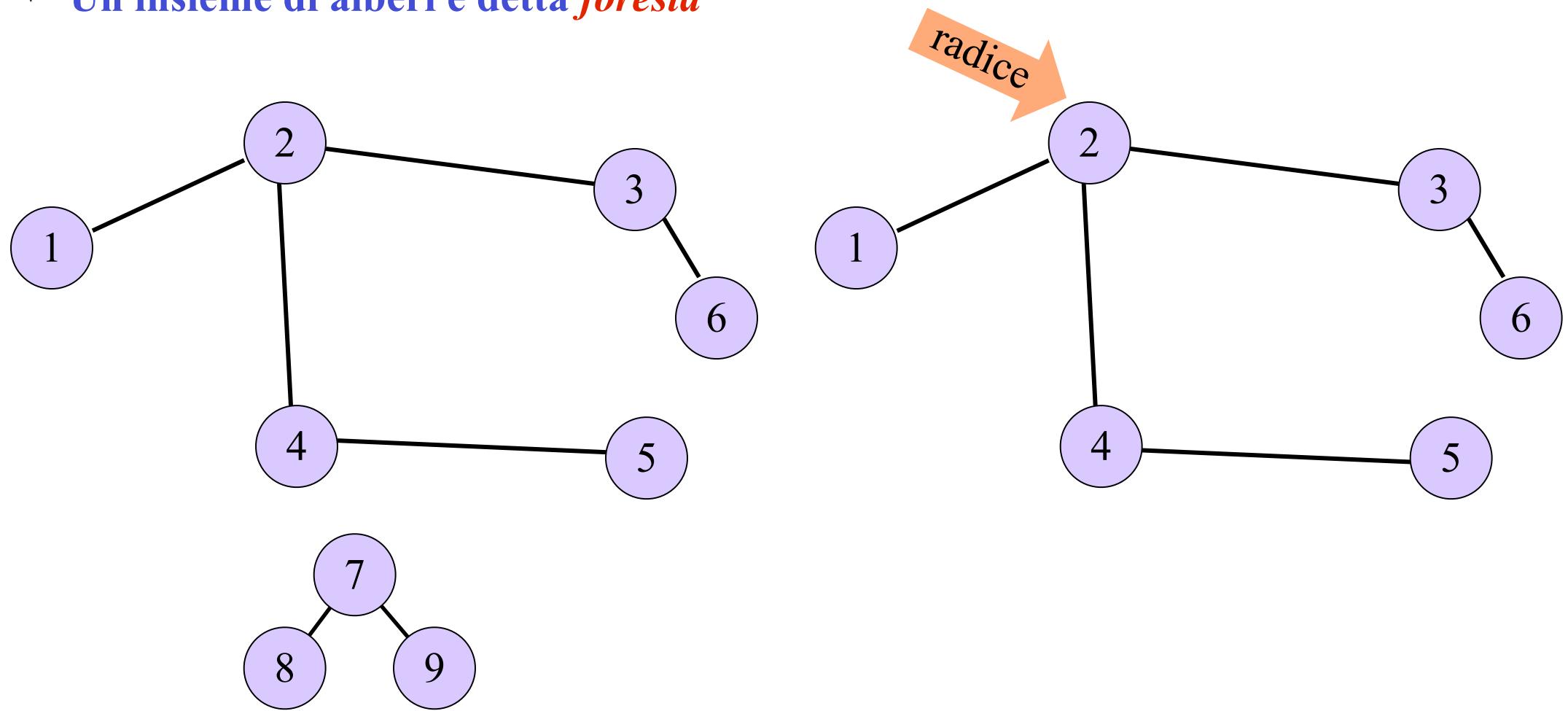
Questo grafo *non è completo*



$$m = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Definizioni: Alberi

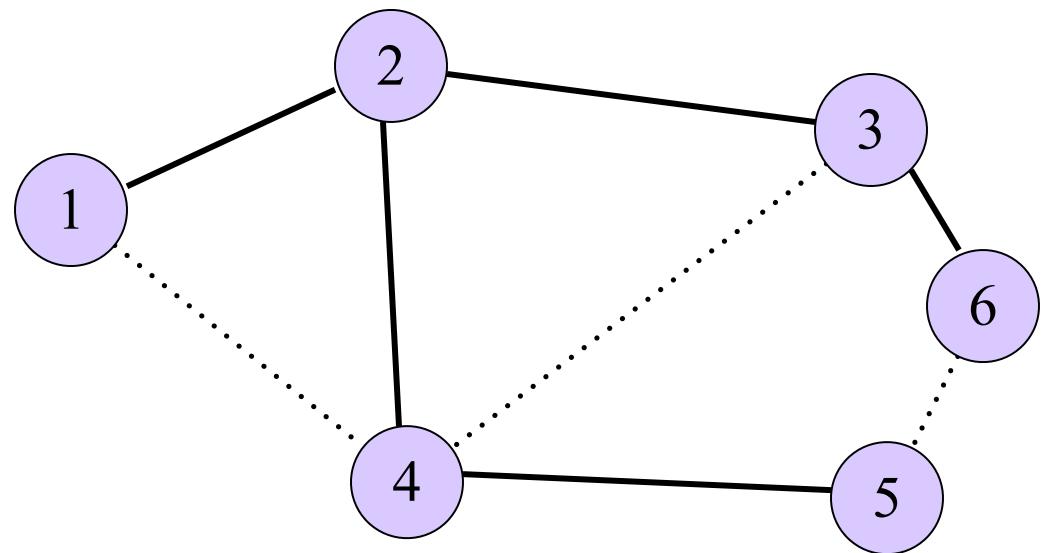
- ♦ Un *albero libero* è un grafo non orientato connesso, aciclico
- ♦ Se qualche vertice è detto radice, otteniamo un *albero radicato*
- ♦ Un insieme di alberi è detta *foresta*



Definizioni: Alberi di copertura

- ♦ **In un grafo non orientato $G=(V, E)$**

- ♦ un albero di copertura T è un albero libero $T = (V, E')$ composto da tutti i nodi di V e da un sottoinsieme degli archi ($E' \subseteq E$), tale per cui tutte le coppie di nodi del grafo sono connesse da una sola catena nell'albero.



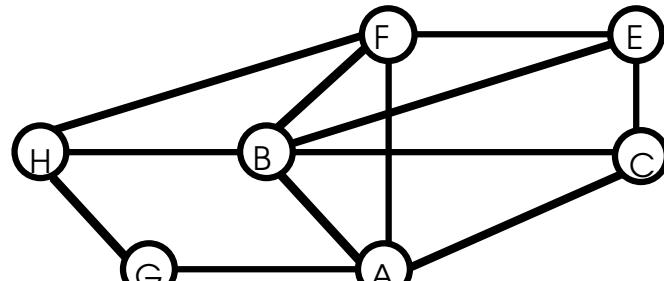
Un esempio di utilizzo dei grafi

- ♦ **Sherlock Holmes indaga sulla morte del duca McPollock, ucciso da un'esplosione nel suo maniero:**
 - ♦ **Watson:** “Ci sono novità, Holmes: pare che il testamento, andato distrutto nell’esplosione, fosse stato favorevole ad una delle sette ‘amiche’ del duca”
 - ♦ **Holmes:** “Ciò che è più strano, è che la bomba sia stata fabbricata appositamente per essere nascosta nell’armatura della camera da letto, il che fa supporre che l’assassino abbia necessariamente fatto più di una visita al castello”
 - ♦ **Watson:** “Ho interrogato personalmente le sette donne, ma ciascuna ha giurato di essere stata nel castello una sola volta nella sua vita.
 - ♦ (1) Ann ha incontrato Betty, Charlotte, Felicia e Georgia;
 - ♦ (2) Betty ha incontrato Ann, Charlotte, Edith, Felicia e Helen;
 - ♦ (3) Charlotte ha incontrato Ann, Betty e Edith;
 - ♦ (4) Edith ha incontrato Betty, Charlotte, Felicia;
 - ♦ (5) Felicia ha incontrato Ann, Betty, Edith, Helen;
 - ♦ (6) Georgia ha incontrato Ann e Helen;
 - ♦ (7) Helen ha incontrato Betty, Felicia e Georgia.

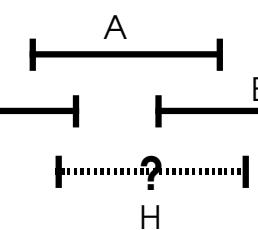
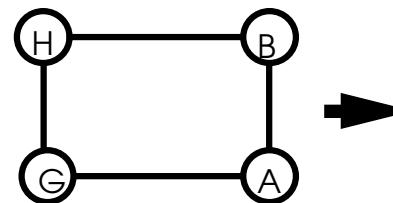
Vedete, Holmes, che le testimonianze concordano. Ma chi sarà l’assassino?”

 - ♦ **Holmes:** “Elementare, Watson: ciò che mi avete detto individua inequivocabilmente l’assassino!”

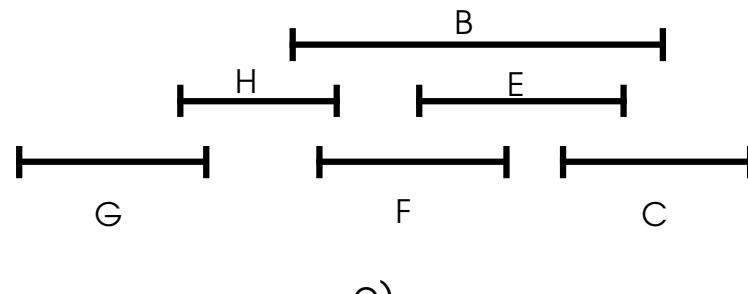
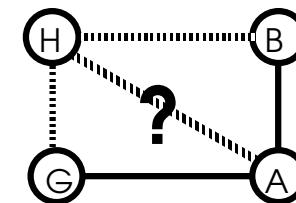
Un esempio di utilizzo dei grafi



a)



b)



c)

Problema: Attraversamento grafi

- ♦ **Definizione del problema**

- ♦ Dato un grafo $G=(V, E)$ ed un vertice r di V (detto *sorgente* o *radice*), visitare ogni vertice raggiungibile nel grafo dal vertice r
- ♦ Ogni nodo deve essere visitato una volta sola

- ♦ **Visita in ampiezza (breadth-first search)**

- ♦ Visita i nodi “espandendo” la frontiera fra nodi scoperti / da scoprire
- ♦ Esempi: Cammini più brevi da singola sorgente

- ♦ **Visita in profondità (depth-first search)**

- ♦ Visita i nodi andando il “più lontano possibile” nel grafo
- ♦ Esempi: Componenti fortemente connesse, ordinamento topologico

Visita: leggermente più difficile di quanto sembri

Un approccio ingenuo alla visita di un grafo potrebbe essere il seguente:

```
visit(GRAPH G)
```

```
foreach  $u \in G.V()$  do
    { visita nodo  $u$  }
    foreach  $v \in G.adj(u)$  do
        { visita arco  $(u, v)$  }
```

- La struttura del grafo non è tenuta in considerazione
- Si itera su tutti i nodi e gli archi senza nessun criterio

Visita: attenzione alle soluzioni “facili”

- ♦ Prendere ispirazione dalla visita degli alberi
- ♦ Ad esempio:
 - ♦ utilizziamo una visita BFS basata su coda
 - ♦ trattiamo i “vertici adiacenti” come se fossero i “figli”

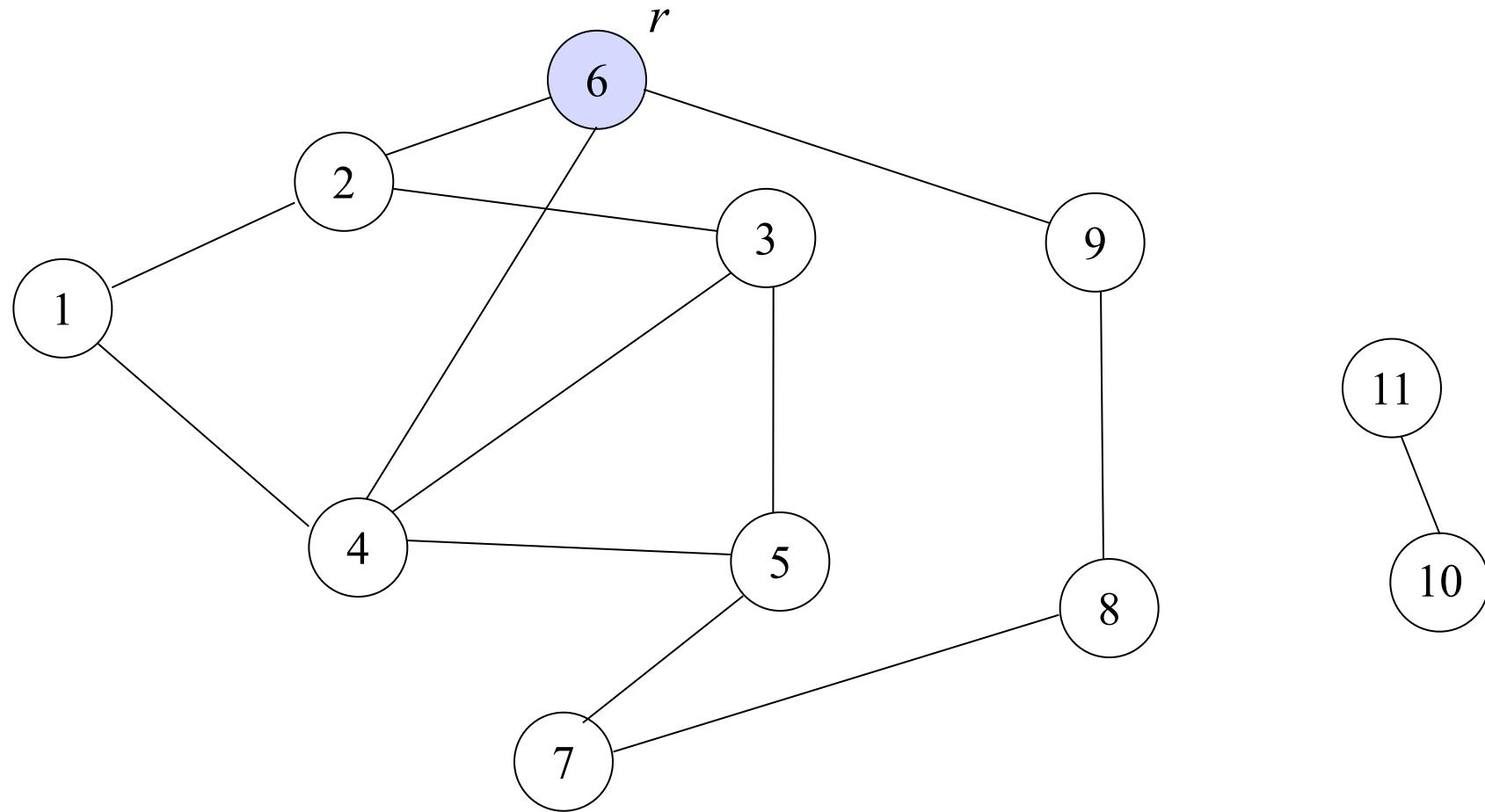
```
visita(GRAPH G, NODE r)
QUEUE S ← Queue()
S.enqueue(r)
while not S.isEmpty() do
    NODE u ← S.dequeue()
    { esamina il nodo u }
    foreach v ∈ G.adj(u) do
        S.enqueue(v)
```

Visita: attenzione alle soluzioni “facili”

- ♦ Prendere ispirazione dalla visita degli alberi
- ♦ Ad esempio:
 - ♦ utilizziamo una visita BFS basata su coda
 - ♦ trattiamo i “vertici adiacenti” come se fossero i “figli”

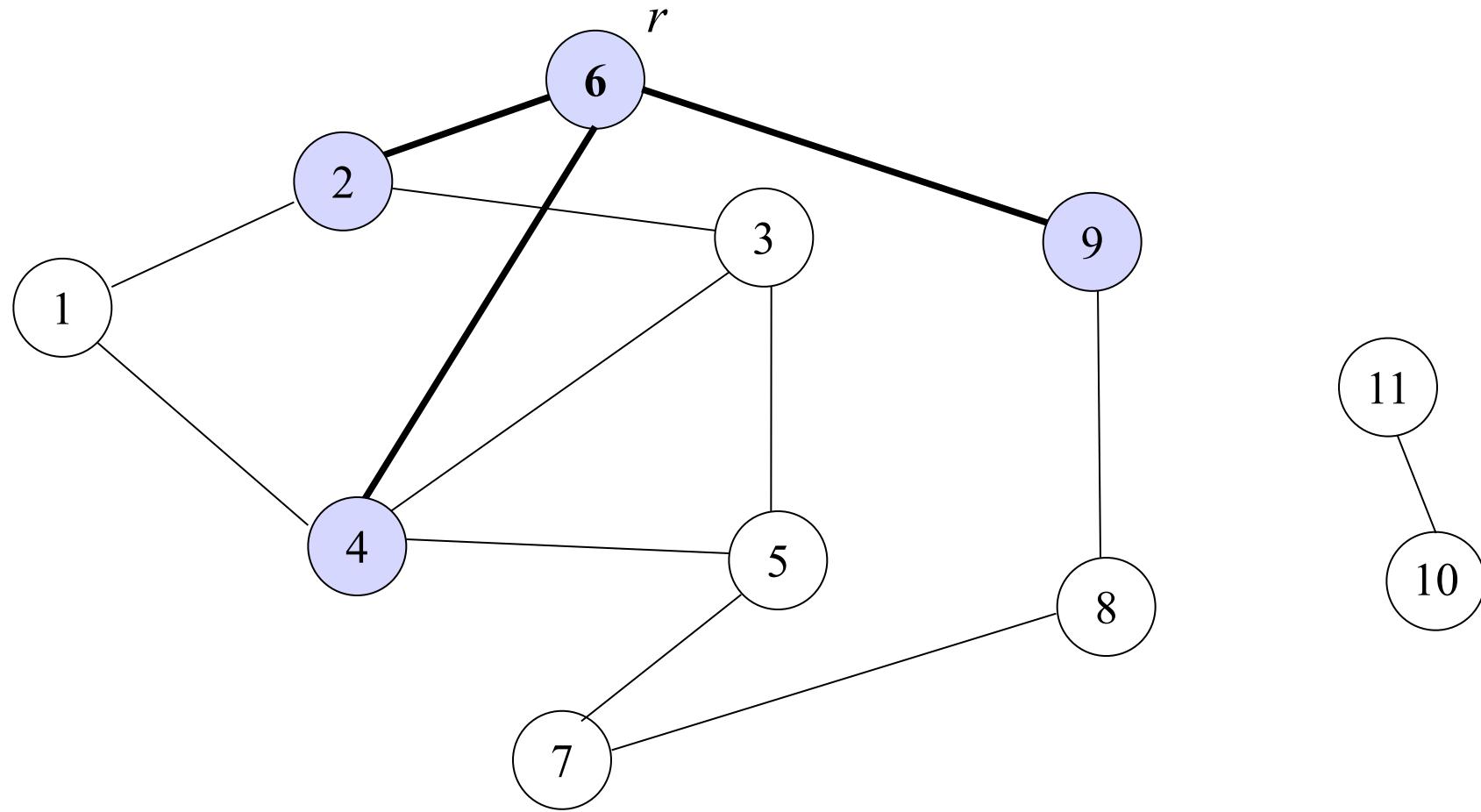
```
visita(GRAPH G, NODE r)
QUEUE S ← Queue()
S.enqueue(r)
while not S.isEmpty() do
    NODE u ← S.dequeue()
    { esamina il nodo u }
    foreach v ∈ G.adj(u) do
        S.enqueue(v)
```

Esempio di visita - errata



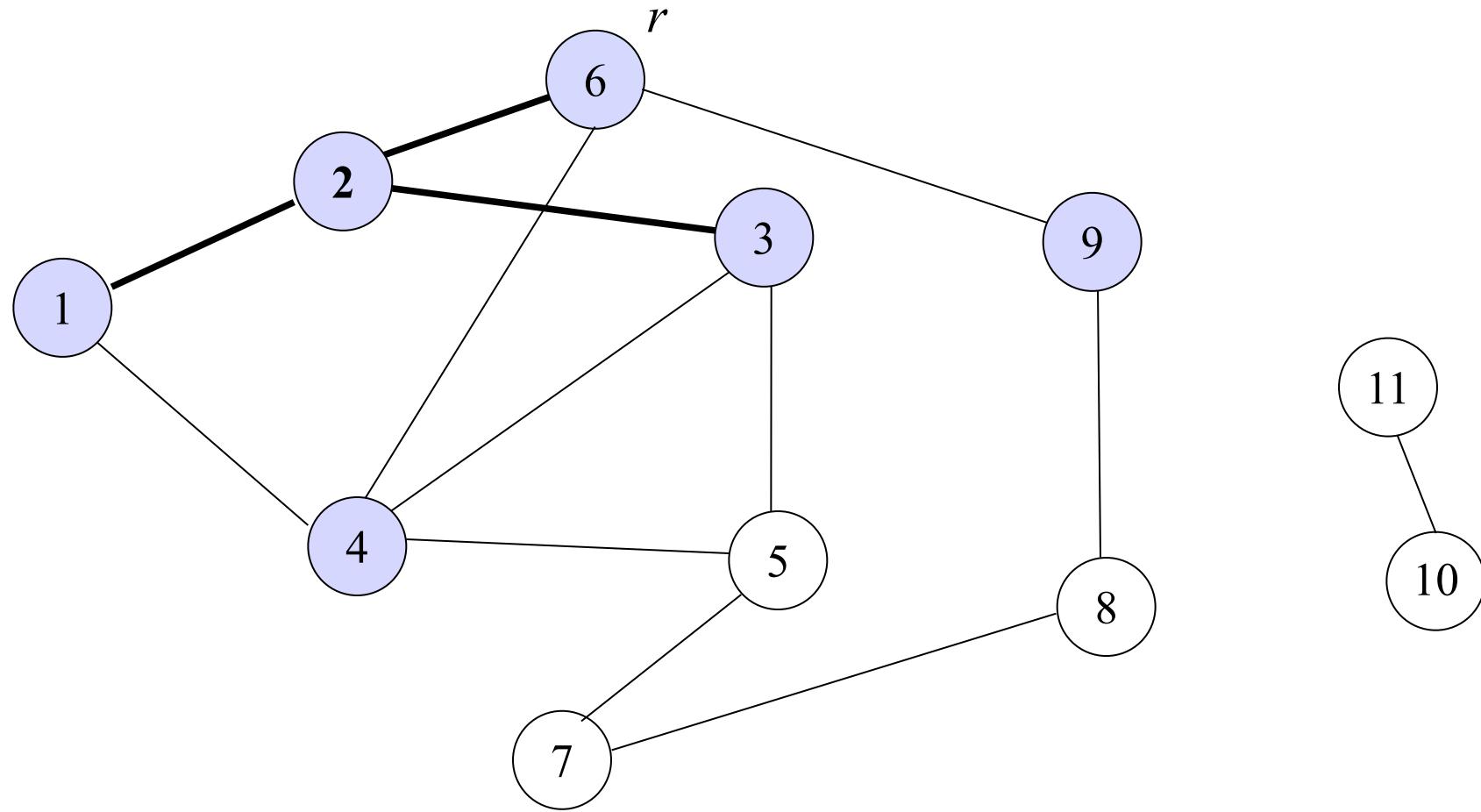
Coda : { 6 }

Esempio di visita - errata



Coda: {2, 4, 9}

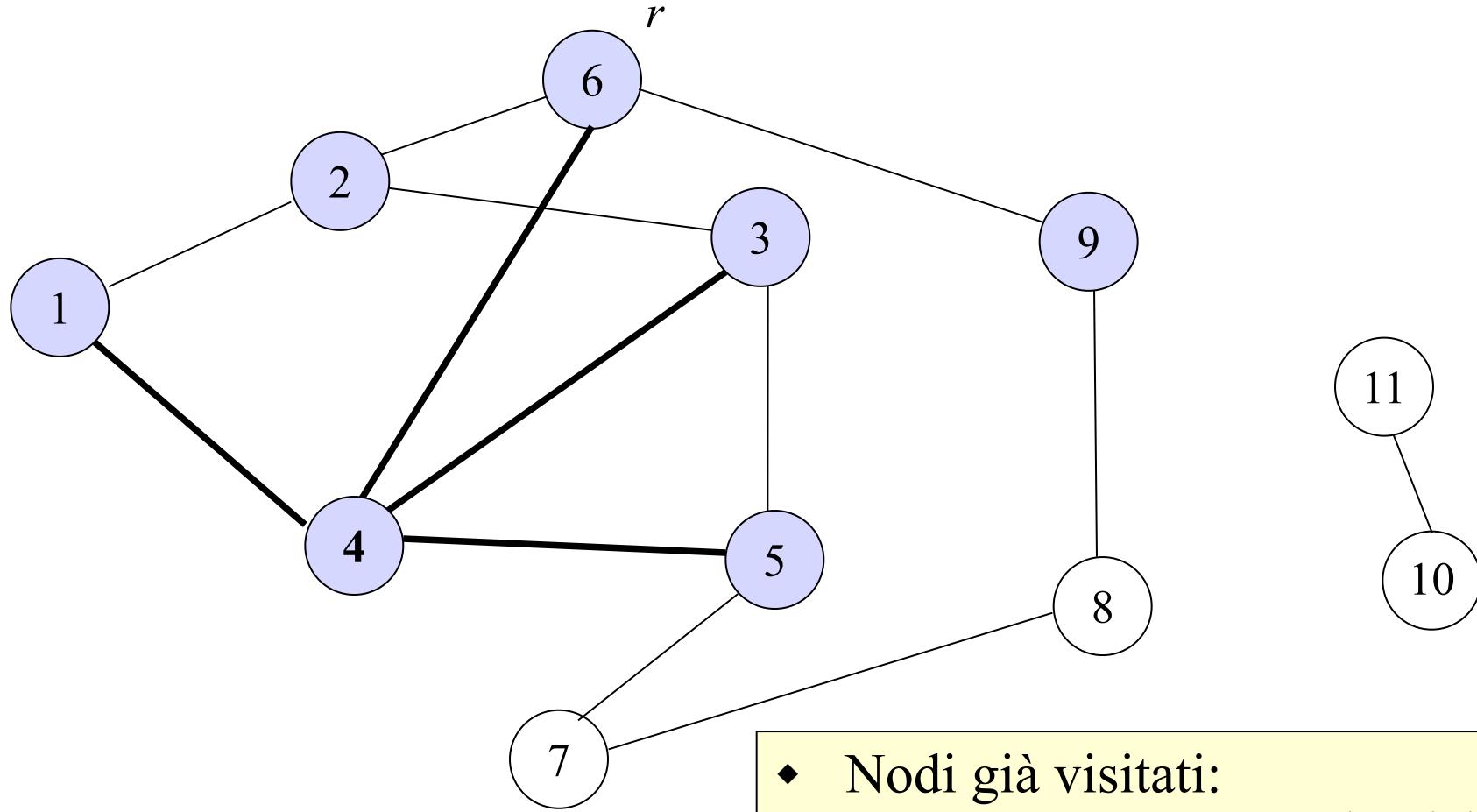
Esempio di visita - errata



Coda: {4, 9, 3, 1, 6}

u = 2

Esempio di visita - errata



Coda: {9, 3, 1, 6, 5, 6, 1, 3}

- ◆ Nodi già visitati:
 - ◆ “Marcare” un nodo visitato in modo che non possa essere visitato di nuovo
 - ◆ Bit di marcatura: nel vertice, array separato, etc.

Algoritmo generico per la visita

```
visita(GRAPH  $G$ , NODE  $r$ )
```

```
    SET  $S \leftarrow \text{Set}()$ 
```

```
     $S.\text{insert}(r)$ 
```

```
    { marca il nodo  $r$  come "scoperto" }
```

```
    while  $S.\text{size}() > 0$  do
```

```
        NODE  $u \leftarrow S.\text{remove}()$ 
```

```
        { esamina il nodo  $u$  }
```

```
        foreach  $v \in G.\text{adj}(u)$  do
```

```
            { esamina l'arco  $(u, v)$  }
```

```
            if  $v$  non è già stato scoperto then
```

```
                { marca il nodo  $v$  come "scoperto" }
```

```
                 $S.\text{insert}(v)$ 
```

- S è l'insieme *frontiera*

- Il funzionamento di **insert()** e **remove()** non è specificato

Visita in ampiezza (breadth first search, BFS)

Cosa vogliamo fare?

- ♦ **Visitare i nodi a distanze crescenti dalla sorgente**
 - ♦ visitare i nodi a distanza k prima di visitare i nodi a distanza $k+1$
- ♦ **Generare un albero BF (breadth-first)**
 - ♦ albero contenente tutti i vertici raggiungibili da r e tale che il cammino da r ad un nodo nell'albero corrisponde al cammino più breve nel grafo
- ♦ **Calcolare la distanza minima da s a tutti i vertici raggiungibili**
 - ♦ numero di archi attraversati per andare da r ad un vertice

Visita in ampiezza (breadth first search, BFS)

```
bfs(GRAPH  $G$ , NODE  $r$ )
```

```
QUEUE  $S \leftarrow \text{Queue}()$ 
```

```
 $S.\text{enqueue}(r)$ 
```

```
boolean[]  $\text{visitato} \leftarrow \text{new boolean}[1 \dots G.n]$ 
```

```
foreach  $u \in G.V() - \{r\}$  do  $\text{visitato}[u] \leftarrow \text{false}$ 
```

```
 $\text{visitato}[r] \leftarrow \text{true}$ 
```

```
while not  $S.\text{isEmpty}()$  do
```

```
    NODE  $u \leftarrow S.\text{dequeue}()$ 
```

```
    { esamina il nodo  $u$  }
```

```
    foreach  $v \in G.\text{adj}(u)$  do
```

```
        { esamina l'arco  $(u, v)$  }
```

```
        if not  $\text{visitato}[v]$  then
```

```
             $\text{visitato}[v] \leftarrow \text{true}$ 
```

```
             $S.\text{enqueue}(v)$ 
```

- Insieme S gestito tramite una coda
- $\text{visitato}[v]$ corrisponde alla marcatura del nodo v

Applicazione BFS: Numero di Erdös

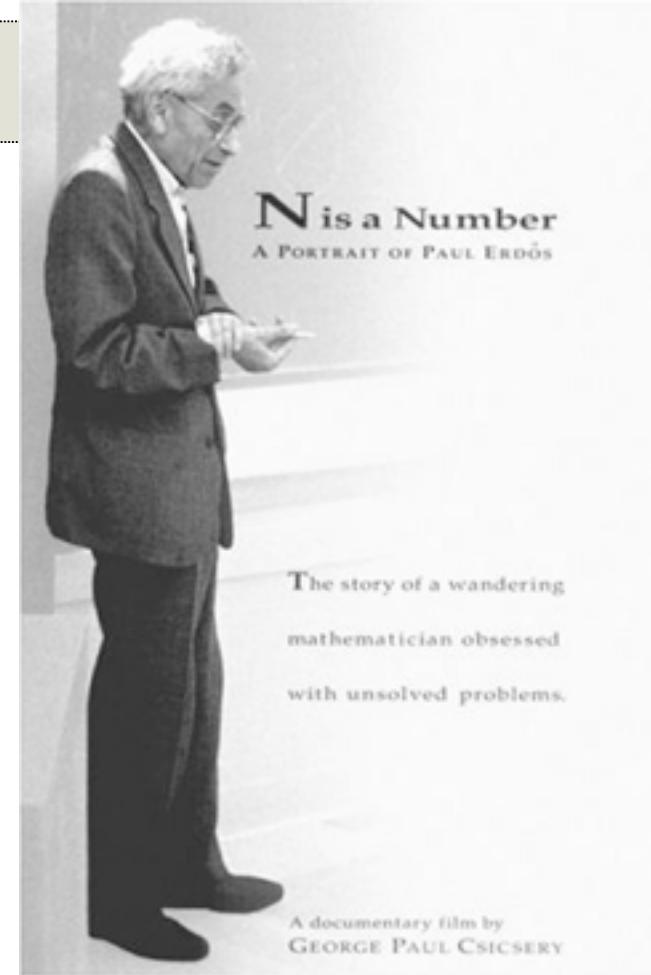
- ♦ **Paul Erdös (1913-1996)**

- ♦ Matematico
- ♦ Più di 1500 articoli, con più di 500 co-autori

- ♦ **Numero di Erdös**

- ♦ Erdös ha $erdös = 0$
- ♦ I co-autori di Erdös hanno $erdös = 1$
- ♦ Se X ha scritto una pubblicazione scientifica con un co-autore con $erdös = k$, ma non con un co-autore con $erdös < k$, X ha $erdös = k + 1$
- ♦ Chi non è raggiunto da questa definizione ha $erdös = +\infty$

- ♦ **Vediamo un'applicazione di BFS per calcolare il numero di Erdös**



Calcolo del numero di Erdős

```
erdos(GRAPH G, NODE r, integer[] erdos, NODE[] p)
```

```
QUEUE S ← Queue()
```

```
S.enqueue(r)
```

```
foreach  $u \in G.V() - \{r\}$  do  $erdős[u] = \infty$ 
```

```
 $erdős[r] \leftarrow 0$ 
```

```
 $p[r] \leftarrow \text{nil}$ 
```

```
while not  $S.isEmpty()$  do
```

```
    NODE  $u \leftarrow S.dequeue()$ 
```

```
    foreach  $v \in G.adj(u)$  do
```

```
        if  $erdős[v] = \infty$  then
```

```
             $erdős[v] \leftarrow erdős[u] + 1$ 
```

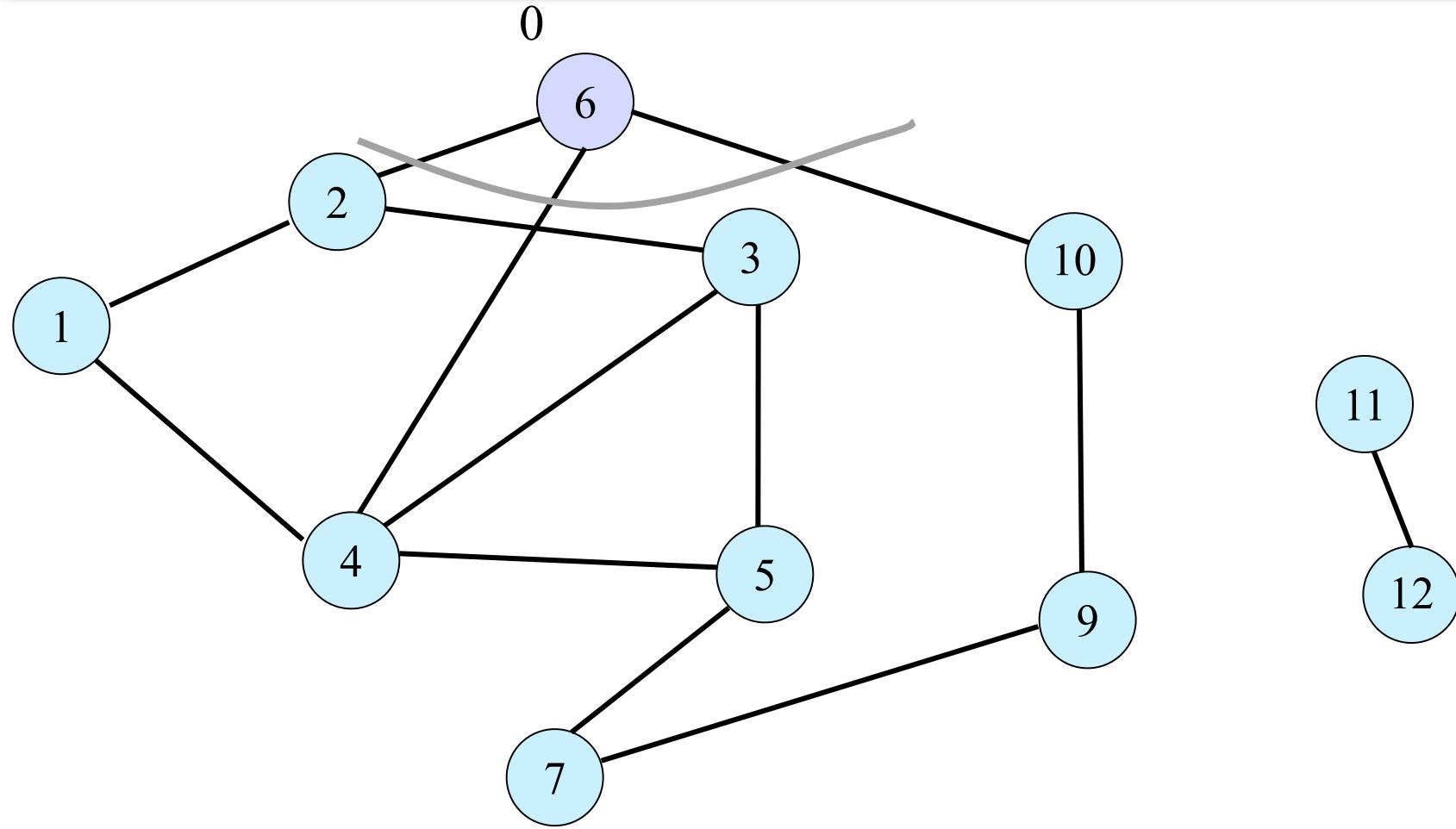
```
             $p[v] \leftarrow u$ 
```

```
            S.enqueue(v)
```

% Esamina l'arco (u, v)

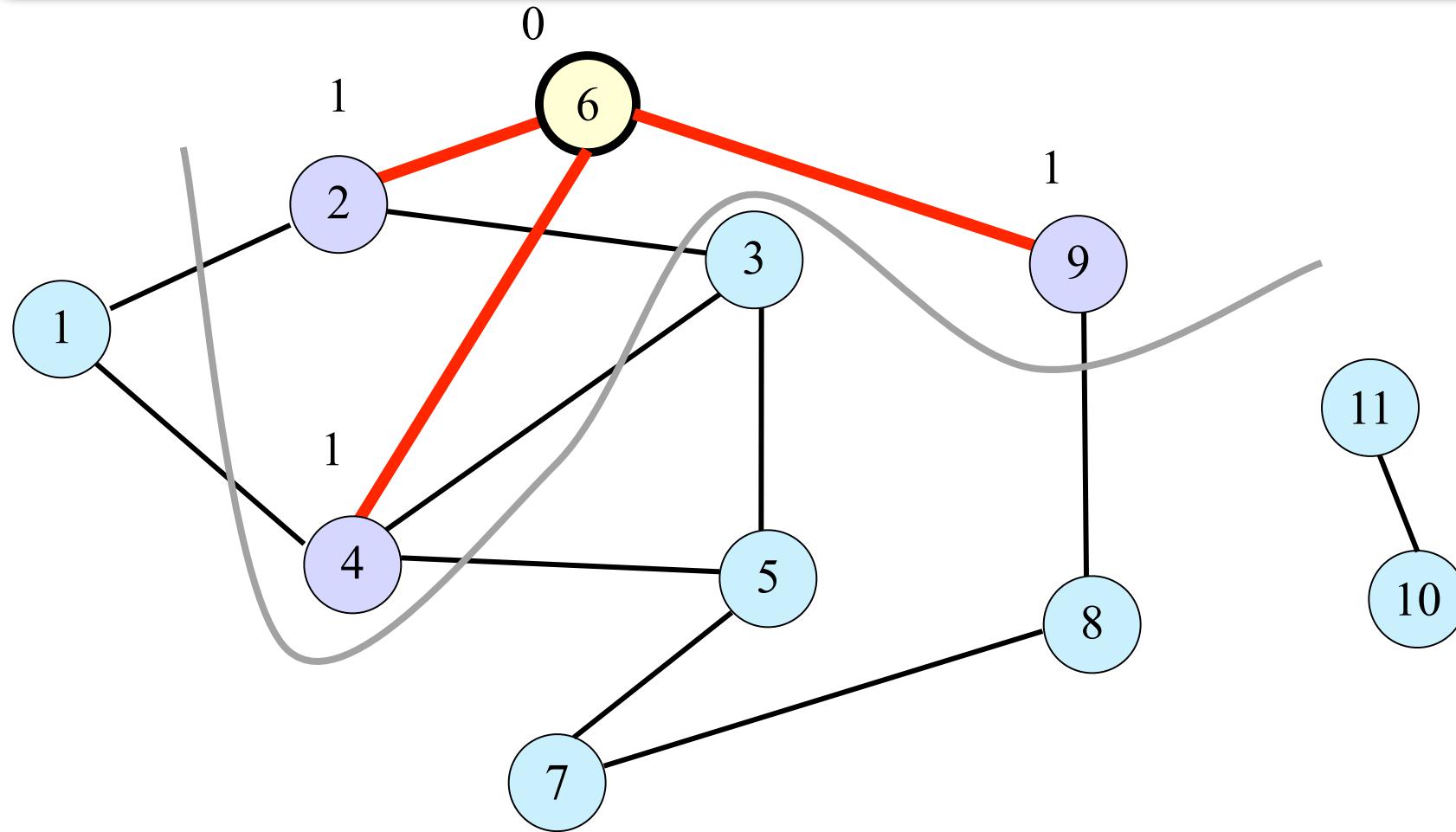
% Il nodo u non è già stato scoperto

Esempio



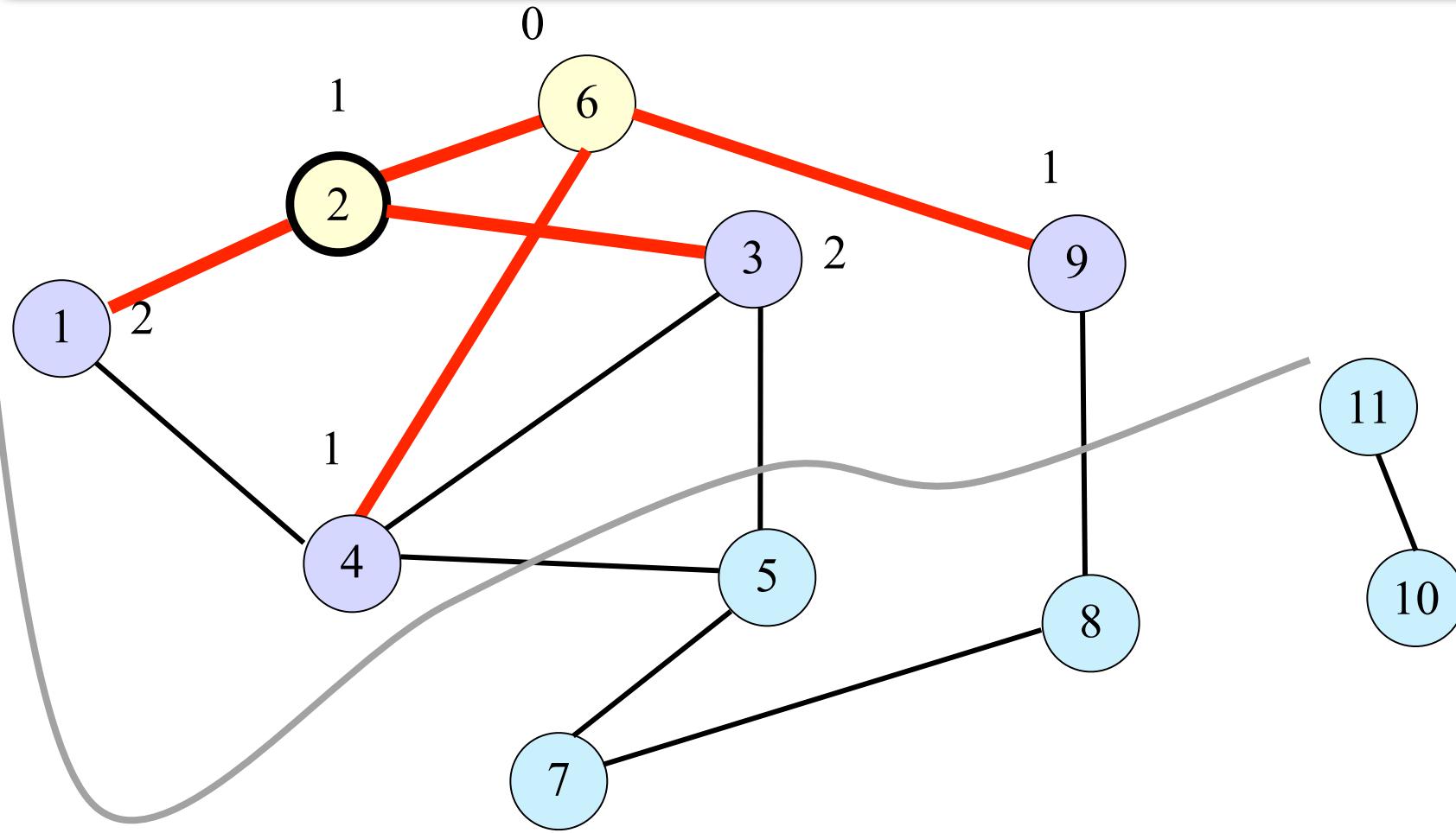
Coda : {6}

Esempio



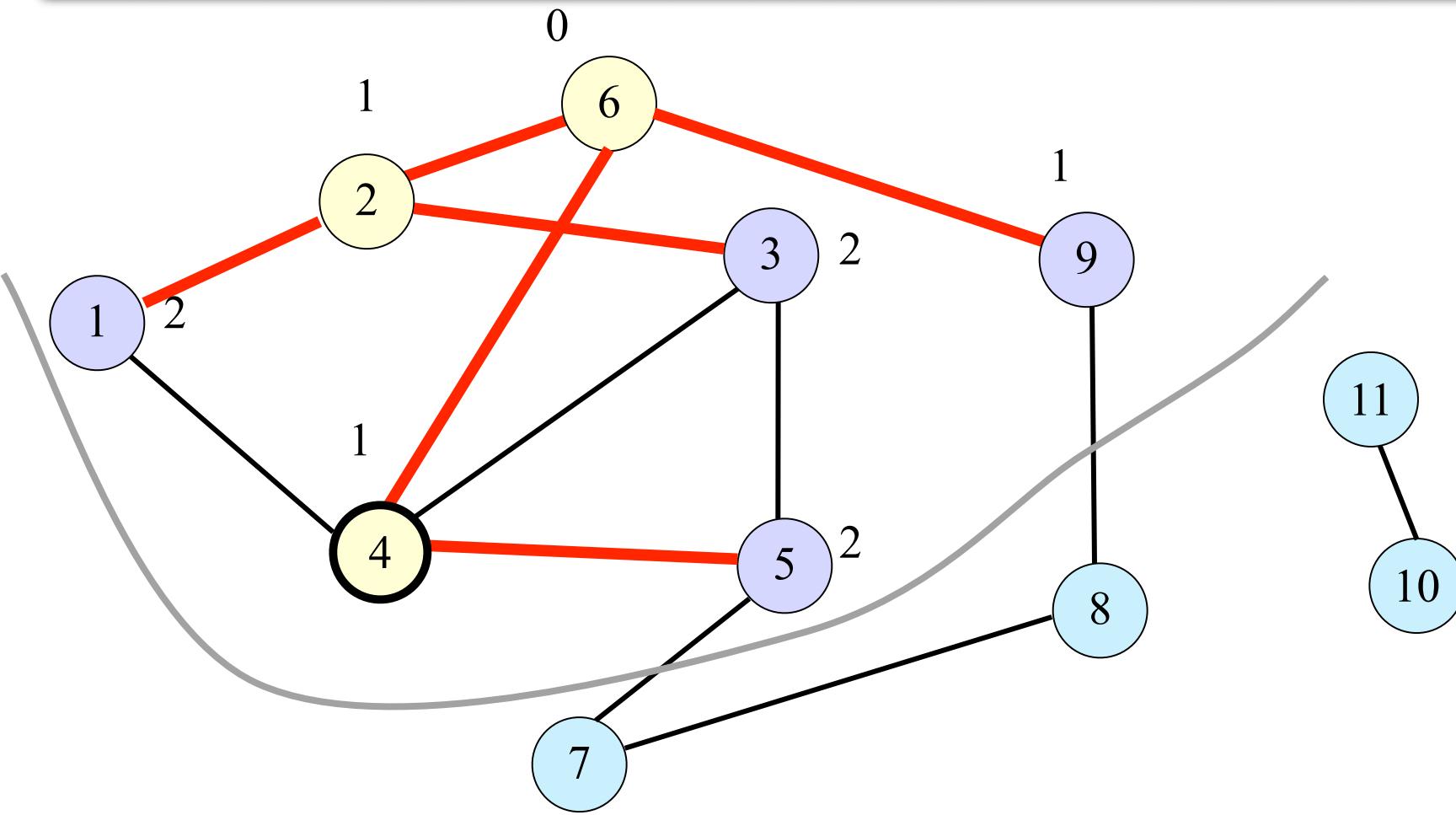
Coda: {2, 4, 9}

Esempio



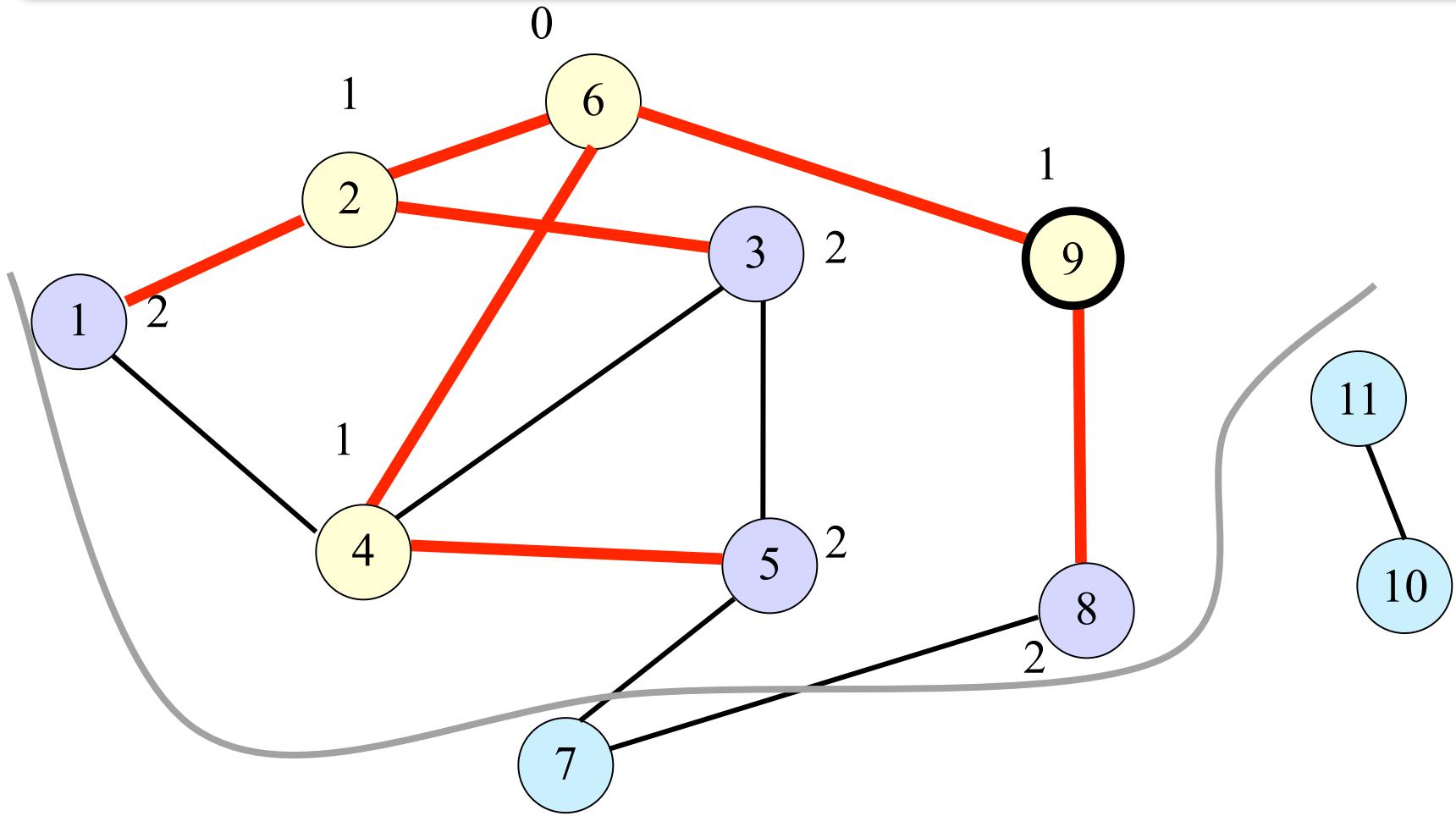
Coda: {4, 9, 3, 1}

Esempio



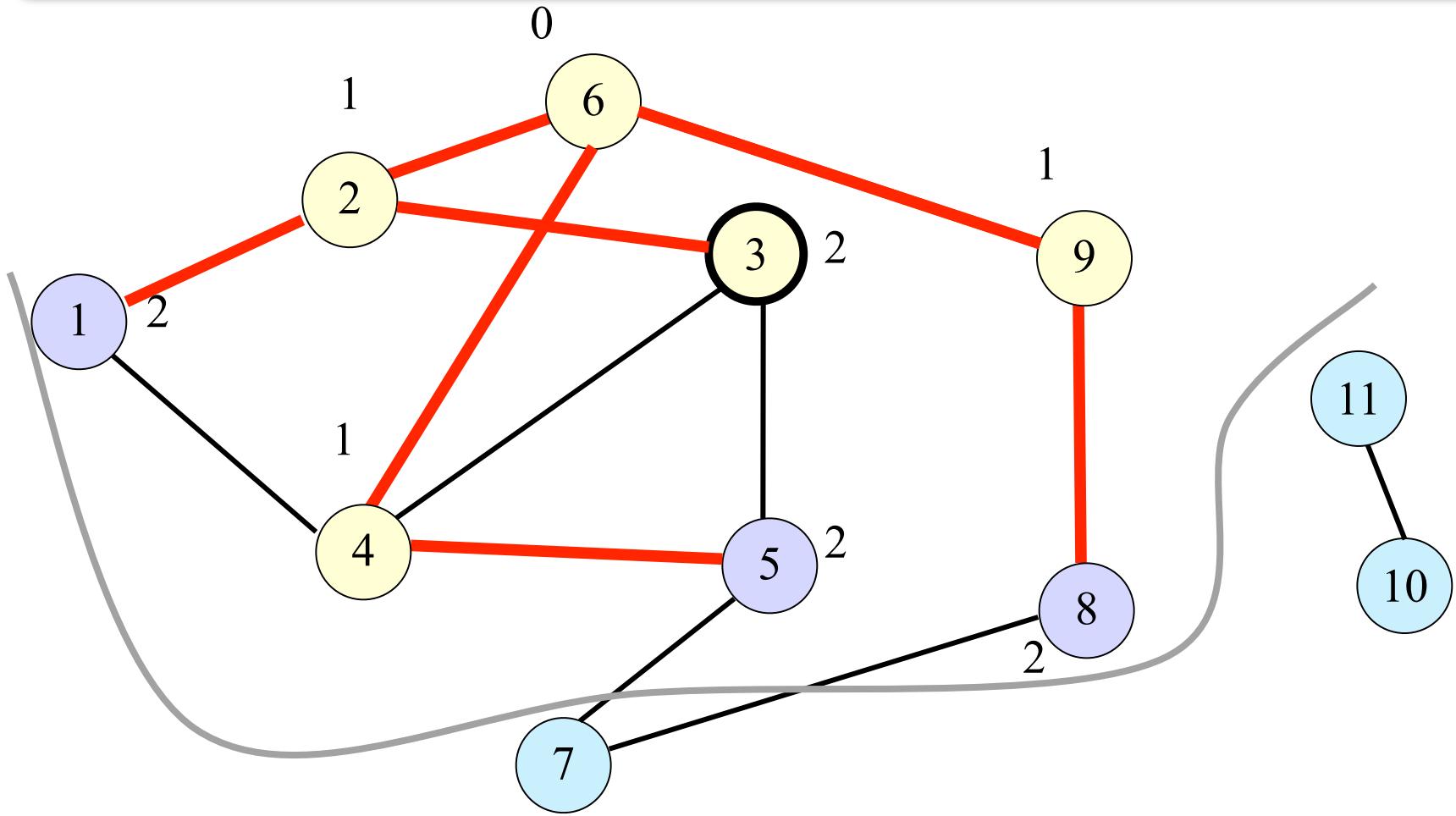
Coda: {9, 3, 1, 5}

Esempio



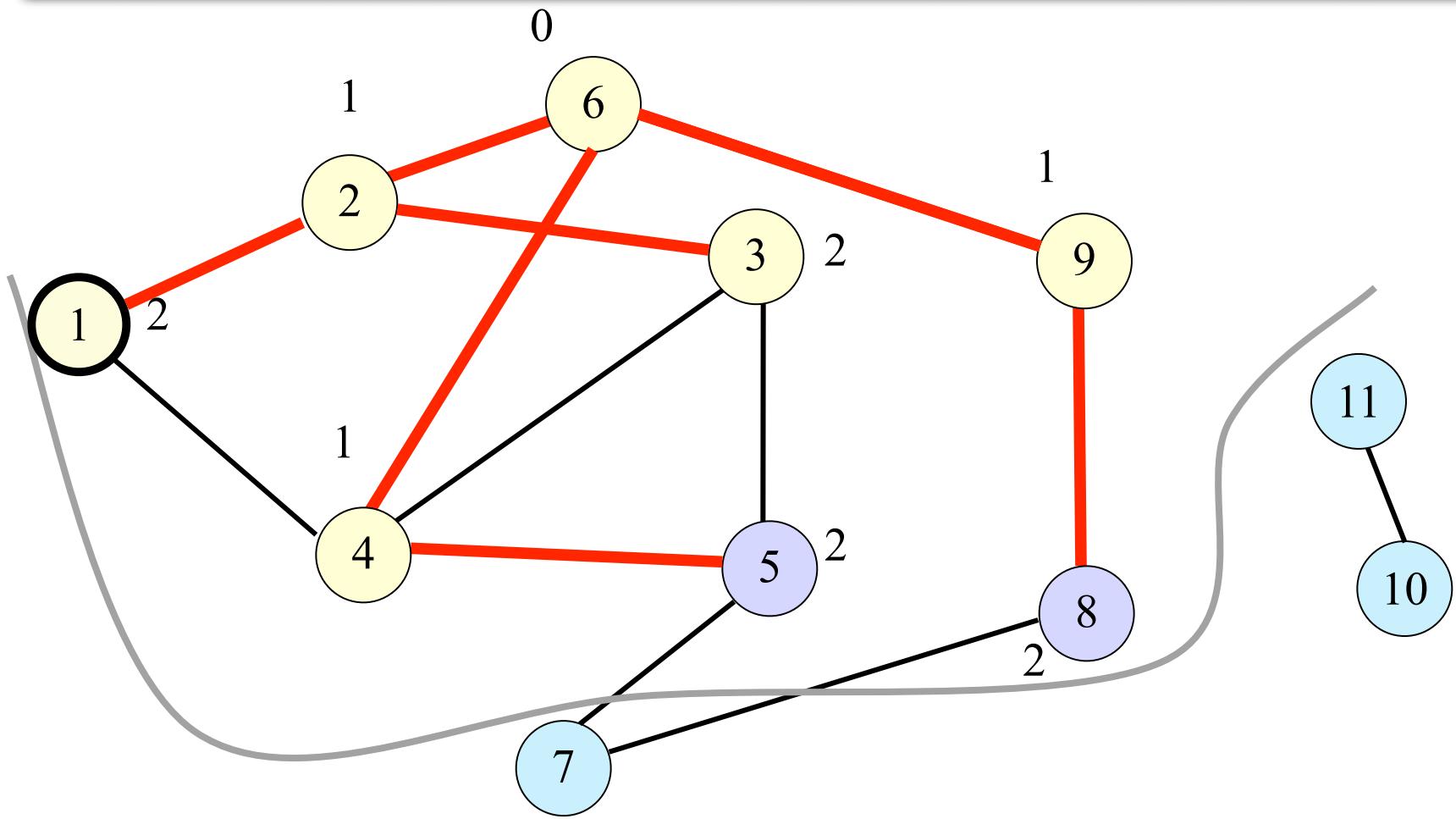
Coda: {3, 1, 5, 8}

Esempio



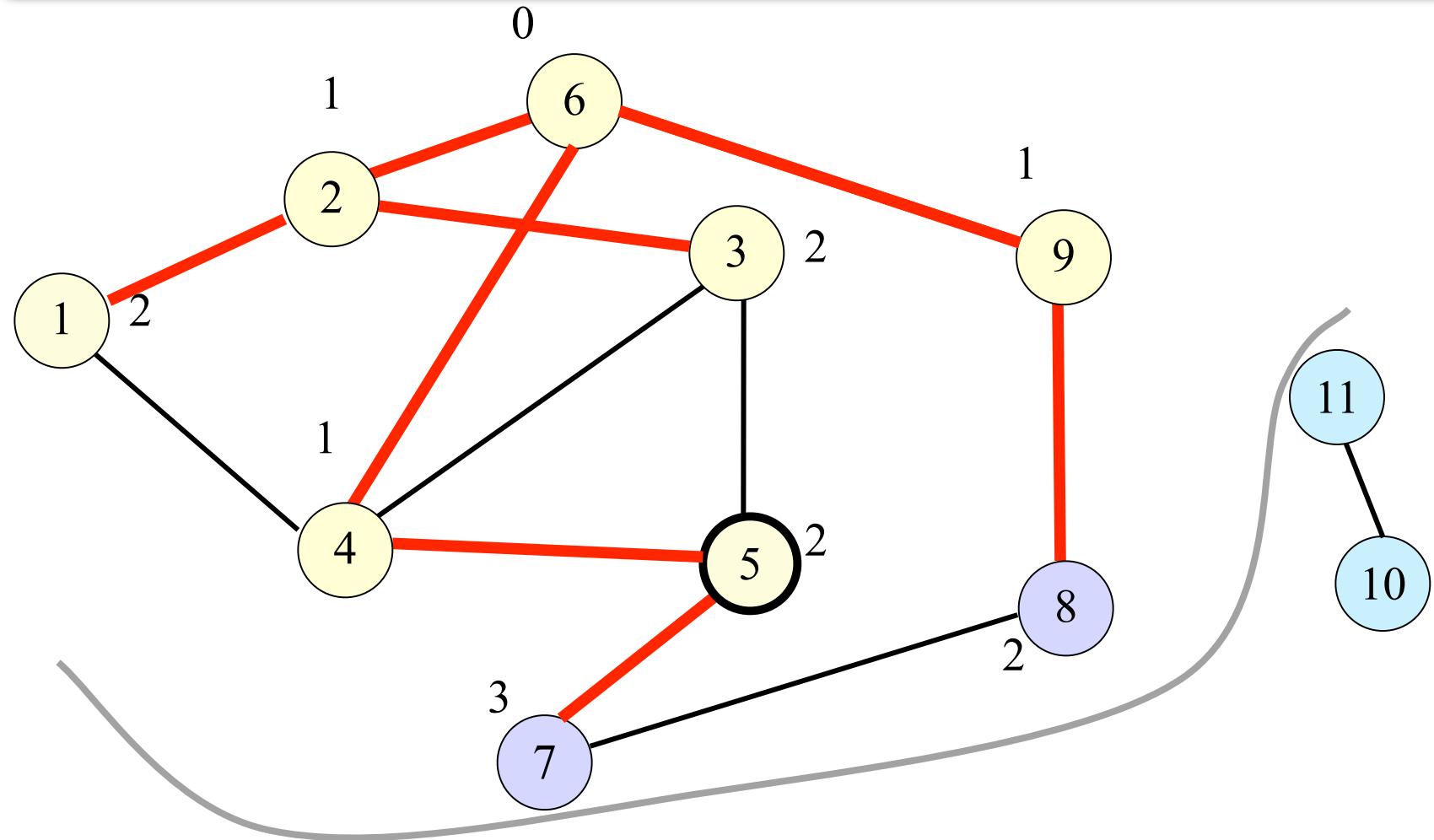
Coda: {1,5,8}

Esempio



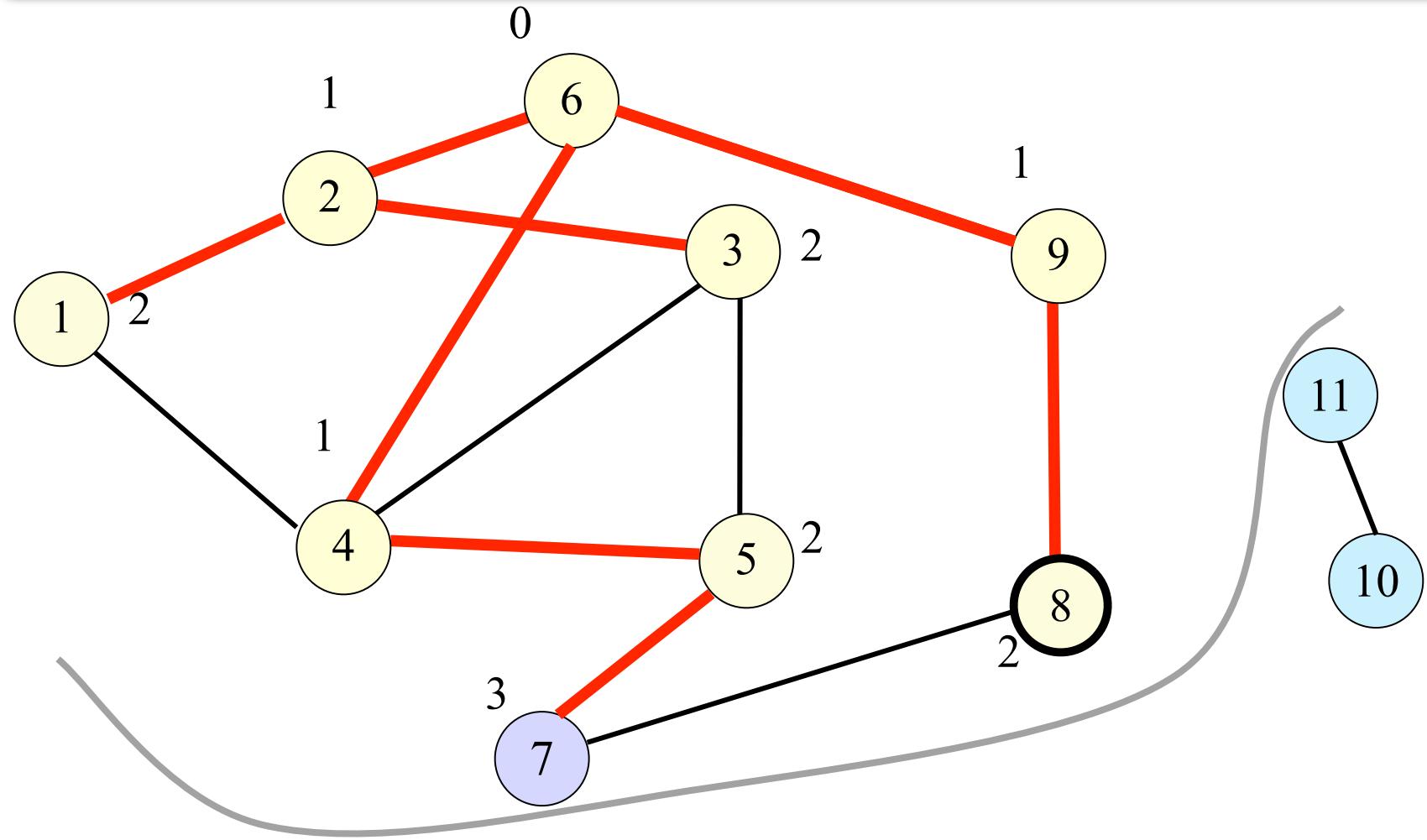
Coda: {5, 8}

Esempio



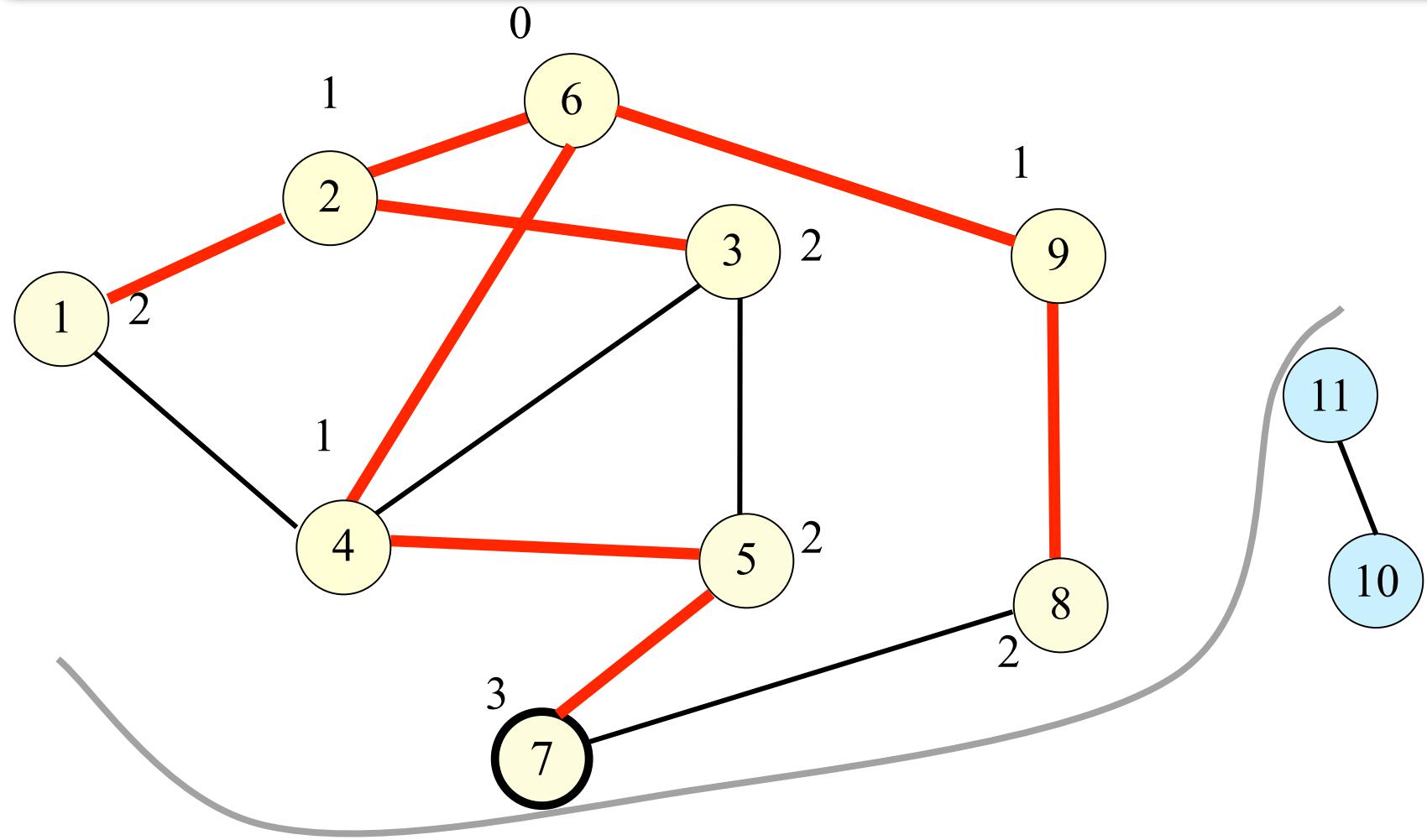
Coda: {8, 7}

Esempio



Coda : {7}

Esempio



Coda : {}

Albero dei cammini BFS

- La visita BFS può essere utilizzata per ottenere il cammino più breve fra due vertici (numero di archi)

- Albero di copertura di G radicato in r
- Memorizzato tramite vettore dei padri p
- Figli di u - nodi v tali che $(u,v) \in E$ e v non è ancora visitato

```
if erdős[v] = ∞ then
    erdős[v] ← erdős[u] + 1
    p[v] ← u
    S.enqueue(v)
```

```
stampaCammino(GRAPH G, NODE r, NODE s, NODE[] p)
```

```
if r = s then print s
else if p[s] = nil then
    print "nessun cammino da r a s"
else
    stampaCammino(G, r, p[s], p)
    print s
```

Algoritmo generico per la visita

♦ Alcune definizioni

- ♦ L'albero T contiene i *vertici visitati*
- ♦ $S \subseteq T$ contiene i *vertici aperti*: vertici i cui archi uscenti non sono ancora stati percorsi
- ♦ $T-S \subseteq T$ contiene i *vertici chiusi*: vertici i cui archi uscenti sono stati tutti percorsi
- ♦ $V-T$ contiene i vertici non visitati
- ♦ Se u si trova lungo il cammino che va da r al nodo v , diciamo che:
 - ♦ u è un antenato di v
 - ♦ v è un discendente di u

♦ Alcune cose da notare:

- ♦ I nodi vengono visitati al più una volta (marcatura)
- ♦ Tutti i nodi raggiungibili da r vengono visitati
- ♦ Ne segue che T contiene esattamente tutti i nodi raggiungibili da r

Visita in profondità (depth first search, DFS)

♦ Visita in profondità

- ♦ E' spesso una "subroutine" della soluzione di altri problemi
- ♦ Utilizzata per coprire l'intero grafo, non solo i nodi raggiungibili da una singola sorgente (diversamente da BFS)

♦ Output

- ♦ Invece di un albero, una *foresta* DF (depth-first) $G_\pi = (V, E_\pi)$
 - ♦ Contenente un insieme di alberi DF

♦ Struttura di dati

- ♦ Ricorsione al posto di una pila esplicita

```
dfs(GRAPH G, NODE u, boolean[] visitato)
```

visitato[*u*] $\leftarrow \text{true}$

(1) { esamina il nodo *u* (caso *previsita*) }

foreach *v* $\in G.\text{adj}(u)$ **do**

{ esamina l'arco (*u*, *v*) }

if not *visitato*[*v*] **then**

 dfs(*G*, *v*, *visitato*)

(2) { esamina il nodo *u* (caso *postvisita*) }

Depth-First Search (Ricorsiva, stack implicito)

```
dfs(GRAPH  $G$ , NODE  $u$ , boolean[]  $visited$ )
```

$visited[u] = \text{true}$

{ visita il nodo u (pre-order) }

foreach $v \in G.\text{adj}(u)$ **do**

if not $visited[v]$ **then**

 { visita l'arco (u, v) }

 dfs($G, v, visited$)

{ visita il nodo u (post-order) }

Complessità: $O(m + n)$

BFS vs DFS

- Eseguire una DFS basata su chiamate ricorsive può essere rischioso in grafi molto grandi e connessi
- È possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio
- In tali casi, si preferisce utilizzare una BFS oppure una DFS basata su stack esplicito

Stack size in Java

Platform	Default
Windows IA32	64 KB
Linux IA32	128 KB
Windows x86_64	128 KB
Linux x86_64	256 KB
Windows IA64	320 KB
Linux IA64	1024 KB (1 MB)
Solaris Sparc	512 KB

DFS (Iterativa, stack esplicito, pre-order)

dfs(GRAPH G , NODE r)

```

STACK  $S = \text{Stack}()$ 
 $S.\text{push}(r)$ 
boolean[]  $\text{visited} =$ 
    new boolean[1 ...  $G.\text{size}()$ ]
foreach  $u \in G.\text{V}()$  do
     $\text{visited}[u] = \text{false}$ 
while not  $S.\text{isEmpty}()$  do
    NODE  $u = S.\text{pop}()$ 
    if not  $\text{visited}[u]$  then
        { visita il nodo  $u$  (pre-order) }
         $\text{visited}[v] = \text{true}$ 
        foreach  $v \in G.\text{adj}(u)$  do
            { visita l'arco  $(u, v)$  }
             $S.\text{push}(v)$ 

```

Note

- Un nodo può essere inserito nella pila più volte
- Il controllo se un nodo è già stato visitato viene fatto all'estrazione, non all'inserimento
- Complessità $O(m + n)$
 - $O(m)$ visite degli archi
 - $O(m)$ inserimenti, estrazioni
 - $O(n)$ visite dei nodi

DFS (Iterativa, stack esplicito, post-order)

Visita post-order

- Quando un nodo viene scoperto:
 - viene inserito nello stack con il tag **discovery**
- Quando un nodo viene estratto dalla coda con tag **discovery**:
 - Viene re-inserito con il tag **finish**
 - Tutti i suoi vicini vengono inseriti
- Quando un nodo viene estratto dalla coda con tag **finish**:
 - Viene effettuata la post-visita

Componenti fortemente connesse

- ◆ **Terminologia**

- ◆ Componenti connesse (connected components, CC)
- ◆ Componenti fortemente connesse (strongly connected components, SCC)

- ◆ **Motivazioni**

- ◆ Molti algoritmi che operano sui grafi iniziano decomponendo il grafo nelle sue componenti
- ◆ L'algoritmo viene poi eseguito su ognuna delle componenti
- ◆ I risultati vengono poi ricomposti assieme

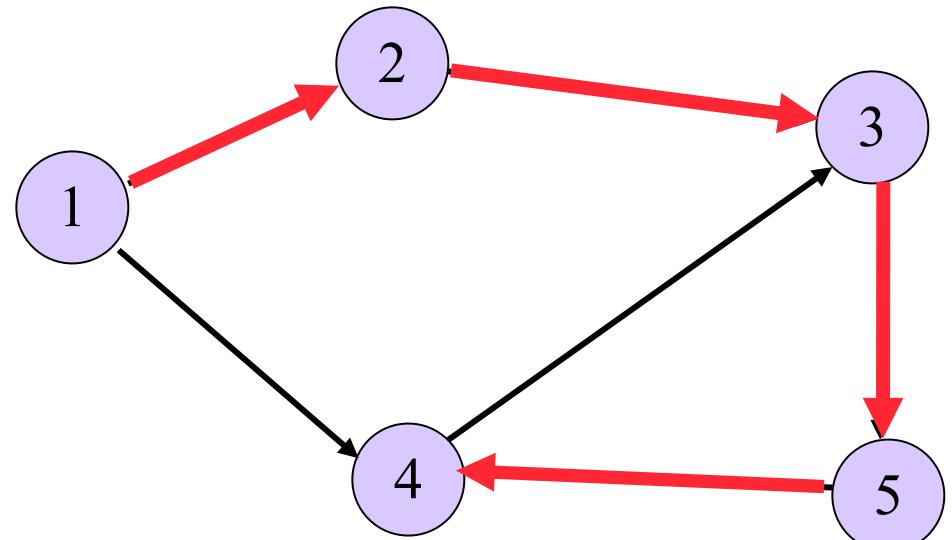
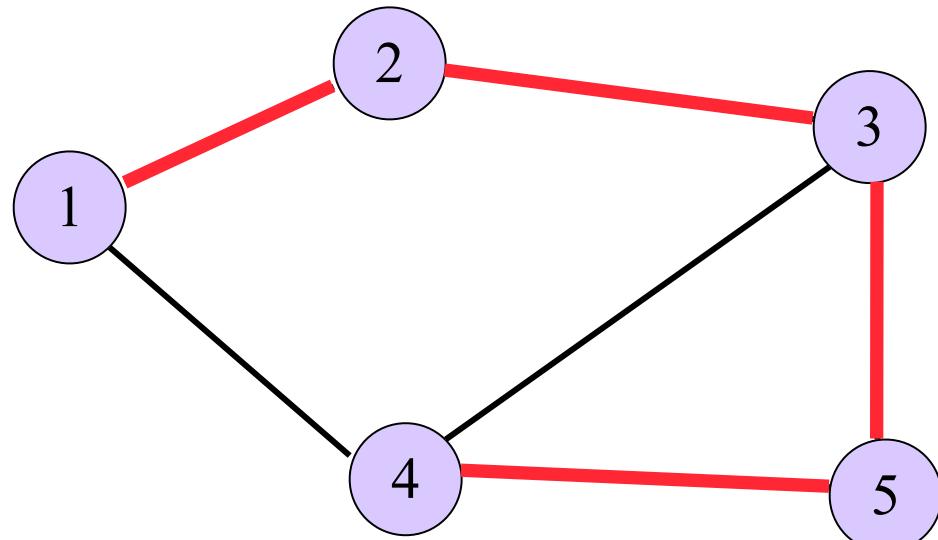
Definizioni: Raggiungibilità

- ♦ **In grafo orientato (non orientato)**

- ♦ Se esiste un cammino (catena) c tra i vertici u e v , si dice che v è *raggiungibile* da u tramite c

1 è raggiungibile da 4 e viceversa

4 è raggiungibile da 1 ma non viceversa



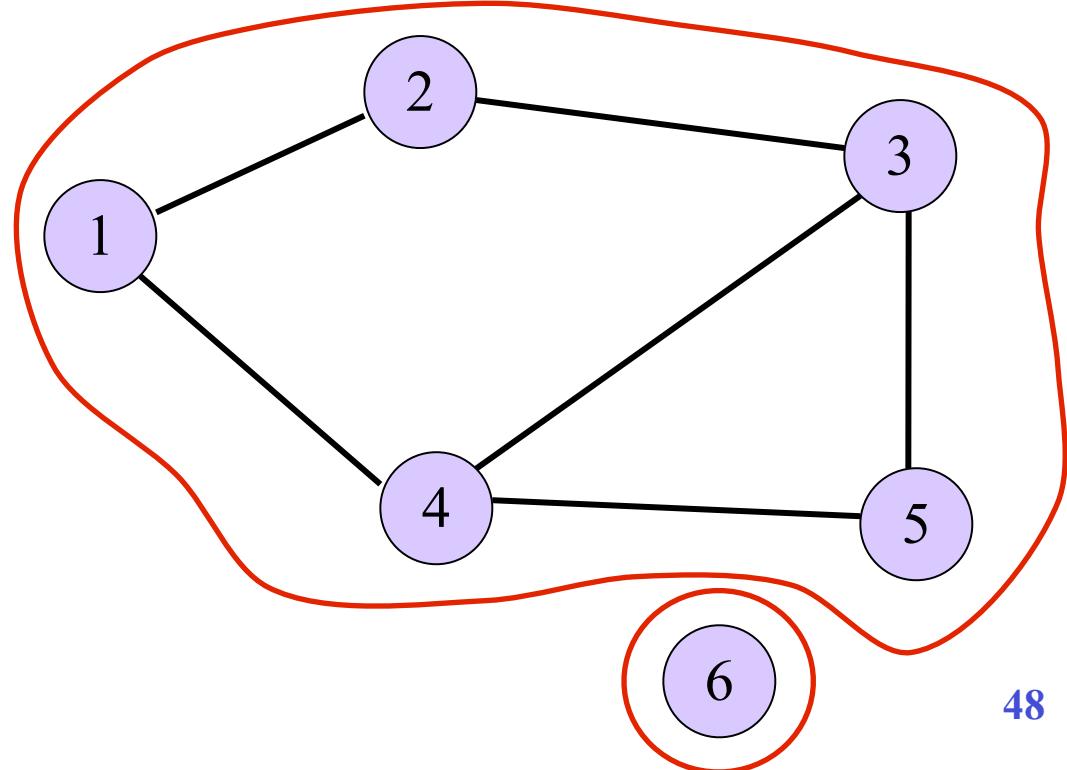
Definizioni: Grafi connessi e componenti connesse

♦ In un grafo non orientato G

- G è *connesso* \Leftrightarrow esiste un cammino da ogni vertice ad ogni altro vertice
- Un grafo $G' = (V', E')$ è una *componente连通的* di $G \Leftrightarrow$ è un sottografo di G fortemente e massimale

♦ Definizioni

- G' è un *sottografo* di G ($G' \subseteq G$) se e solo se $V' \subseteq V$ e $E' \subseteq E$
- G' è *massimale* \Leftrightarrow non esiste un sottografo G'' di G che sia connesso e “più grande” di G' , ovvero tale per cui $G' \subseteq G'' \subseteq G$



Applicazioni DFS: Componente connesse

♦ Problema

- ♦ Verificare se un grafo non orientato è connesso
- ♦ Identificare le componenti connesse di cui è composto

♦ Soluzione

- ♦ Un grafo è connesso se, al termine della DFS, tutti i nodi sono stati marcati
- ♦ Altrimenti, una singola passata non è sufficiente e la visita deve ripartire da un nodo non marcato, scoprendo una nuova porzione del grafo

♦ Strutture dati

- ♦ Vettore id degli identificatori di componente
- ♦ $id[u]$ è l'identificatore della componente connessa a cui appartiene u

Componenti connesse

```
integer[] cc(GRAPH G, NODE[] ordine)
```

```
    integer[] id ← new integer[1 ... G.n]
```

```
    foreach  $u \in G.V()$  do  $id[u] \leftarrow 0$ 
```

```
    integer conta ← 0
```

```
    for integer  $i \leftarrow 1$  to  $G.n$  do
```

```
        if  $id[ordine[i]] = 0$  then
```

```
            conta ← conta + 1
```

```
            ccdfs( $G, conta, ordine[i], id$ )
```

```
    return id
```

```
ccdfs(GRAPH G, integer conta, NODE u, integer[] id)
```

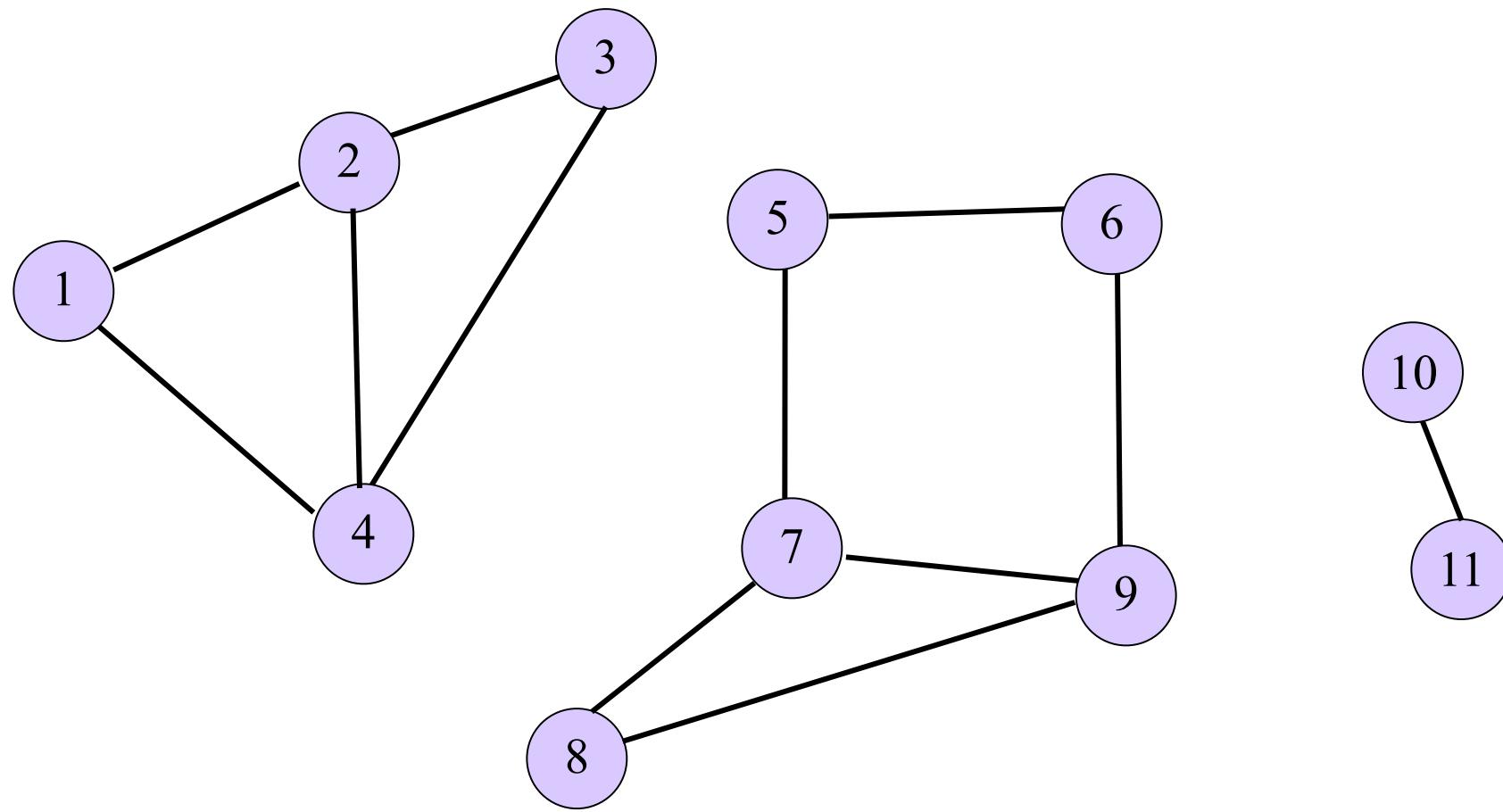
```
     $id[u] \leftarrow conta$ 
```

```
    foreach  $v \in G.adj(u)$  do
```

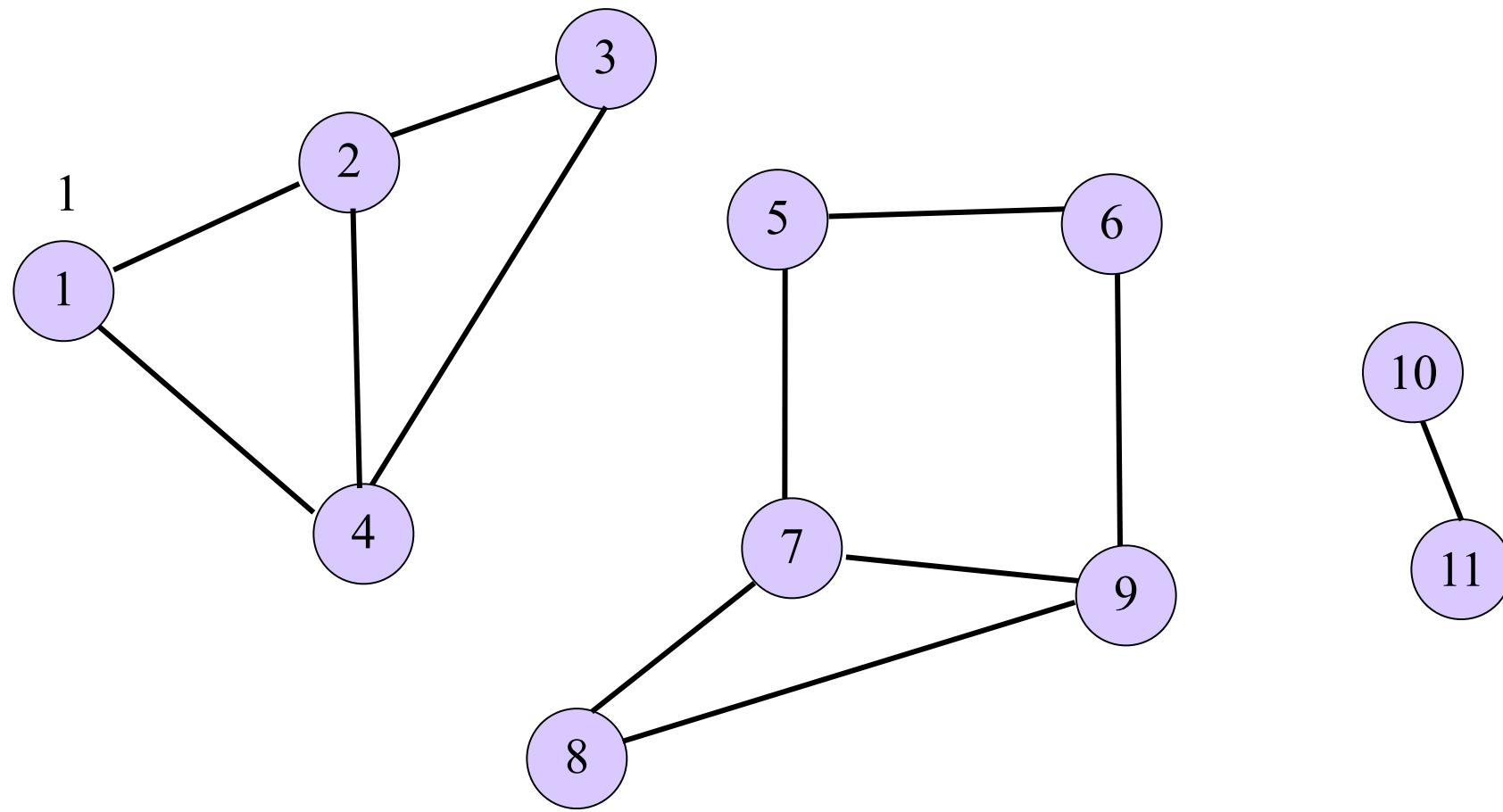
```
        if  $id[v] = 0$  then
```

```
            ccdfs( $G, conta, v, id$ )
```

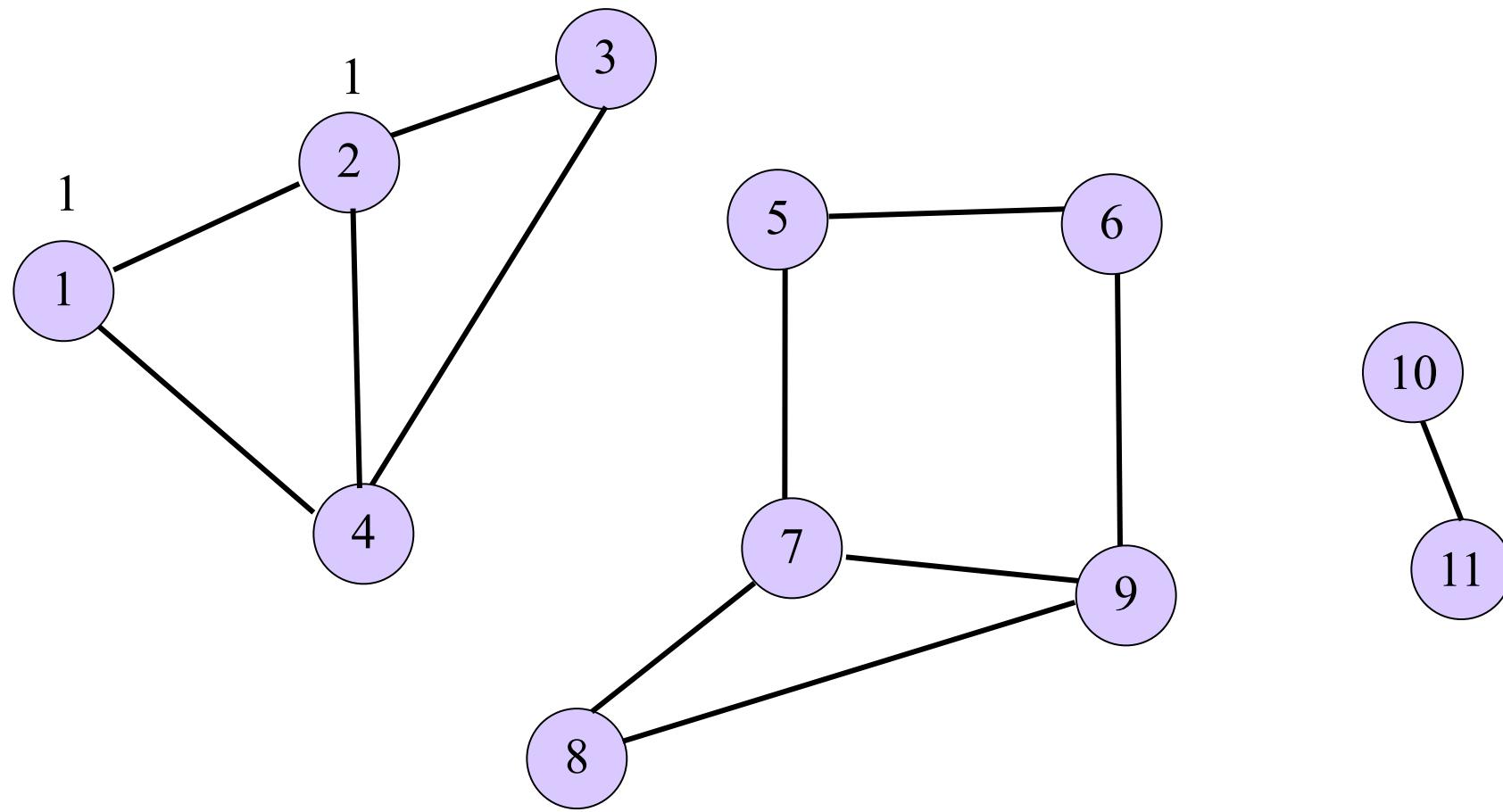
Componenti connesse



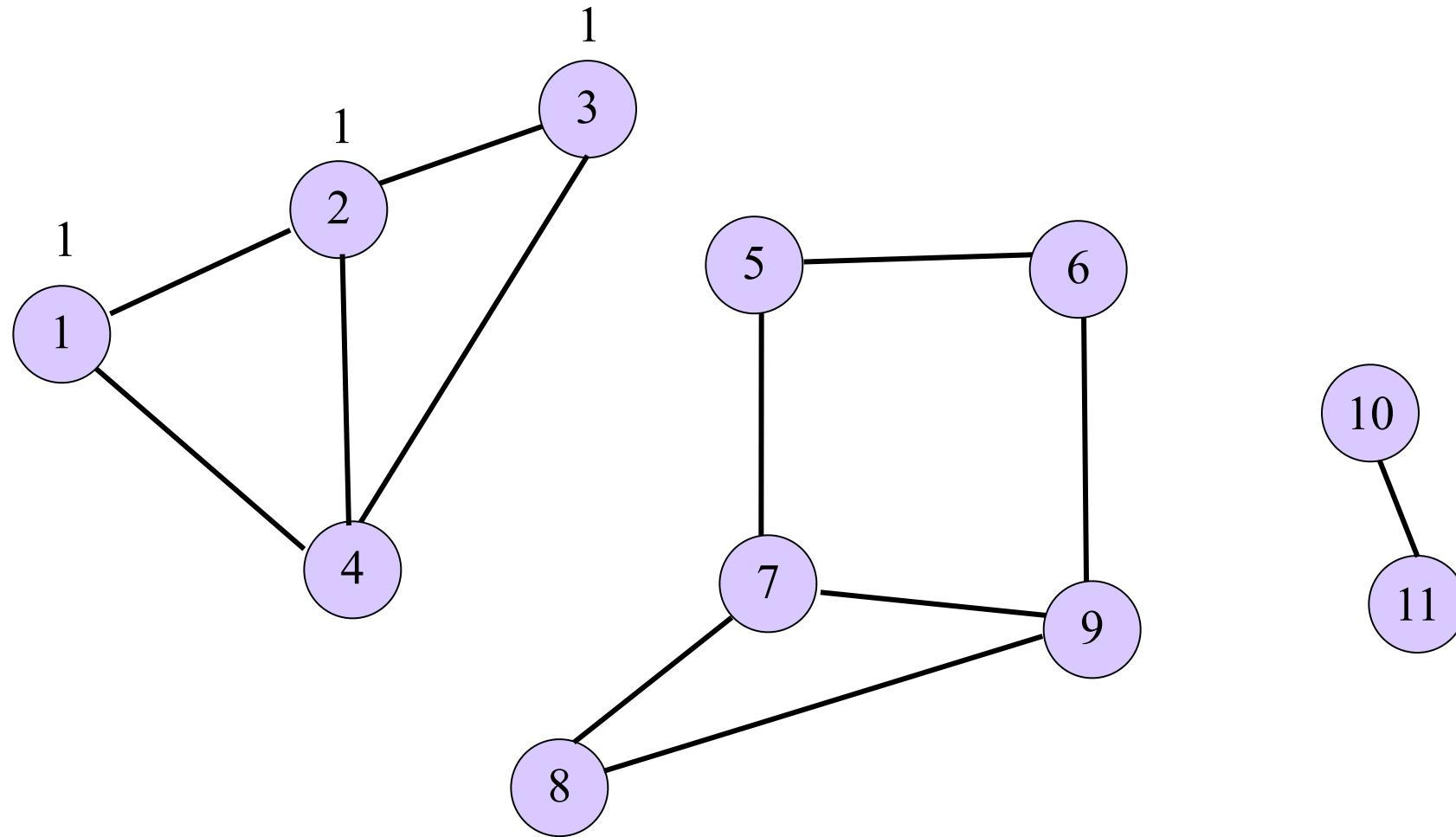
Componenti connesse



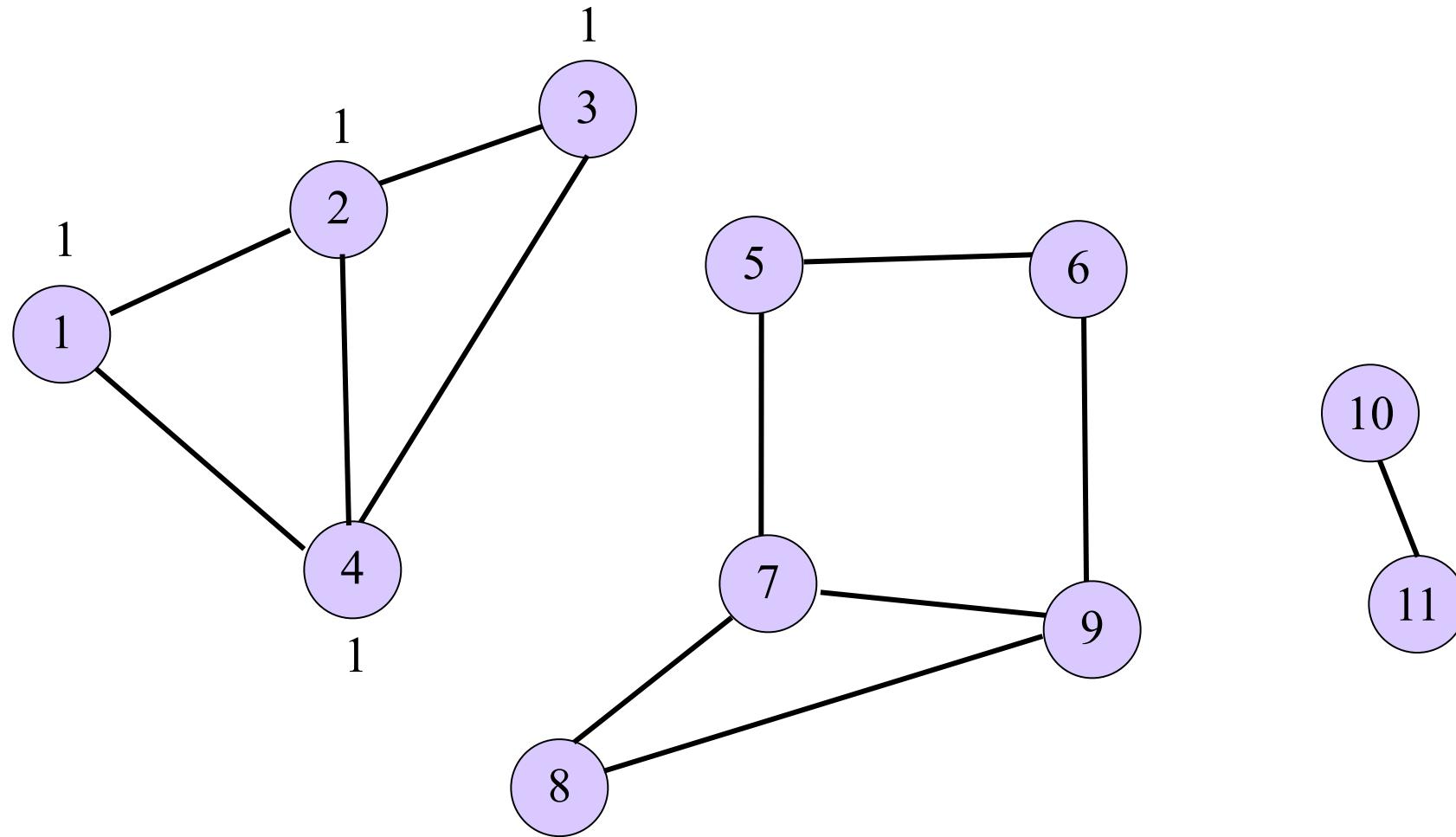
Componenti connesse



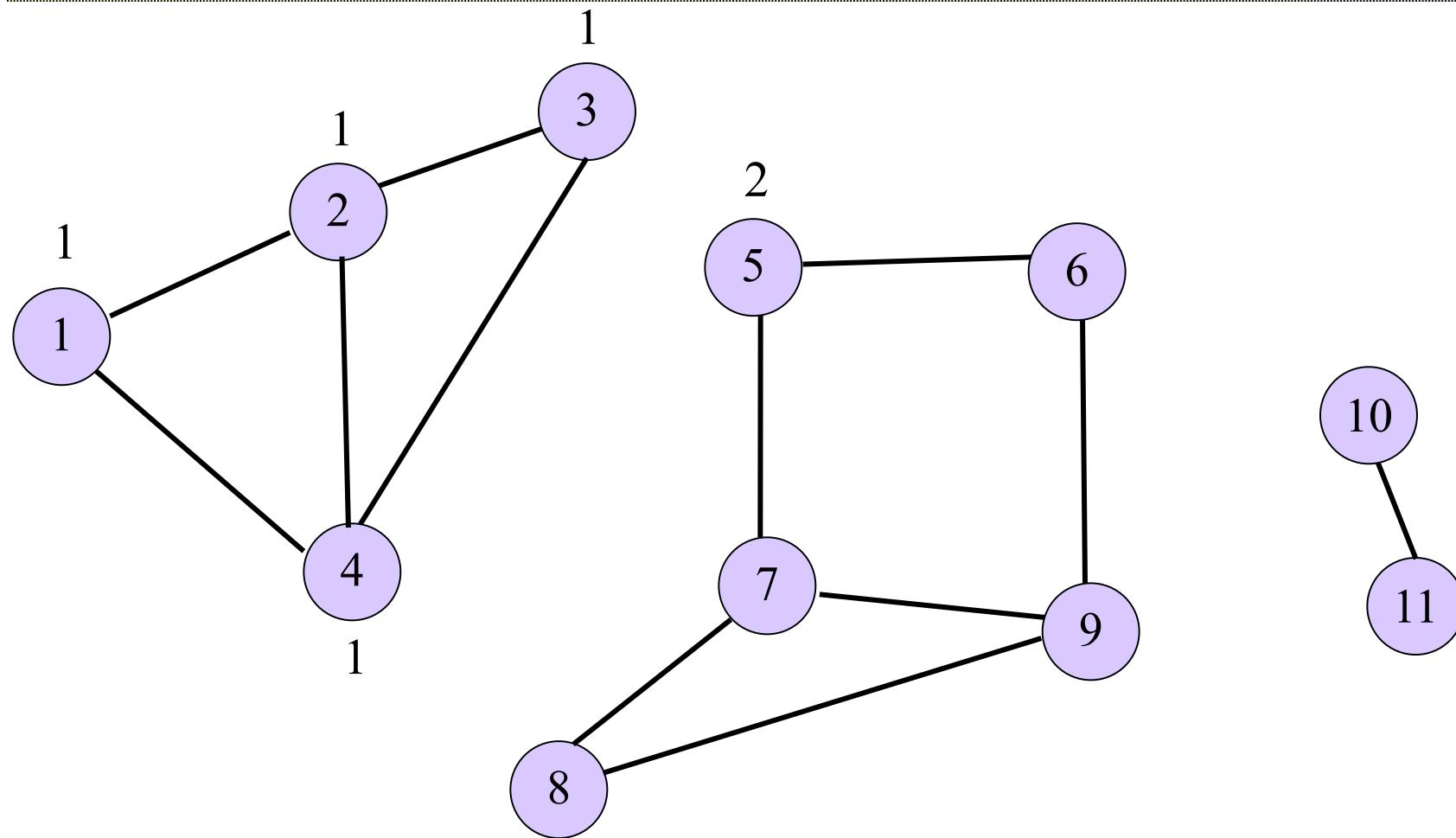
Componenti connesse



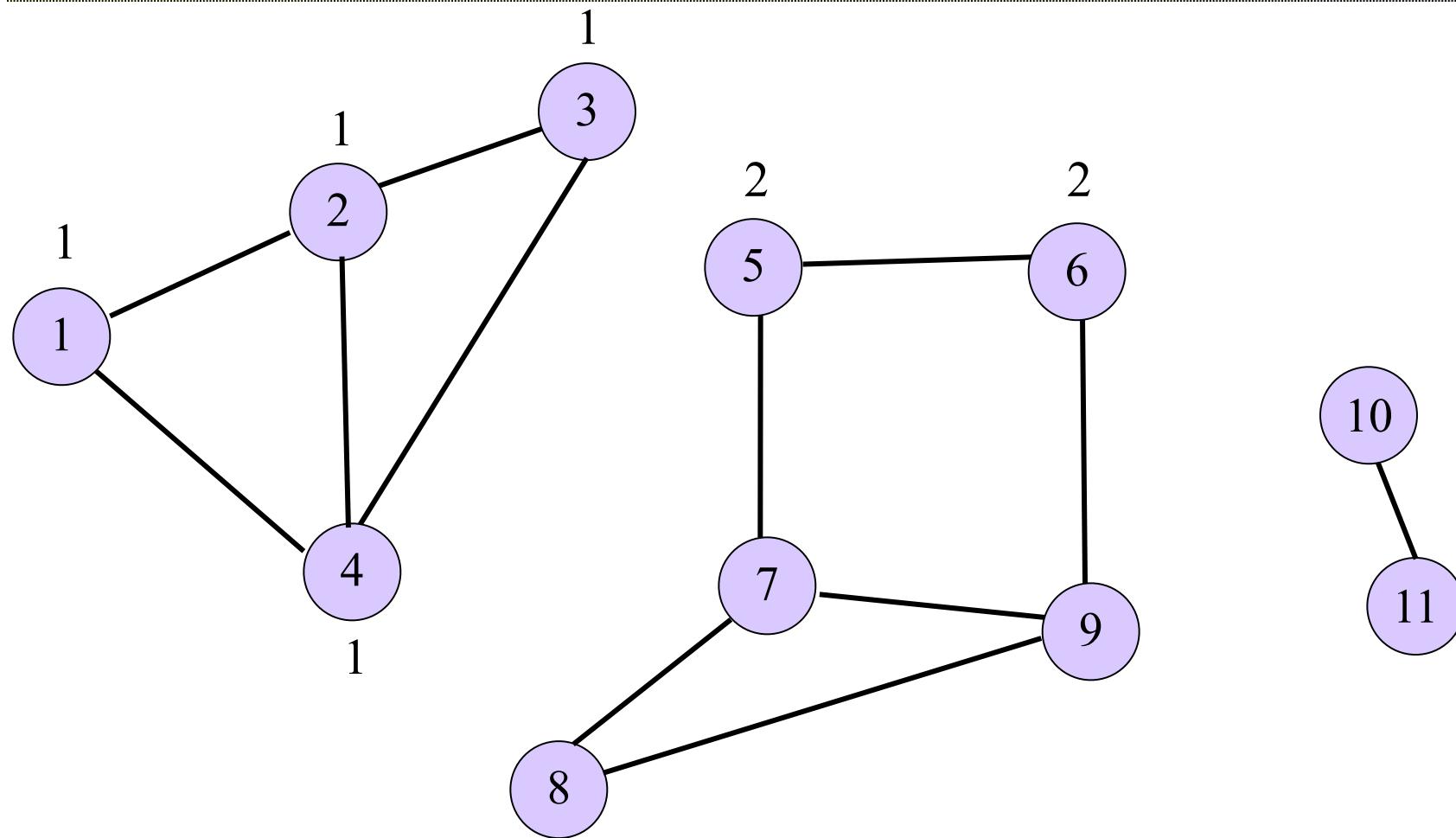
Componenti connesse



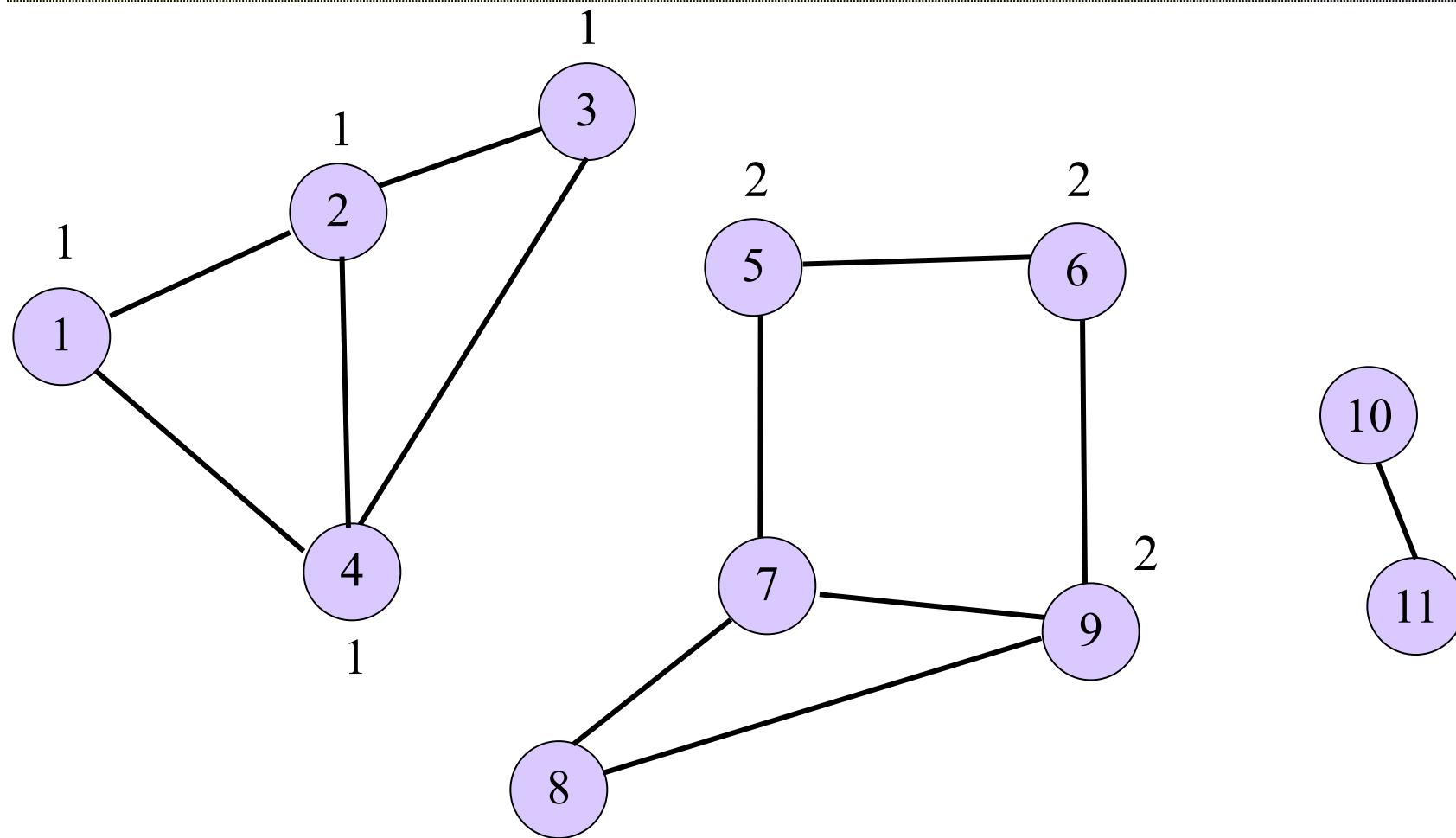
Componenti connesse



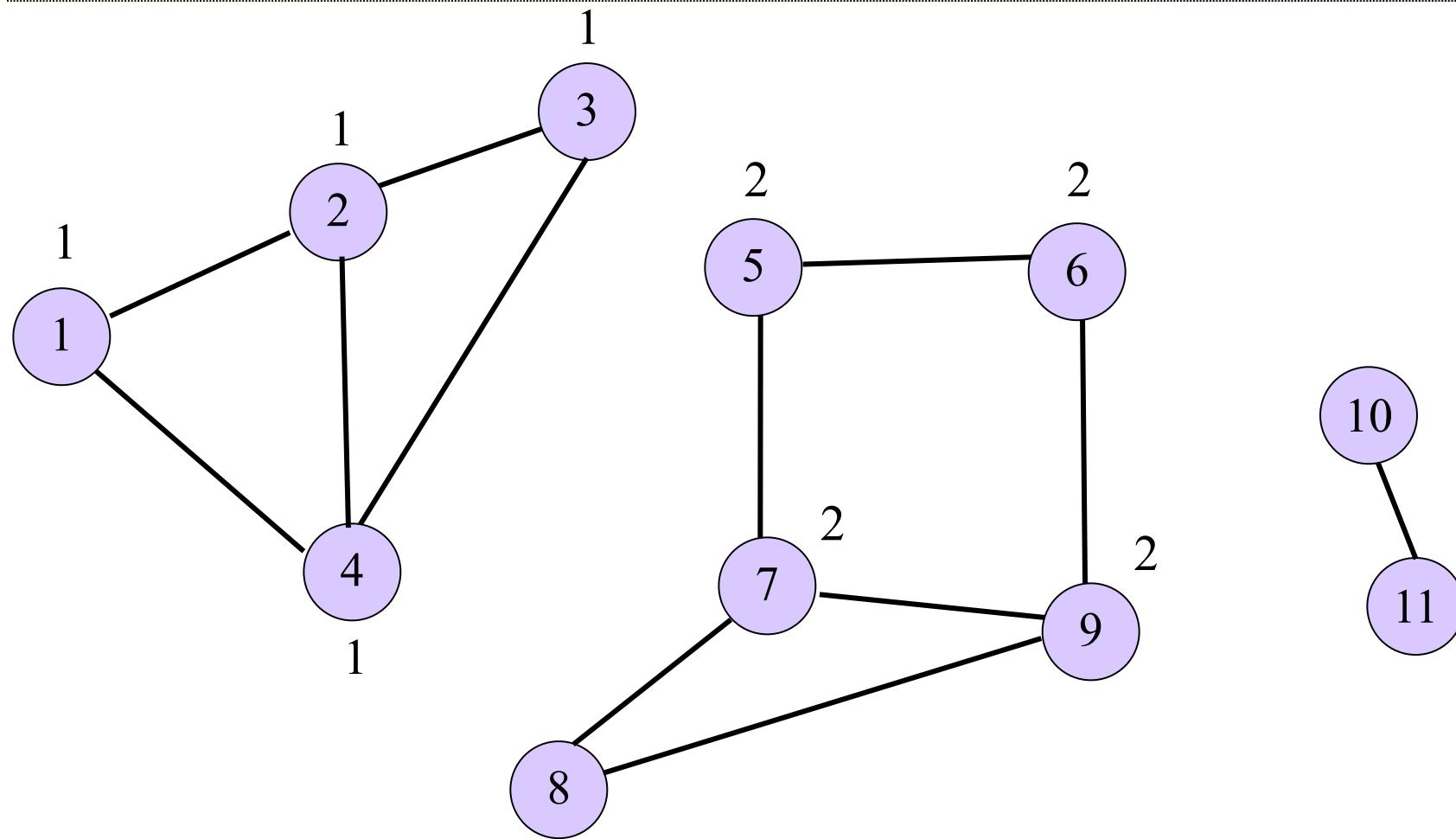
Componenti connesse



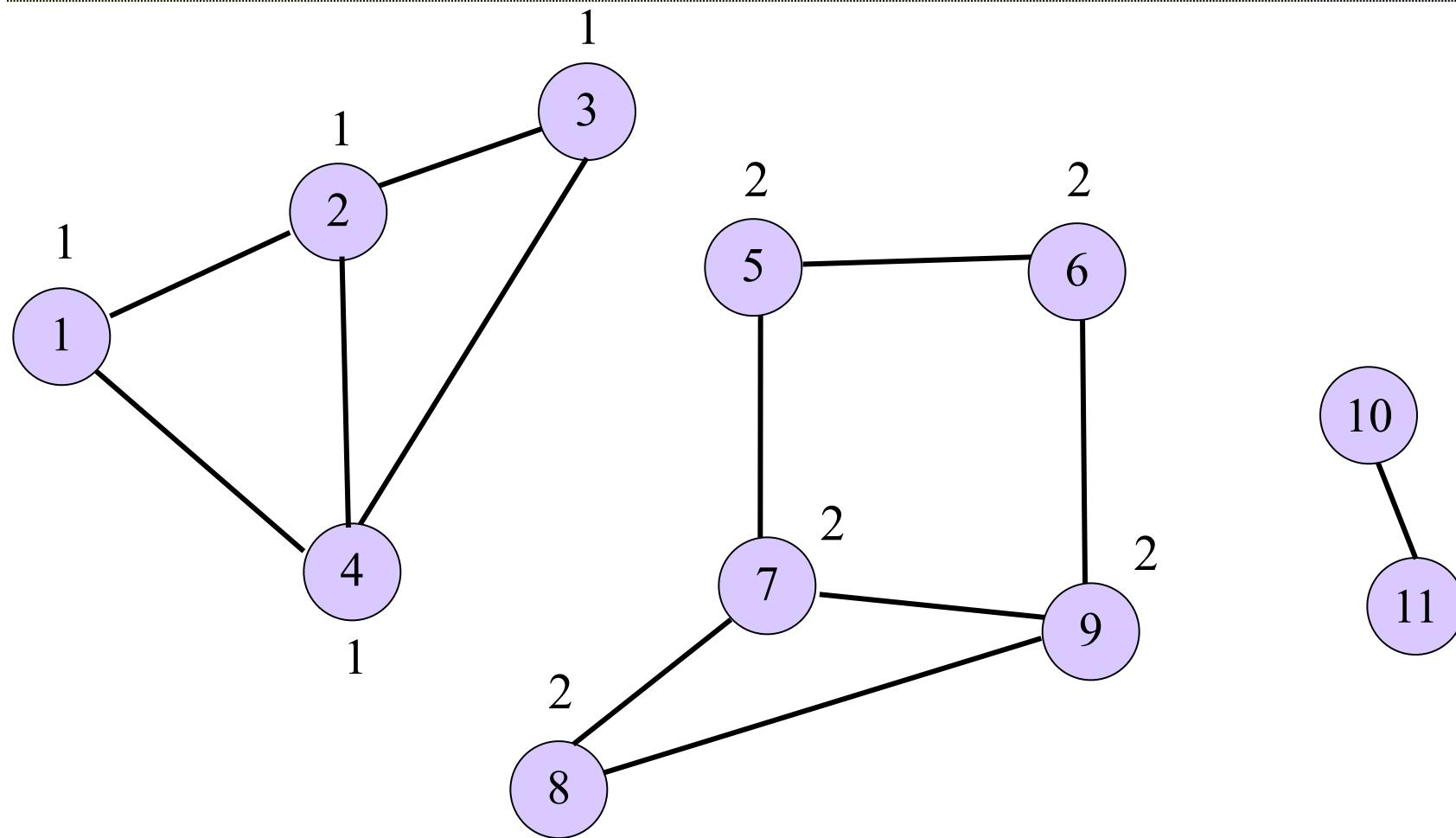
Componenti connesse



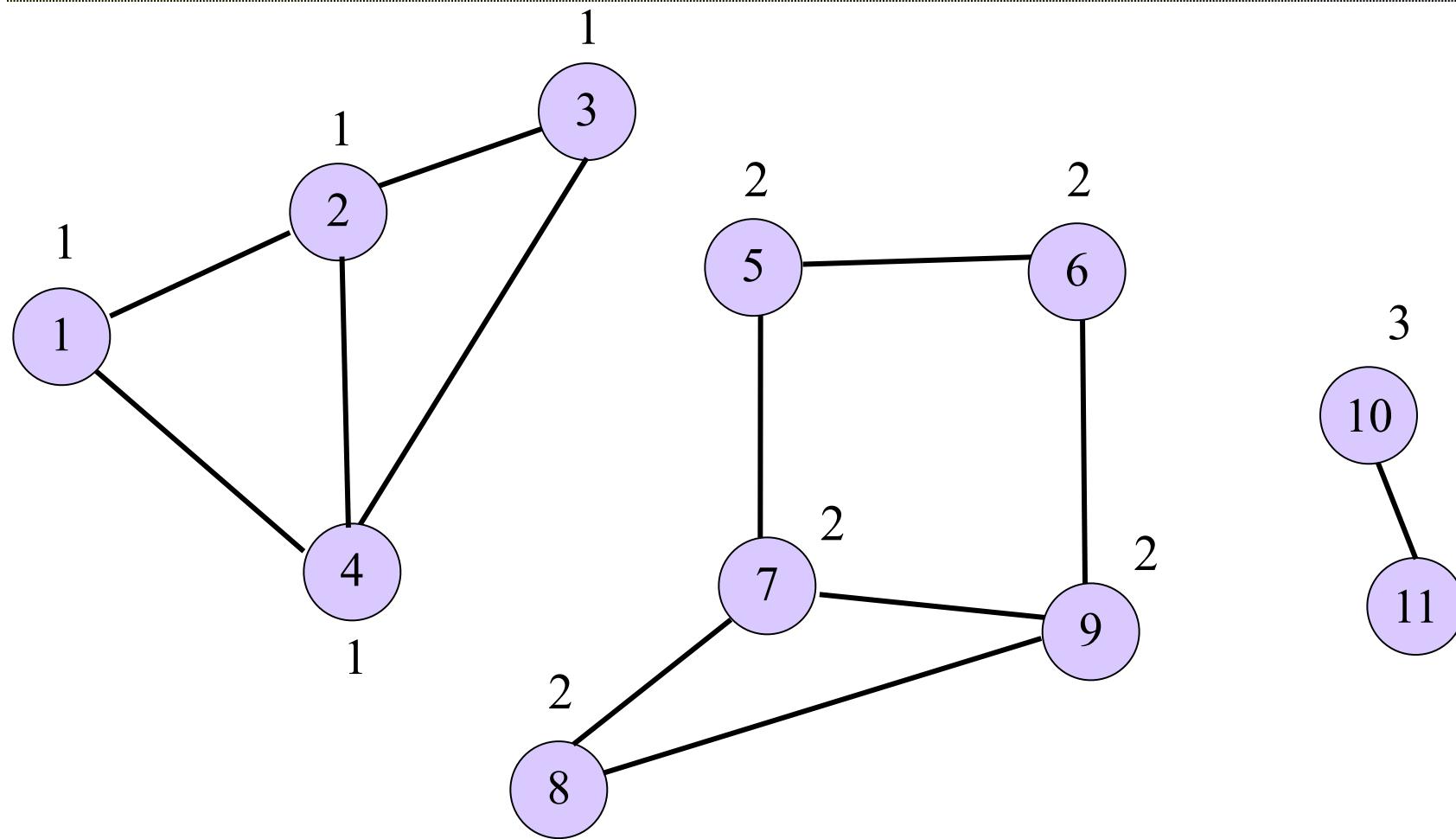
Componenti connesse



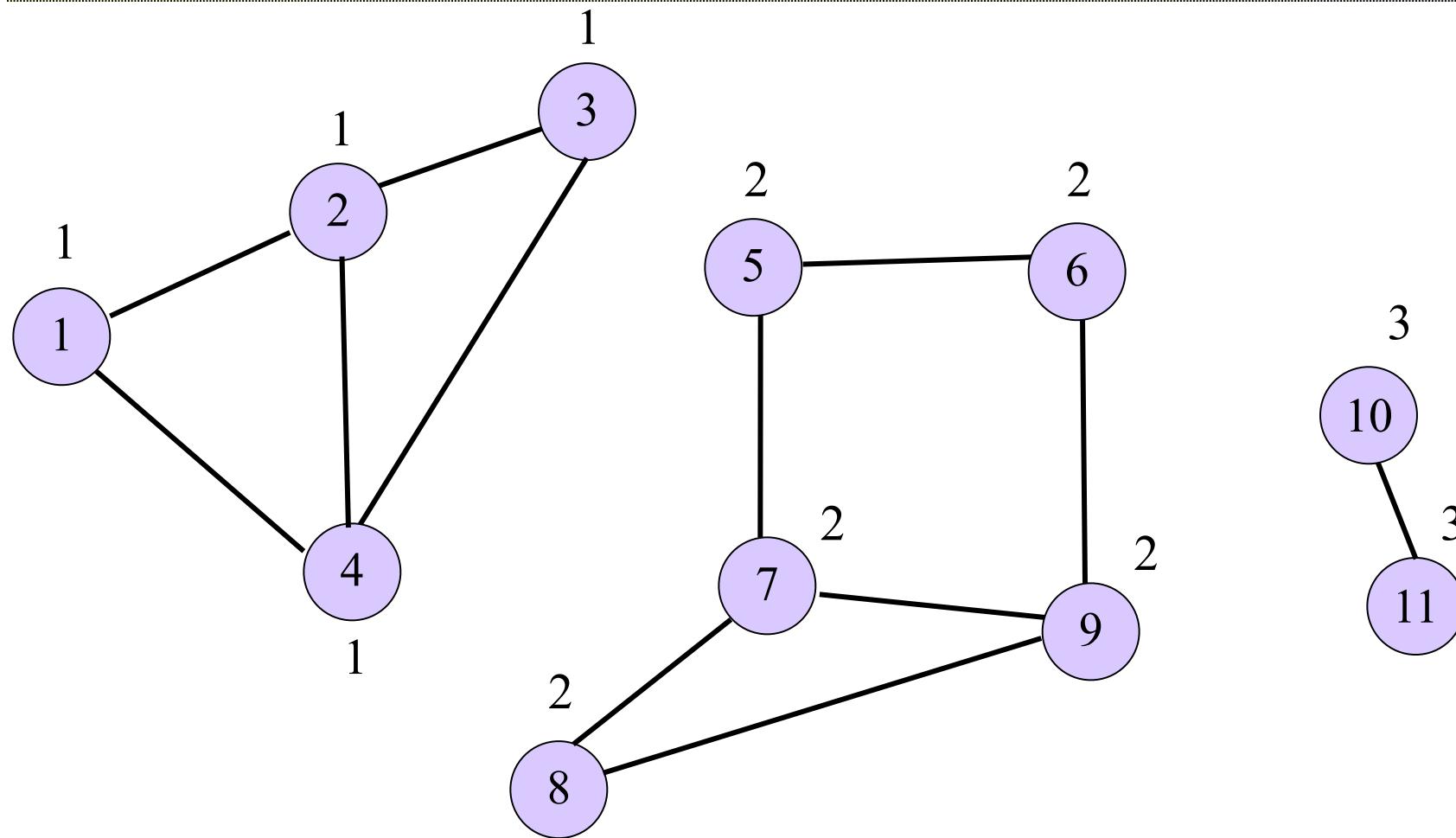
Componenti connesse



Componenti connesse



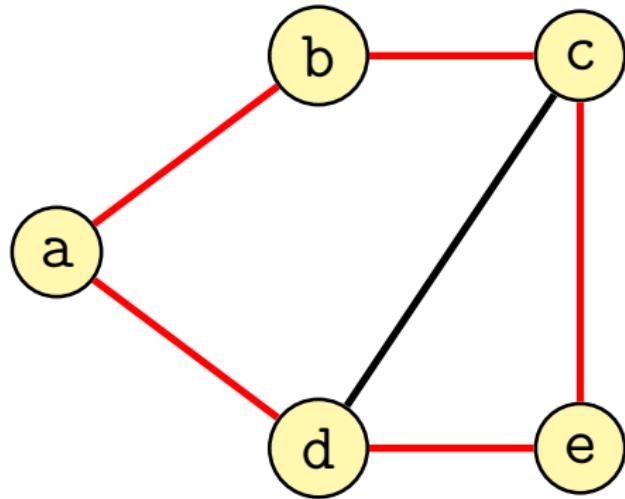
Componenti connesse



Definizioni: Ciclo

Ciclo (cycle)

In un grafo non orientato $G = (V, E)$, un **ciclo** C di lunghezza $k > 2$ è una sequenza di nodi u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1} \in E)$ per $0 \leq i \leq k - 1$ e $u_0 = u_k$.

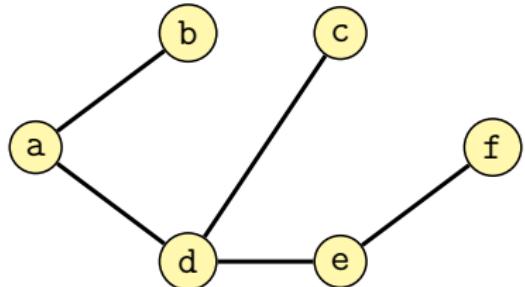


$k > 2$ esclude cicli banali composti da coppie di archi (u, v) e (v, u) , che sono onnipresenti nei grafi non orientati.

Definizioni: Grafo aciclico

Grafo aciclico

Un grafo non orientato che non contiene cicli è detto **aciclico**.



Problema

Dato un grafo non orientato G , scrivere un algoritmo che restituisca **true** se G contiene un ciclo, **false** altrimenti.

Applicazione DFS: Grafo non orientato aciclico

boolean hasCycleRec(GRAPH G , NODE u , NODE p , boolean[] $visited$)

$visited[u] = \text{true}$

foreach $v \in G.\text{adj}(u) - \{p\}$ do

if $visited[v]$ then

return true

else if hasCycleRec($G, v, u, visited$) then

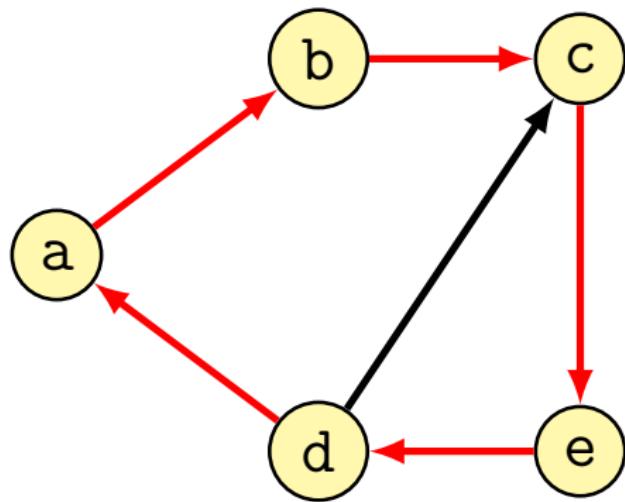
return true

return false

Definizioni: Ciclo

Ciclo (cycle)

In un grafo orientato $G = (V, E)$, un **ciclo** C di lunghezza $k \geq 2$ è una sequenza di nodi u_0, u_1, \dots, u_k tale che $(u_i, u_{i+1} \in E)$ per $0 \leq i \leq k - 1$ e $u_0 = u_k$.



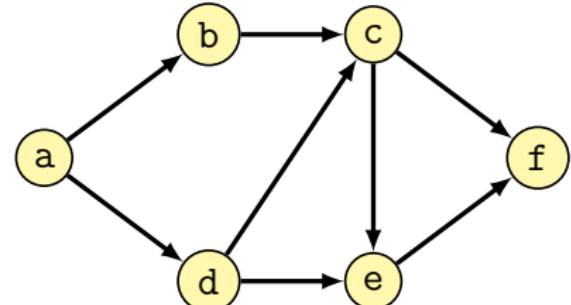
Esempio: a, b, c, e, d, a è un cammino nel grafo di lunghezza 5

Note: un ciclo è detto **semplice** se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)

Definizioni: Grafo orientato aciclico (DAG)

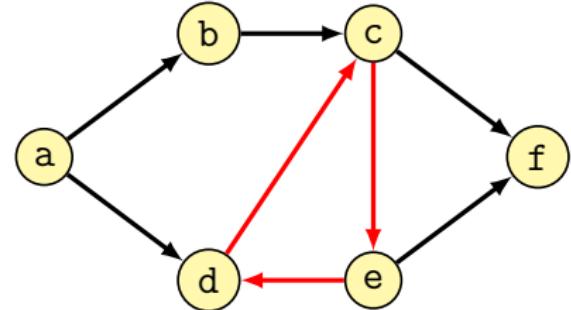
DAG

Un grafo orientato che non contiene cicli è detto **DAG** (directed acyclic graph).



Grafo ciclico

Un grafo è **ciclico** se contiene un ciclo.



Applicazione DFS: Grafo orientato aciclico

Problema

Dato un grafo orientato G , scrivere un algoritmo che restituisca **true** se G contiene un ciclo, **false** altrimenti.

Problema

Riuscite a concepire un grafo orientato per cui l'algoritmo appena visto non si comporta correttamente?

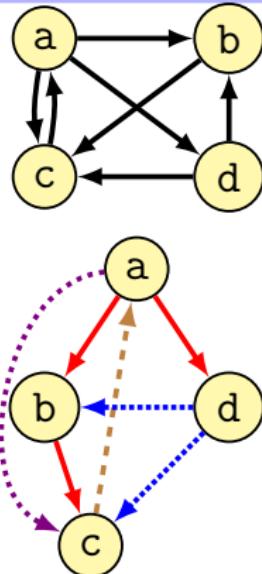
Classificazione degli archi

Albero di copertura DFS

Ogni volta che si esamina un arco da un nodo marcato ad un nodo non marcato, tale arco viene **arco dell'albero**

Gli archi (u, v) non inclusi nell'albero possono essere divisi in tre categorie

- Se u è un antenato di v in T , (u, v) è detto **arco in avanti**
- Se u è un discendente di v in T , (u, v) è detto **arco all'indietro**
- Altrimenti, viene detto **arco di attraversamento**



DFS Schema

```
dfs-schema(GRAPH G, NODE u, int &time, int[] dt,
int[] ft)
```

{ visita il nodo u (pre-order) }

$time = time + 1$; $dt[u] = time$

foreach $v \in G.\text{adj}(u)$ **do**

{ visita l'arco (u, v) (qualsiasi) }

if $dt[v] == 0$ **then**

{ visita l'arco (u, v) (albero) }
 dfs-schema($G, v, time, dt, ft$)

else if $dt[u] > dt[v]$ **and** $ft[v] == 0$ **then**

{ visita l'arco (u, v) (indietro) }

else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**

{ visita l'arco (u, v) (avanti) }

else

{ visita l'arco (u, v) (attraversamento) }

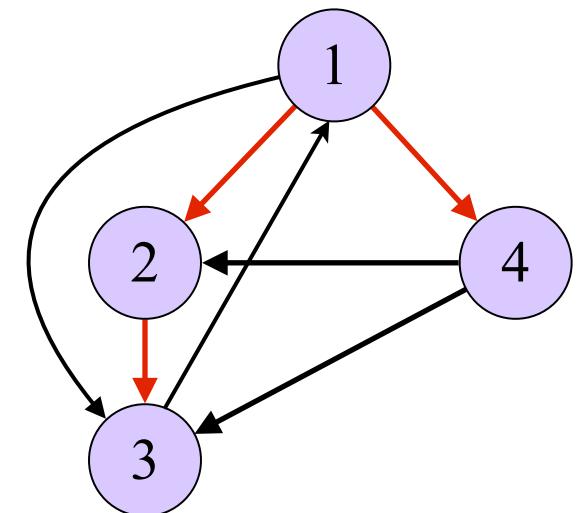
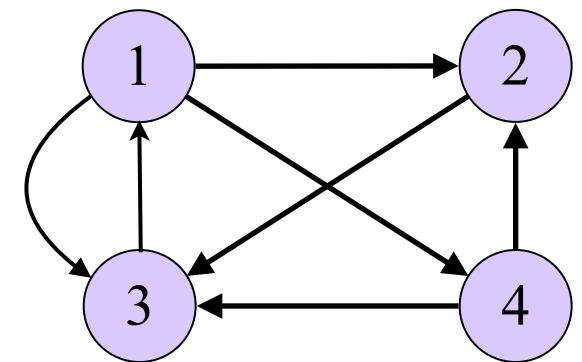
{ visita il nodo u (post-order) }

$time = time + 1$; $ft[u] = time$

- $time$: contatore
- dt : discovery time
(tempo di scoperta)
- ft : finish time
(tempo di fine)

Alberi di copertura DFS

- ♦ La visita DFS genera l'albero (foresta) dei cammini DFS
 - ♦ Tutte le volte che viene incontrato un arco che connette un nodo marcato ad uno non marcato, esso viene inserito nell'albero T
- ♦ Gli archi non inclusi in T possono essere divisi in tre categorie durante la visita:
 - ♦ se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T , è detto *arco all'indietro*
 - ♦ se l'arco è esaminato passando da un nodo di T ad un suo discendente (che non sia figlio) in T è detto *arco in avanti*
 - ♦ altrimenti, è detto *arco di attraversamento*



Schema DFS

♦ Variabili globali

- ♦ $time$ orologio
- ♦ dt discovery time
- ♦ ft finish time

dfs-schema(GRAPH G , NODE u)

esamina il nodo u prima (caso *pre-visita*)

$time \leftarrow time + 1; dt[u] \leftarrow time$

foreach $v \in G.\text{adj}(u)$ **do**

esamina l'arco (u, v) di qualsiasi tipo

if $dt[v] = 0$ **then**

 esamina l'arco (u, v) in T

 dfs-schema(g, v)

else if $dt[u] > dt[v]$ **and** $ft[v] = 0$ **then**

 esamina l'arco (u, v) all'indietro

else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**

 esamina l'arco (u, v) in avanti

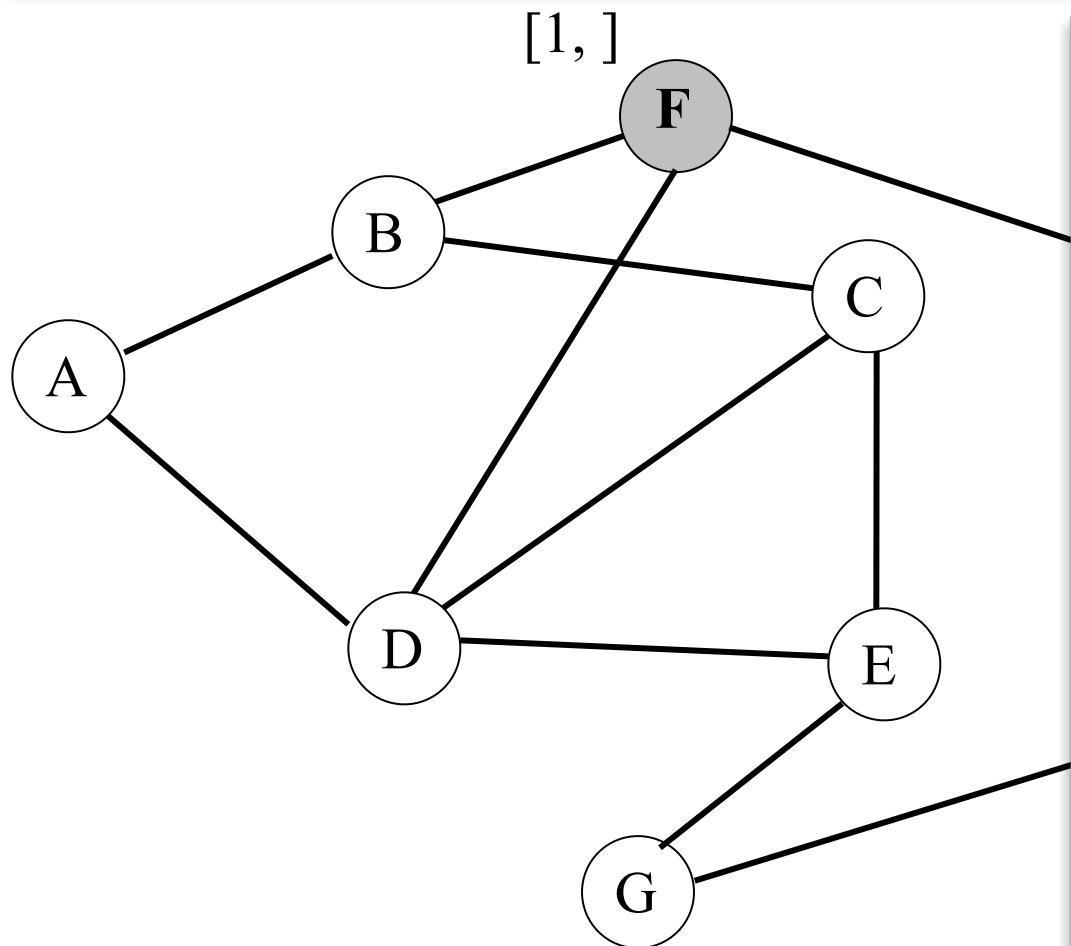
else

 esamina l'arco (u, v) di attraversamento

esamina il nodo u dopo (caso *post-visita*)

$time \leftarrow time + 1; ft[u] \leftarrow time$

Esempio



dfs-schema(GRAPH G , NODE u)

esamina il nodo u prima (caso *pre-visita*)

$time \leftarrow time + 1$; $dt[u] \leftarrow time$

foreach $v \in G.\text{adj}(u)$ **do**

esamina l'arco (u, v) di qualsiasi tipo

if $dt[v] = 0$ **then**

 | esamina l'arco (u, v) in T
 | dfs-schema(g, v)

else if $dt[u] > dt[v]$ **and** $ft[v] = 0$ **then**

 | esamina l'arco (u, v) all'indietro

else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**

 | esamina l'arco (u, v) in avanti

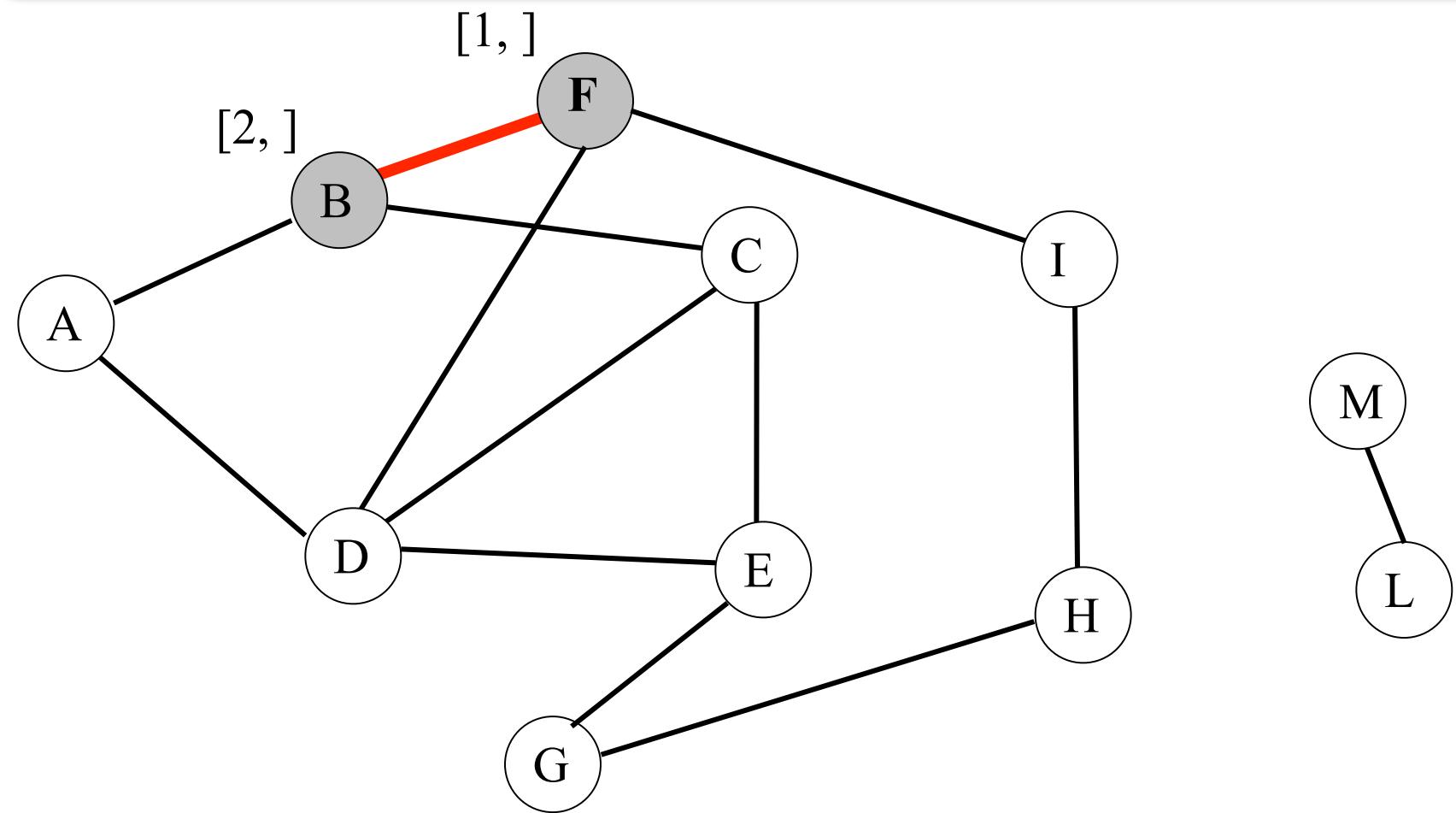
else

 | esamina l'arco (u, v) di attraversamento

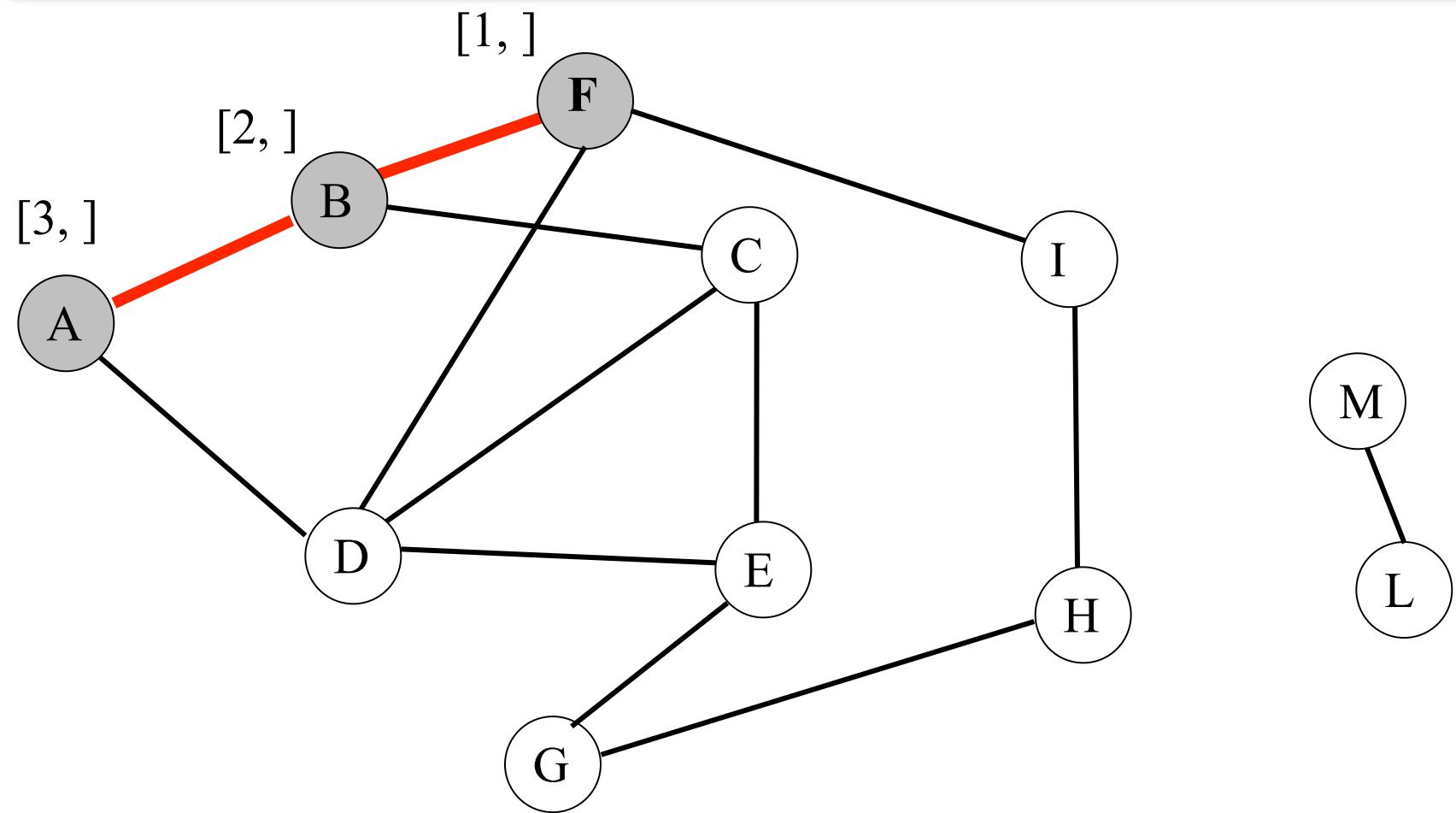
esamina il nodo u dopo (caso *post-visita*)

$time \leftarrow time + 1$; $ft[u] \leftarrow time$

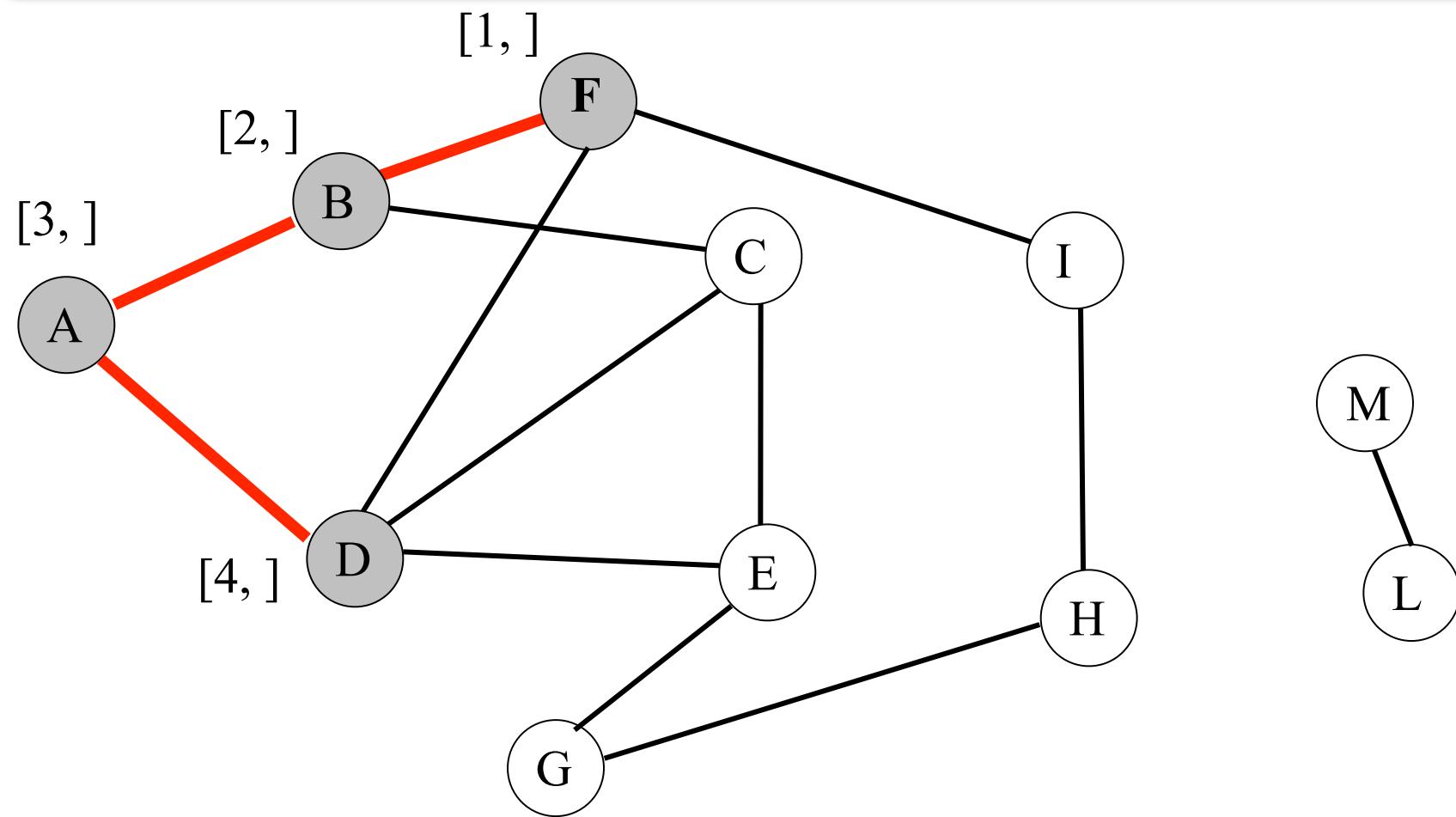
Esempio



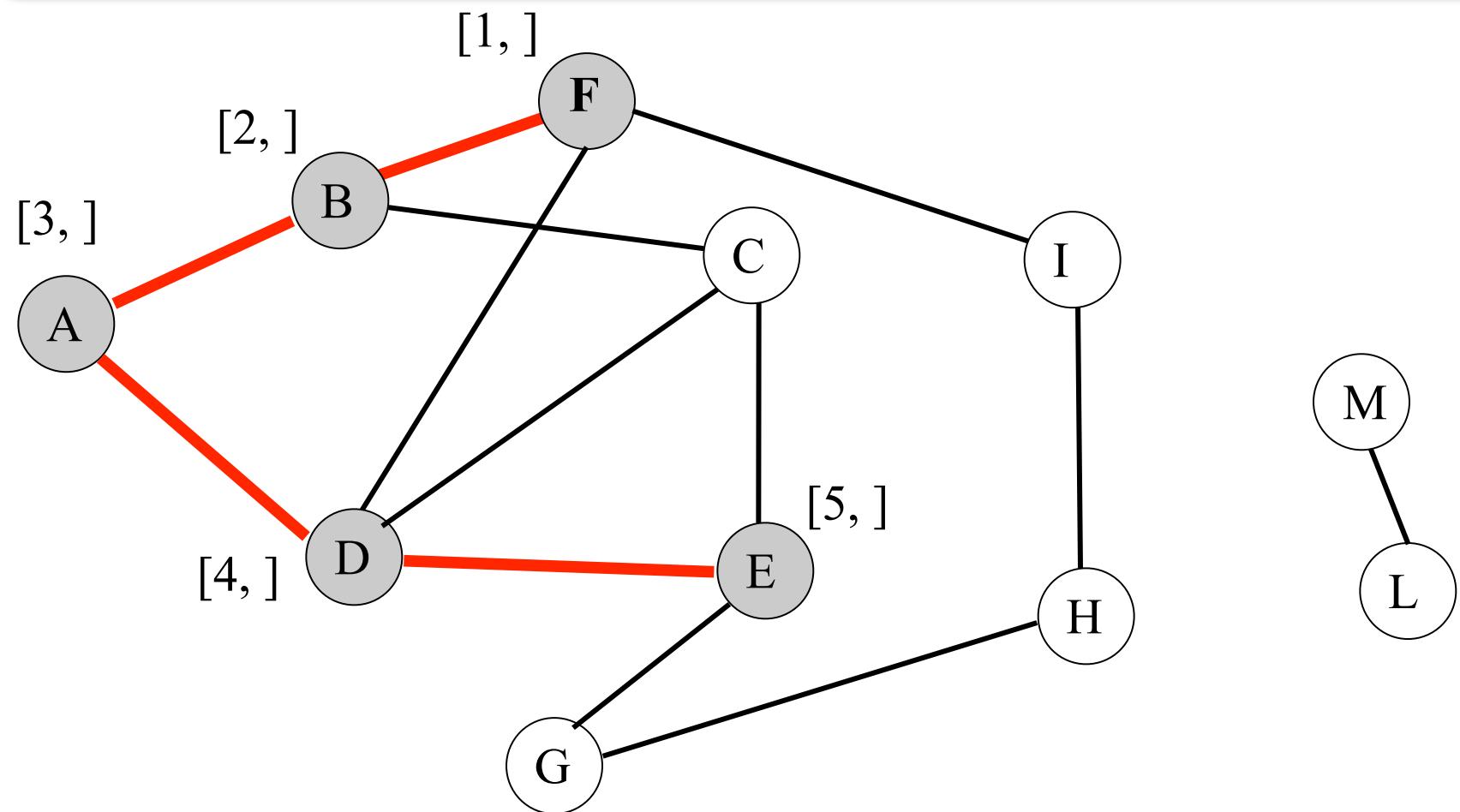
Esempio



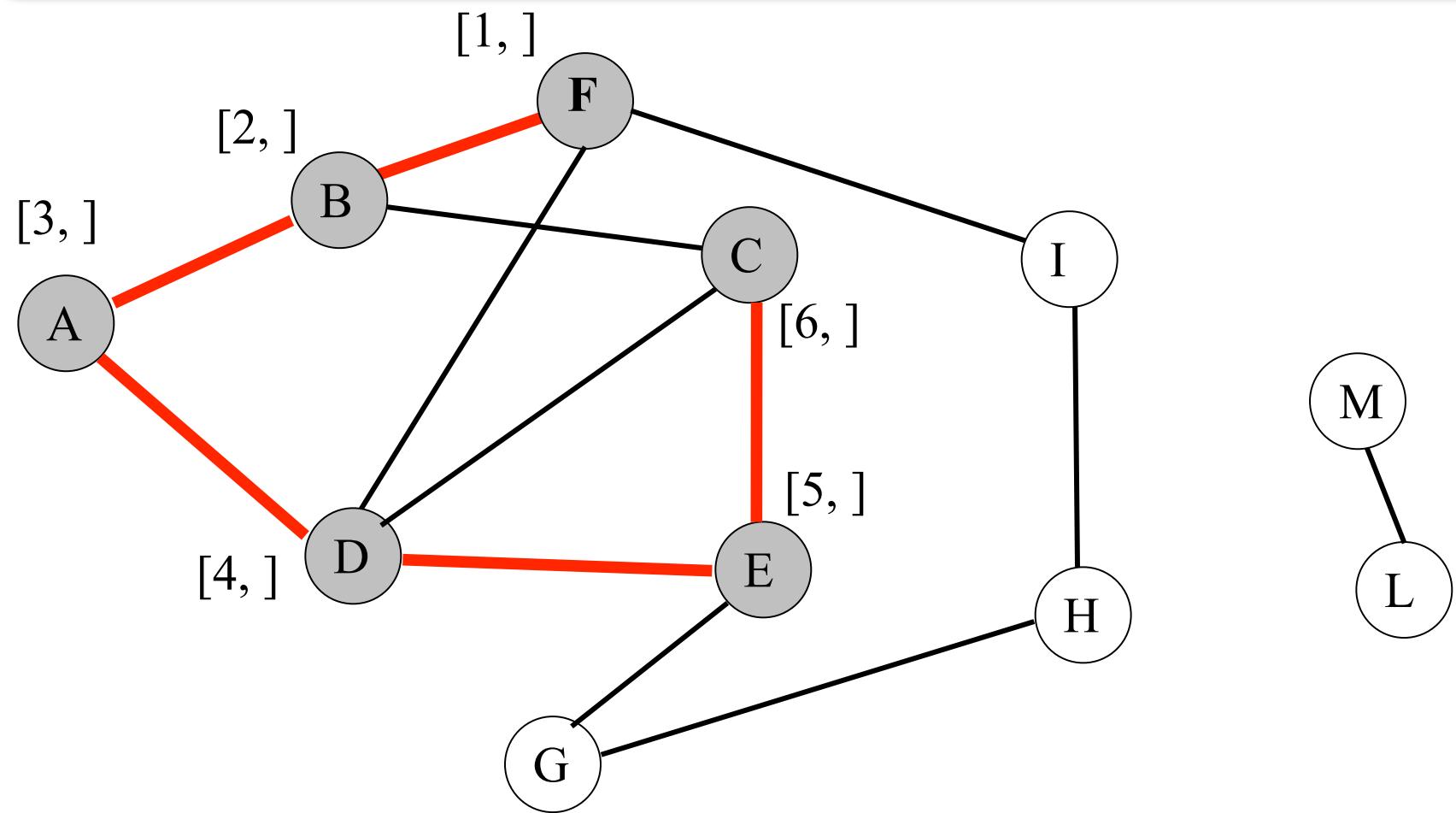
Esempio



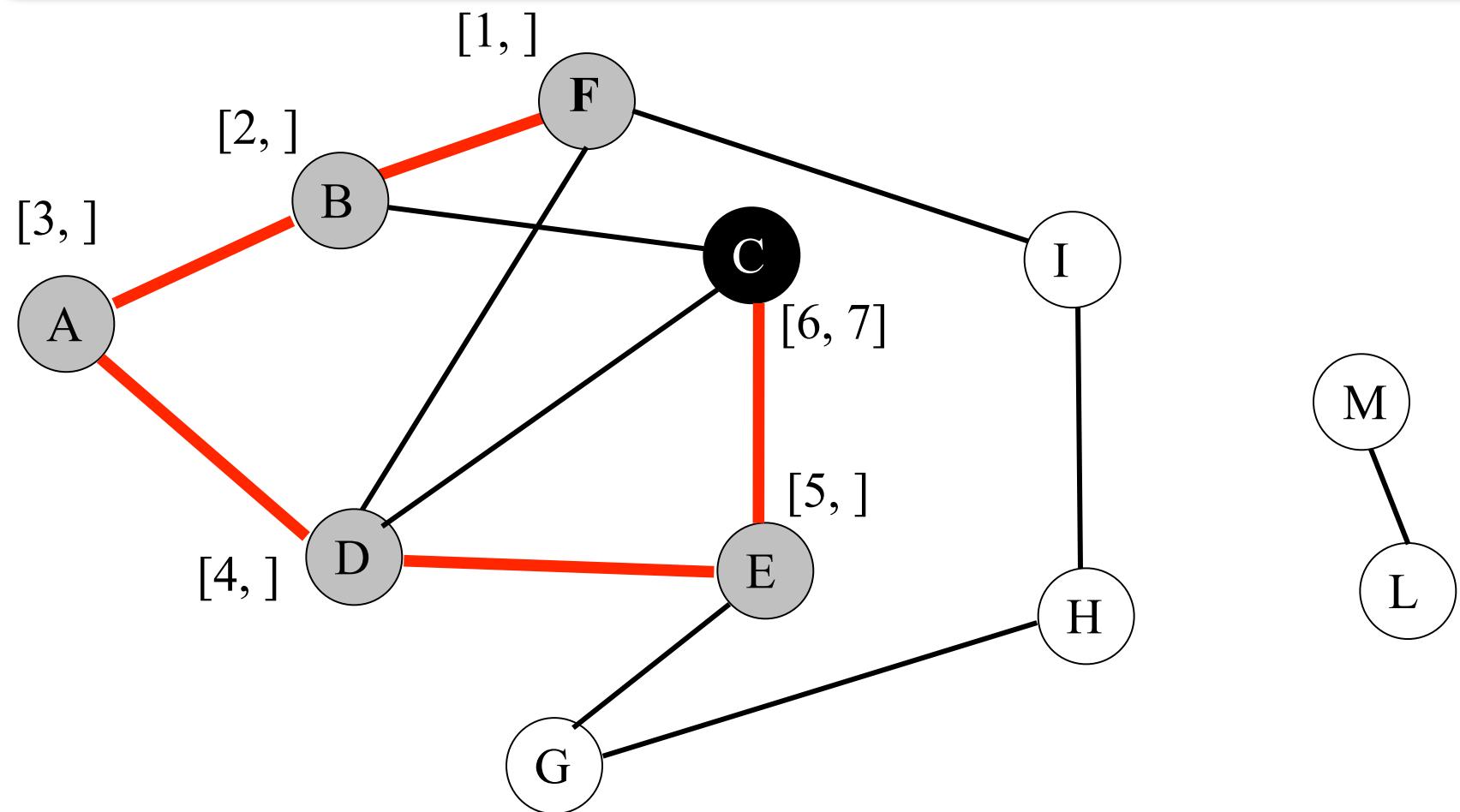
Esempio



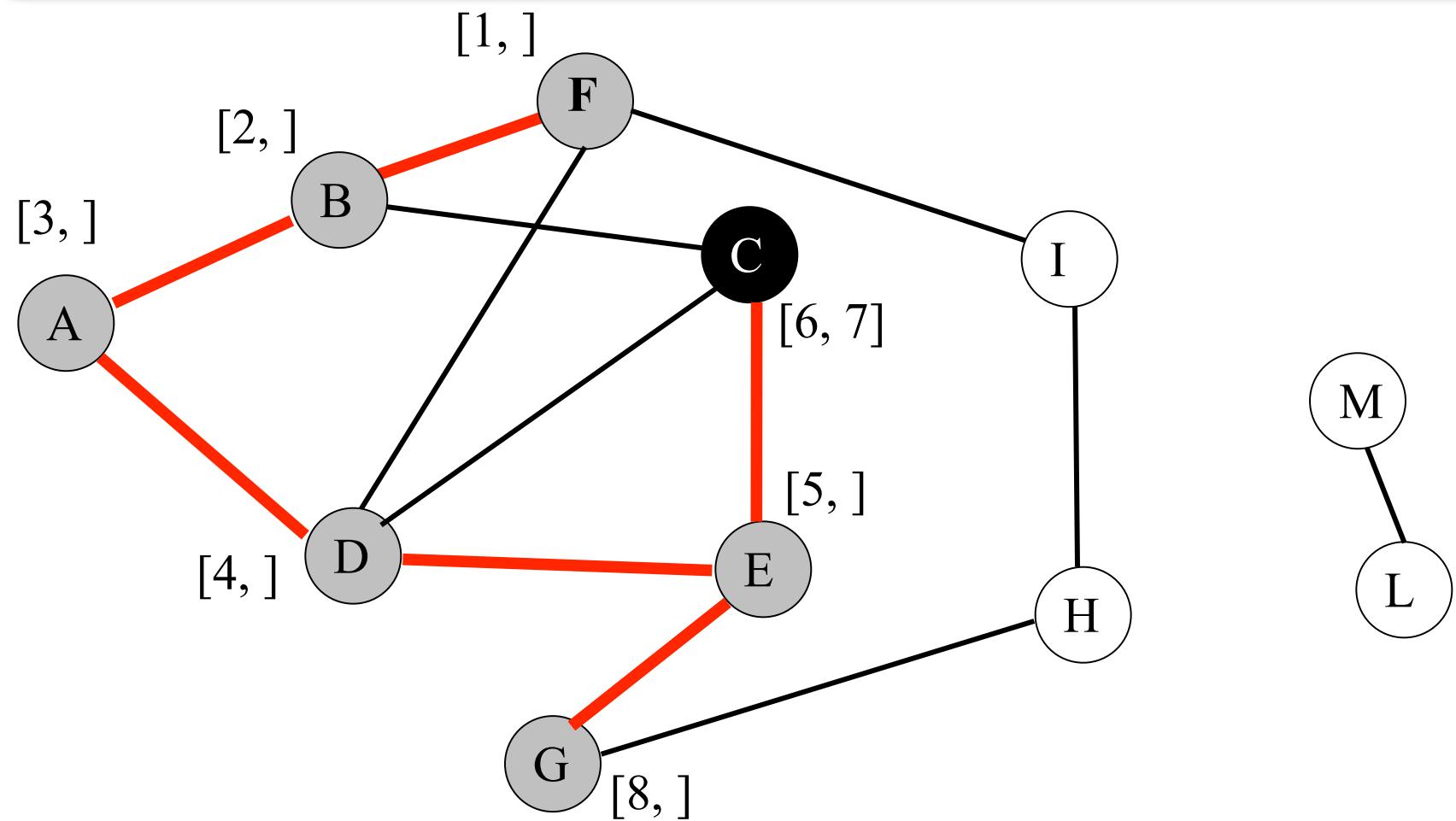
Esempio



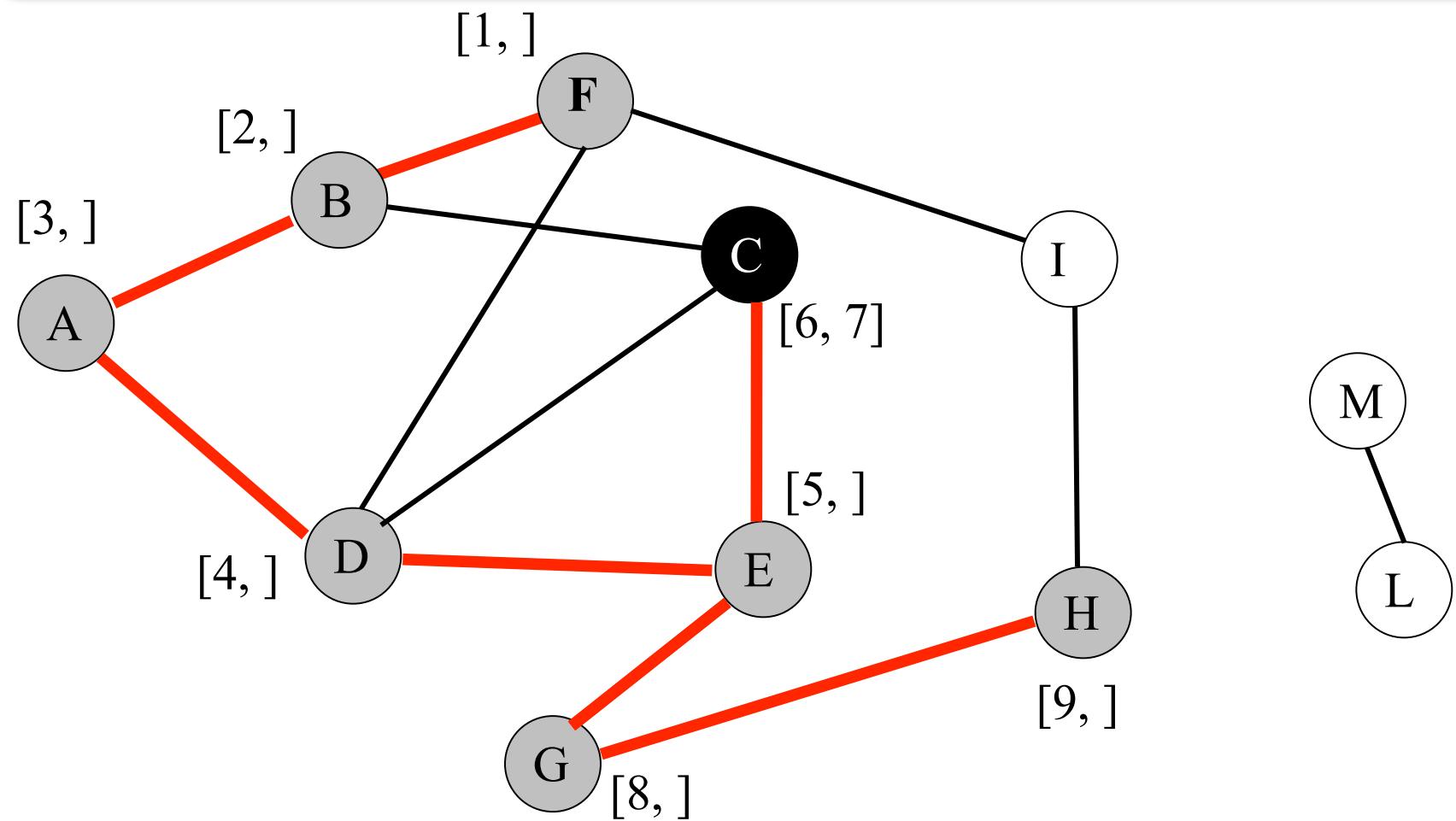
Esempio



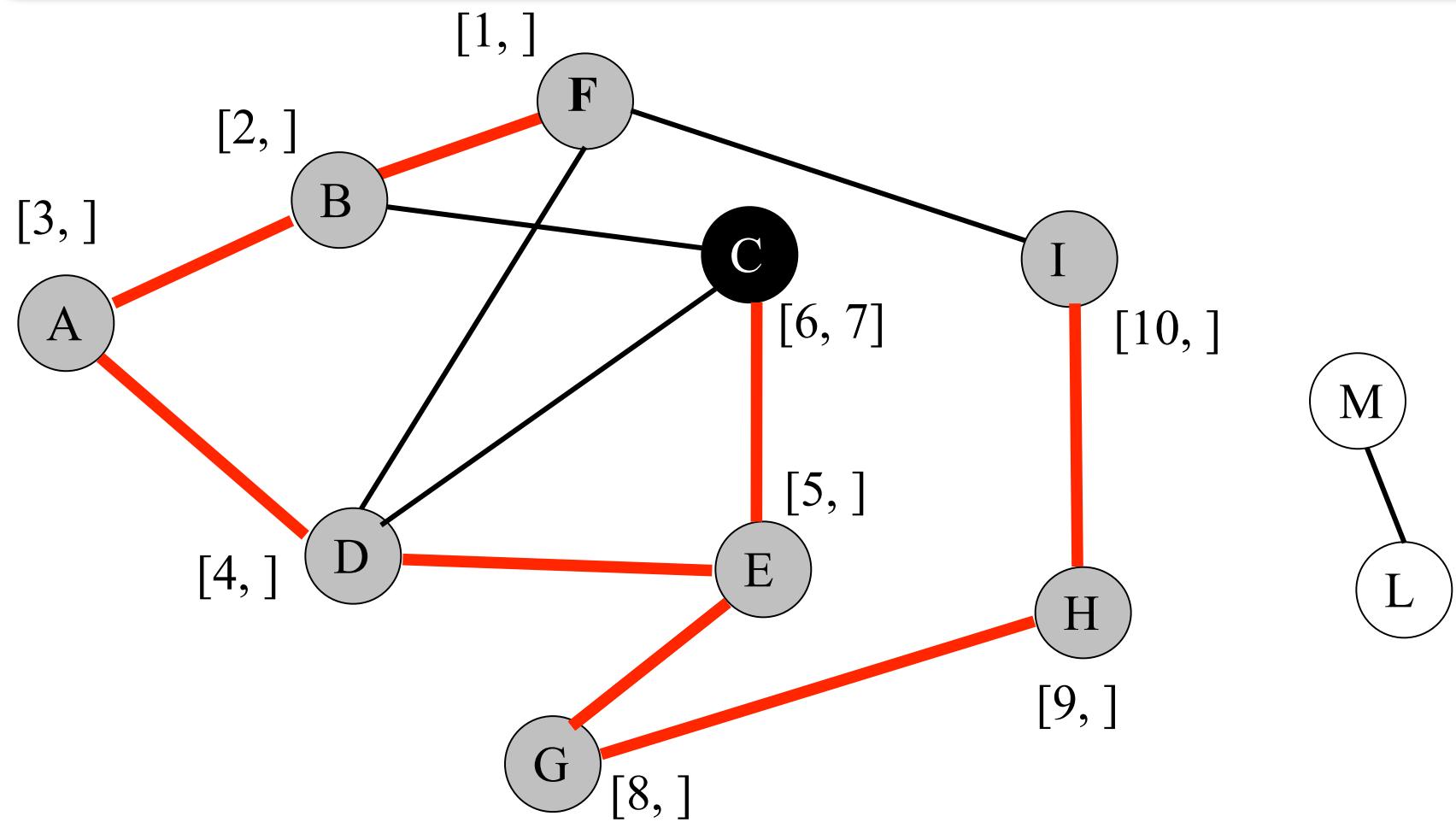
Esempio



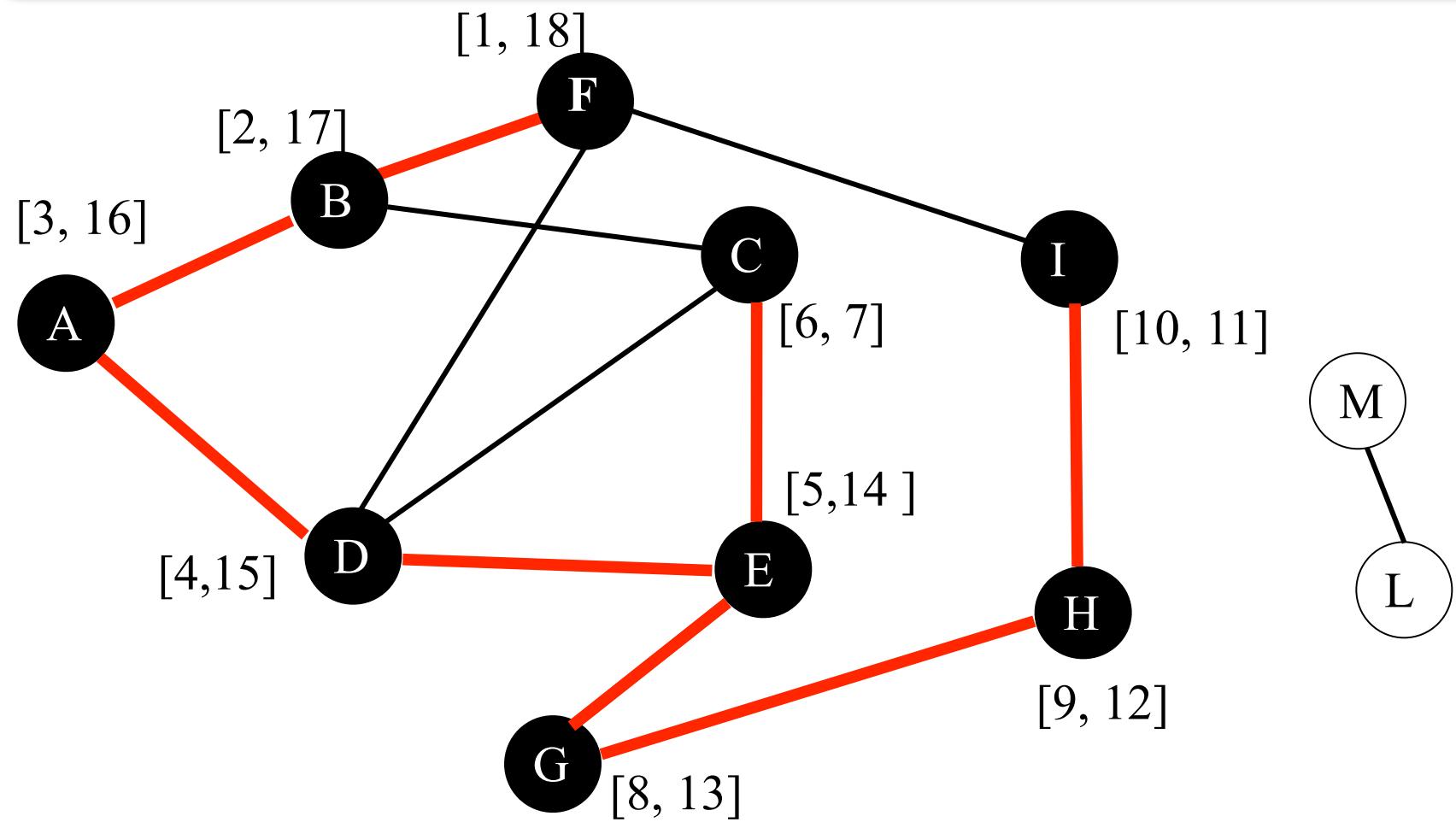
Esempio



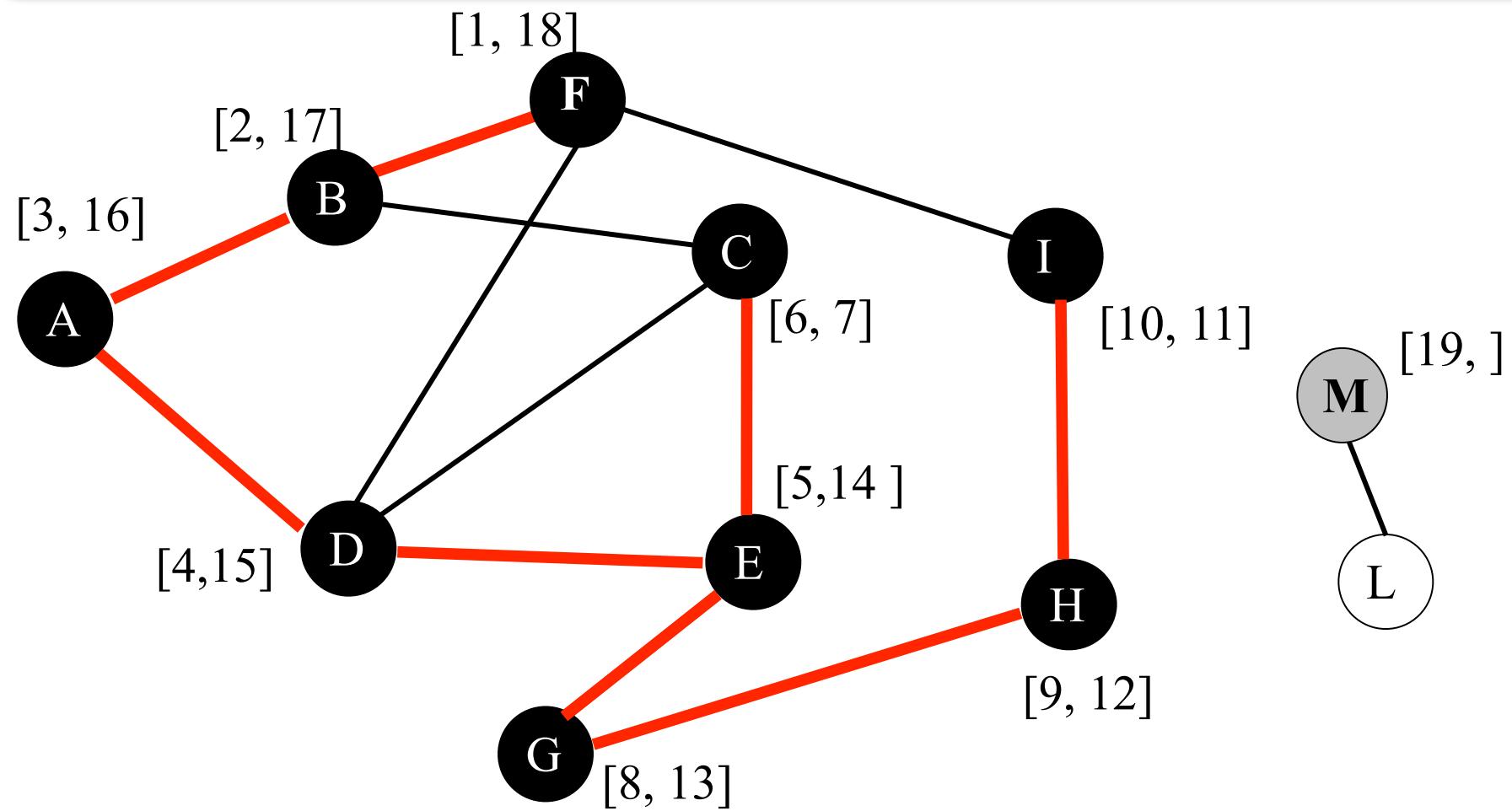
Esempio



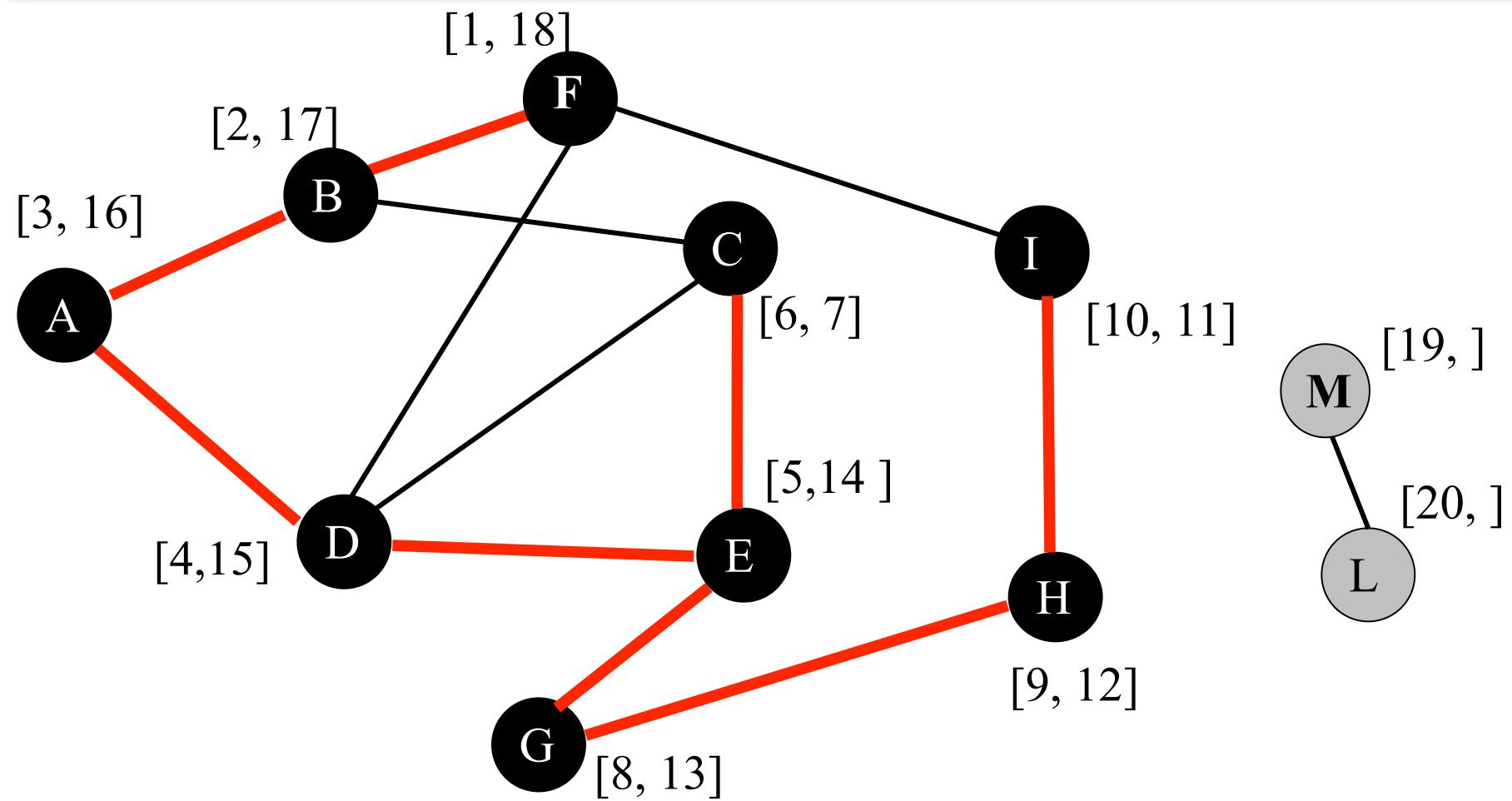
Esempio



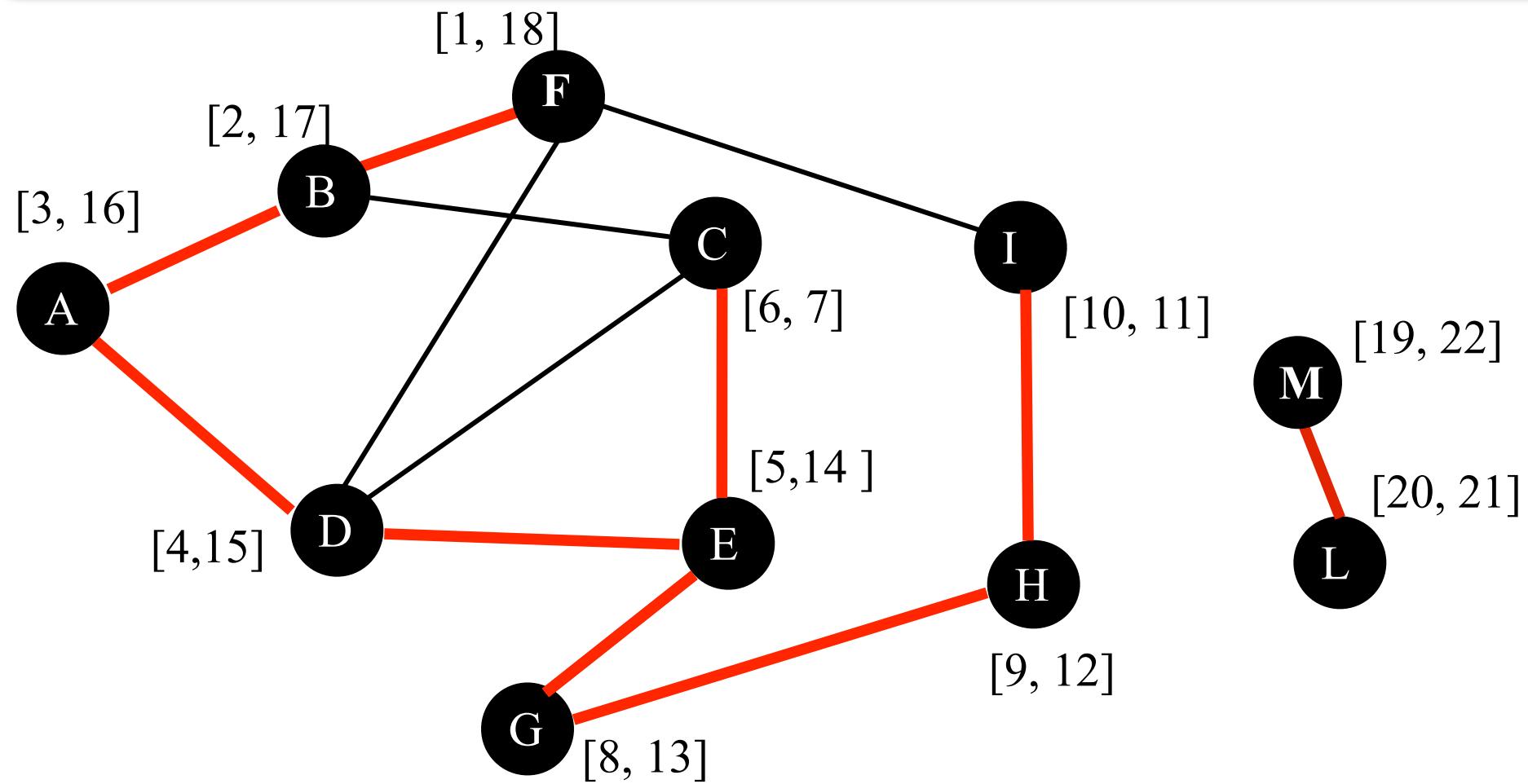
Esempio



Esempio



Esempio



DFS Schema

```
dfs-schema(GRAPH G, NODE u, int &time, int[] dt,
int[] ft)
```

time = time + 1; dt[u] = time

foreach $v \in G.\text{adj}(u)$ **do**

if $dt[v] == 0$ **then**

 { visita l'arco (u, v) (albero) }
 dfs-schema($G, v, time, dt, ft$)

else if $dt[u] > dt[v]$ **and** $ft[v] == 0$ **then**

 { visita l'arco (u, v) (indietro) }

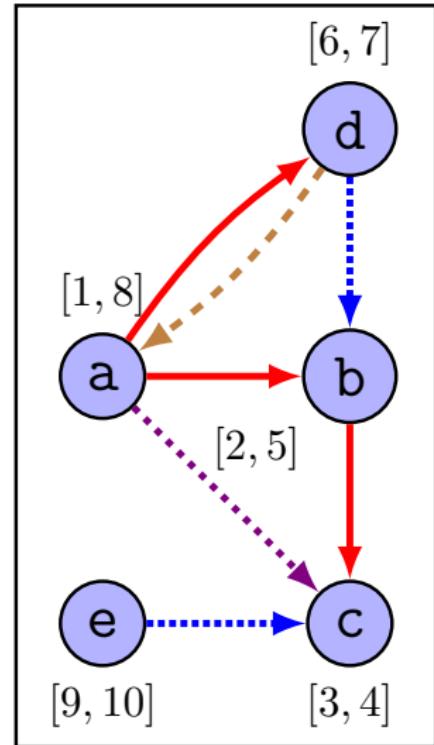
else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**

 { visita l'arco (u, v) (avanti) }

else

 { visita l'arco (u, v) (attraversamento) }

time = time + 1; ft[u] = time



Classificazione degli archi

Perchè classificare gli archi?

Possiamo dimostrare proprietà sul tipo degli archi e usare queste proprietà per costruire algoritmi migliori

Teorema

Data una visita DFS di un grafo $G = (V, E)$, per ogni coppia di nodi $u, v \in V$, solo una delle condizioni seguenti è vera:

- Gli intervalli $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ sono non-sovrapposti;
u, v non sono discendenti l'uno dell'altro nella foresta DF
- L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$;
u è un discendente di v in un albero DF
- L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$;
v è un discendente di u in un albero DF

Teoria

Teorema

Un grafo orientato è aciclico \Leftrightarrow non esistono archi all'indietro nel grafo.

Dimostrazione

- **se:** Se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e sia (v, u) un arco del ciclo. Il cammino che connette u ad v verrà prima o poi visitato, e da v verrà scoperto l'arco all'indietro (v, u) .
- **solo se:** Se esiste un arco all'indietro (u, v) , dove v è un antenato di u , allora esiste un cammino da v a u e un arco da u a v , ovvero un ciclo.

Applicazione DFS: DAG

boolean hasCycle(GRAPH G , NODE u , int &time, int[] dt, int[] ft)

$time = time + 1; \quad dt[u] = time$

foreach $v \in G.\text{adj}(u)$ **do**

if $dt[v] == 0$ **then**

if $\text{hasCycle}(G, v, time, dt, ft)$ **then**
 └ **return true**

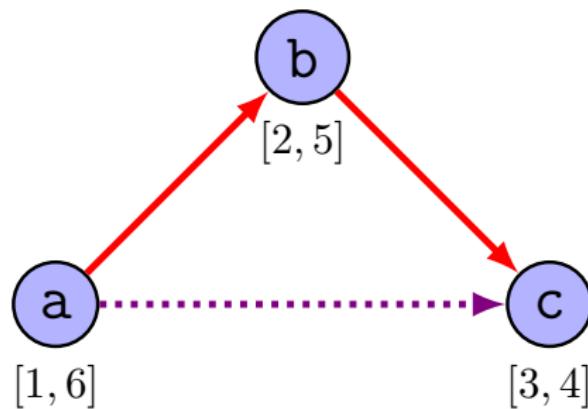
else if $dt[u] > dt[v]$ **and** $ft[v] == 0$ **then**

 └ **return true**

$time = time + 1; \quad ft[u] = time$

return false

Applicazione DFS: DAG



Arco dell'albero $dt[v] == 0$

Arco all'indietro: $dt[u] > dt[v]$ and $ft[v] = 0$

Arco in avanti: $dt[u] < dt[v]$ and $ft[v] \neq 0$

Arco attraversamento: altrimenti

Applicazione DFS: DAG

Non viene individuato nessun arco all'indietro, quindi tutte le chiamate ricorsive arriveranno al termine e ritorneranno **false**.

```
boolean hasCycle(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
```

time = time + 1; dt[u] = time

foreach $v \in G.\text{adj}(u)$ **do**

if $dt[v] == 0$ **then**

if $\text{hasCycle}(G, v, time, dt, ft)$ **then**

\sqcup **return true**

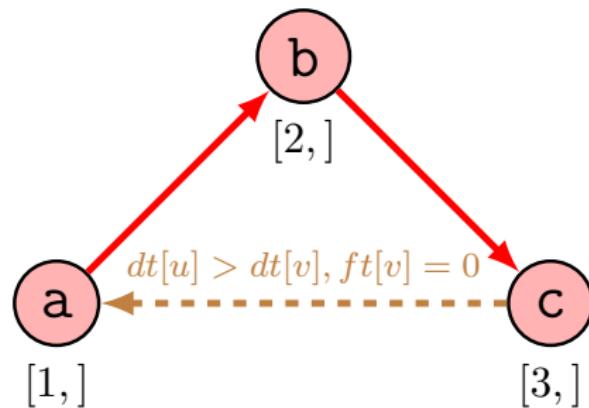
else if $dt[u] > dt[v]$ **and** $ft[v] == 0$ **then**

\sqcup **return true**

time = time + 1; ft[u] = time

return false

Applicazione DFS: DAG



Arco dell'albero $dt[v] == 0$

Arco all'indietro: $dt[u] > dt[v]$ and $ft[v] = 0$

Arco in avanti: $dt[u] < dt[v]$ and $ft[v] \neq 0$

Arco attraversamento: altrimenti

Applicazione DFS: DAG

Viene individuato un arco all'indietro, che causa la restituzione di **true** in una chiamata e la conseguente restituzione di **true** da parte di tutte le chiamate ricorsive precedenti.

boolean hasCycle(GRAPH G , NODE u , int &time, int[] dt, int[] ft)

$time = time + 1;$ $dt[u] = time$

foreach $v \in G.\text{adj}(u)$ **do**

if $dt[v] == 0$ **then**

if $\text{hasCycle}(G, v, time, dt, ft)$ **then**
 └ **return true**

else if $dt[u] > dt[v]$ **and** $ft[v] == 0$ **then**

 └ **return true**

$time = time + 1;$ $ft[u] = time$

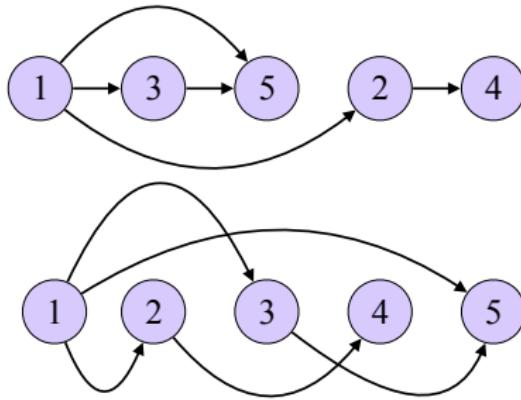
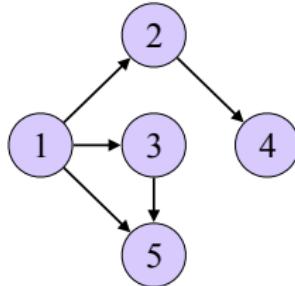
return false

Ordinamento topologico

Definizione

Dato un DAG G , un **ordinamento topologico** di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

- Esistono più ordinamenti topologici
- Se il grafo contiene un ciclo, non esiste un ordinamento topologico.



Ordinamento topologico

Problema

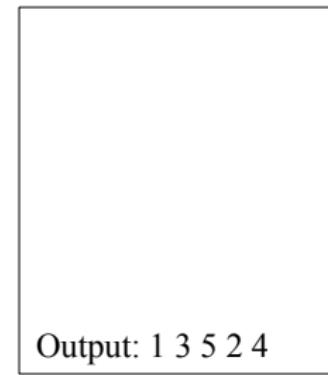
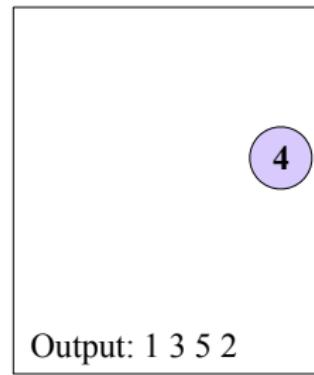
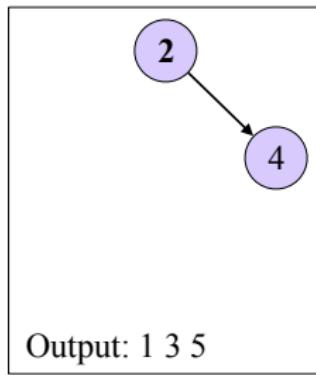
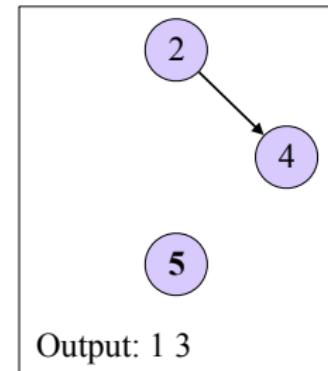
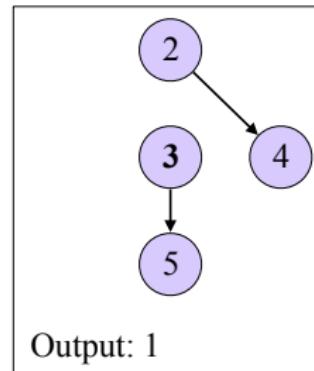
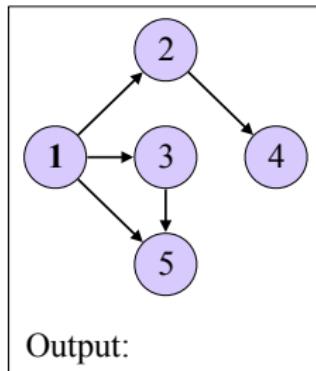
Scrivere un algoritmo che prende in input un DAG e ritorna un ordinamento topologico per esso.

Naive solution

- Trovare un nodo senza archi entranti
- Aggiungere questo nodo nell'ordinamento e rimuoverlo, insieme a tutti i suoi archi
- Ripetere questa procedura fino a quando tutti i nodi sono stati rimossi

Arthur B. Kahn. *Topological sorting of large networks*. Communications of the ACM, 5(11):558–562, 1962.

Ordinamento topologico - Algoritmi naive



Ordinamento topologico basato su DFS

Algoritmo

- Eseguire una DFS nel quale l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista, "at finish time" (post-ordine)
- Restituire la lista così ottenuta.

Output

- La sequenza dei nodi, ordinati per tempo decrescente di fine.

Perchè funziona?

- Quando un nodo è "finito", tutti i suoi discendenti sono stati scoperti e aggiunti alla lista. Aggiungendolo in testa alla lista, il nodo è in ordine corretto.

Ordinamento topologico - L'algoritmo

```
STACK topSort(GRAPH G)
```

```
STACK S = Stack()
```

```
boolean[] visited = boolean[1 ... G.size()]
```

```
foreach u ∈ G.V() do visited[u] = false
```

```
foreach u ∈ G.V() do
```

```
    if not visited[u] then
```

```
        ts-dfs(G, u, visited, S)
```

```
return S
```

```
ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
```

```
visited[u] = true
```

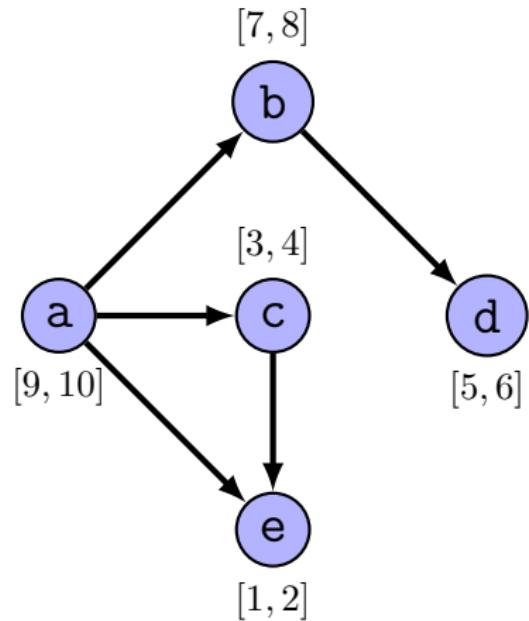
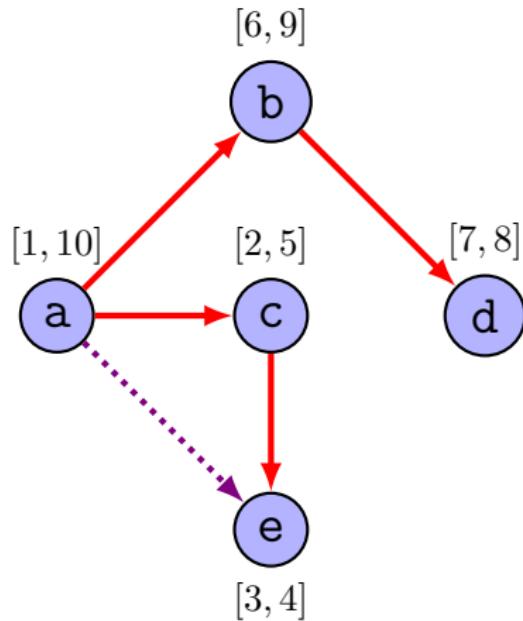
```
foreach v ∈ G.adj(u) do
```

```
    if not visited[v] then
```

```
        ts-dfs(G, v, visited, S)
```

```
S.push(u)
```

Ordinamento topologico – Esempio



Stack = { a, b, d, c, e }

Stack = { a, b, d, c, e }

Reality check

Applicazioni dell'ordinamento topologico

- Ordine di valutazione delle celle in uno spreadsheet
- Ordine di compilazione in un `Makefile`
- Risoluzione delle dipendenze nei linker
- Risoluzione delle dipendenze nei gestori di pacchetti software

Grafi e componenti fortemente connessi

Definizioni

- Un grafo orientato $G = (V, E)$ è **fortemente connesso** \Leftrightarrow ogni suo nodo è raggiungibile da ogni altro suo nodo
- Un grafo $G' = (V', E')$ è una **componente fortemente connessa** di $G \Leftrightarrow G'$ è un sottografo connesso e massimale di G

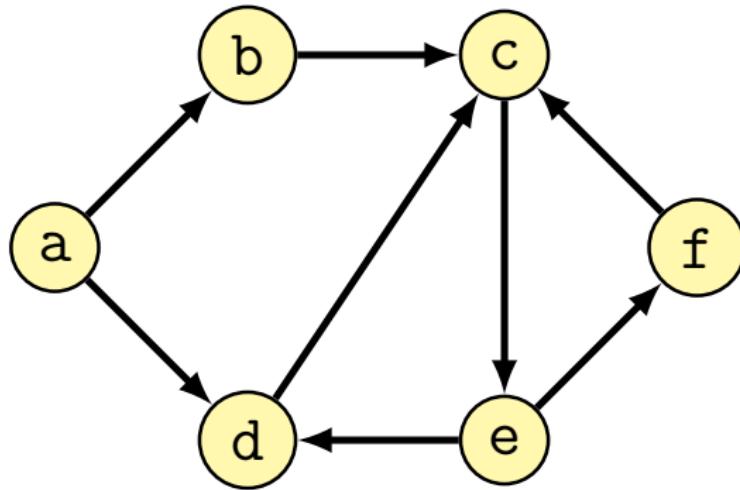
Repetita iuvant

- G' è un **sottografo** di G ($G' \subseteq G$) $\Leftrightarrow V' \subseteq V$ e $E' \subseteq E$
- G' è **massimale** \Leftrightarrow non esiste un altro sottografo G'' di G tale che:
 - G'' è connesso
 - G'' è più grande di G' (i.e. $G' \subseteq G'' \subseteq G$)

Connessione forte

Domanda

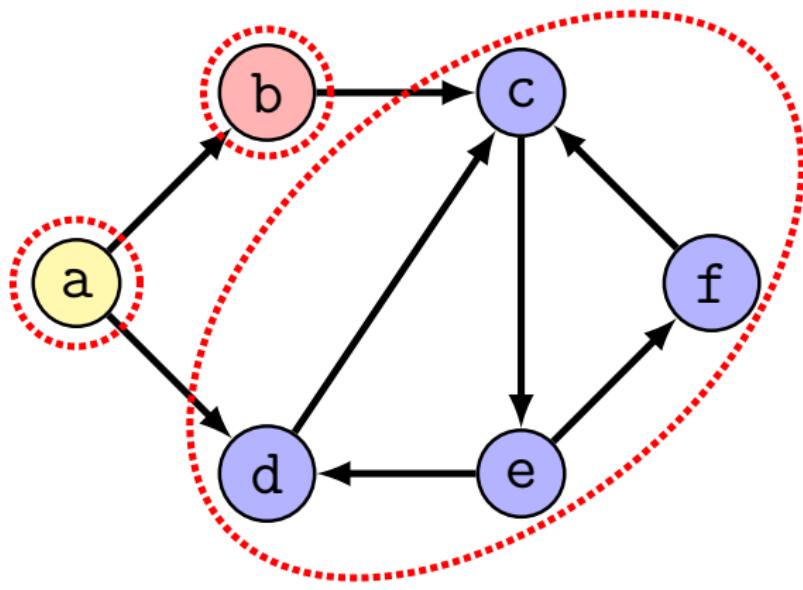
Questo grafo è fortemente connesso? No



Componenti fortemente connesse

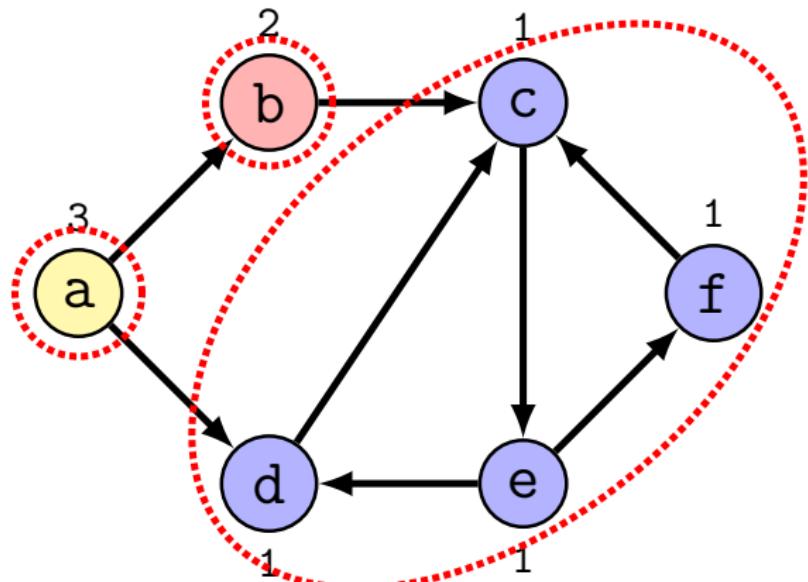
Domanda

Quali sono le componenti fortemente connesse di questo grafo?



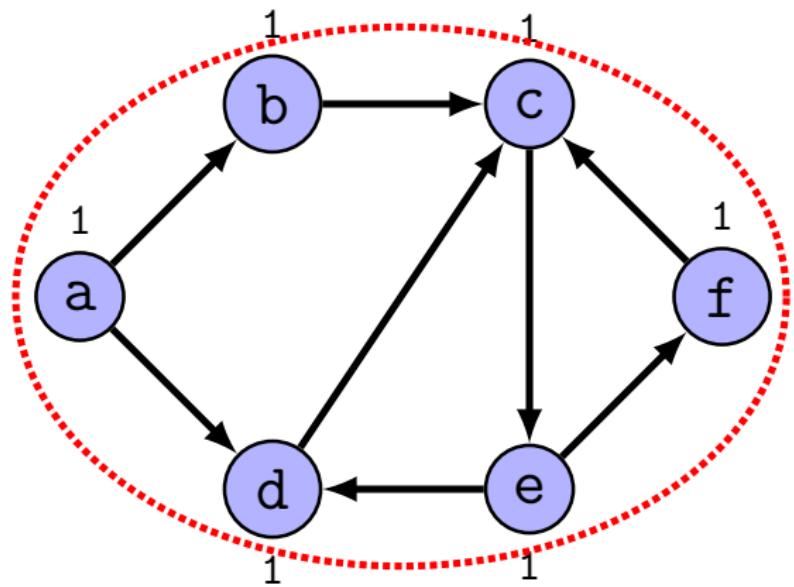
Soluzione "ingenua" (e non corretta)

- Si applica l'algoritmo `cc()` al grafo
- Purtroppo, il risultato dipende dal nodo di partenza



Soluzione "ingenua" (e non corretta)

- Si applica l'algoritmo `cc()` al grafo
- Purtroppo, il risultato dipende dal nodo di partenza



Algoritmo di Kosaraju

Kosaraju Algorithm (1978)

- Effettua una visita DFS del grafo G
- Calcola il grafo trasposto G_t
- Esegui una visita DFS sul grafo G_t utilizzando cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita
- Le componenti connesse (e i relativi alberi DF) rappresentano le componenti fortemente connesse di G

```
int[] scc(GRAPH G)
```

STACK $S = \text{topSort}(G)$

% First visit

$G^T = \text{transpose}(G)$

% Graph transposal

return cc(G^T, S)

% Second visit

Ordinamento topologico su grafi generali

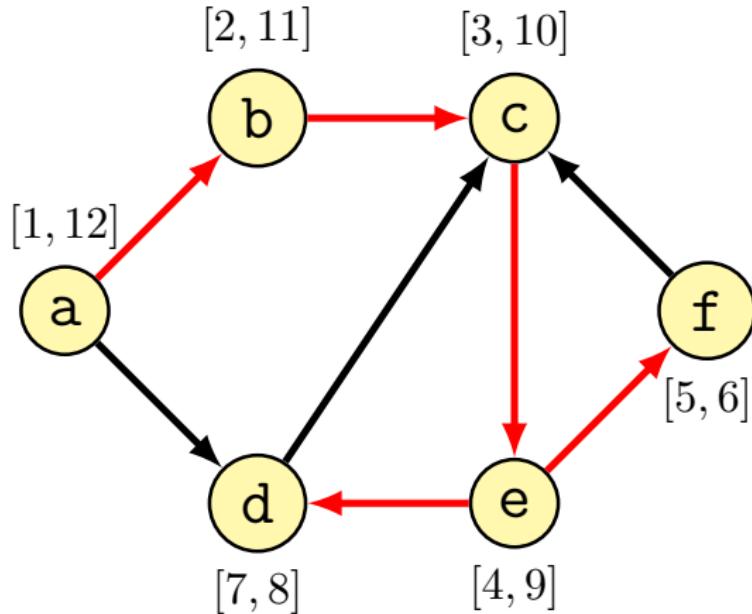
Idea generale

Applicando l'algoritmo di ordinamento topologico su un grafo generale, siamo sicuri che:

- se un arco (u, v) non appartiene ad un ciclo, allora u viene listato prima di v nella sequenza ordinata
- gli archi di un ciclo vengono listati in qualche ordine, ininfluente

Utilizziamo quindi `topsort()` per ottenere i nodi in ordine decrescente di tempo di fine

Esecuzione 1: Ordinamento topologico



Stack = { a, b, c, e, d, f }

Calcolo del grafo trasposto

Grafo trasposto (Transpose graph)

Dato un grafo orientato $G = (V, E)$, il **grafo trasposto** $G_t = (V, E_T)$ ha gli stessi nodi e gli archi orientati in senso opposto.:

$$E_T = \{(u, v) \mid (v, u) \in E\}$$

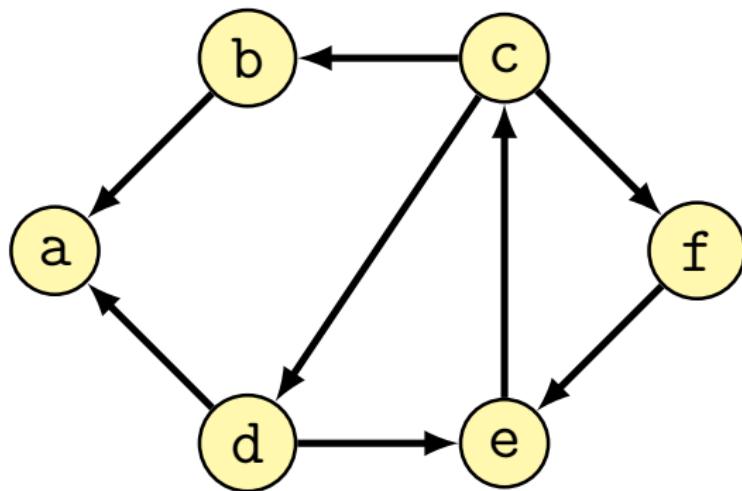
int[] transpose(GRAPH G)

GRAPH $G^T = \text{Graph}()$
foreach $u \in G.V()$ **do**
 $G^T.\text{insertNode}(u)$
foreach $u \in G.V()$ **do**
foreach $v \in G.\text{adj}(u)$ **do**
 $G^T.\text{insertEdge}(v, u)$
return G^T

Costo computazionale: $O(m+n)$

- $O(n)$ nodi aggiunti
- $O(m)$ archi aggiunti
- Ogni operazione costa $O(1)$

Esecuzione 1: Grafo trasposto



Calcolo delle componenti connesse

Invece di esaminare i nodi in ordine arbitrario, questa versione di `cc()` li esamina nell'ordine LIFO memorizzato nello stack.

`cc(GRAPH G, STACK S)`

`int[] id =
new int[1...G.size()]`

`foreach u ∈ G.V() do
 id[u] = 0`

`int counter = 0`

`while not S.isEmpty() do`

`u = S.pop()`

`if id[u] == 0 then`

`counter = counter + 1
 ccdfs(G, counter, u, id)`

`ccdfs(GRAPH G, int counter,
NODE u, int[] id)`

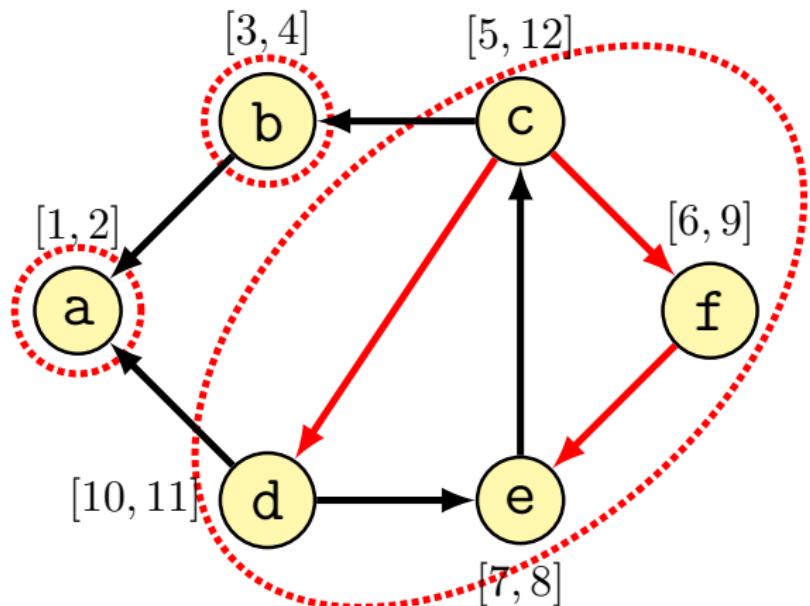
`id[u] = counter`

`foreach v ∈ G.adj(u) do`

`if id[v] == 0 then`

`ccdfs(G, counter, v, id)`

Esecuzione 1: Componenti connesse



Stack = { a, b, c, e, d, f }

SCC: The algorithm

int[] scc(GRAPH G)

STACK S = topSort(G)

% First visit

G^T = transpose(G)

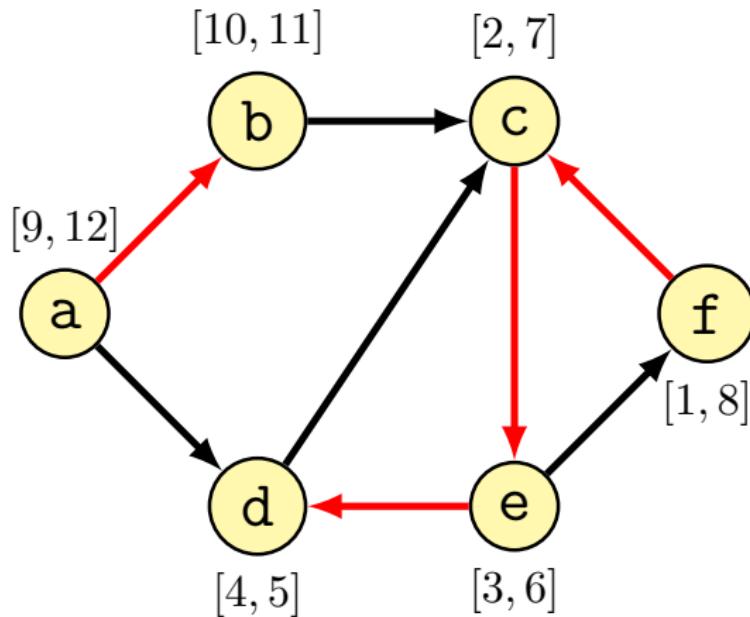
% Graph transposal

return cc(G^T, S)% Second visit

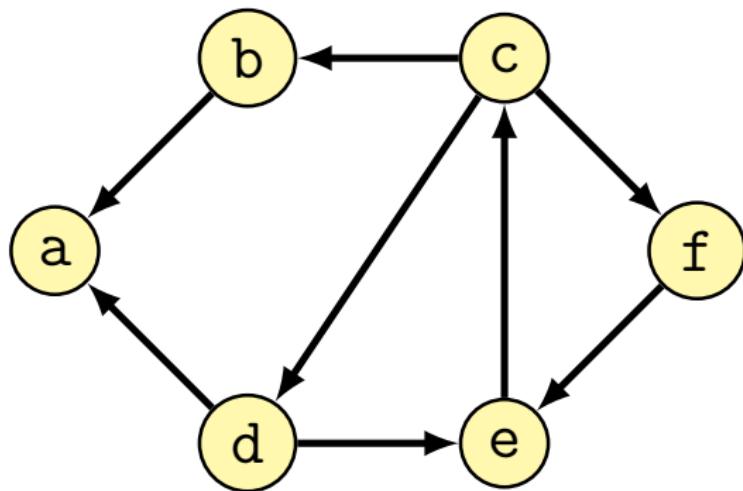
Costo computazionale: $O(m + n)$

- Ogni fase richiede $O(m + n)$

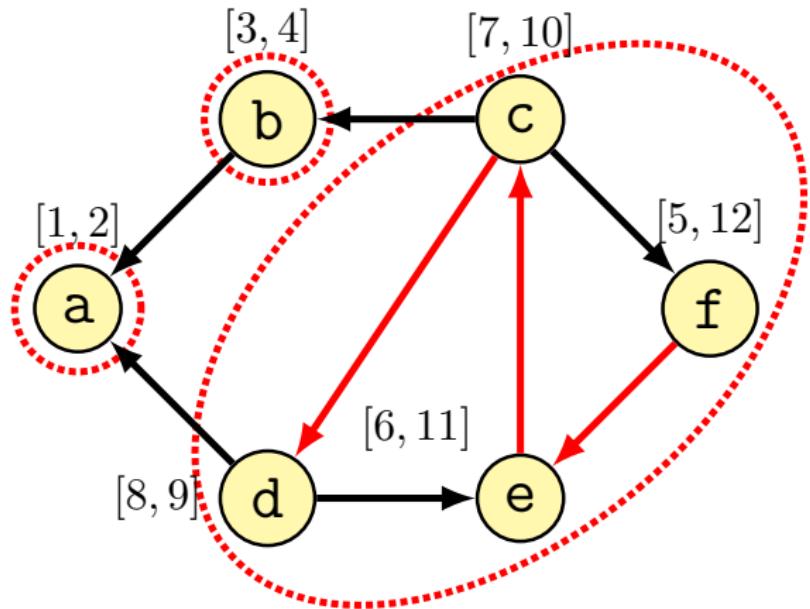
Esecuzione 2: Ordinamento topologico



Esecuzione 2: Grafo trasposto



Esecuzione 2: Componenti connesse

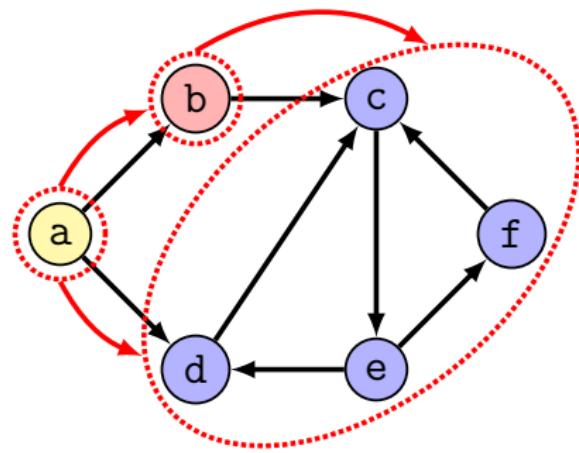


Dimostrazione di correttezza

Grafo delle componenti

$$C(G) = (V_c, E_c)$$

- $V_c = \{C_1, C_2, \dots, C_k\}$, dove C_i è la i -esima SCC of G
- $E_c = \{(C_i, C_j) | \exists (u_i, v_i) \in E : u_i \in C_i \wedge v_i \in C_j\}$



Reality check

Algoritmo di Tarjan (1972)

- Tarjan, R. E. "Depth-first search and linear graph algorithms", SIAM Journal on Computing 1(2): 146–160 (1972)
- Algoritmo con costo $O(m + n)$ come Kosaraju
- È preferito a Kosaraju in quanto necessita di una sola visita e non richiede il grafo trasposto

Applicazioni

Gli algoritmi per SCC possono essere utilizzati per risolvere il problema **2-satisfiability (2-SAT)**, un problema di soddisfacibilità booleana con clausole composte da coppie di letterali.

Conclusioni

113 Pages in category "Graph algorithms"

A

- [A* search algorithm](#)
- [Algorithmic version for Szemerédi regularity partition](#)
- [Alpha-beta pruning](#)
- [Aperiodic graph](#)

B

- [B*](#)
- [Barabási-Albert model](#)
- [Belief propagation](#)
- [Bellman-Ford algorithm](#)
- [Bianconi-Barabási model](#)
- [Bidirectional search](#)
- [Borůvka's algorithm](#)
- [Bottleneck traveling salesman problem](#)
- [Breadth-first search](#)
- [Bron-Kerbosch algorithm](#)
- [Bulky algorithm](#)

C

- [Centrality](#)
- [Chaitin's algorithm](#)
- [Christofides algorithm](#)
- [Clique percolation method](#)
- [Closure problem](#)
- [Color-coding](#)
- [Contraction hierarchies](#)
- [Courcelle's theorem](#)
- [Cuthill-McKee algorithm](#)

D

- [D*](#)
- [Degeneracy \(graph theory\)](#)
- [Depth-first search](#)
- [Dijkstra-Scholten algorithm](#)
- [Dijkstra's algorithm](#)
- [Dinic's algorithm](#)

- [Disparity filter algorithm of weighted network](#)
- [Double pushout graph rewriting](#)
- [Dulmage-Mendelsohn decomposition](#)
- [Dynamic connectivity](#)
- [Dynamic link matching](#)

E

- [Edmonds-Karp algorithm](#)
- [Edmonds' algorithm](#)
- [Blossom algorithm](#)
- [Euler tour technique](#)

F

- [FKT algorithm](#)
- [Flooding algorithm](#)
- [Floyd-Warshall algorithm](#)
- [Force-directed graph drawing](#)
- [Ford-Fulkerson algorithm](#)
- [Fringe search](#)

G

- [Girvan-Newman algorithm](#)
- [Goal node \(computer science\)](#)
- [Gomory-Hu tree](#)
- [Graph bandwidth](#)
- [Graph edit distance](#)
- [Graph embedding](#)
- [Graph isomorphism](#)
- [Graph isomorphism problem](#)
- [Graph kernel](#)
- [Graph reduction](#)
- [Graph traversal](#)

H

- [Havel-Hakimi algorithm](#)
- [Hierarchical closeness](#)
- [Hierarchical clustering of networks](#)
- [Hooper-Karp algorithm](#)

I

- [Iterative deepening A*](#)
- [Initial attractiveness](#)
- [Iterative compression](#)
- [Iterative deepening depth-first search](#)

J

- [Johnson's algorithm](#)
- [Journal of Graph Algorithms and Applications](#)
- [Jump point search](#)
- [Junction tree algorithm](#)

K

- [K shortest path routing](#)
- [Karger's algorithm](#)
- [Kleinman-Wang algorithms](#)
- [Knight's tour](#)
- [Knuth's Simpath algorithm](#)
- [Kosaraju's algorithm](#)
- [Kruskal's algorithm](#)

L

- [Lexicographic breadth-first search](#)
- [Longest path problem](#)

M

- [MaxCliqueDyn maximum clique algorithm](#)
- [Minimax](#)
- [Minimum bottleneck spanning tree](#)
- [Misra & Gries edge coloring algorithm](#)

N

- [Nearest neighbour algorithm](#)
- [Network flow problem](#)
- [Network simplex algorithm](#)
- [Nonblocking minimal spanning switch](#)

P

- [PageRank](#)

- [Parallel all-pairs shortest path algorithm](#)
- [Path-based strong component algorithm](#)
- [Pre-topological order](#)
- [Prim's algorithm](#)
- [Proof-number search](#)
- [Push-relabel maximum flow algorithm](#)

R

- [Reverse-delete algorithm](#)
- [Rocha-Thatte cycle detection algorithm](#)

S

- [Sethi-Ullman algorithm](#)
- [Shortest Path Faster Algorithm](#)
- [SMA*](#)
- [Spectral layout](#)
- [Spreading activation](#)
- [Stoe-Wagner algorithm](#)
- [Subgraph isomorphism problem](#)
- [Saurabh's algorithm](#)

T

- [Tarjan's off-line lowest common ancestors algorithm](#)
- [Tarjan's strongly connected components algorithm](#)
- [Theta*](#)
- [Topological sorting](#)
- [Transitive closure](#)
- [Transitive reduction](#)
- [Travelling salesman problem](#)
- [Tree traversal](#)

W

- [Widest path problem](#)
- [Wiener connector](#)

Y

- [Yen's algorithm](#)

Classificazione degli archi

dfs-schema(GRAPH G , NODE u)

esamina il nodo u prima (caso *pre-visita*)

$time \leftarrow time + 1; dt[u] \leftarrow time$

foreach $v \in G.\text{adj}(u)$ **do**

 esamina l'arco (u, v) di qualsiasi tipo

if $dt[v] = 0$ **then**

 | esamina l'arco (u, v) in T

 | dfs-schema(g, v)

else if $dt[u] > dt[v]$ **and** $ft[v] = 0$ **then**

 | esamina l'arco (u, v) all'indietro

else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**

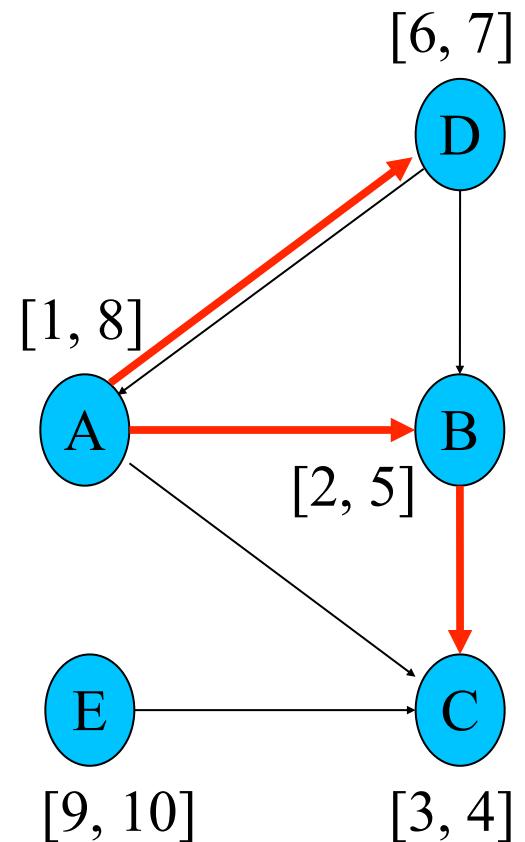
 | esamina l'arco (u, v) in avanti

else

 | esamina l'arco (u, v) di attraversamento

esamina il nodo u dopo (caso *post-visita*)

$time \leftarrow time + 1; ft[u] \leftarrow time$



Classificazione degli archi

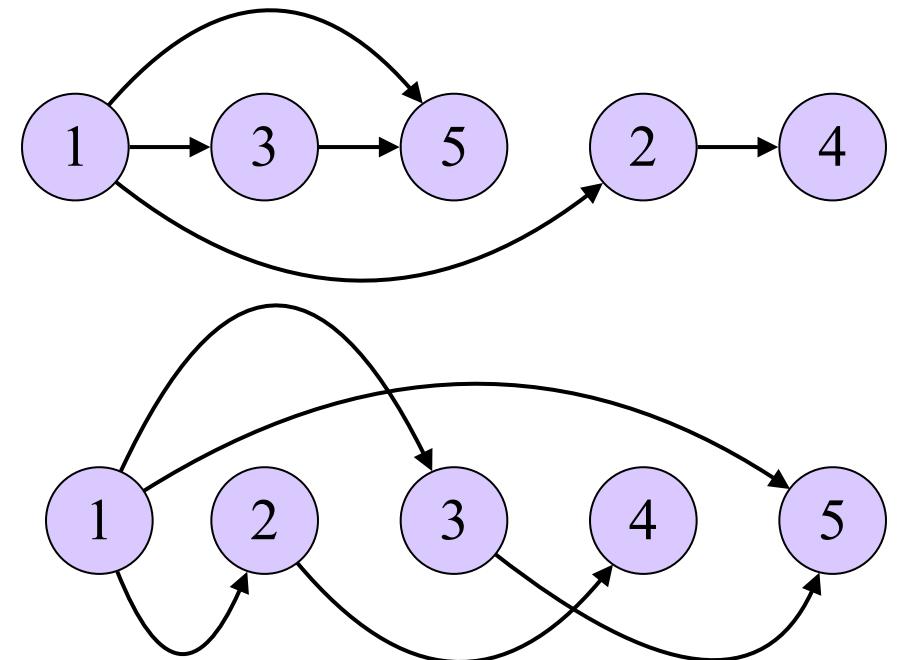
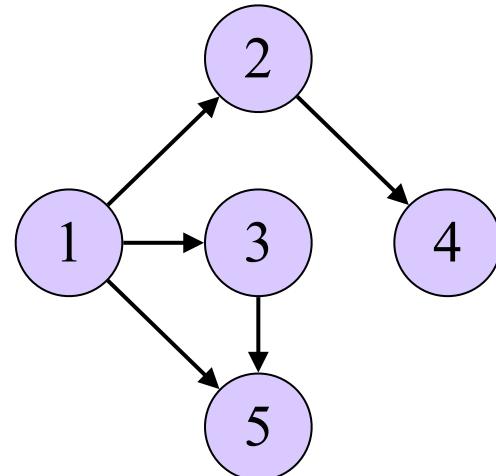
- ◆ **Cosa serve le classificazione?**
 - ◆ E' possibile dimostrare alcune proprietà, che poi possono essere sfruttate negli algoritmi
- ◆ **Esempio:**
 - ◆ DAG non hanno archi all'indietro (dimostrare)

boolean ciclico(GRAPH G , NODE u)

```
time ← time + 1; dt[u] ← time
foreach  $v \in G.\text{adj}(u)$  do
    if  $dt[v] = 0$  then
        | if ciclico( $G, v$ ) then return true
        | else if  $dt[u] > dt[v]$  and  $ft[v] = 0$  then
            |   return true
time ← time + 1; ft[u] ← time
return false;
```

Ordinamento topologico

- Dato un DAG G (direct acyclic graph), un ordinamento topologico su G è un ordinamento lineare dei suoi vertici tale per cui:
 - se G contiene l'arco (u, v) , allora u compare prima di v nell'ordinamento
 - Per transitività, ne consegue che se v è raggiungibile da u , allora u compare prima di v nell'ordinamento
 - Nota: possono esserci più ordinamenti topologici



Ordinamento topologico

♦ Problema:

- ♦ Fornire un algoritmo che dato un grafo orientato aciclico, ritorni un ordinamento topologico

♦ Soluzioni

- ♦ Diretta

Trovare ogni vertice che non ha alcun arco incidente in ingresso

Stampare questo vertice e rimuoverlo, insieme ai suoi archi

Ripetere la procedura finché tutti i vertici risultano rimossi

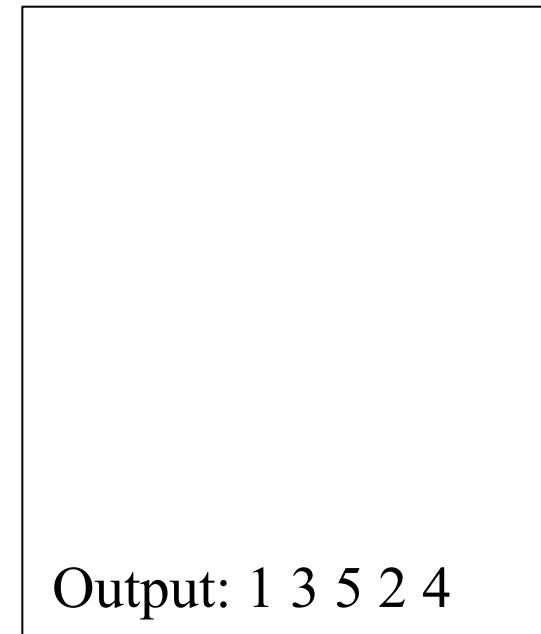
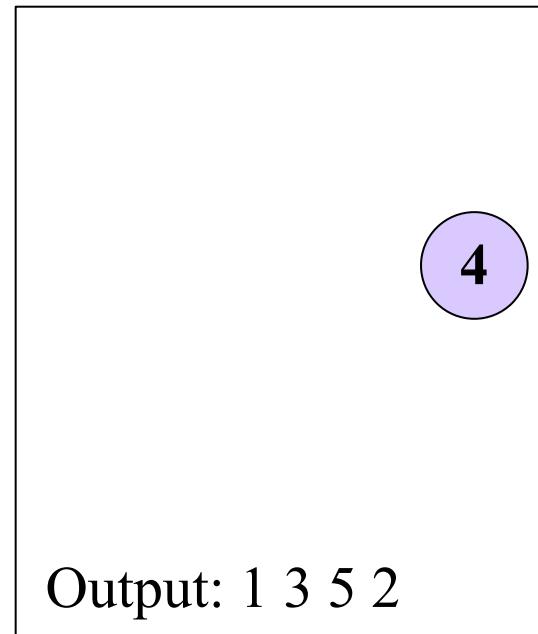
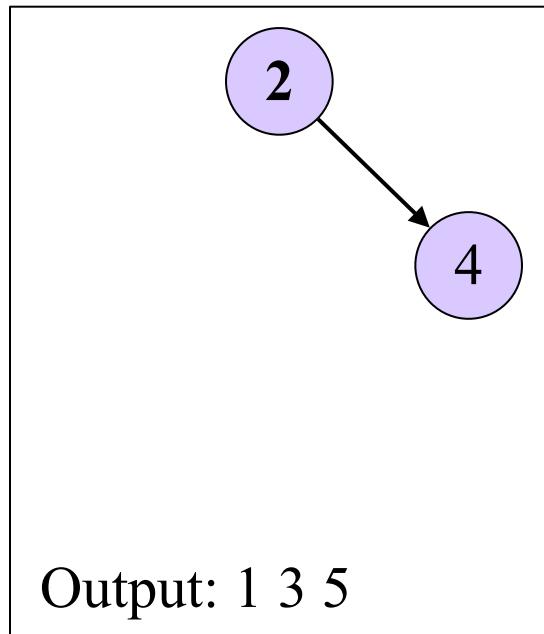
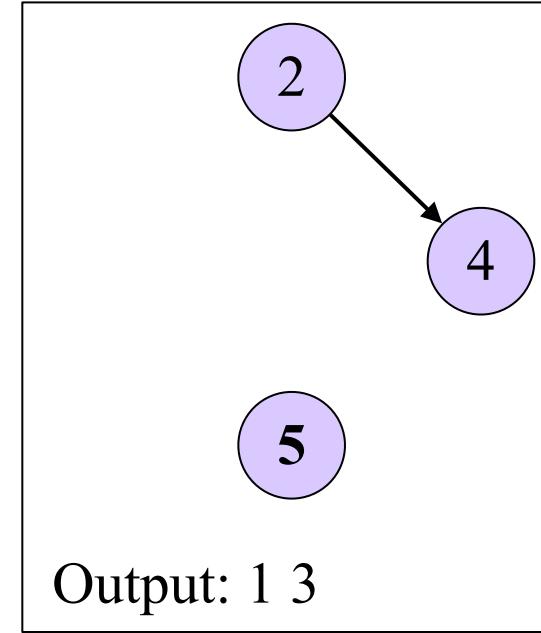
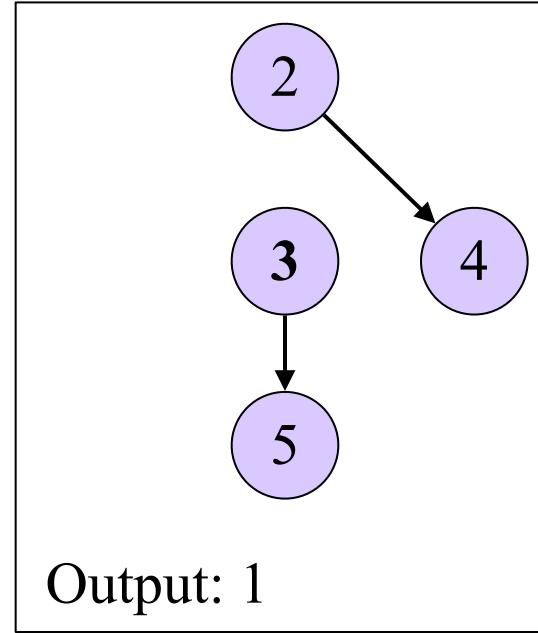
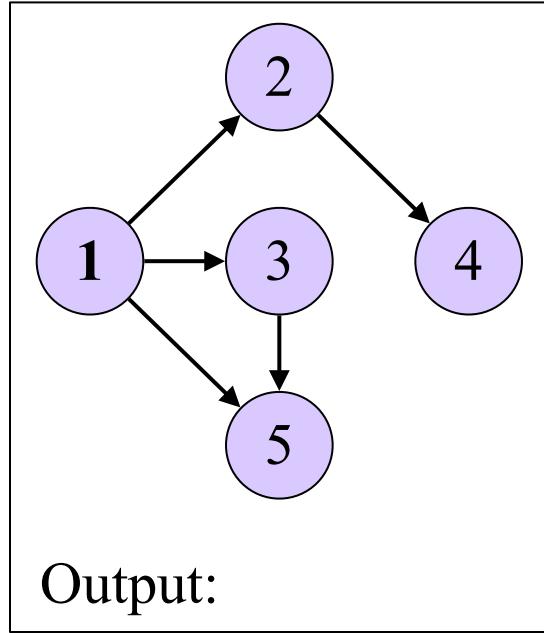
- ♦ Basata su DFS

Esercizio: scrivere lo pseudocodice per questo algoritmo

Qual è la complessità?

- con matrici di adiacenza
- con liste di adiacenza

Soluzione diretta



Ordinamento topologico basato su DFS

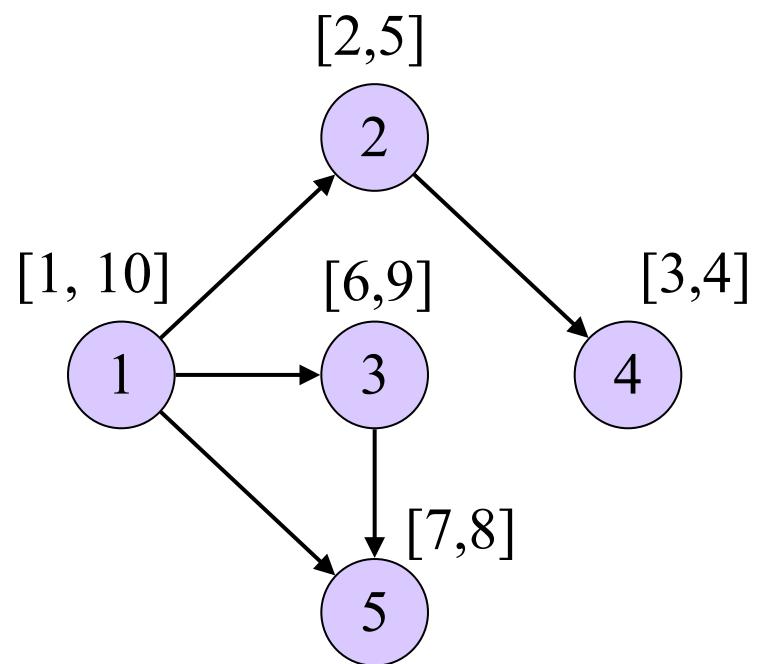
♦ Algoritmo

- ♦ Si effettua una DFS
- ♦ L'operazione di visita consiste nell'aggiungere il vertice alla testa di una lista "at finish time"
- ♦ Restituire la lista di vertici

♦ Output

- ♦ Sequenza ordinata di vertici, in ordine inverso di finish time

♦ Perché funziona?



Ordinamento topologico basato su DFS

```
topSort(GRAPH  $G$ , integer [ ]  $ordine$ )
```

```
    boolean[ ]  $visitato \leftarrow \text{boolean}[1 \dots G.n]$ 
    foreach  $u \in G.V()$  do  $visitato[u] \leftarrow \text{false}$ 
    integer  $i \leftarrow 1$ 
    foreach  $u \in G.V()$  do
        if not  $visitato[u]$  then
             $i \leftarrow \text{ts-dfs}(G, u, i, visitato, ordine)$ 
```

```
integer ts-dfs(GRAPH  $G$ , NODE  $u$ , integer  $i$ , boolean[ ]  $visitato$ , NODE[ ]  $ordine$ )
     $visitato[u] \leftarrow \text{true}$ 
    foreach  $v \in G.\text{adj}(u)$  do
        if not  $visitato[v]$  then
             $i \leftarrow \text{ts-dfs}(G, v, i, visitato, ordine)$ 
     $ordine[G.n - i + 1] \leftarrow u$ 
    return  $i + 1$ 
```

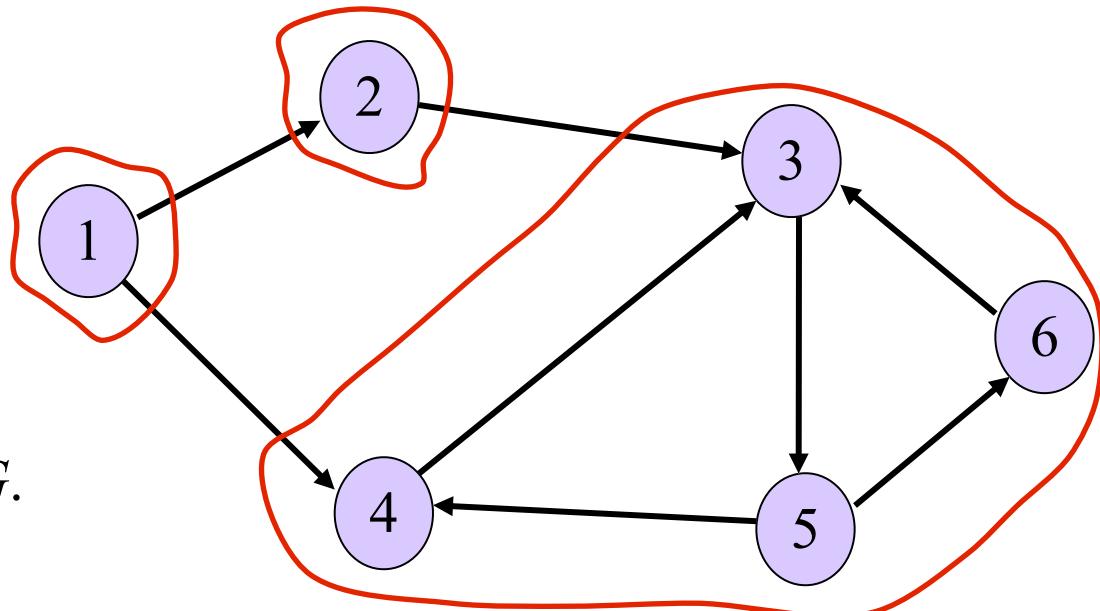
Definizioni: Grafi fortemente connessi e componenti fortemente connesse

♦ In un grafo orientato G

- ♦ G è *fortemente connesso* \Leftrightarrow esiste un cammino da ogni vertice ad ogni altro vertice
- ♦ Un grafo $G' = (V', E')$ è una *componente fortemente connessa* di $G \Leftrightarrow$ è un sottografo di G fortemente connesso e massimale

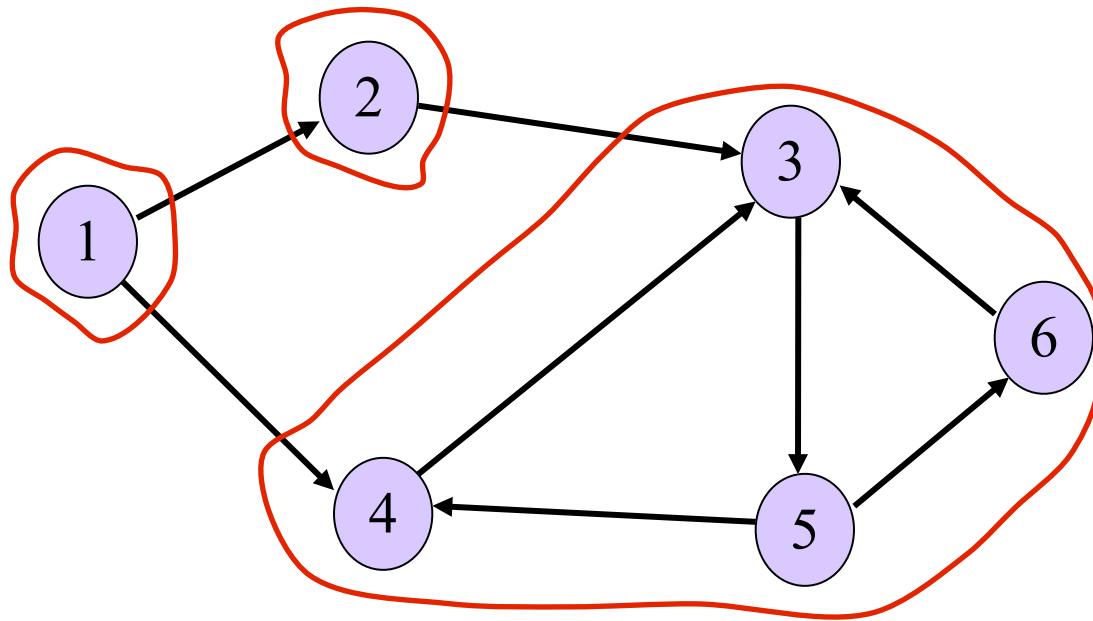
♦ Definizioni

- ♦ G' è un *sottografo* di G ($G' \subseteq G$) se e solo se $V' \subseteq V$ e $E' \subseteq E$
- ♦ G' è *massimale* \Leftrightarrow non esiste un sottografo G'' di G che sia fortemente connesso e “più grande” di G' , ovvero tale per cui $G' \subseteq G'' \subseteq G$.



Soluzione errata

- ♦ Proviamo ad utilizzare l'algoritmo per le componenti connesse?
 - ♦ Risultati variano a seconda del nodo da cui si parte



Componenti fortemente connesse

- ♦ **Algoritmo (Kosaraju, 1978)**

1. Effettua una DFS di G
2. Calcola il grafo trasposto G^T
3. Effettua una DFS di G^T esaminando i vertici in ordine inverso di tempo di fine
4. Fornisci i vertici di ogni albero della foresta depth-first prodotta al passo 3. come una diversa SCC

- ♦ **Grafo trasposto**

- ♦ Dato un grafo $G = (V, E)$, il grafo trasposto $G^T = (V, E^T)$ è formato dagli stessi nodi, mentre gli archi hanno direzioni invertite: i.e, $E^T = \{(u,v) \mid (v,u) \in E\}$

Componenti fortemente connesse - algoritmo

integer[] scc(GRAPH G)

STACK $S \leftarrow \text{Stack}()$ % Prima visita

boolean[] $\text{visitato} \leftarrow \text{new boolean}[1 \dots G.n]$

foreach $u \in G.V()$ **do** $\text{visitato}[u] \leftarrow \text{false}$

foreach $u \in G.V()$ **do** $\text{dfsStack}(G, \text{visitato}, S, u)$

GRAPH $G^T \leftarrow \text{Graph}()$ % Calcolo grafo trasposto

foreach $u \in G.V()$ **do** $G^T.\text{insertNode}(u)$

foreach $u \in G.V()$ **do**

foreach $v \in G.\text{adj}(u)$ **do**

$G^T.\text{insertEdge}(v, u)$

integer[] $\text{ordine} \leftarrow \text{new integer}[1 \dots G.n]$ % Seconda visita

for integer $i \leftarrow 1$ **to** n **do** $\text{ordine}[i] \leftarrow S.\text{pop}()$

return $\text{cc}(G^T, \text{ordine})$

Componenti fortemente connesse - algoritmo

dfsStack(GRAPH G , boolean [] $visitato$, STACK S , NODE u)

```
visitato[ $u$ ]  $\leftarrow$  true
foreach  $v \in G.\text{adj}(u)$  do
    if not visitato[ $v$ ] then
        dfsStack( $G, visitato, S, v$ )
    S.push( $u$ )
```

Componenti fortemente connesse - dimostrazione correttezza

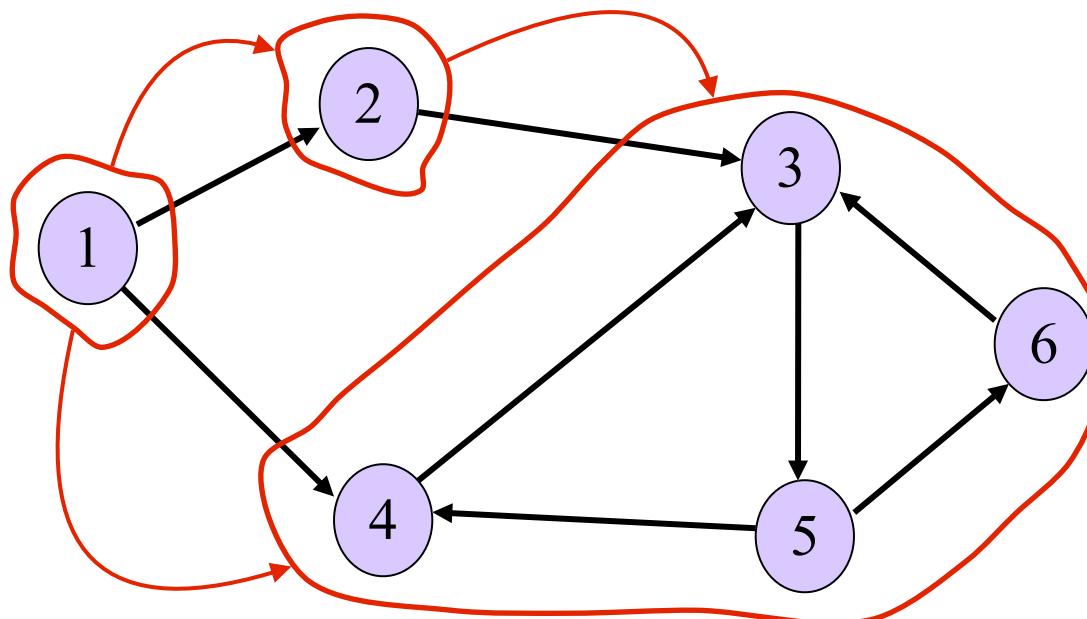
♦ Domanda

- ♦ Che rapporto c'è fra le SCC del grafo G e del suo trasposto G^T

♦ Grafo delle componenti $G^c = (V^c, E^c)$

- ♦ $V^c = \{C_1, C_2, \dots, C_k\}$, dove C_i è l' i -esima componente连通的 di G ;
- ♦ $E^c = \{(C_i, C_j) : \exists (u_i, u_j) \in E : u_i \in C_i, u_j \in C_j\}$

Grafo delle componenti
è aciclico?



Componenti fortemente connesse - dimostrazione correttezza

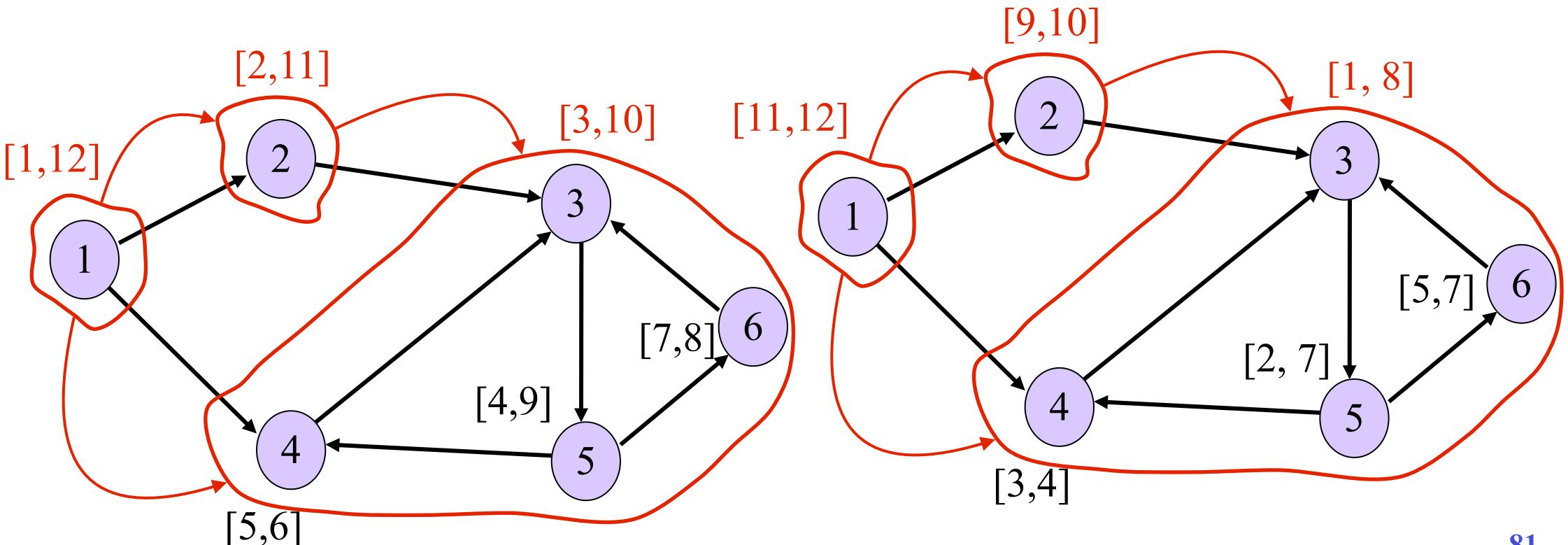
- ♦ Si può estendere il concetto di dt e ft al grafo delle componenti

$$dt(C) = \min\{dt[u] : u \in C\}$$

$$ft(C) = \max\{ft[u] : u \in C\}$$

- ♦ Teorema

- ♦ Siano C e C' due componenti distinte nel grafo orientato $G = (V, E)$. Supponiamo che esista un arco $(C, C') \in E^c$. Allora $ft(C) > ft(C')$



Componenti fortemente connesse - dimostrazione correttezza

- ♦ Si può estendere il concetto di dt e ft al grafo delle componenti

$$dt(C) = \min\{dt[u] : u \in C\}$$

$$ft(C) = \max\{ft[u] : u \in C\}$$

- ♦ **Teorema**

- ♦ Siano C e C' due componenti distinte nel grafo orientato $G = (V, E)$. Supponiamo che esista un arco $(C, C') \in E^c$. Allora $ft(C) > ft(C')$

