Palestra di Algoritmi

Olimpiadi Italiane di Informatica

Parte del contenuto di queste slide è basato su materiale del prof. Alberto Montresor (UNITN) https://cricca.disi.unitn.it/montresor/teaching/asd/materiale/lucidi/

Liceo Galilei - Trento

#7 - 27/01/2022

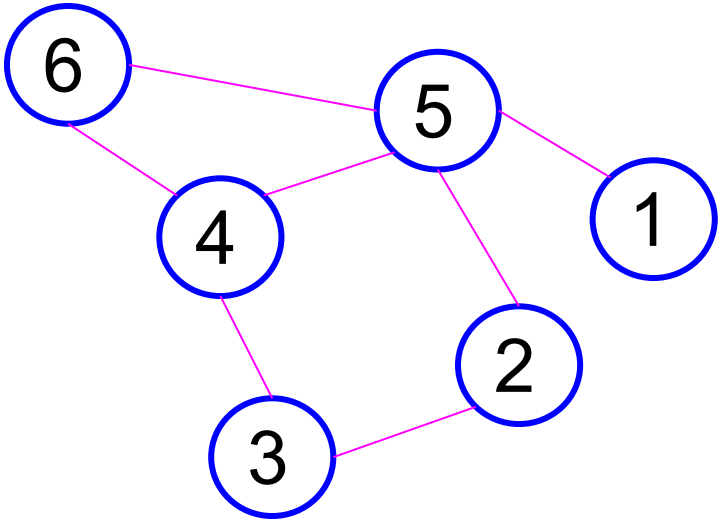
0. Calendario

Prossime lezioni: ONLINE 15-17

- ➔ #8 giovedì 3 febbraio 2022
- ➔ OII martedì 8 febbraio 2022

Siete pronti? Partiamo!

Che cosa sono i GRAFI?



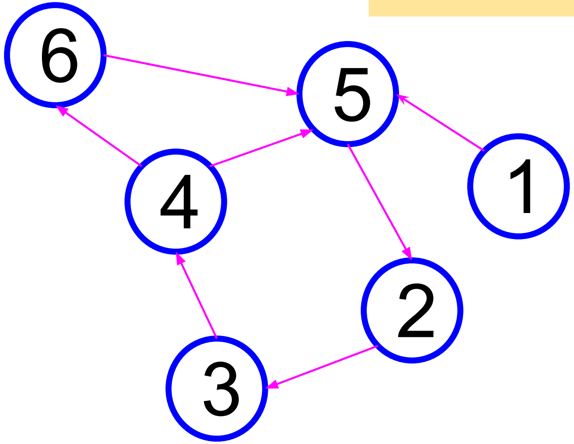
Un grafo è una **struttura dati** composta di **nodi** (o vertici) e **archi**

G = (V,E) V → Vertexes, E → Edges

Gli archi possono essere:

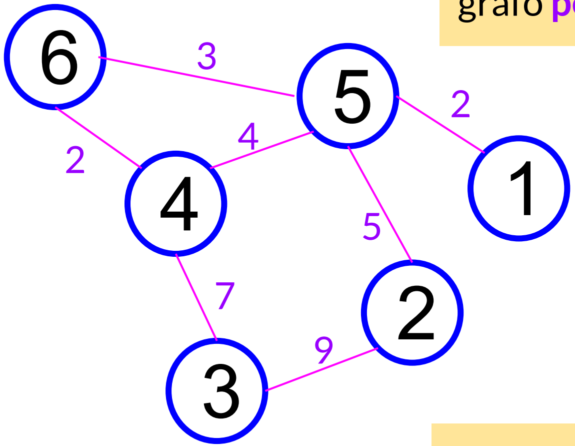
- *orientati*
- non *orientati*
- *pesati*
- non *pesati*
- *connessi*
- non *connessi*

grafo **orientato**

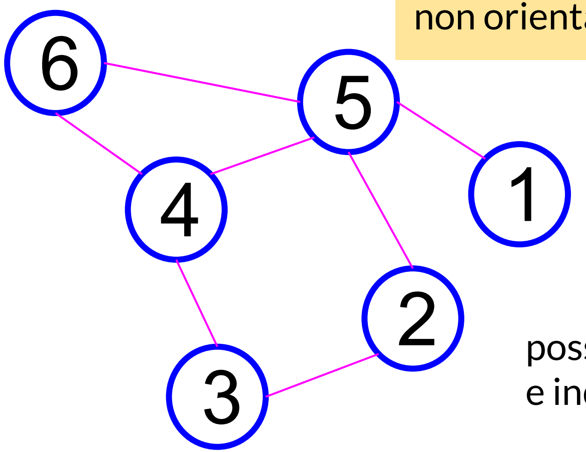


le frecce indicano la direzione nella quale posso muovermi

grafo **pesato**

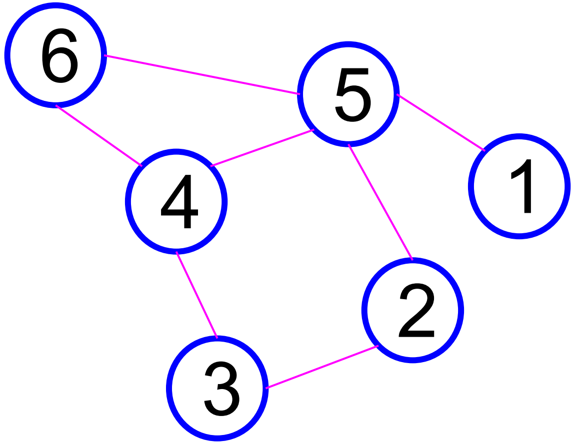


non orientato

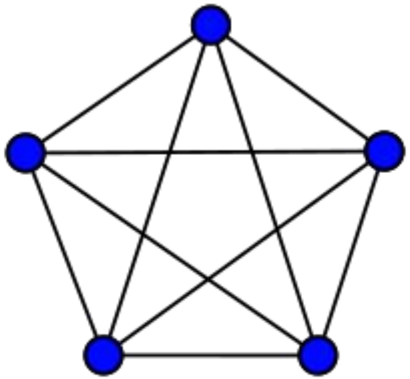


posso andare “avanti e indietro”

non pesato

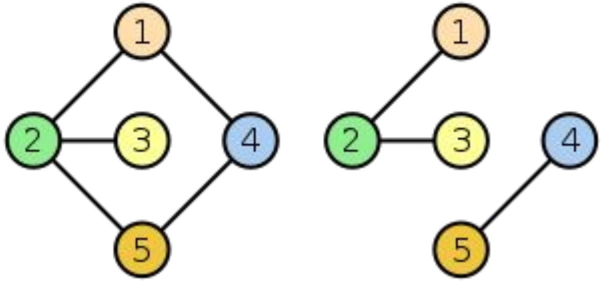


grafo completo



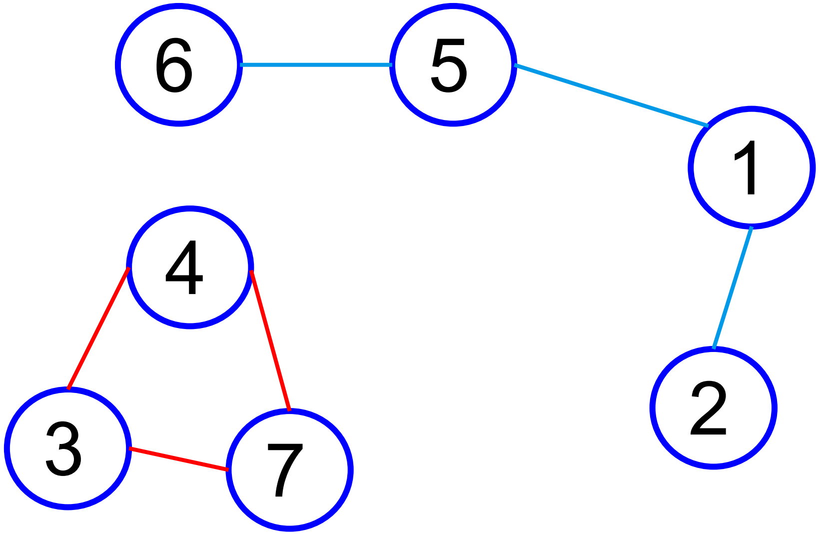
ogni nodo è
connesso ad ogni
altro nodo

grafo connesso



un grafo si dice connesso
se da ciascun nodo è
possibile raggiungere
tutti gli altri

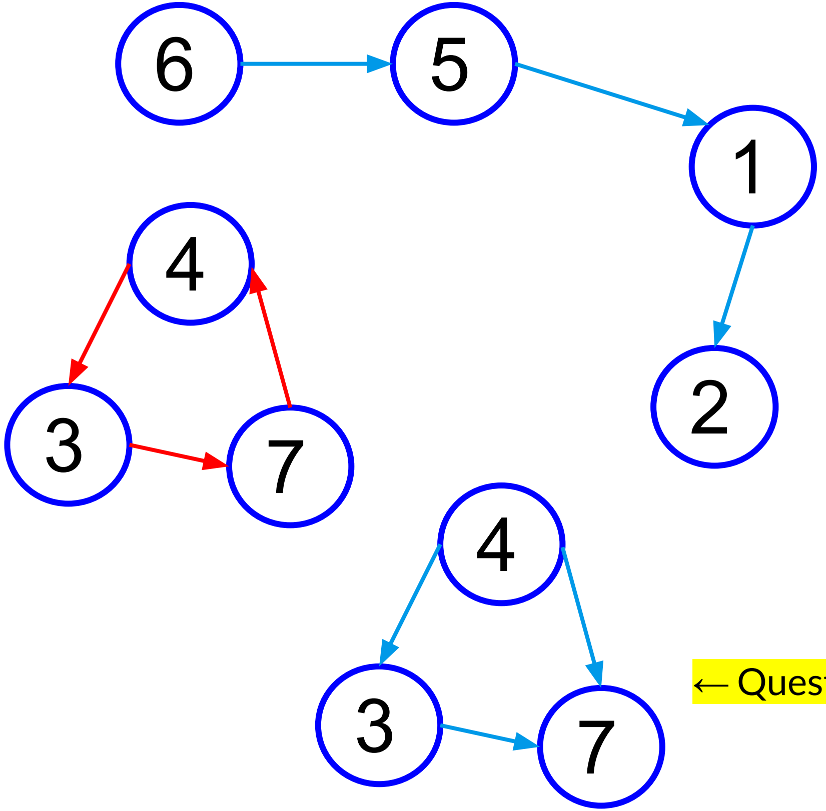
Cammini e cicli *nei grafi __non__ orientati*



Un **cammino** è un insieme di nodi collegati da archi
Nel caso qui a fianco,
$C_1 = \{6, 5, 1, 2\}$ è un cammino (cammino semplice)

Un **ciclo** è un cammino tale che il nodo finale è uguale a quello iniziale.
$C_2 = \{3, 4, 7, 3\}$ è un ciclo

Cammini e cicli *nei grafi orientati*



Un **cammino** è un insieme di nodi collegati da archi
Nel caso qui a fianco,
$C_1 = \{6, 5, 1, 2\}$ è un cammino (cammino semplice)

Un **ciclo** è un cammino tale che il nodo finale è uguale a quello iniziale.
$C_2 = \{3, 4, 7, 3\}$ è un ciclo

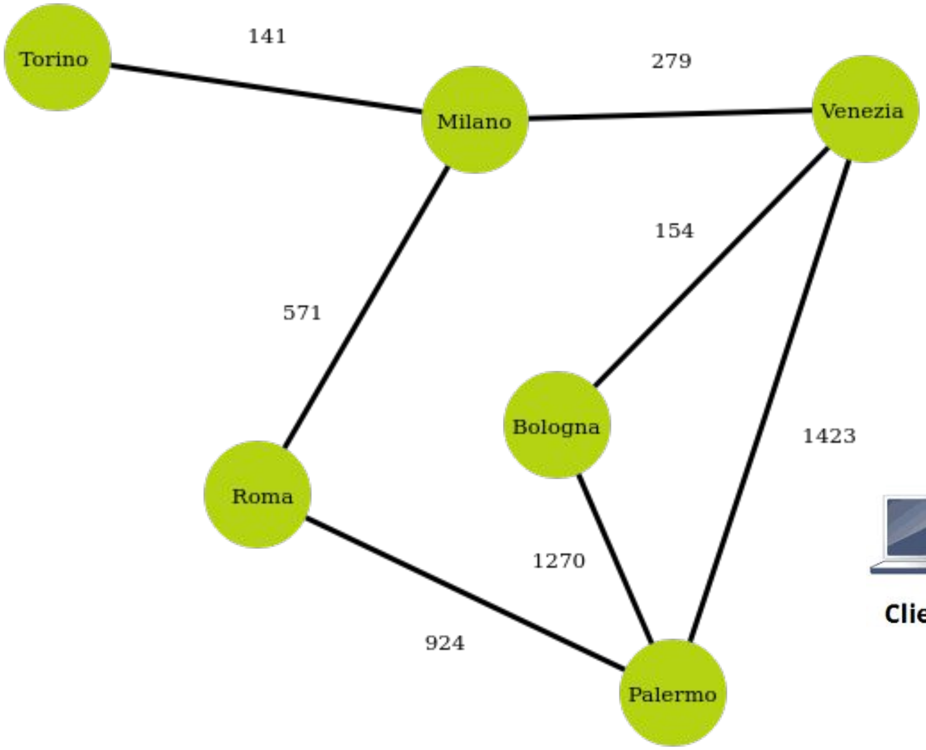
← Questo non è un ciclo!

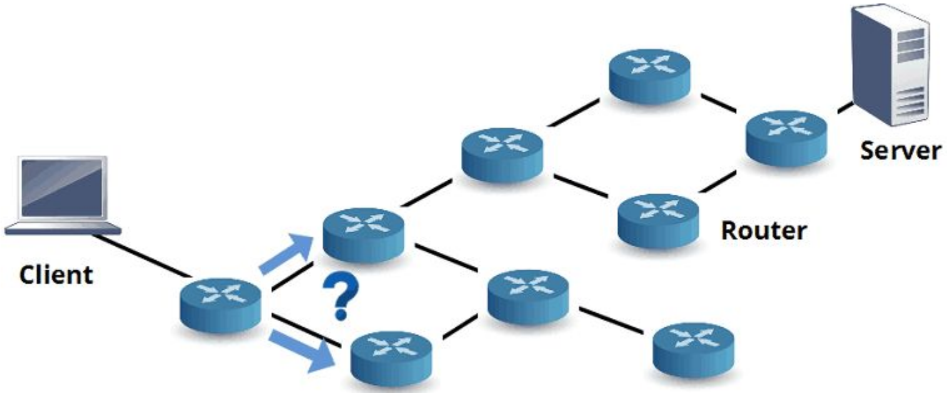
Perché usiamo i GRAFI?

Spesso i dati sono legati tra di loro e vogliamo usare questo “legame” nei nostri programmi. Possiamo risolvere molti problemi utilizzando i grafi.

es. Google Maps: quali sono i possibili percorsi da Roma a Venezia? Qual è il percorso più breve?

es. instradamento pacchetti IP





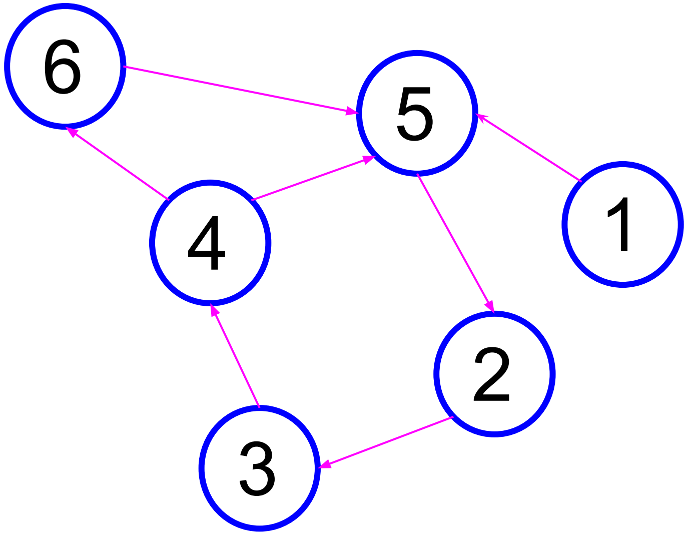
Memorizzare i GRAFI - Liste di adiacenza

- Ogni nodo è rappresentato da un numero intero nell’insieme `[0, …, n-1]`, dove n è il numero di nodi
- Per ogni nodo, salviamo tutti i nodi **adiacenti** (i nodi a cui è collegato tramite degli archi) in una **lista** (array o vector)
- Rappresentiamo il grafo come `vector<vector<int>>` (o in alternativa `int grafo[][]`)

Scenari differenti:

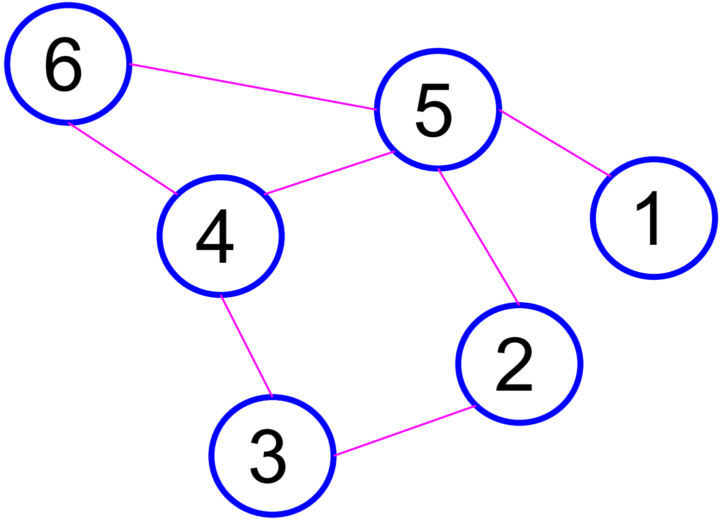
- grafo orientato e non orientato
- grafo pesato e non pesato

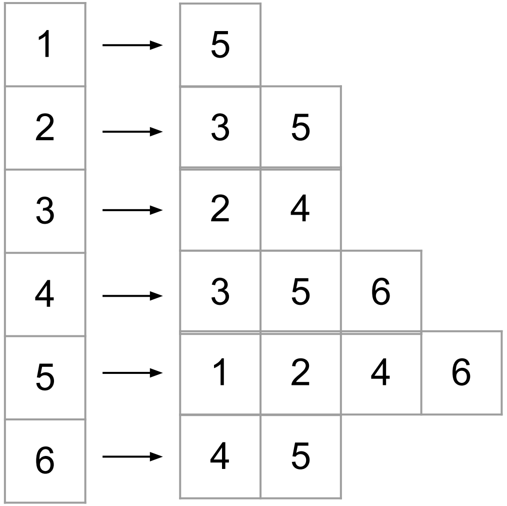
GRAFO orientato - Liste di adiacenza



1	→	5	
2	→	3	
3	→	4	
4	→	5	6
5	→	2	
6	→	5	

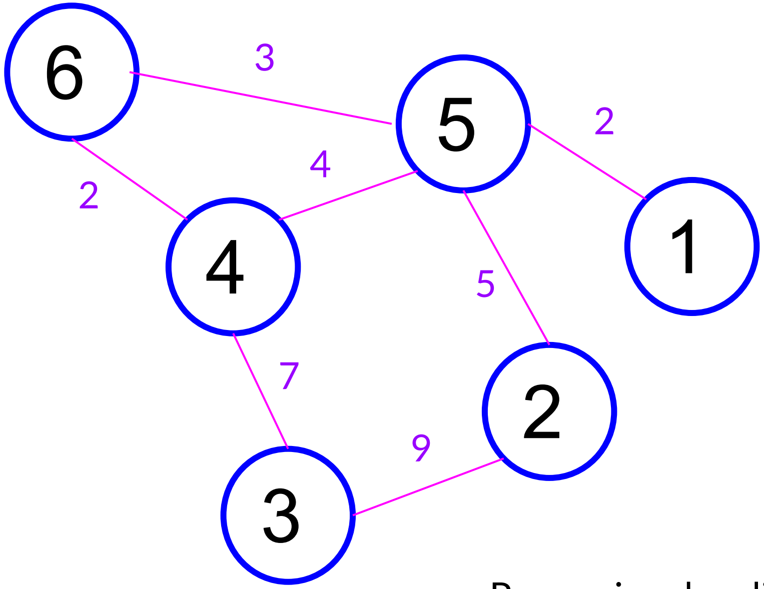
GRAFO NON orientato - Liste di adiacenza





Usando le liste di adiacenza, in questo caso abbiamo delle ridondanze… spazio occupato inutilmente

GRAFO pesato - Liste di adiacenza



1	→	(5, 2)			
2	→	(3, 9)	(5, 5)		
3	→	(2, 9)	(4, 7)		
4	→	(3, 7)	(5, 4)	(6, 2)	
5	→	(1, 2)	(2, 5)	(4, 4)	(6, 3)
6	→	(4, 2)	(5, 3)		

Per ogni nodo adiacente, memorizziamo una coppia (nodo, peso)

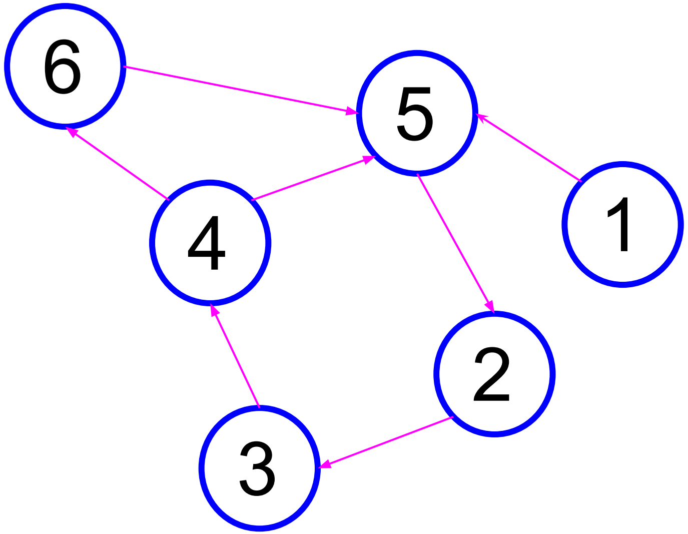
Memorizzare i GRAFI - Matrice di adiacenza

- Ogni nodo è rappresentato da un numero intero nell'insieme `[0, …, n-1]`, dove n è il numero di nodi
- Creiamo una matrice di dimensione `n*n`
- Per ogni coppia di nodi `(i, j)` indichiamo nella **matrice** se esiste un arco da i a j, *scrivendo* `1` *oppure* `0` (se non esiste)

Scenari differenti:

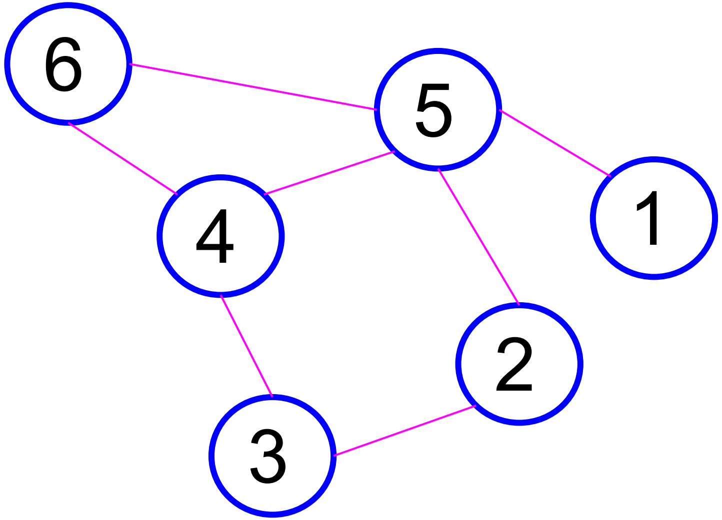
- grafo orientato e non orientato
- grafo pesato e non pesato

GRAFO orientato - Matrice di adiacenza



	0	0	0	1	0
0		1	0	0	0
0	0		1	0	0
0	0	0		1	1
0	1	0	0		0
0	0	0	0	1	

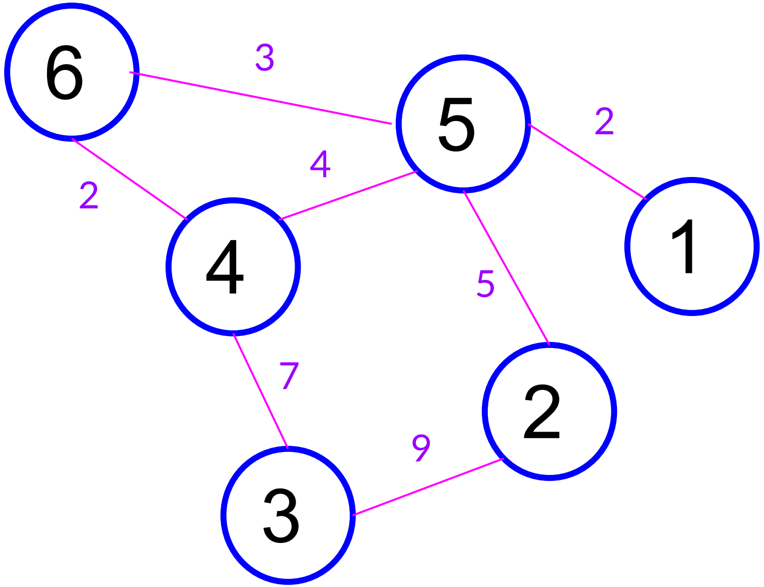
GRAFO NON orientato - Matrice di adiacenza



	0	0	0	1	0
0		1	0	1	0
0	1		1	0	0
0	0	1		1	1
1	1	0	1		1
0	0	0	1	1	

Ci basta mezza matrice, l’altra è speculare

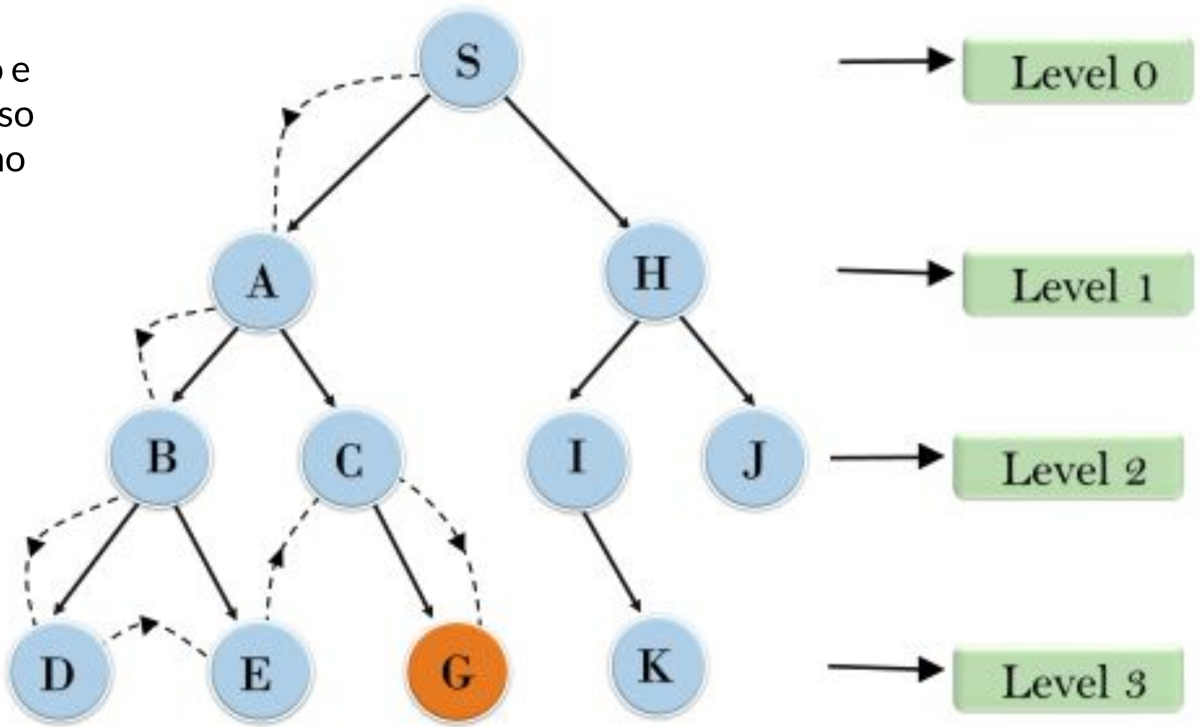
GRAFO pesato - Matrice di adiacenza

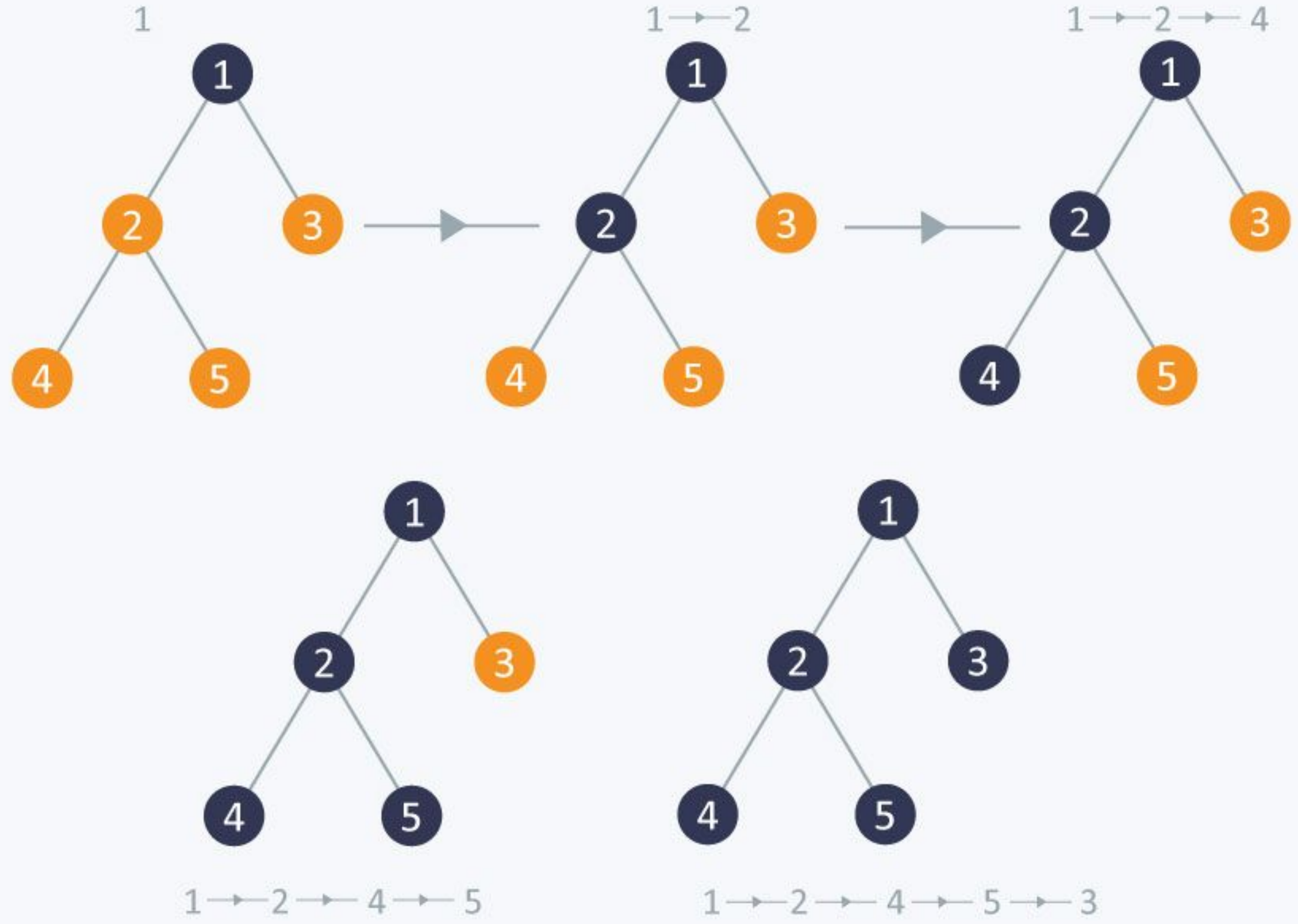


	0	0	0	2	0
0		9	0	5	0
0	9		7	0	0
0	0	7		4	2
2	5	0	4		3
0	0	0	2	3	

Visite dei GRAFI - Depth First Search

parto da un nodo e vado sempre verso il primo vicino fino a quando non ne trovo più




1
1 → 2
1 → 2 → 4
1 → 2 → 4 → 5
1 → 2 → 4 → 5 → 3
1
2
3
4
5

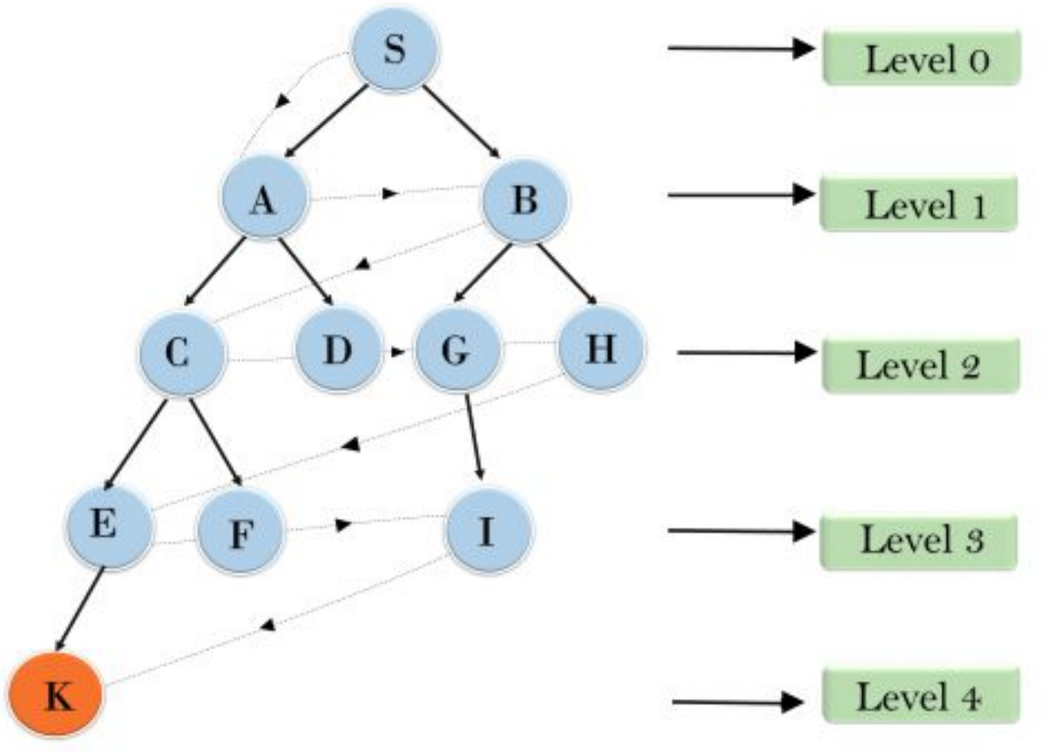
Pseudocodice

```
dfs(GRAPH G, NODE u, boolean[] visited)
visited[u] = true
{ visita il nodo u (pre-order) }
foreach v ∈ G.adj(u) do
    if not visited[v] then
        { visita l'arco (u, v) }
        dfs(G, v, visited)
{ visita il nodo u (post-order) }
```

Complessità: $O(m + n)$

N.B. Vettore visited inizializzato a false

Visite dei GRAFI - Breadth First Search



nodo di partenza

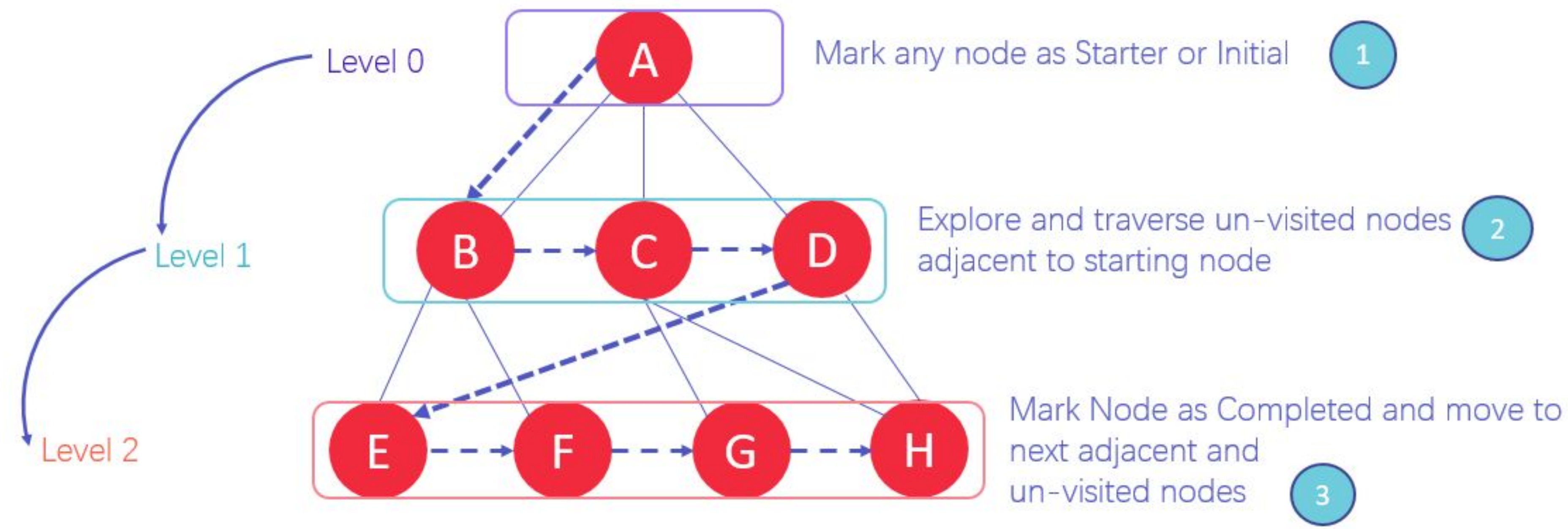
nodi a distanza 1

nodi a distanza 2

nodi a distanza 3

nodi a distanza 4

CONCEPT DIAGRAM



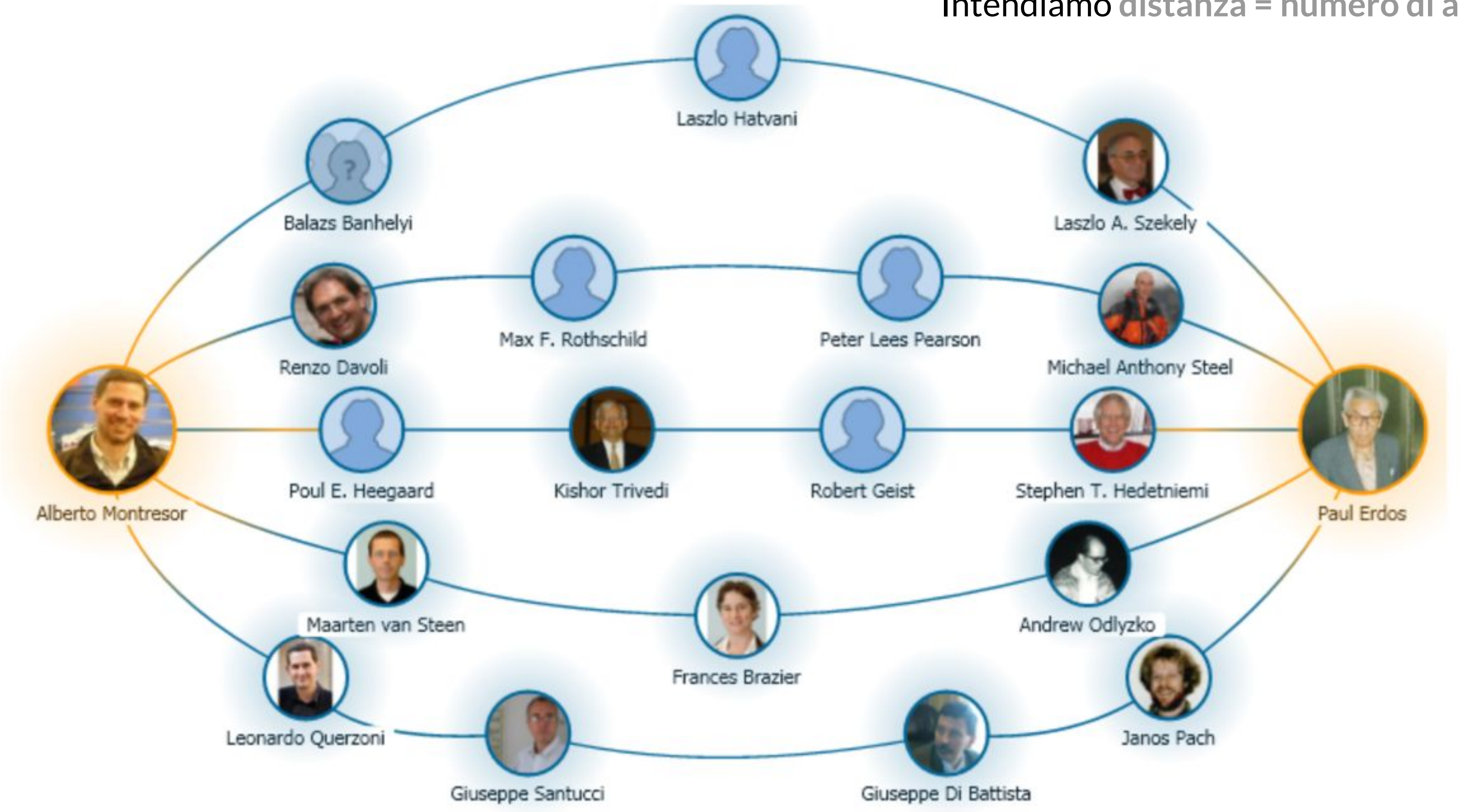
Guru99.com

Pseudocodice

```
bfs(GRAPH G, NODE r)
─────────────────────────────────────────
QUEUE Q = Queue( )
S.enqueue(r)
boolean[ ] visited = new boolean[G.size()]
foreach u ∈ G.V() − {r} do
    visited[u] = false
visited[r] = true
while not Q.isEmpty() do
    NODE u = Q.dequeue()
    { visita il nodo u}
    foreach v ∈ G.adj(u) do
        { visita l'arco (u, v) }
        if not visited[v] then
            visited[v] = true
            Q.enqueue(v)
```

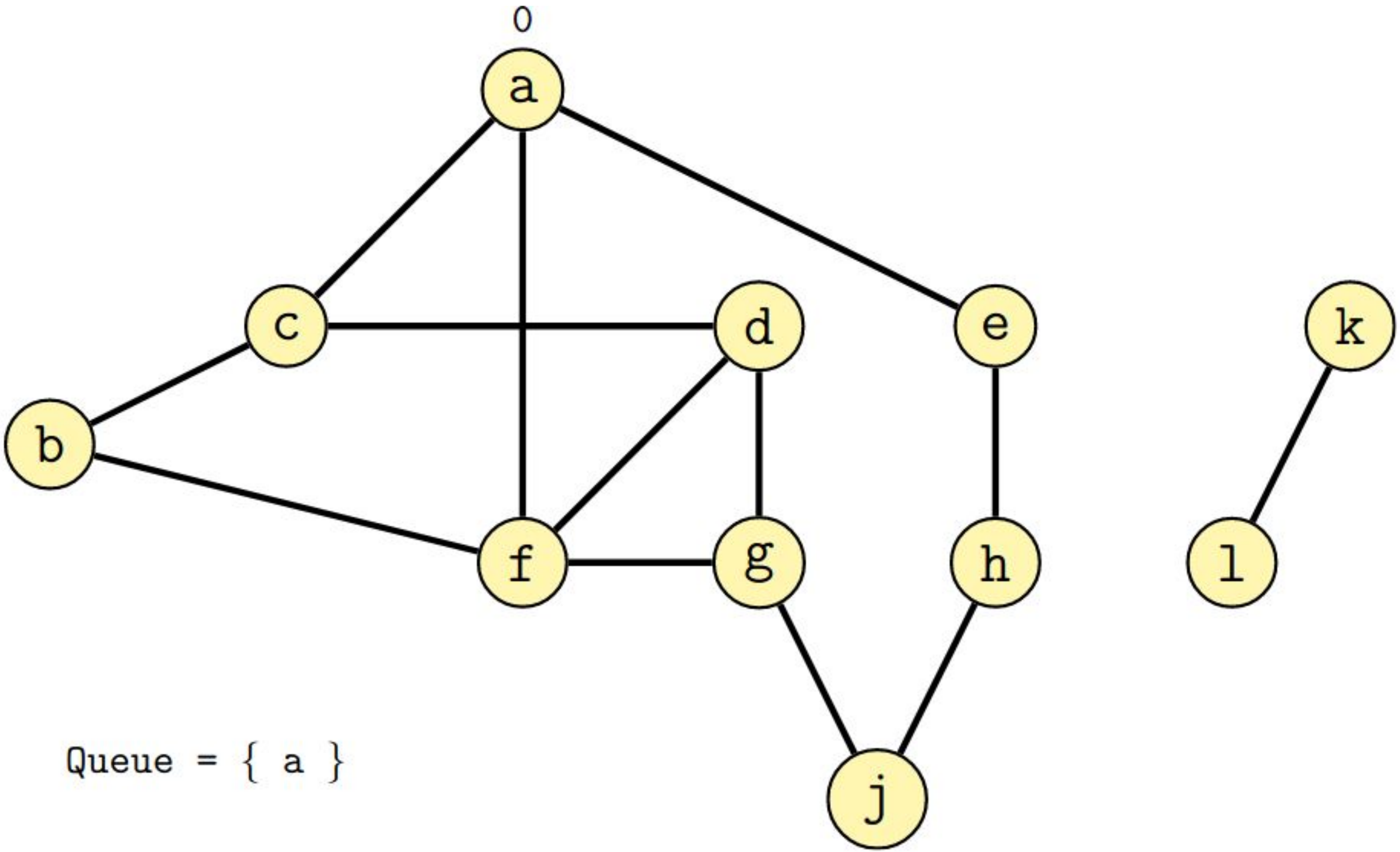
Cammini minimi

Vogliamo calcolare la distanza minima tra un tutti i nodi ed Erdos.
Intendiamo **distanza = numero di archi**

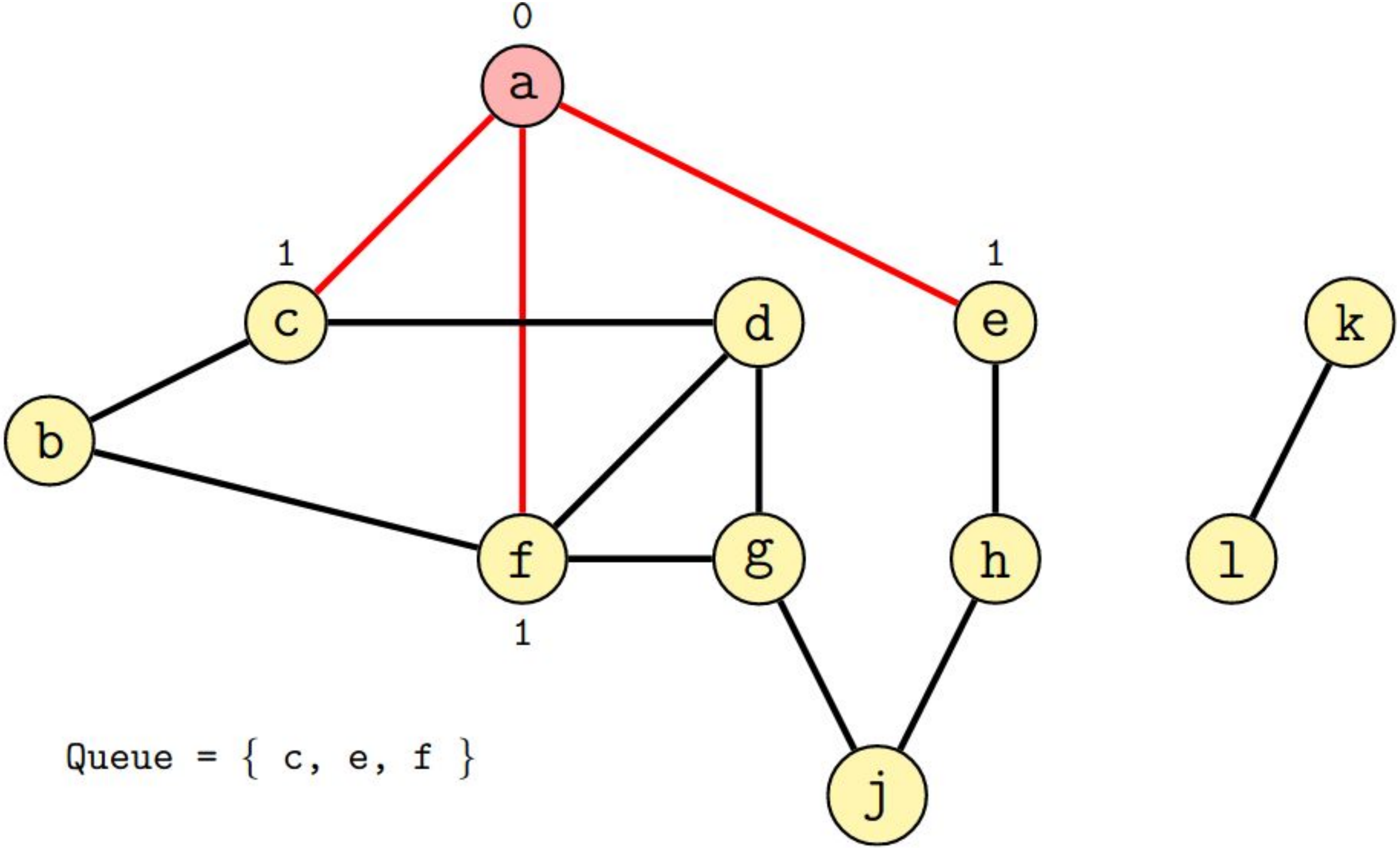


Cammini minimi

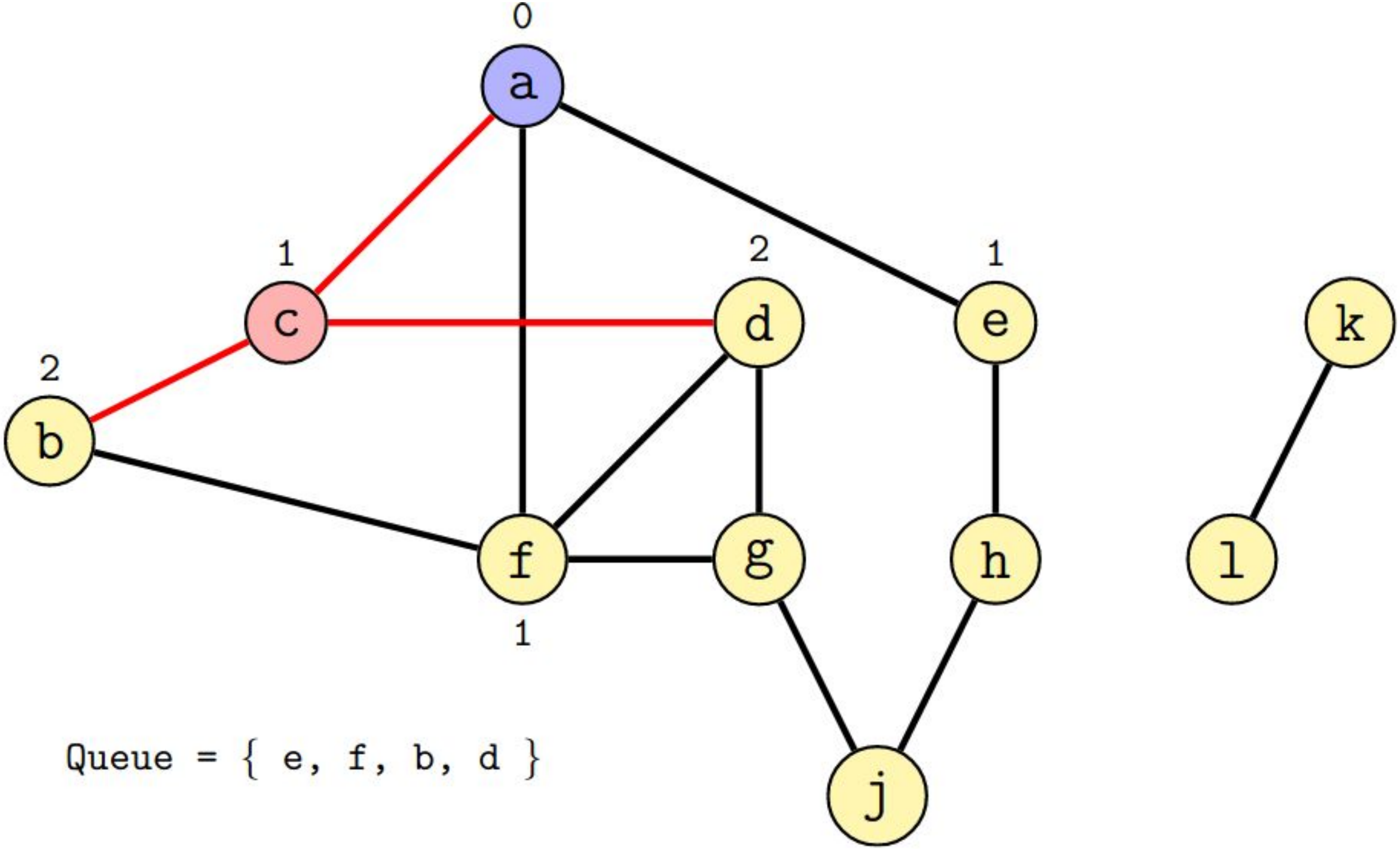
```
distance(GRAPH G, NODE r, int[] distance)
-----------------------------------------
QUEUE Q = Queue()
Q.enqueue(r)
foreach u ∈ G.V() − {r} do
    distance[u] = ∞
distance[r] = 0
while not Q.isEmpty() do
    NODE u = Q.dequeue()
    foreach v ∈ G.adj(u) do
        if distance[v] == ∞ then % Se il nodo v non è stato scoperto
            distance[v] = distance[u] + 1
            Q.enqueue(v)
```



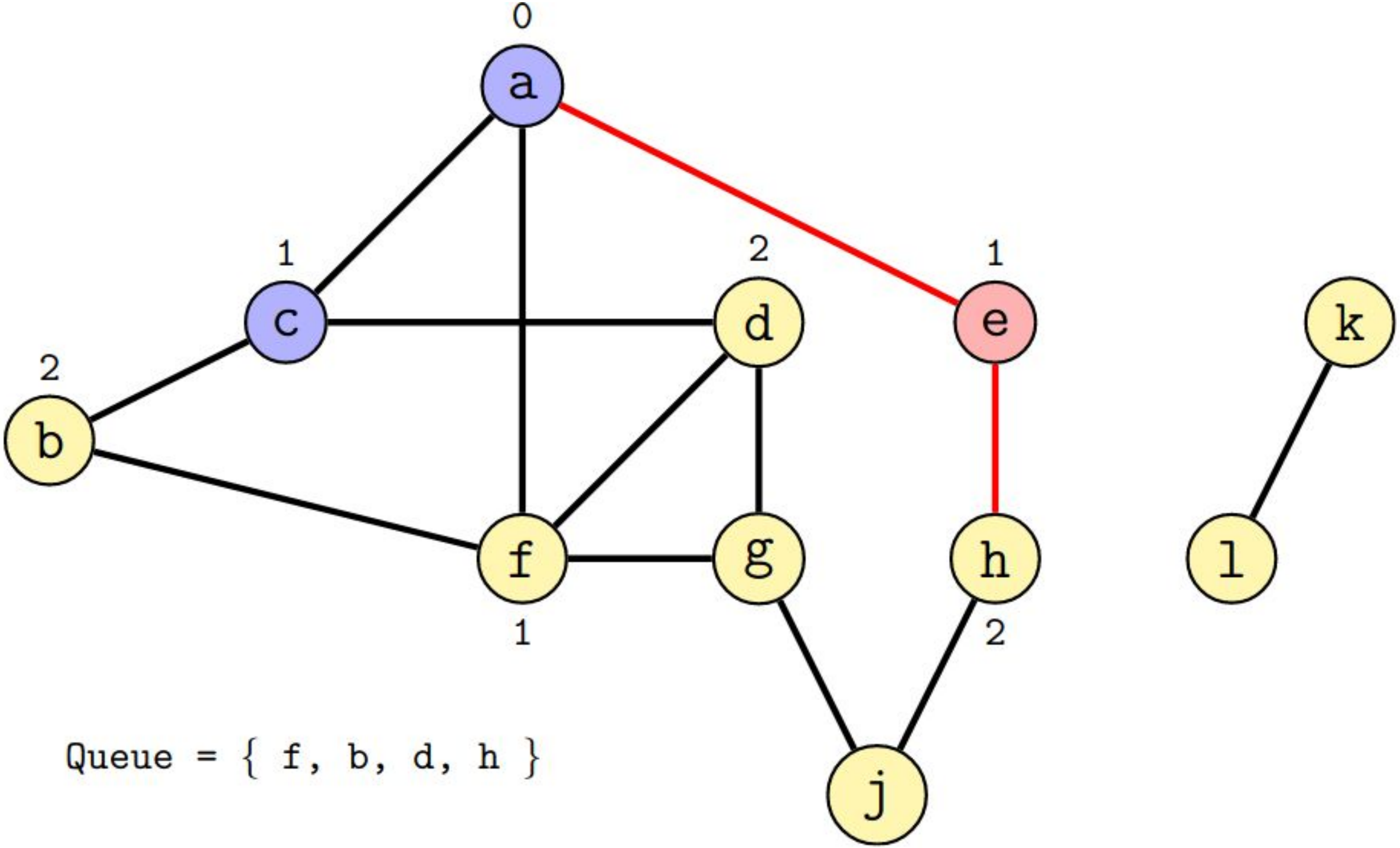
Queue = { a }



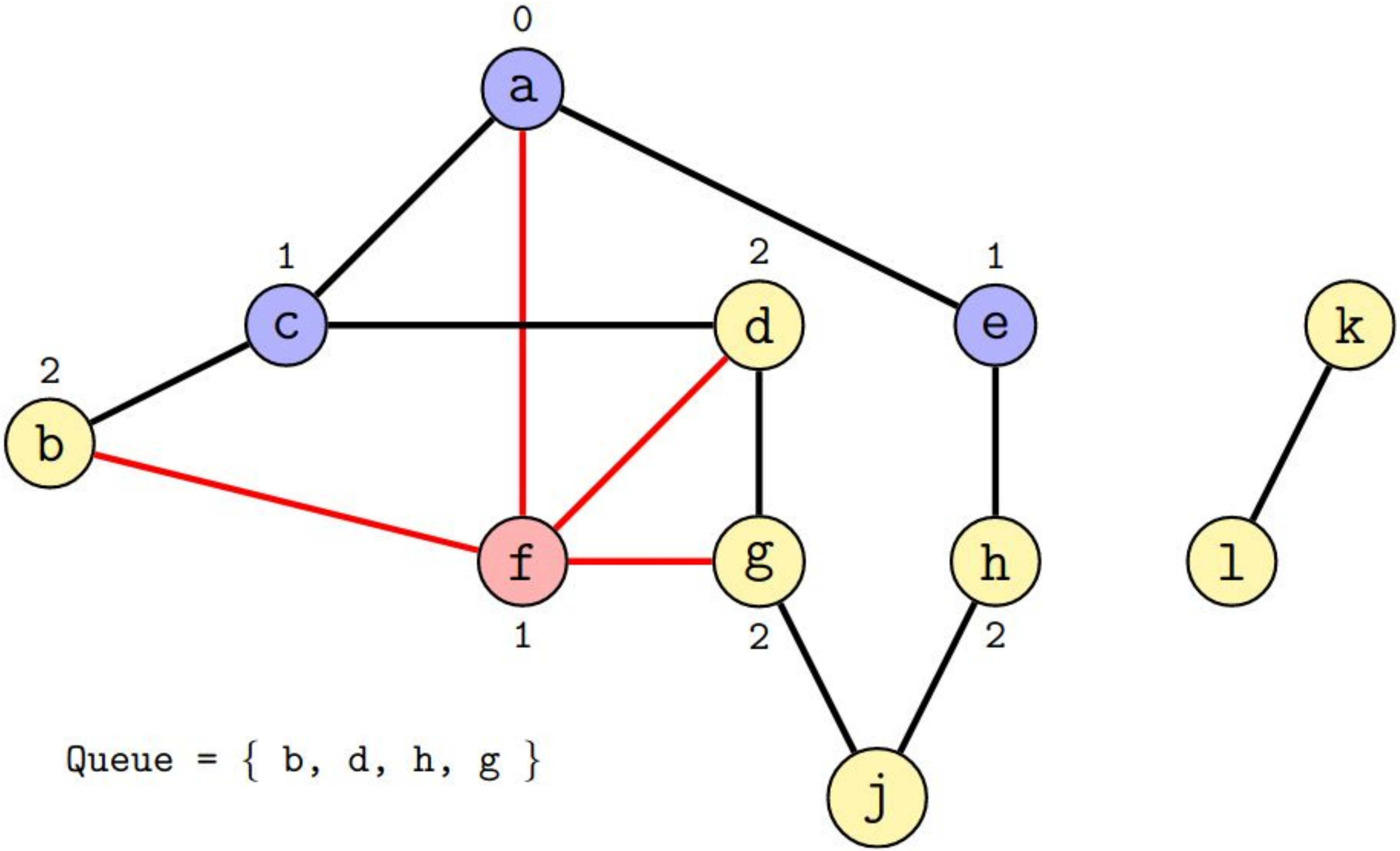
Queue = { c, e, f }



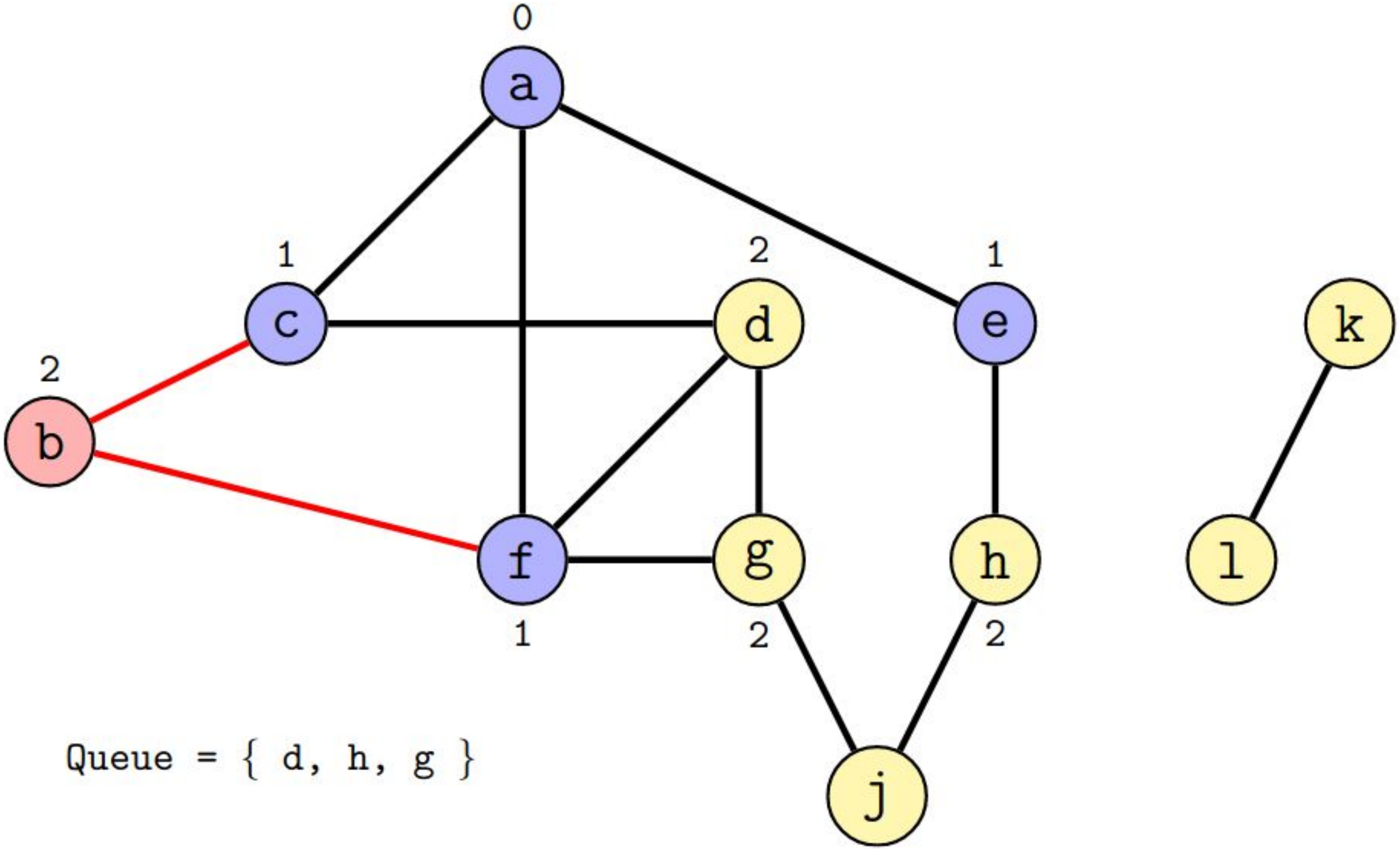
Queue = { e, f, b, d }



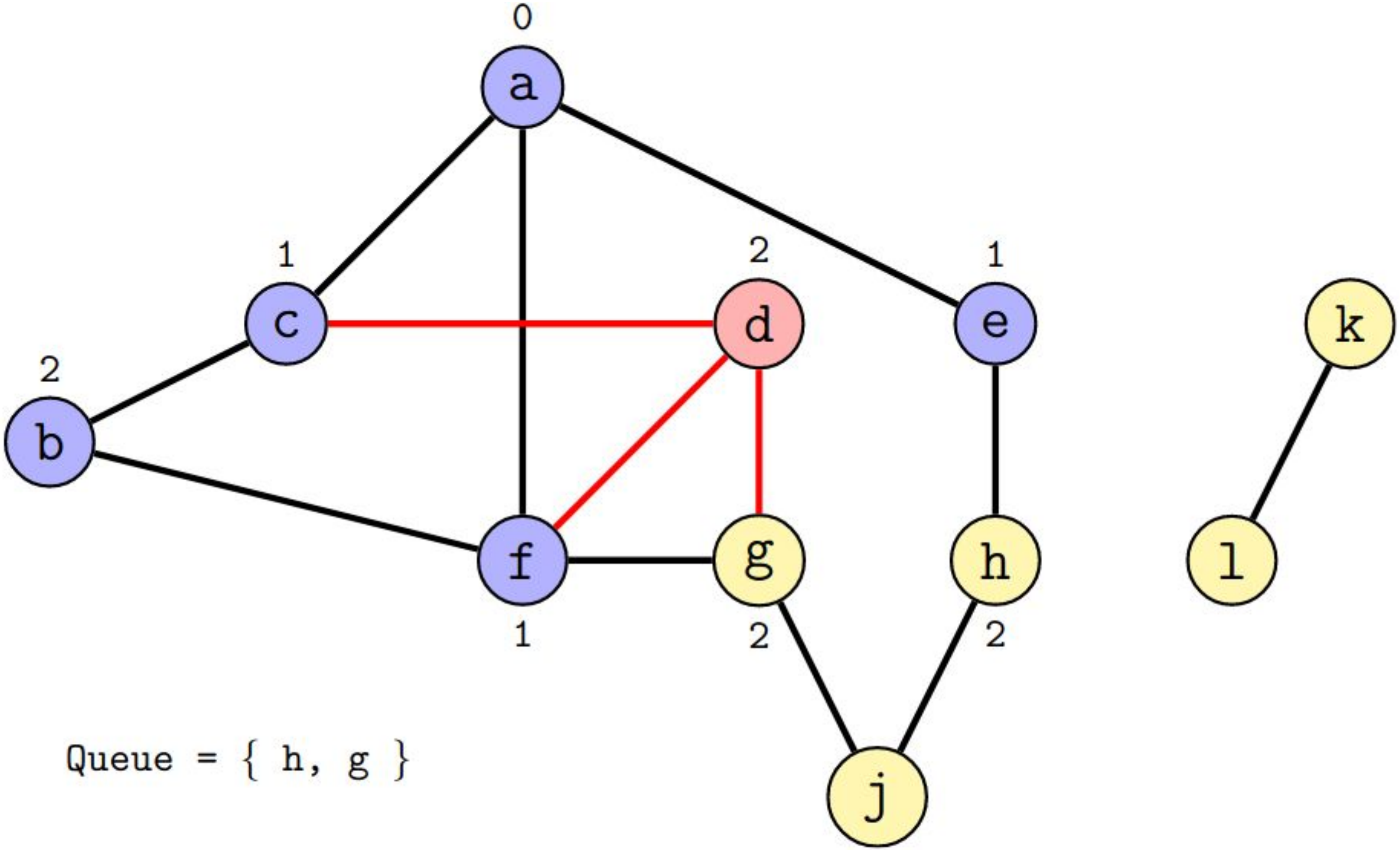
Queue = { f, b, d, h }



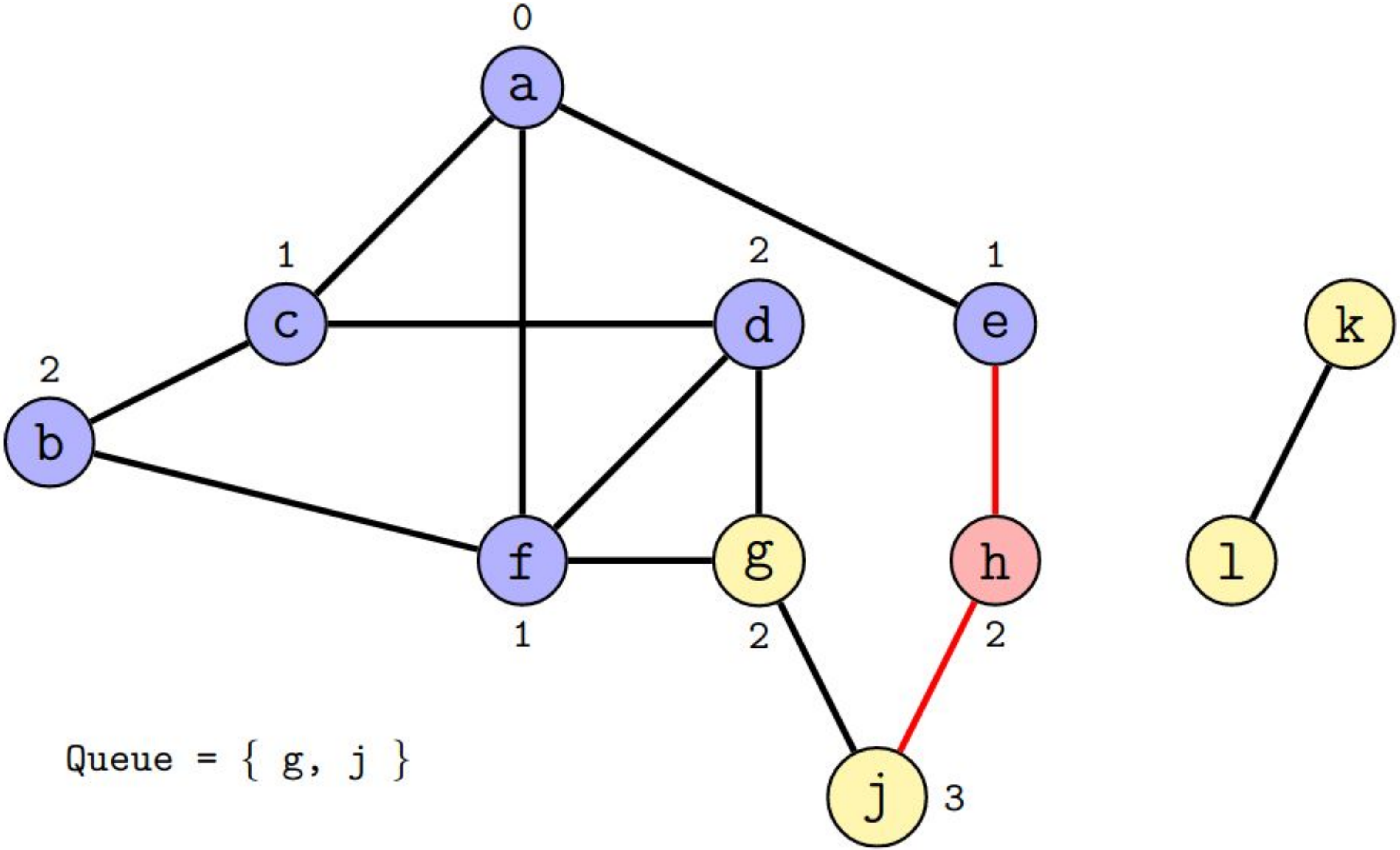
Queue = { b, d, h, g }



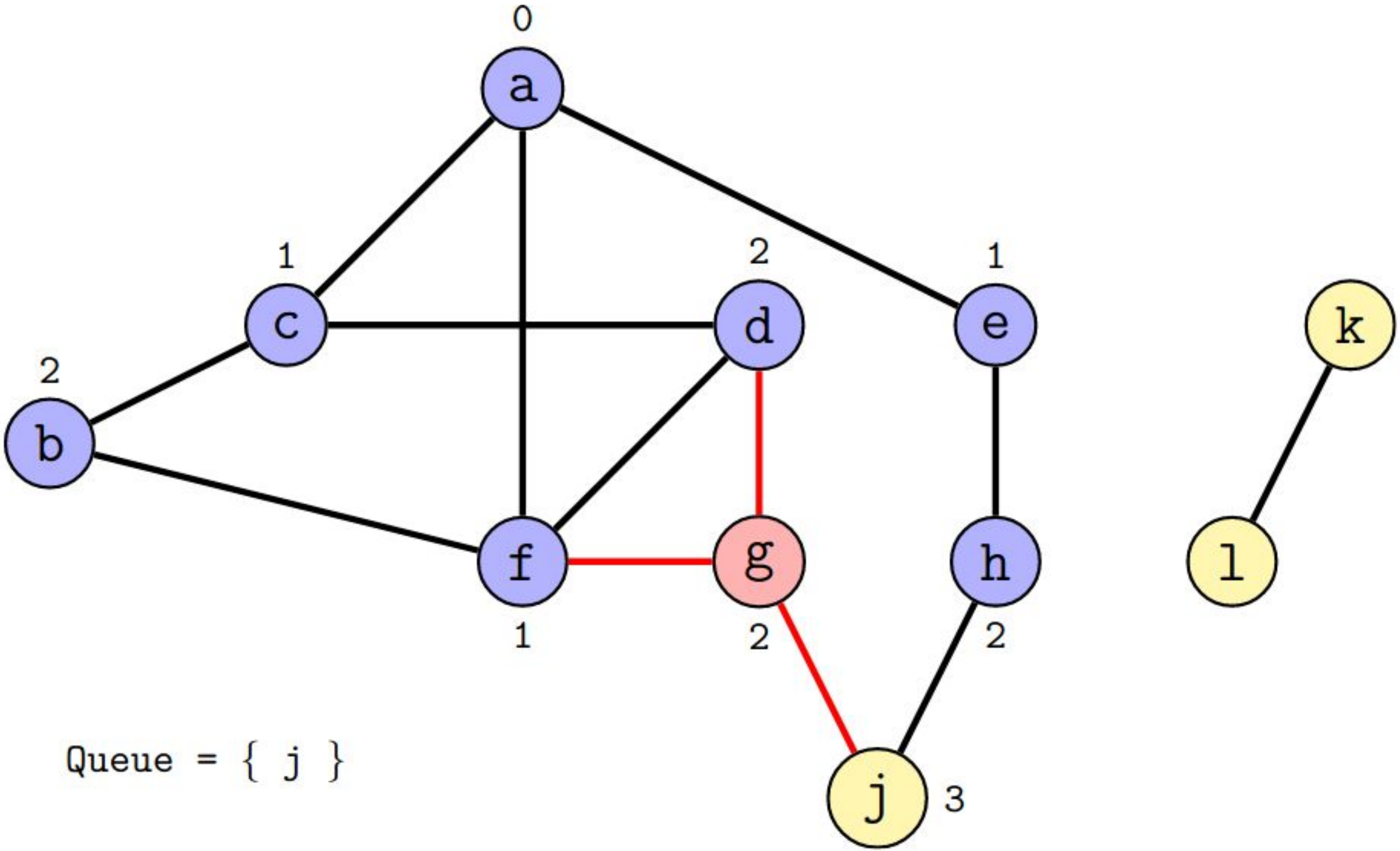
Queue = { d, h, g }



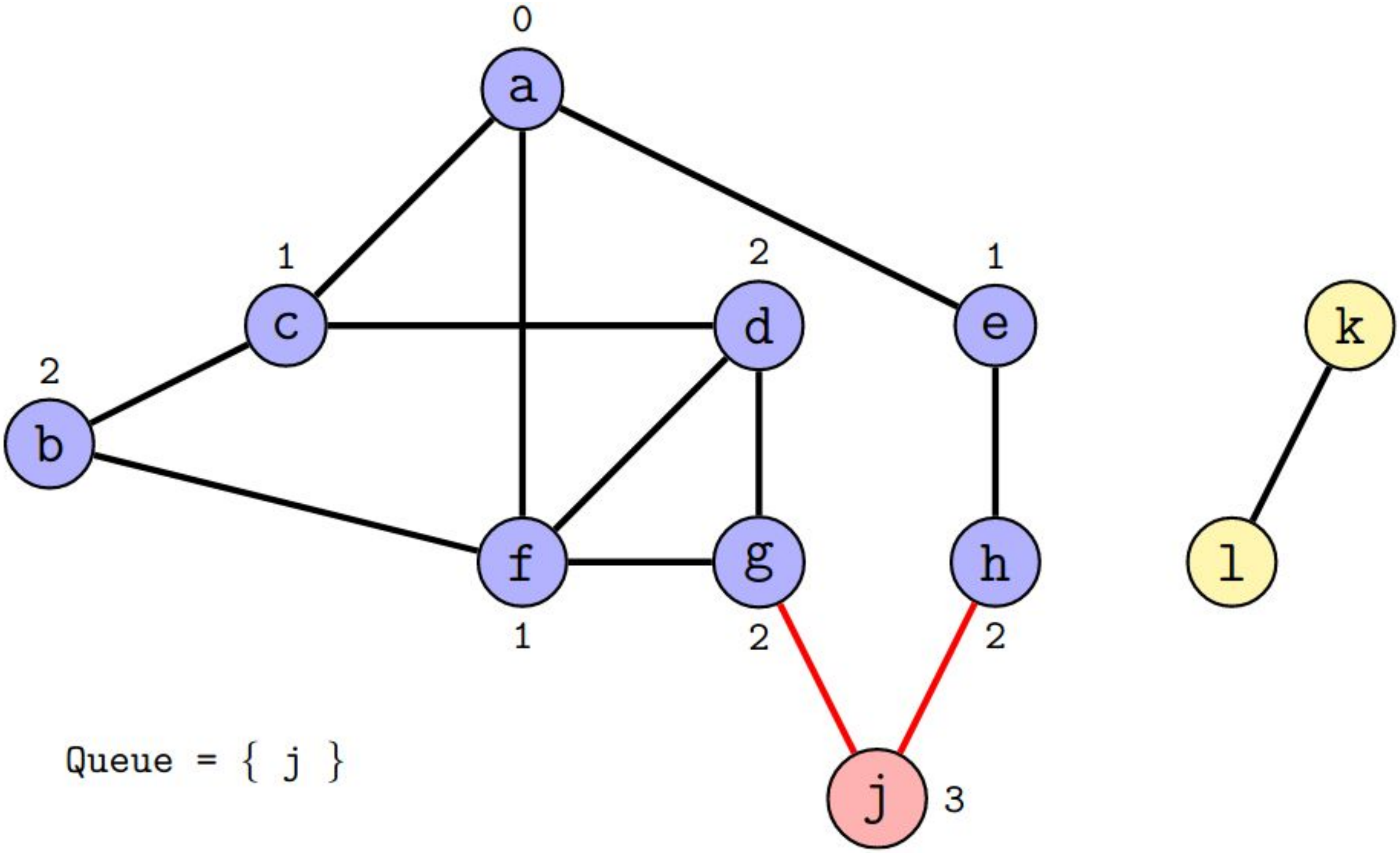
Queue = { h, g }



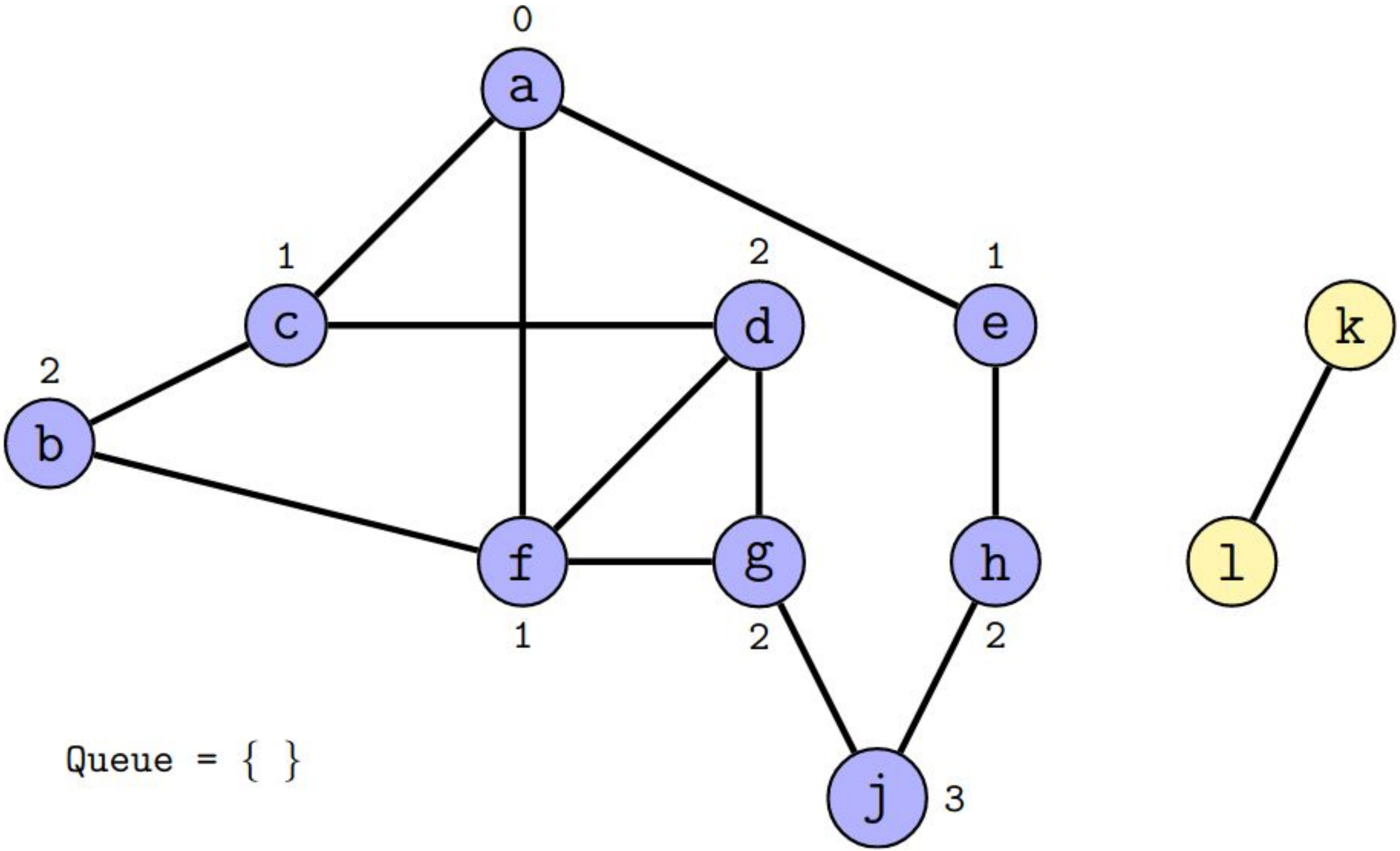
Queue = { g, j }



Queue = { j }



Queue = { j }



Queue = { }

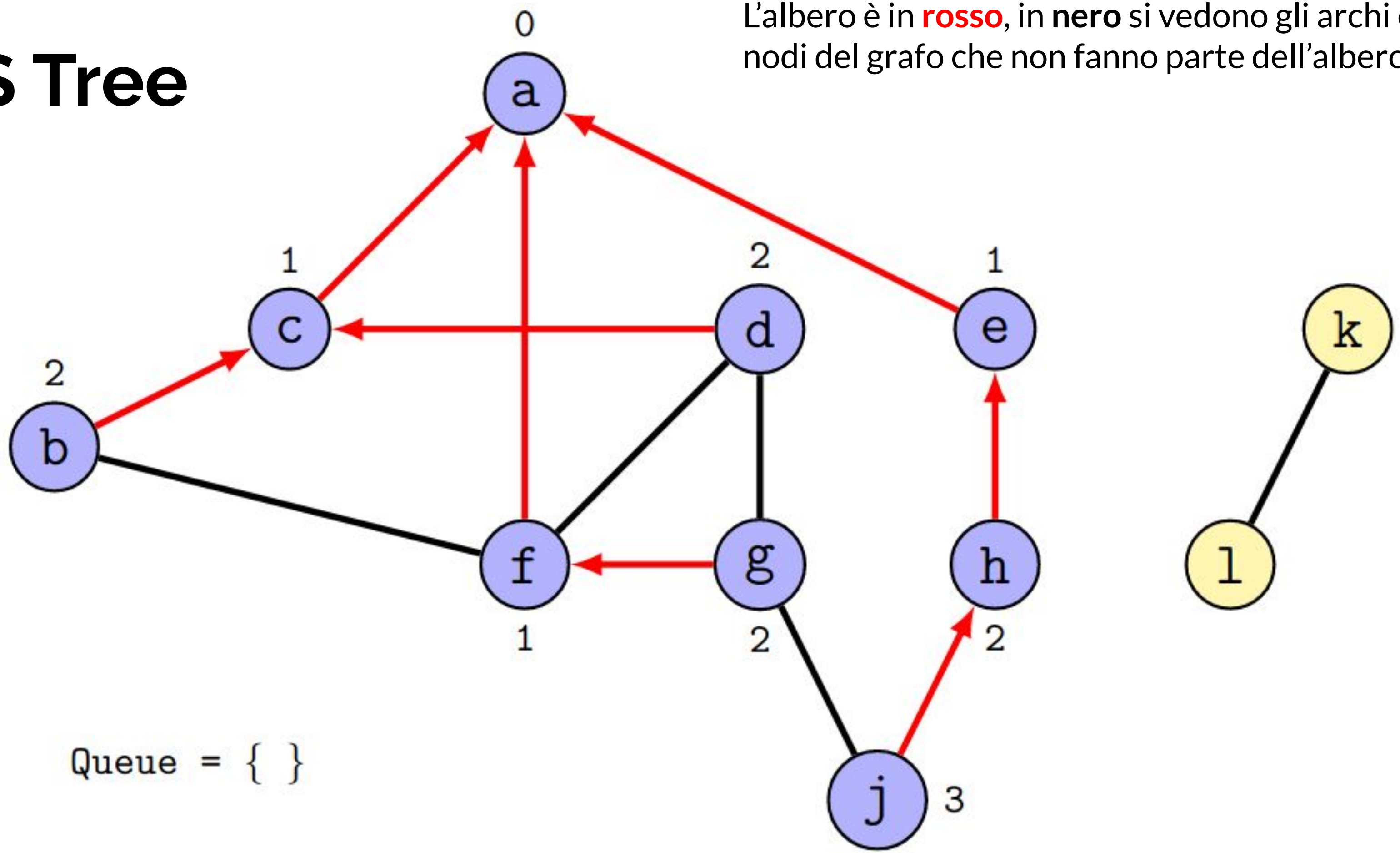
- La visita BFS può essere usata per ottenere il cammino più breve fra due nodi (misurato in numero di archi)
- "Albero di copertura" con radice r
- Memorizzato in un vettore dei padri *parent*

```
distance([...], NODE[] parent)
[...]
parent[r] = nil
while not S.isEmpty() do
    NODE u = S.dequeue()
    foreach v ∈ G.adj(u) do
        if distance[v] == ∞ then
            distance[v] =
             distance[u] + 1
            parent[v] = u
            S.enqueue(v)
```

```
printPath(NODE r, NODE s, NODE[] parent)
if r == s then
    print s
else if parent[s] == nil then
    print "error"
else
    printPath(r, parent[s], parent)
    print s
```

BFS Tree

L'albero è in rosso, in **nero** si vedono gli archi e i nodi del grafo che non fanno parte dell'albero



Piano

Piano di studi

https://training.olinfo.it/#/task/luiss_piano/statement

Qual è il minimo numero di ore necessarie per laurearsi?

Pensa alla strategia / logica del programma

1. *Che tipo di grafo è? Orientato o non orientato? Pesato o non pesato?*
2. *Come memorizziamo il grafo?*
3. *Che tipo di visita devo fare? Da che nodo parto?*
4. *Mi basta solo una visita?*

Pensa alla strategia / logica del programma

1. *Che tipo di grafo è?* **Orientato** e **Pesato**
2. *Come memorizziamo il grafo?* **Matrice** (ce la danno già)
3. *Che tipo di visita devo fare? Da che nodo parto?* **BFS**: sto cercando dei cammini minimi che rispettano delle condizioni
4. *Mi basta solo una visita?* No, devo fare **N visite**: parto da ciascun nodo

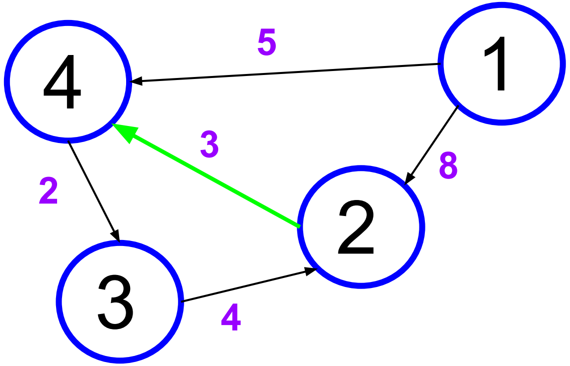
```
INPUT
8  (K)
4  (N)
2  5  3  4  (crediti)
0  8  0  5
0  0  0  3
0  4  0  0
0  0  2  0
```

0	**8**	0	**5**
0	0	0	**3**
0	**4**	0	0
0	0	**2**	0

K = numero di crediti da ottenere
N = numero di esami (nodi del grafo)

Pensa alla strategia / logica del programma

0	8	0	5
0	0	0	3
0	4	0	0
0	0	2	0



Soluzione dell’esempio: 3 ore richieste. Parto con l’esame 2, per il primo esame non si contano le ore; svolgo poi l’esame 4 con costo di 3 ore. La somma dei loro crediti è 5+4 = 9, che è `>= K`, perfetto!

Anche se ottengo più crediti di K, va bene comunque. Devono essere **almeno** K.

SEE YOU NEXT TIME!