

TECNICO SUPERIORE WEB DEVELOPER FULL STACK

DEVELOPMENT

TESTING FEATURES

WEB ANALYTICS

WORKFLOW

#11 - Grafi

GS



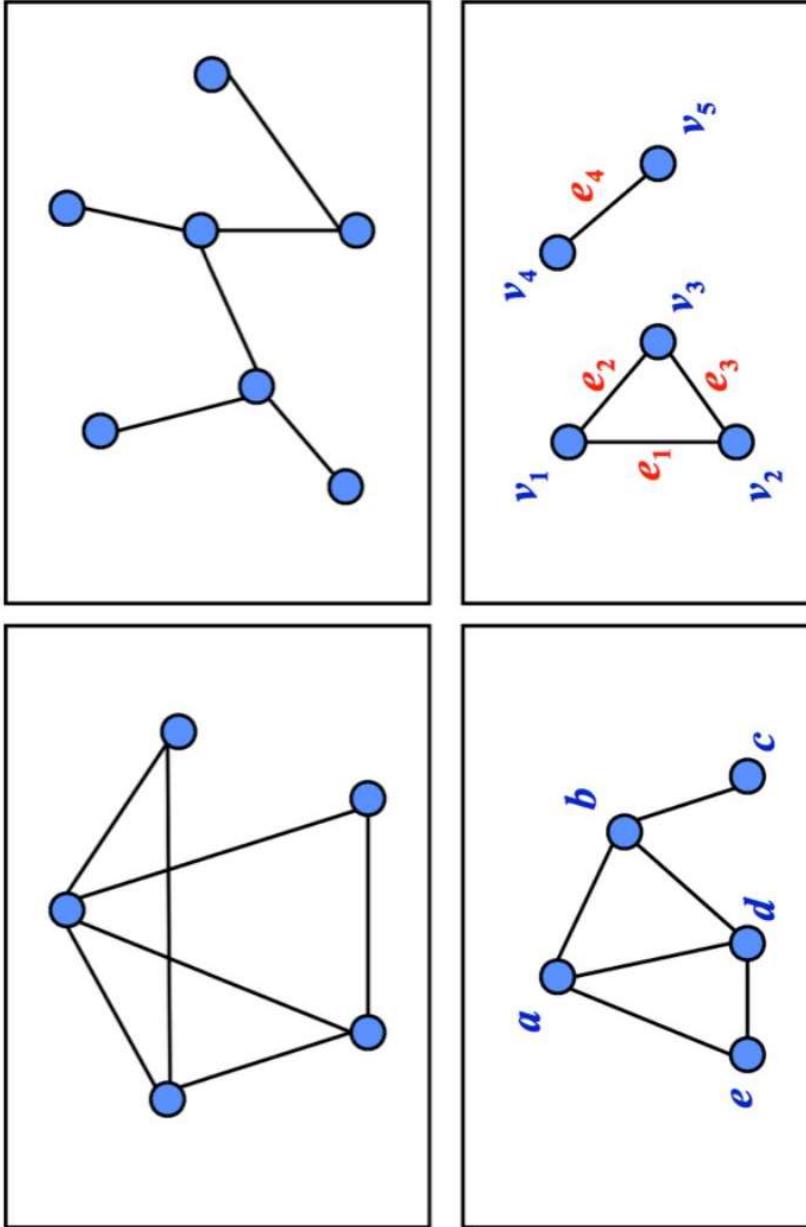
n i u k <>

innovation and knowledge

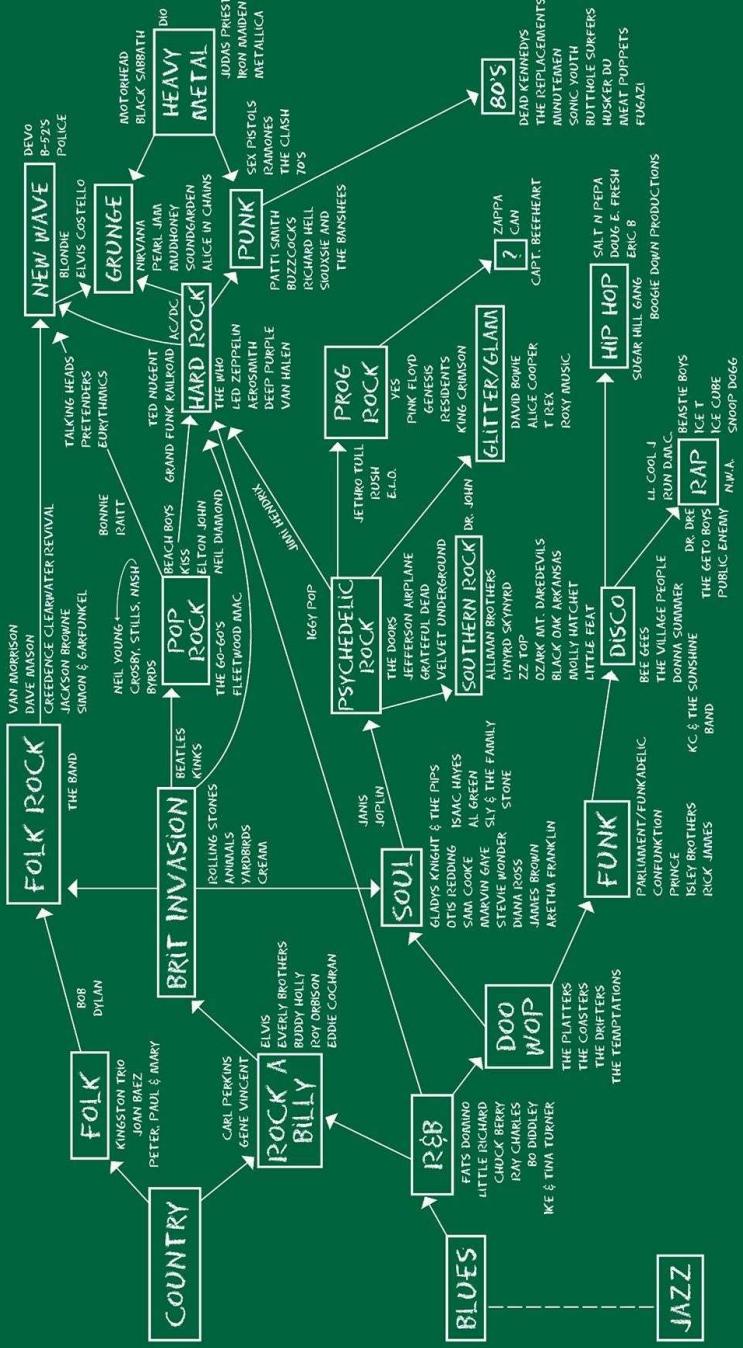
RESPONSIVE DESIGN

CONTENT
MANAGEMENT

Alcuni esempi di grafi:

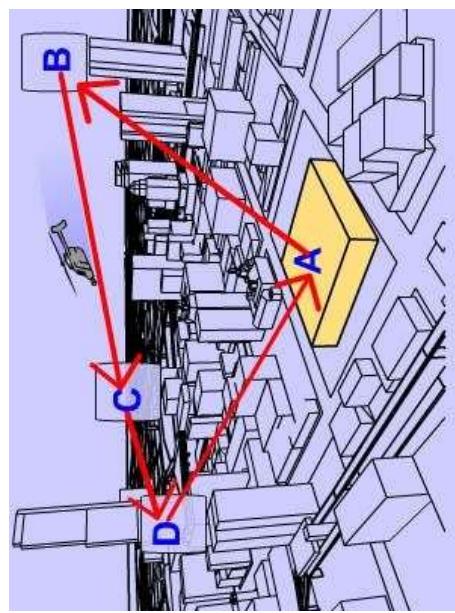
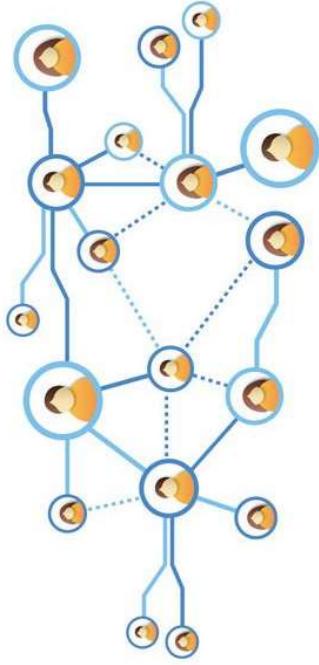
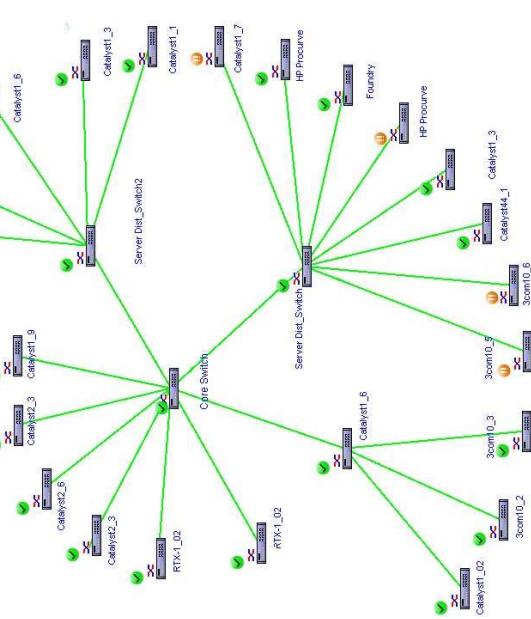
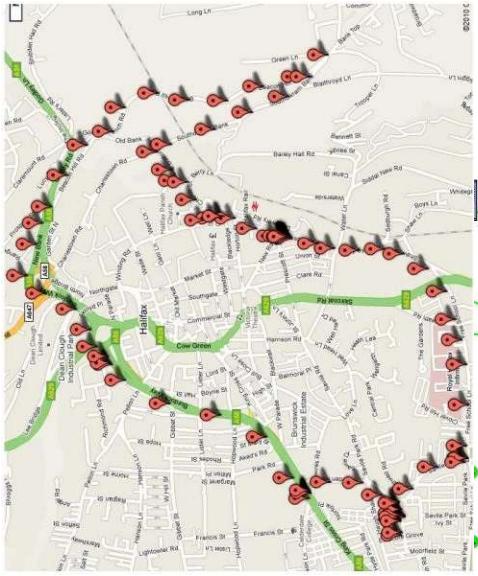


HISTORY OF ROCK



ITS DIGITAL
ACADEMY

nium innovation and knowledge



Problemi relativi ai grafi

Problemi in grafi non pesati

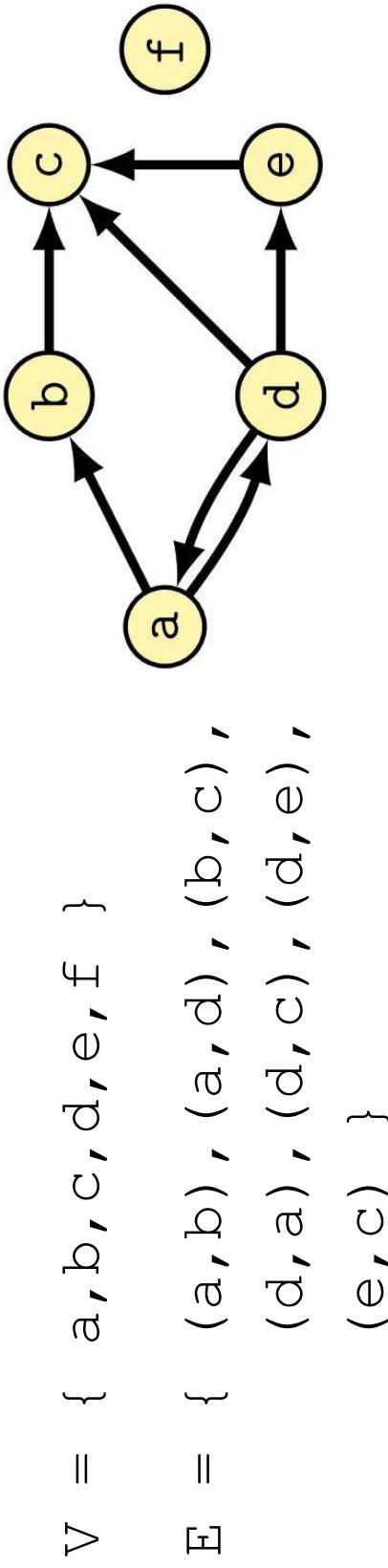
- Ricerca del cammino più breve
(misurato in numero di archi)
- Componenti (fortemente) connesse
- Verifica ciclicità
- Ordinamento topologico
- Cammini di peso minimo
- Alberi di copertura di peso minimo
- Flusso massimo

Problemi in grafi pesati

Grafi orientati: definizione

Un grafo orientato (directed) è una coppia $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ dove:

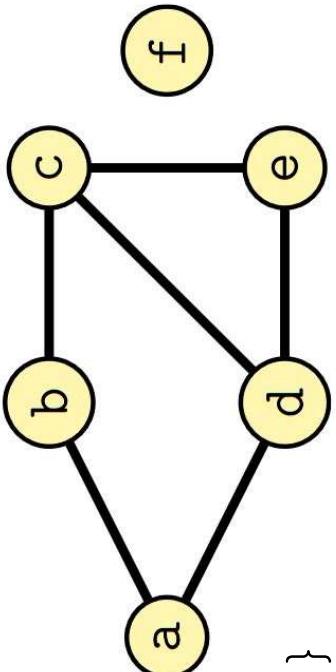
- \mathbf{V} è un insieme di nodi (node) o vertici (vertex)
- \mathbf{E} è un insieme di coppie ordinate (u, v) di nodi dette archi (edge)



Grafi non orientati: definizione

Un grafo non orientato (undirected) è una coppia $\textcolor{red}{G} = (V, E)$ dove:

- V è un insieme di nodi (node) o vertici (vertex)
- E è un insieme di coppie non ordinate (u, v) di nodi dette archi

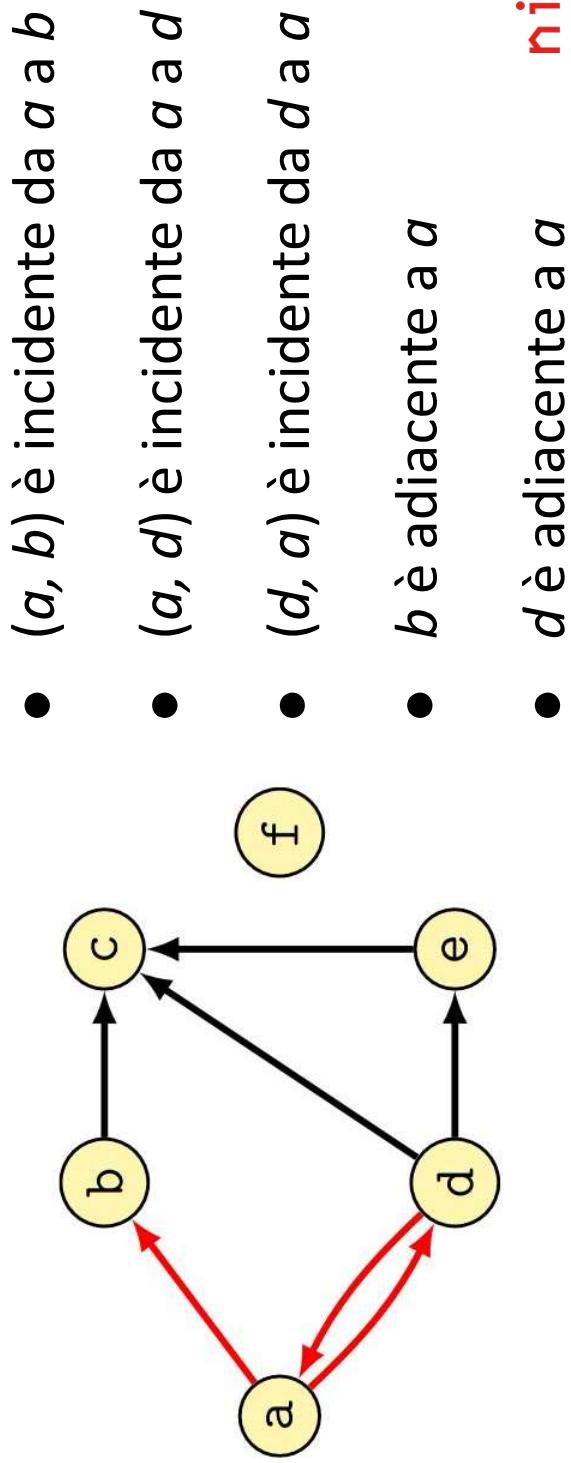


$$V = \{ a, b, c, d, e, f \}$$

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

Grafi, un po' di terminologia

- Un vertice v è detto **adiacente** a u se esiste un arco (u, v)
- Un arco (u, v) è detto **incidente** da u a v
- In un grafo indiretto, la relazione di adiacenza è **simmetrica**



Grafi, un po' di terminologia

Dimensioni del grafo

$n = |V|$ numero di nodi

$m = |E|$ numero di archi

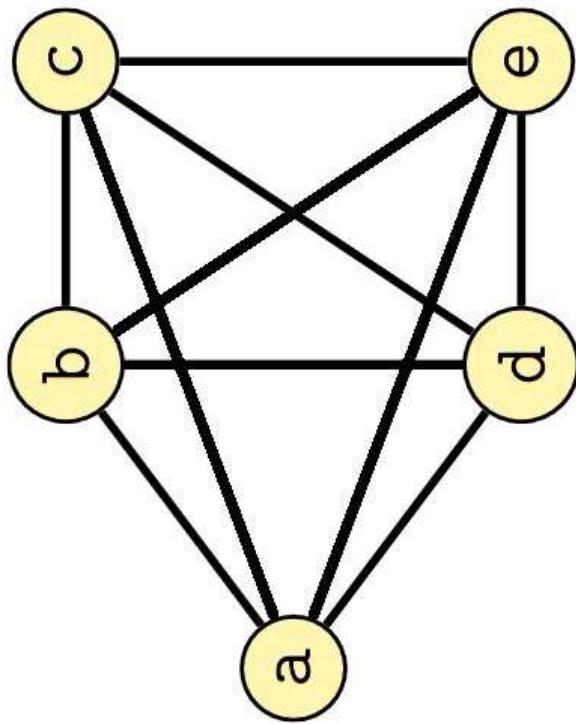
Alcune relazioni fra n e m

- In un grafo non orientato, $m \leq (n(n-1)) / 2 = O(n^2)$
- In un grafo orientato, $m \leq n^2 - n = O(n^2)$

La complessità degli algoritmi sui grafi è espressa sia in termini di
 n sia di m (ad es. $O(n+m)$)

Casi particolari

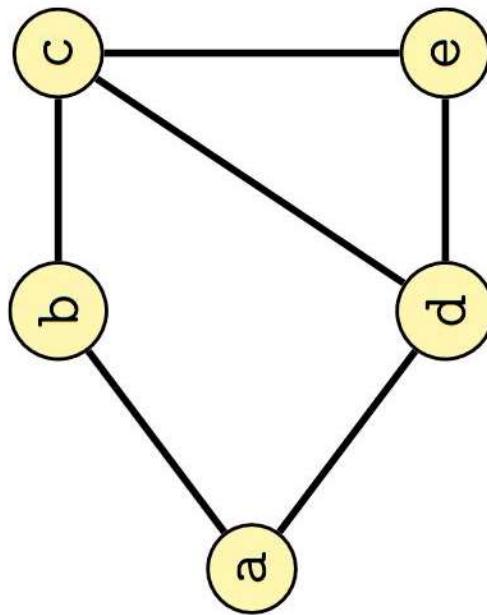
Un grafo con un arco fra tutte le coppie di nodi è detto **completo**.



Casi particolari

Informalmente, un grafo si dice **sparsò** 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”, ad es. $m=\Omega(n^2)$



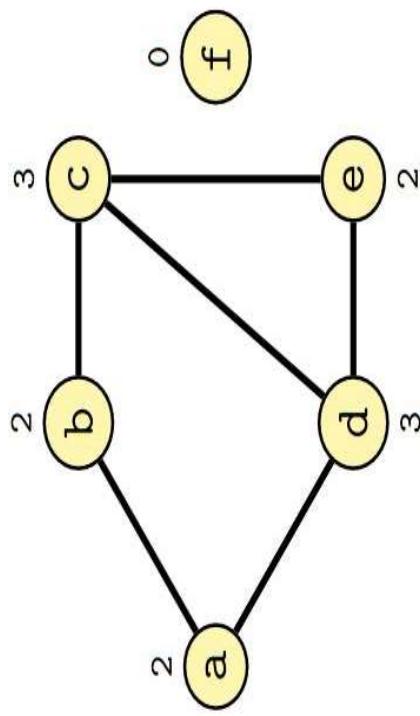
Grado di un nodo in un grafo

12

Nei grafi non orientati il **grado** (degree) di un nodo è il numero di archi incidenti su di esso.

Nei grafi orientati distinguiamo:

- **grado entrante** (in-degree): archi incidenti su di esso
- **grado uscente** (out-degree): archi incidenti da esso



Cammino (Path)

In un grafo $G = (V, E)$ (orientato oppure no), un **cammino** C di lunghezza k è una sequenza di nodi u_0, u_1, \dots, u_k tale che

$$(u_i, u_{i+1}) \in E \text{ per } 0 \leq i \leq k-1$$

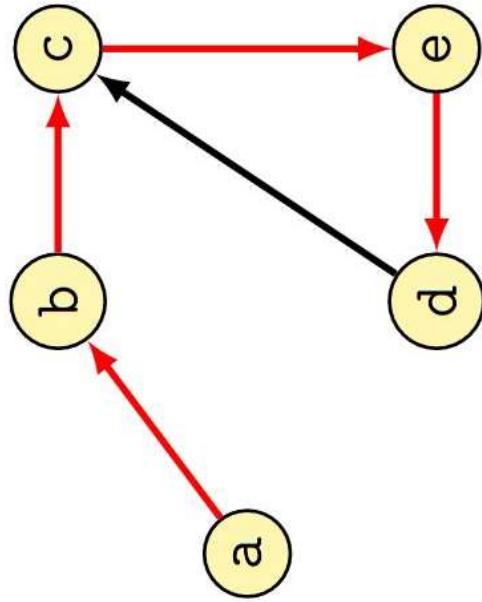
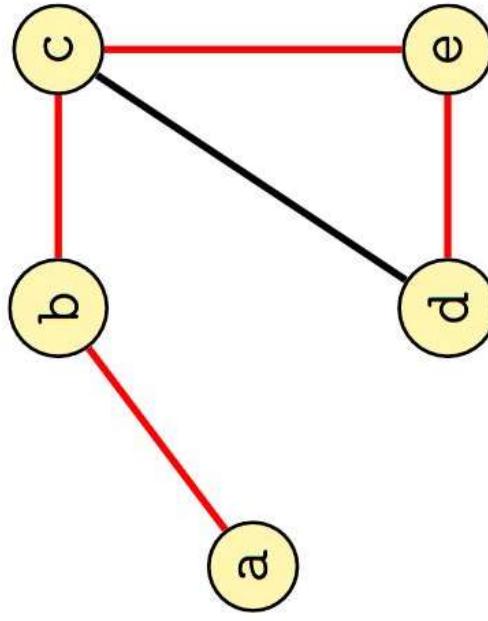
Nota: un cammino è detto **semplice** se tutti i suoi nodi sono distinti

Cammino (Path)

14

Esempio:

$\{a, b, c, e, d\}$ è un cammino semplice nel grafo di lunghezza 4.



Grafi, la specifica

Graph()	Costruisce un nuovo grafo vuoto
Set V()	Restituisce l'insieme di tutti i nodi
int size()	Restituisce il numero di nodi
Set adj(Node u)	Restituisce l'insieme dei nodi adiacenti a u

Grafi, la specifica

insertNode(Node u)	Aggiunge il nodo u al grafo
insertEdge(Node u, Node v)	Aggiunge un arco da u a v
deleteNode(Node u)	Rimuove il nodo u
deleteEdge(Node u, Node v)	Rimuove l'arco tra u e v

Questa parte della specifica è meno usata (principalmente solo insertNode e insertEdge), perché di solito gli algoritmi sui grafi li caricano in memoria, senza poi modificarli.

Memorizzazione dei grafi

Normalmente si utilizzano due possibili approcci:

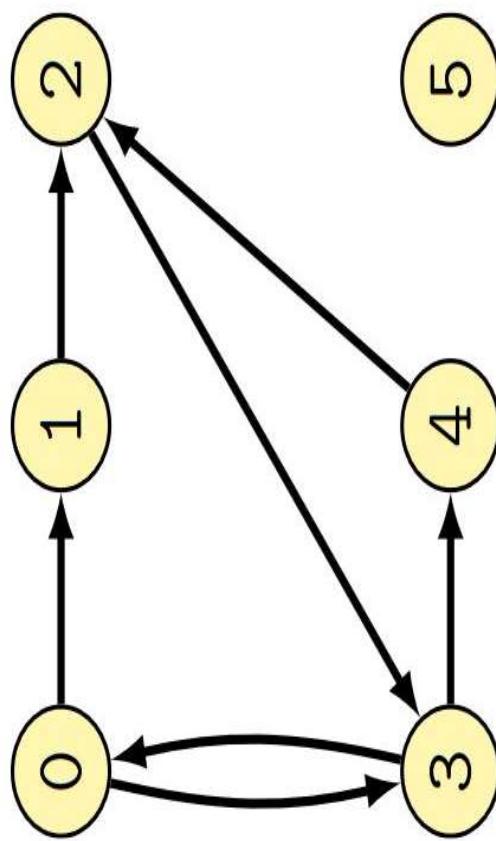
- Matrici di adiacenza
- Liste di adiacenza

Matrici di adiacenza: grafi orientati

18

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

Spazio = n^2 bit



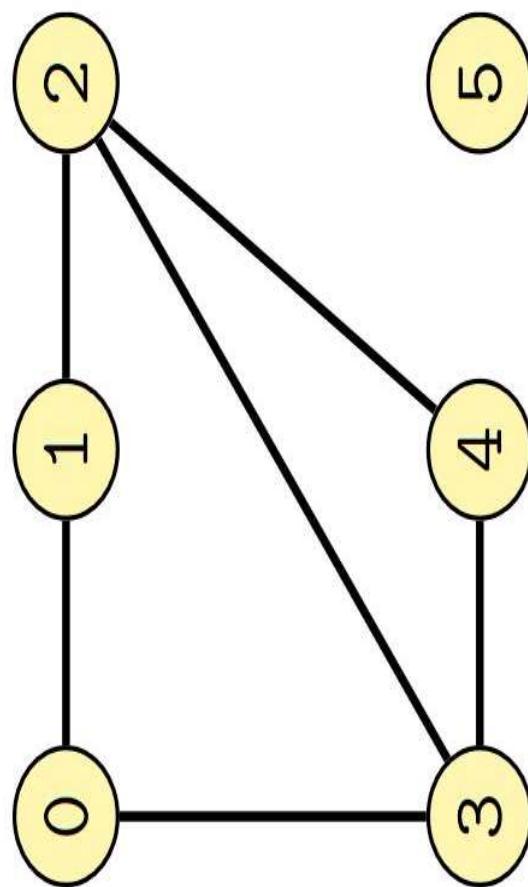
$$\begin{matrix} & & & 5 & 4 & 3 & 2 & 1 & 0 \end{matrix} \left(\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \right)$$

Matrici di adiacenza: grafi non orientati

19

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

Spazio = n^2 oppure $n(n - 1)/2$ bit

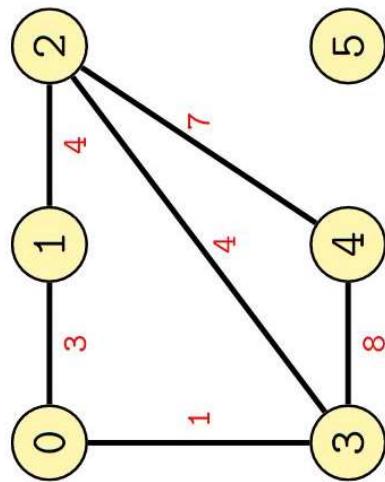


$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 & 1 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Matrici di adiacenza: grafici pesati

Gli archi possono avere un peso (che rappresenta il costo, il profitto, etc. in base al problema)

- Il peso è dato da una funzione di peso $w: V \times V \rightarrow \mathbb{R}$
- Se non esiste arco fra due vertici, il peso assume un valore che dipende dal problema, ad esempio 0 oppure $+\infty$

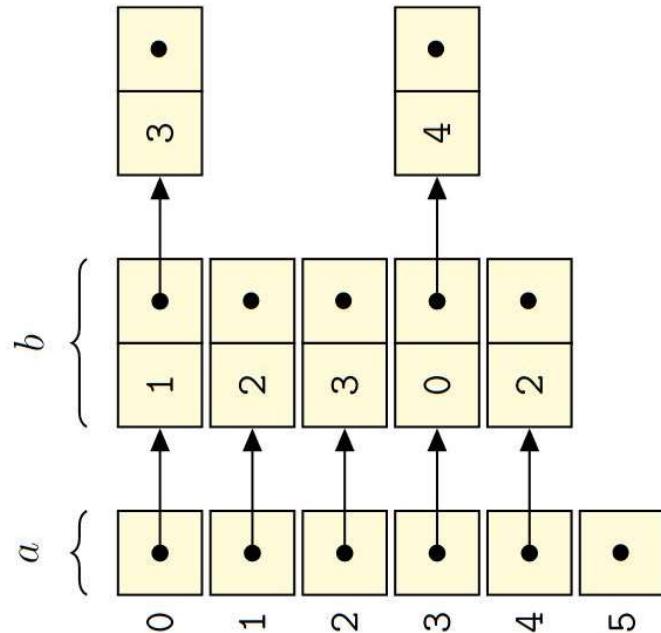
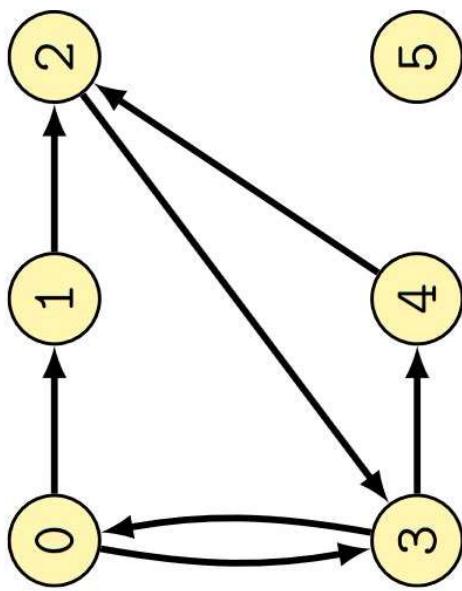


$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 3 & 0 & 1 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 7 & 0 & 0 & 0 & 0 \\ 4 & 8 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Liste di adiacenza: grafi orientati

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

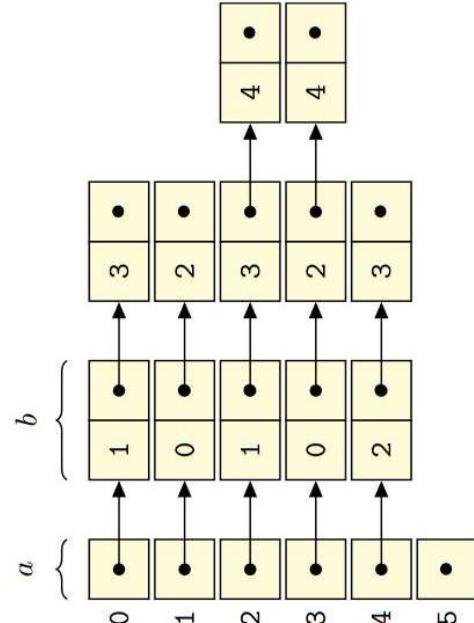
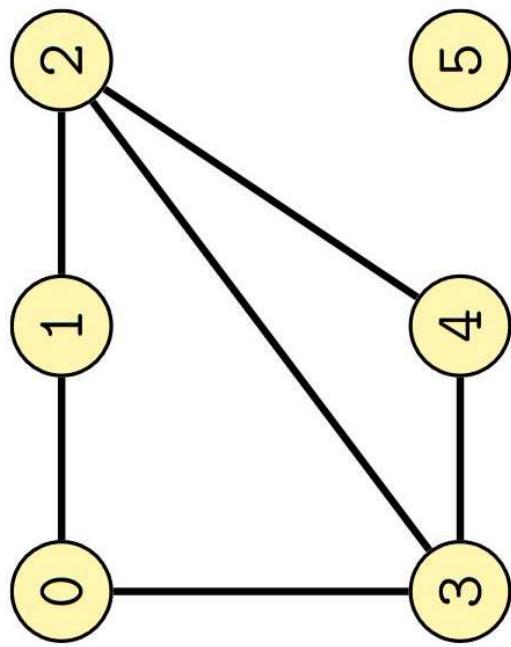
$$\text{Spazio} = an + bm \text{ bit}$$



Liste di adiacenza: grafi non orientati

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

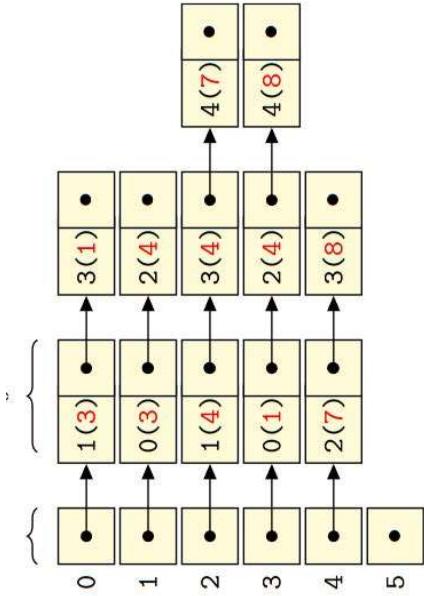
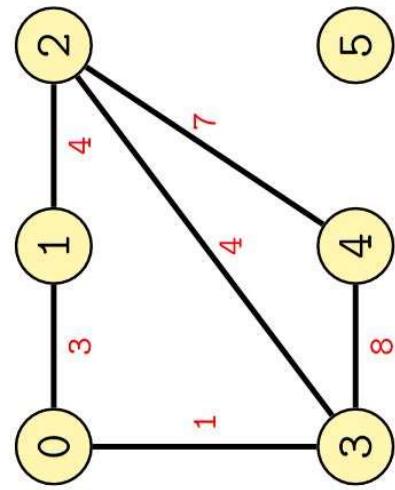
$$\text{Spazio} = an + 2 \cdot bm$$



Liste di adiacenza: grafi pesati

Gli archi possono avere un peso (che rappresenta il costo, il profitto, etc. in base al problema)

- Il peso è dato da una funzione di peso $w: V \times V \rightarrow R$, che va aggiunto al nodo
- Se non esiste arco fra due vertici, il secondo non compare tra gli adiacenti del primo



Iterazione sui grafi

Iterazione su tutti i nodi del grafo

```
foreach  $u \in G.V()$  do  
    { Esegui operazioni sul nodo  $u$  }
```

Costo computazionale $O(n)$ per iterare sui nodi, ma il costo effettivo dipende anche dall'operazione che eseguiamo su ogni nodo (se anche quella è $O(n)$, complessivamente è $O(n^2)$).

Iterazione sui grafi

Iterazione su tutti i nodi e archi del grafo

```
foreach u ∈ G.V() do
    { Esegui operazioni sul nodo u }
        foreach v ∈ G.adj(u) do
            { Esegui operazioni sull'arco (u, v) }
```

Costo computazionale $O(m+n)$ con liste di adiacenza (ci sono m archi e la lista di u contiene solo gli archi che **partono** da u),
 $O(n^2)$ con matrici di adiacenza

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
- 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

Liste di adiacenza

Assunzioni valide da questo punto in poi

- Se non specificato diversamente, l'implementazione è basata su liste di adiacenza
- L'accesso alle informazioni su un nodo avrà costo costo $O(1)$, ad esempio usando un vettore (nodi con valori int) o un dizionario (nodi con altri tipi)
- Le operazioni per aggiungere nodi e archi hanno costo $O(1)$, ossia il caricamento in memoria del grafo non influisce sull'algoritmo in sé
- Dopo l'inizializzazione il grafo è statico

Visite dei grafi

Il problema, stavolta nei grafi

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ detto **radice o sorgente**, visitare *una e una sola volta* tutti i nodi del grafo che possono essere raggiunti attraverso un cammino che parte da r .

Visita in profondità (DFS)

Visita ricorsiva: per ogni nodo adiacente alla radice, si visita tale nodo, visitando ricorsivamente i suoi nodi adiacenti, etc. Come per gli alberi, la visita può essere in pre-ordine o post-ordine.

- Applicazione: ordinamento topologico, componenti connesse, componenti fortemente connesse.

Visita in ampiezza (BFS)

Visita dei nodi per livelli: prima si visita la radice, poi i nodi a distanza 1 dalla radice, poi a distanza 2, etc.

- Applicazione: calcolare i cammini più brevi da una singola sorgente

L'approccio ingenuo

Un'idea corretta, ma ingenua, potrebbe essere “visito ogni nodo, e per ogni nodo tutti i suoi archi”. Ma:

```
visit(GRAPH G)
foreach u ∈ G.V() do
    { visita nodo u }
    foreach v ∈ 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

Riciclare le visite degli alberi

Potremmo riutilizzare gli algoritmi già visti per le visite degli alberi, quindi chiamare ad. es. una BFS a partire dal nodo sorgente. I nodi adiacenti sono quindi trattati come figli.

BFSTraversal(GRAPH G, int r)

QUEUE $Q = \text{Queue}()$

$Q.\text{enqueue}(r)$

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

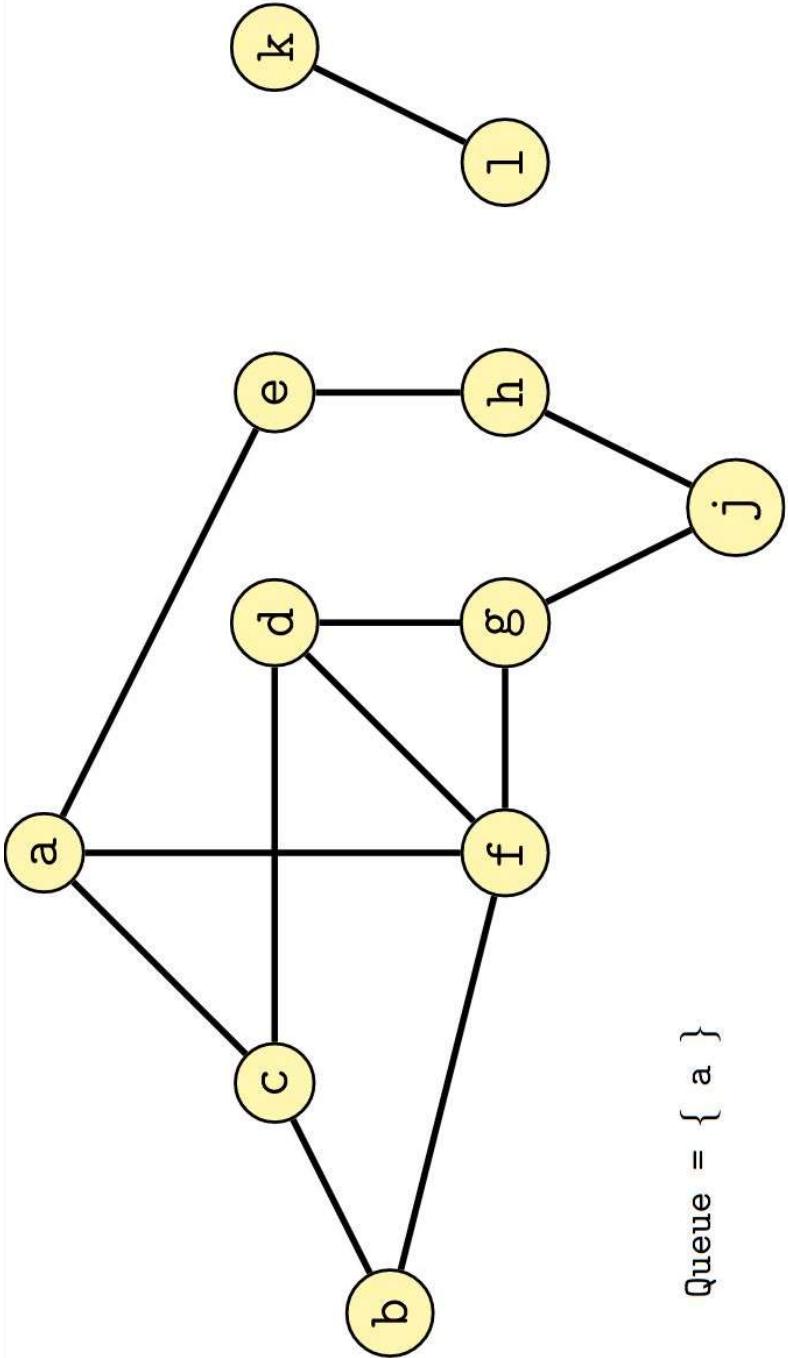
 NODE $u = Q.\text{dequeue}()$

 { visita il nodo u }

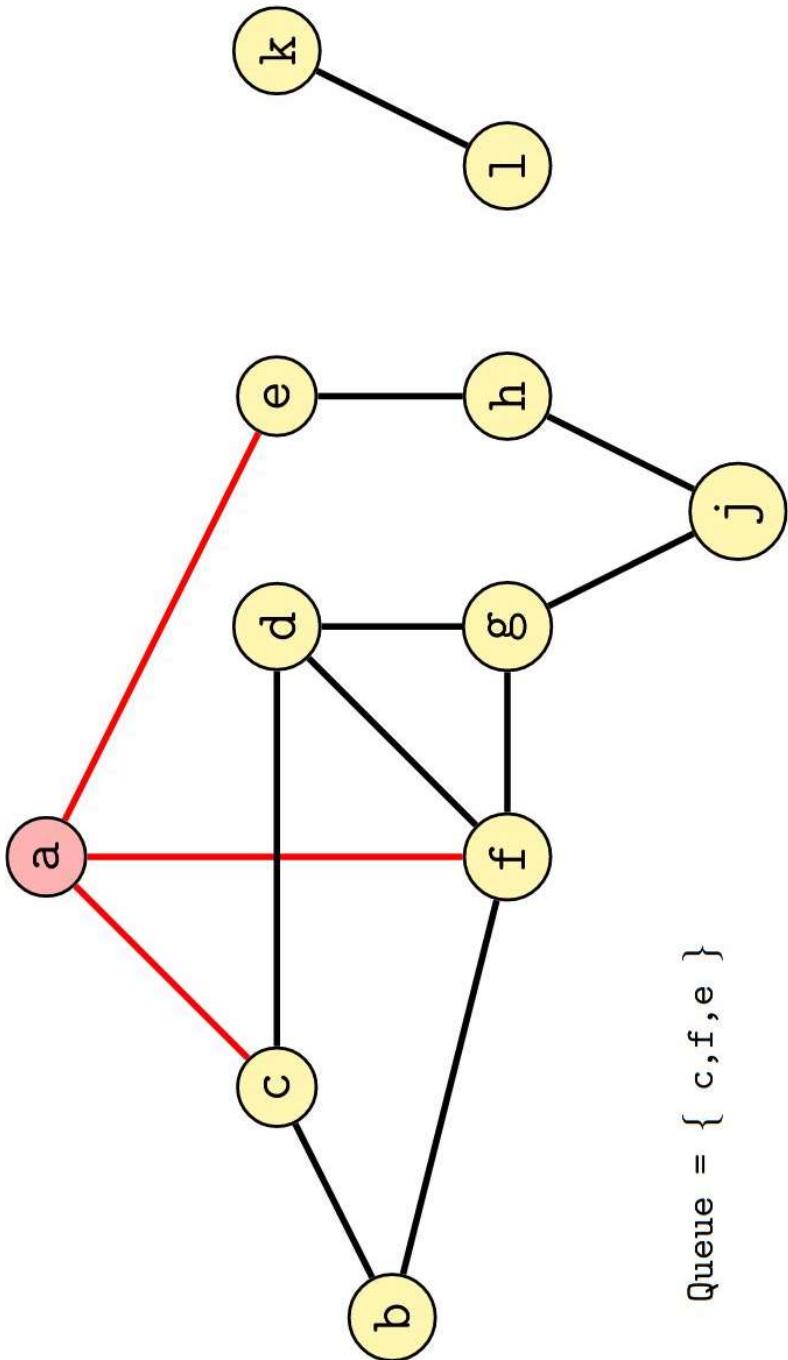
 foreach $v \in G.\text{adj}(u)$ do

$Q.\text{enqueue}(v)$

Il problema della ripetizione

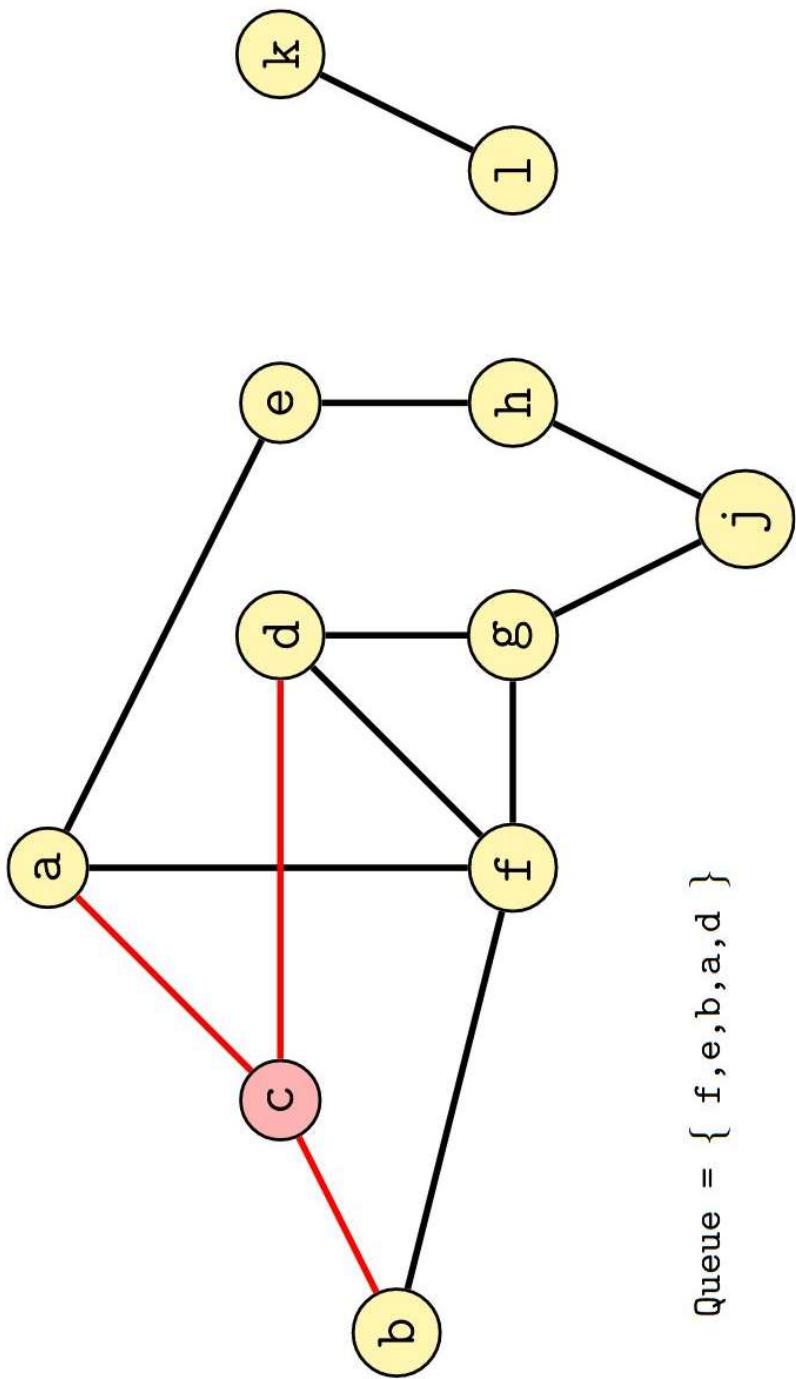


Il problema della ripetizione



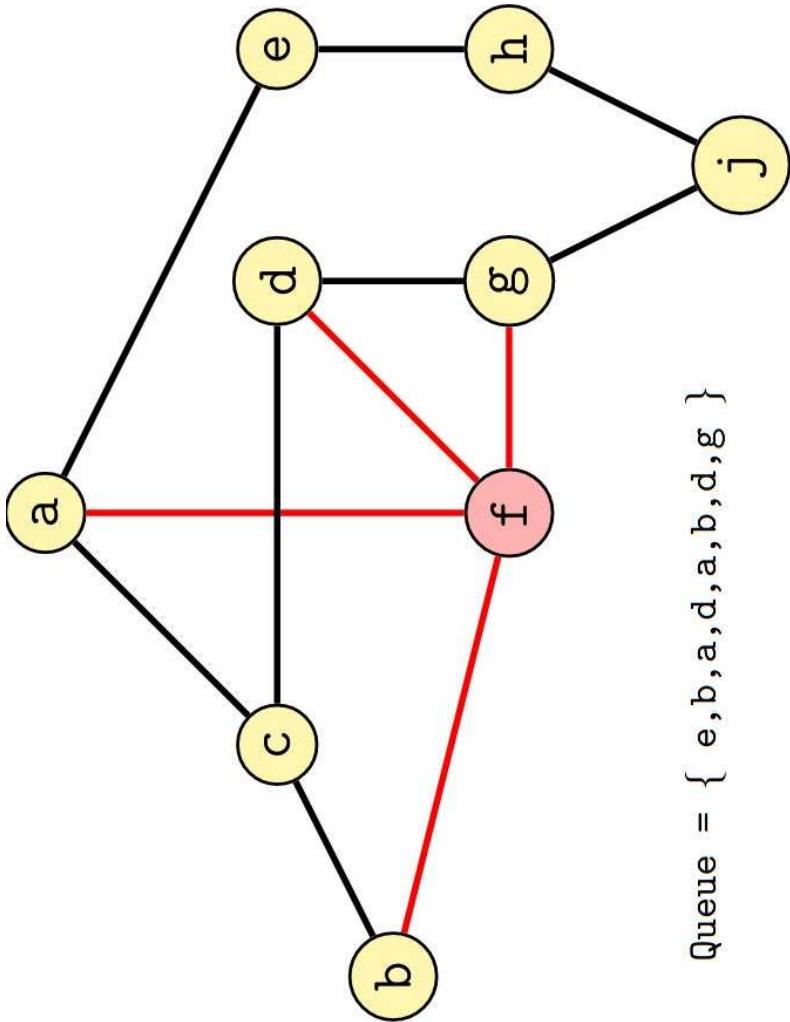
Queue = { c, f, e }

Il problema della ripetizione



Queue = { f, e, b, a, d }

Il problema della ripetizione



Queue = { e, b, a, d, a, b, d, g }

Soluzione: marcare i nodi visitati

Ogni volta che attraversiamo un nodo, segnamo (nel nodo, o esternamente) che esso è già stato attraversato.

Attraversamento generico di un grafo

39

graph Traversal(GRAPH G, NODE r)

SET S = Set()

S.insert(*r*)
{ **marca il nodo *r*** }

while S.size() > 0 do

 NODE *u* = S.remove()
 { **visita il nodo *u*** }

 foreach *v* ∈ *G*.adj(*u*) do

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

 if *v* non è ancora stato marcato then

 { **marca il nodo *v*** }

 S.insert(*v*)

Note:

- *S* è un insieme generico,
bisogna specificare di volta in
volta che struttura è (Stack,
Queue, altro...)
 - Le operazioni *S.insert()* e
S.remove() andranno adattate
di conseguenza.
-

BFS, seriamente

Tre obiettivi:

1. **Visitare i nodi** a distanze crescenti dalla sorgente:
 - Visitare i nodi a distanza 1, poi visitare i nodi a distanza 2...
2. Calcolare il **cammino più breve da r** a tutti gli altri nodi
 - Le distanze sono misurate come il numero di archi attraversati
3. Generare un **albero breadth-first**
 - Generare un albero contenente tutti i nodi raggiungibili da r , tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Breadth-first search, lo pseudocodice

41

```
graph Traversal(GRAPH G, NODE r)
SET S = Set()
S.insert(r)
{ marca il nodo r }
while S.size() > 0 do
    NODE u = S.remove()
    { visita il nodo u }
    foreach v ∈ G.adj(u) do
        { visita l'arco (u,v) }
        if v non è ancora stato marcato then
            { marca il nodo v }
            S.insert(v)
```

```
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)
```

1. Visitare i nodi a distanza crescente

42

```
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)
```

- Set è diventato Queue

- Memorizziamo i nodi visitati in un vettore di booleani (accesso in $O(1)$) dove inizialmente ogni nodo è falso (da visitare)
- Dopo avere visitato un nodo, il valore nel vettore è true

2. Calcolare i cammini più brevi

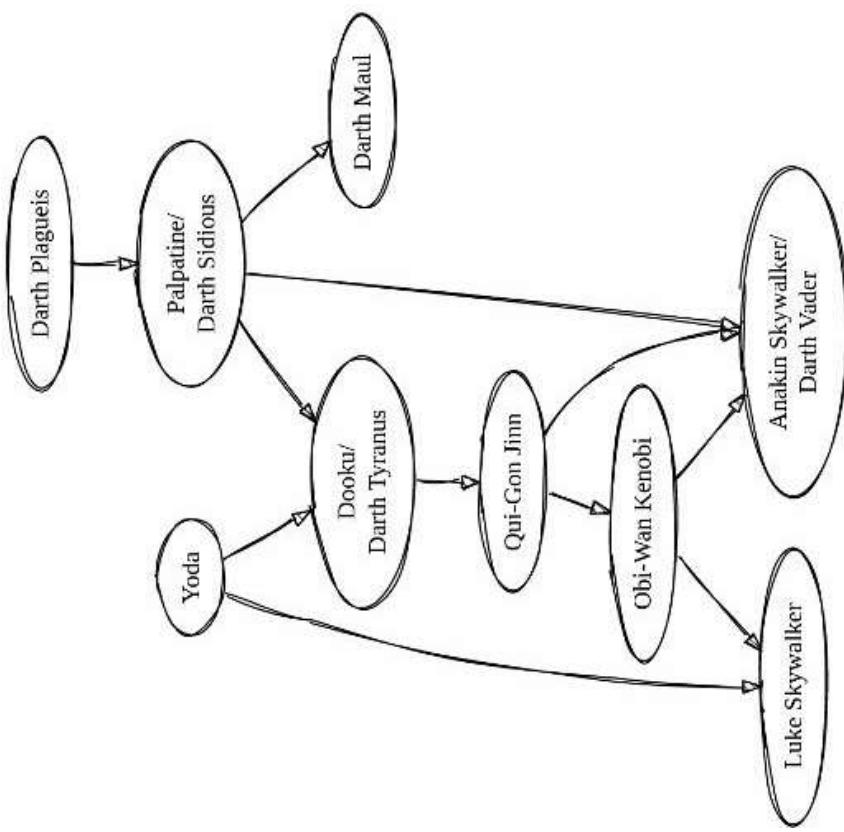
La teoria dei sei gradi di separazione è un'ipotesi secondo la quale ogni persona può essere collegata a qualunque altra persona o cosa attraverso una catena di conoscenze e relazioni con non più di 5 intermediari.

Definiamo d la distanza a partire da un individuo X

- La persona X ha valore $d=0$
- Le persone direttamente connesse con X hanno $d=1$
- Se una persona T è vicina a qualche nodo con $d=k$ e non è vicina a qualcuno con $d < k$, allora T ha $d=k+1$

Le persone non raggiunte da questa definizione hanno $d=+\infty$  Innovation and knowledge

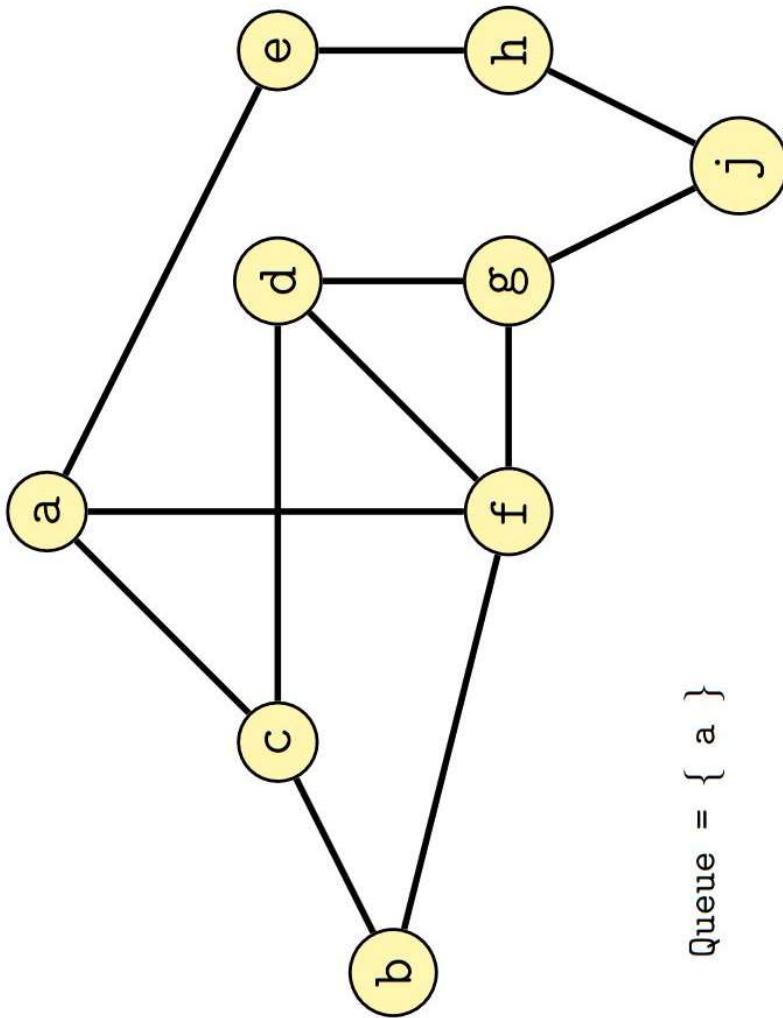
Six degrees of Star Wars



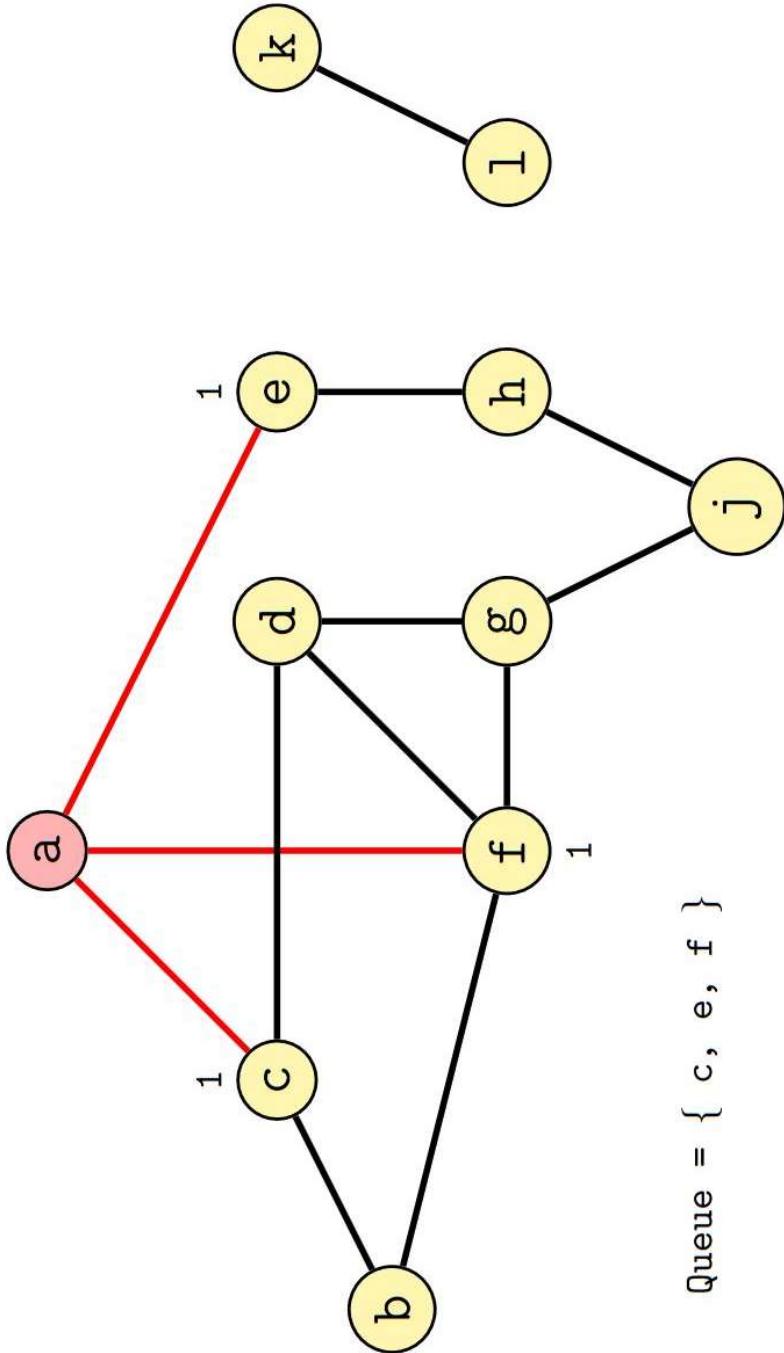
Distanza dalla radice, lo pseudocodice

```
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
            distance[v] = distance[u] + 1
            Q.enqueue(v)
```

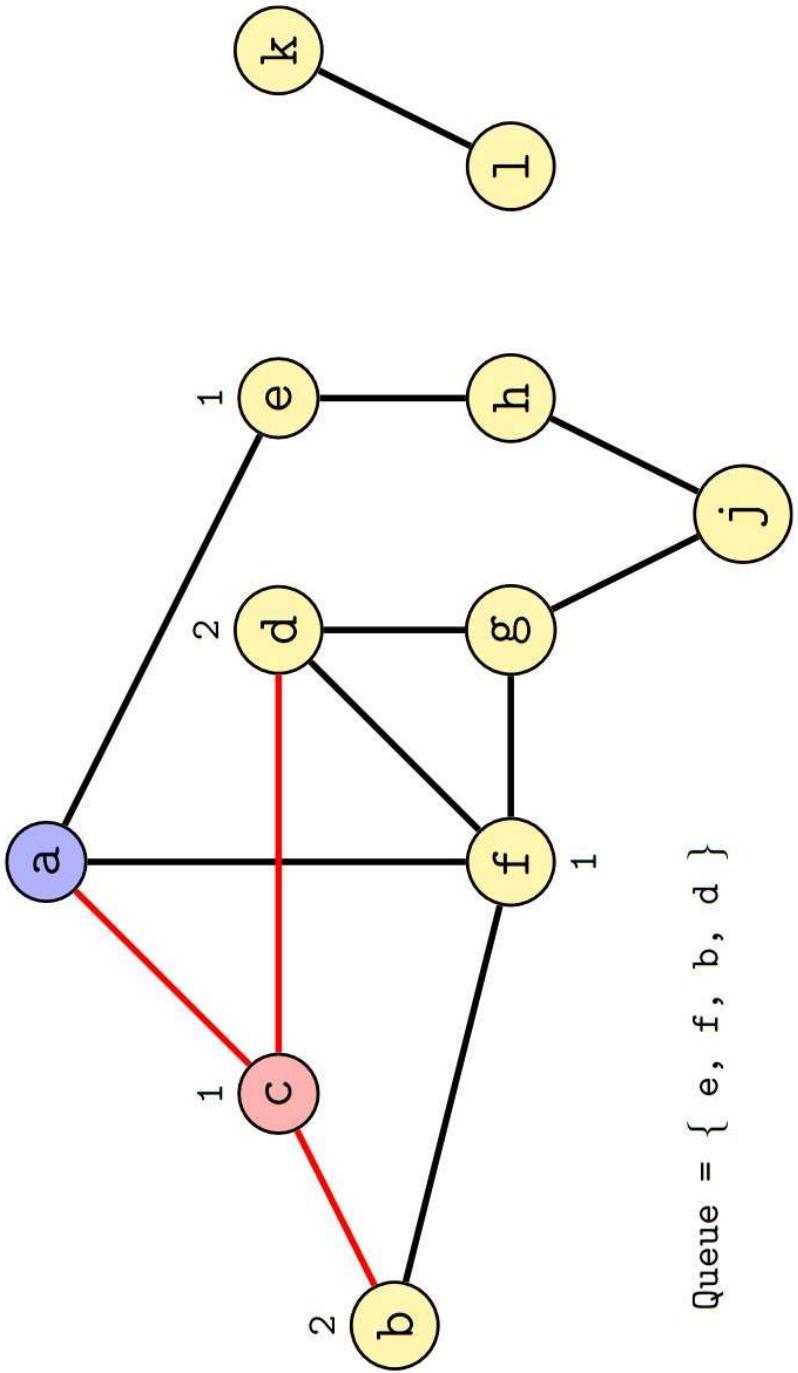
Esempio



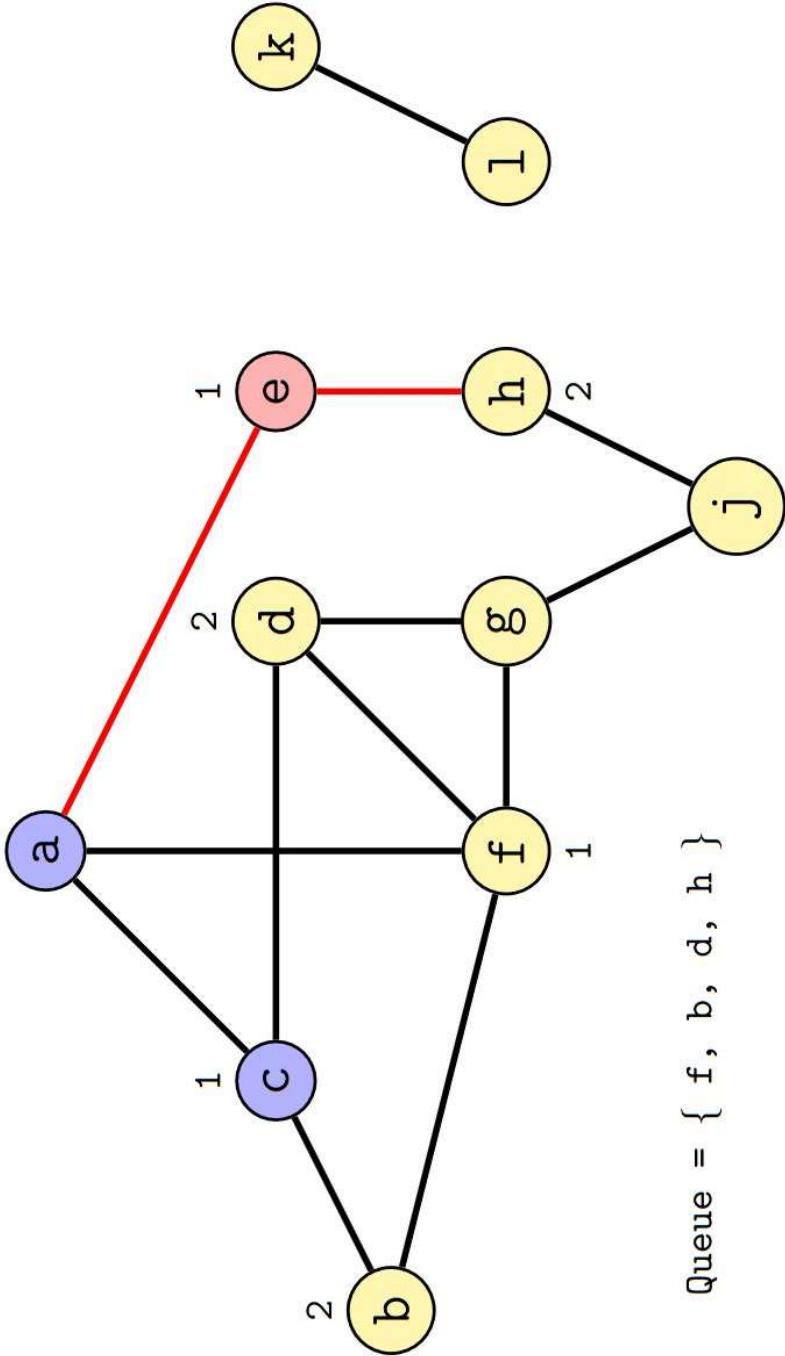
Esempio



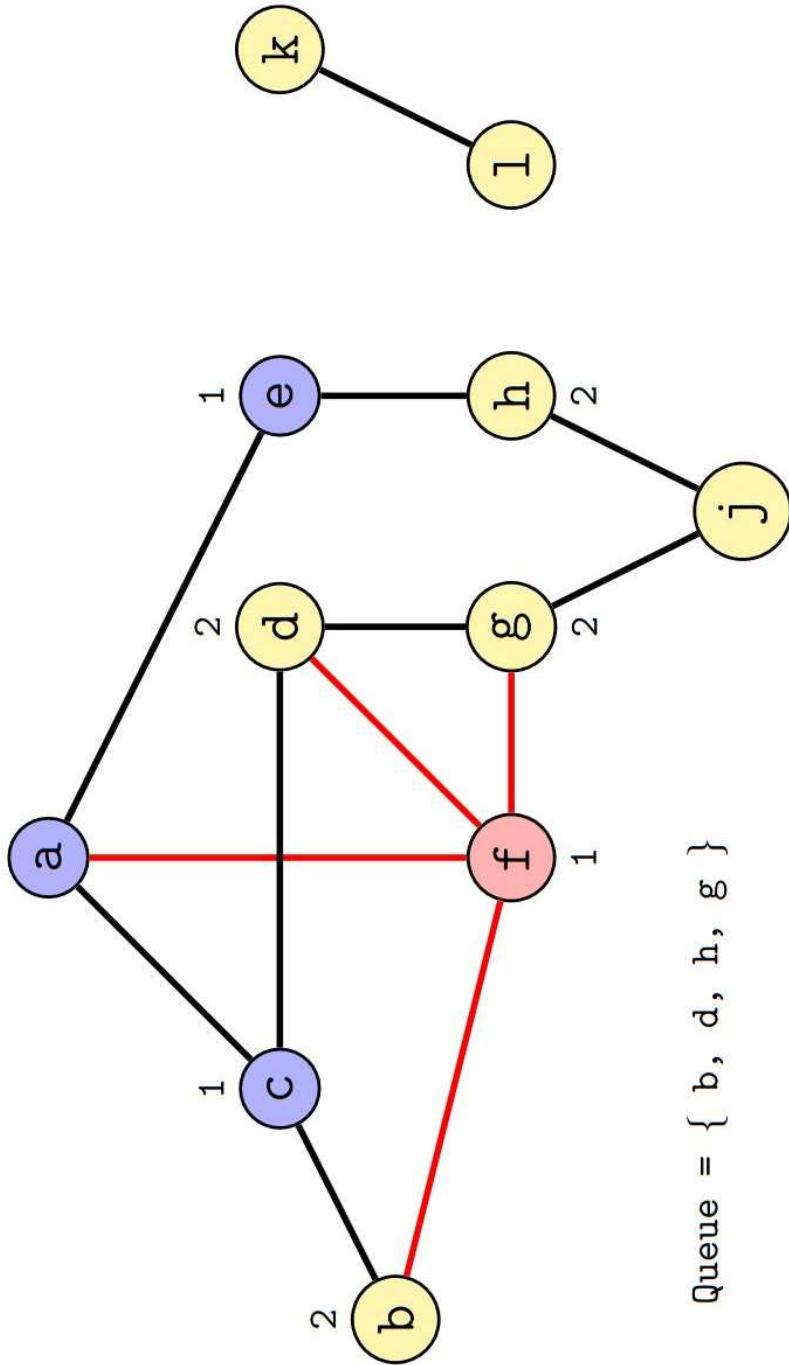
Esempio



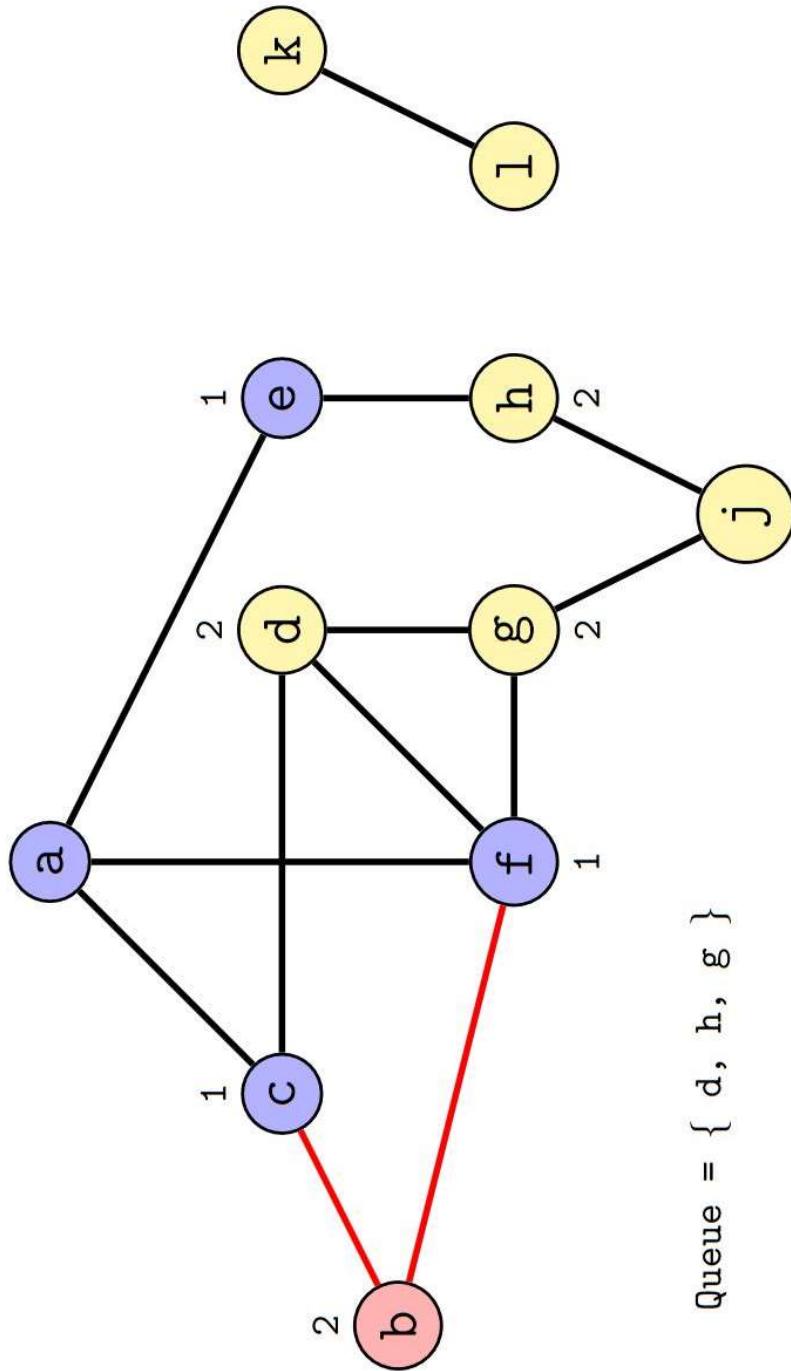
Esempio



Esempio

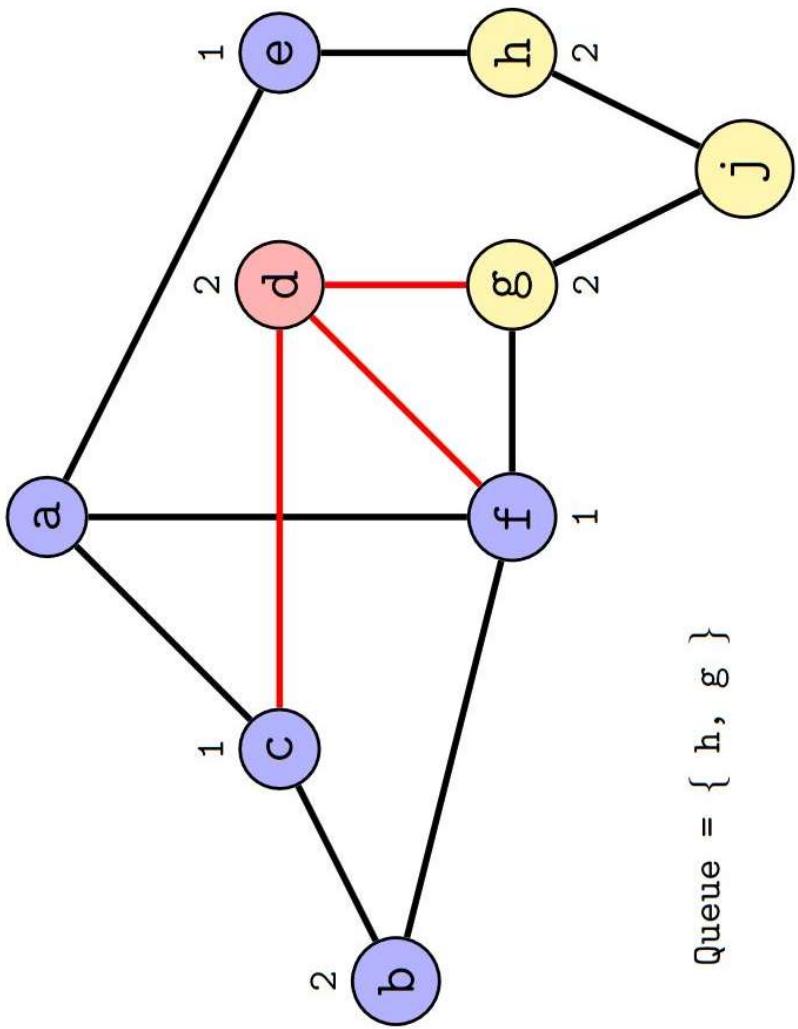


Esempio



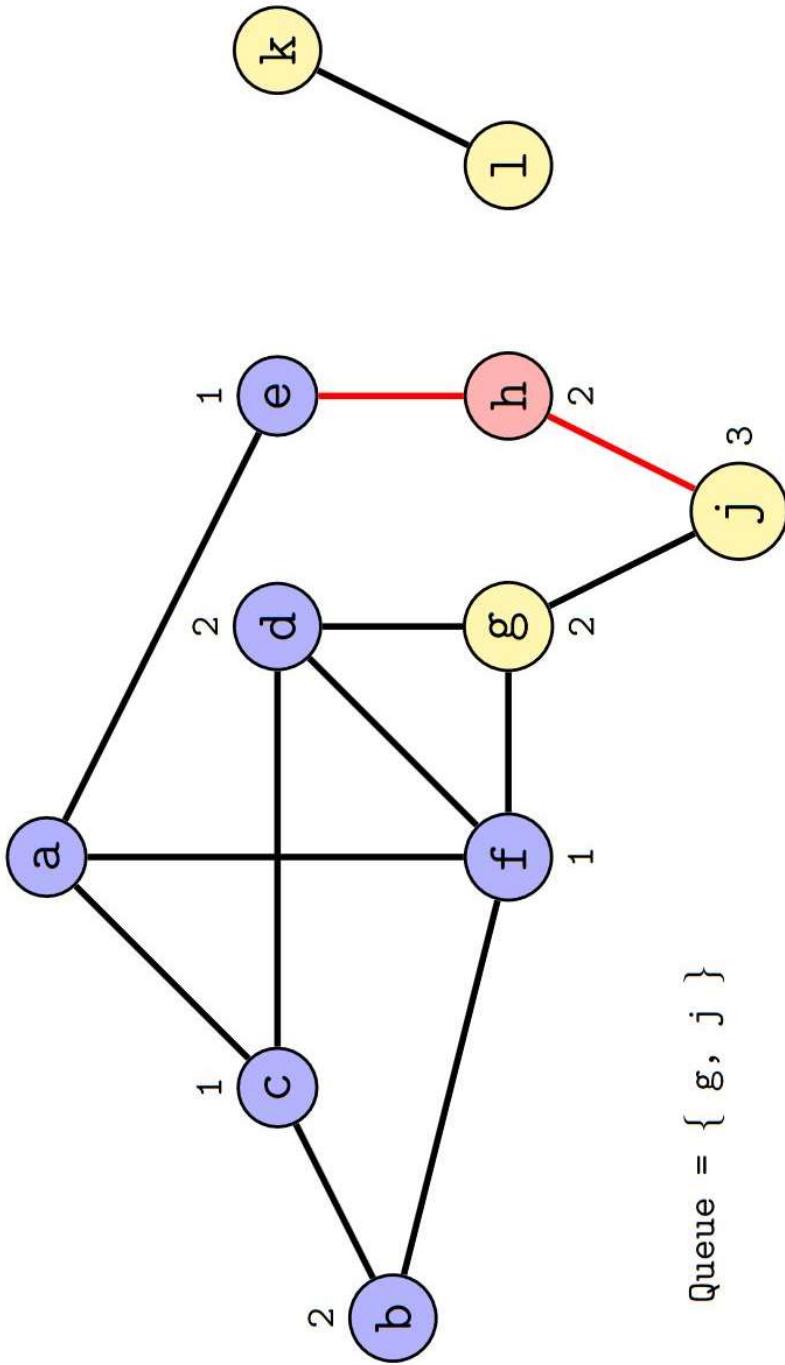
Queue = { d, h, g }

Esempio



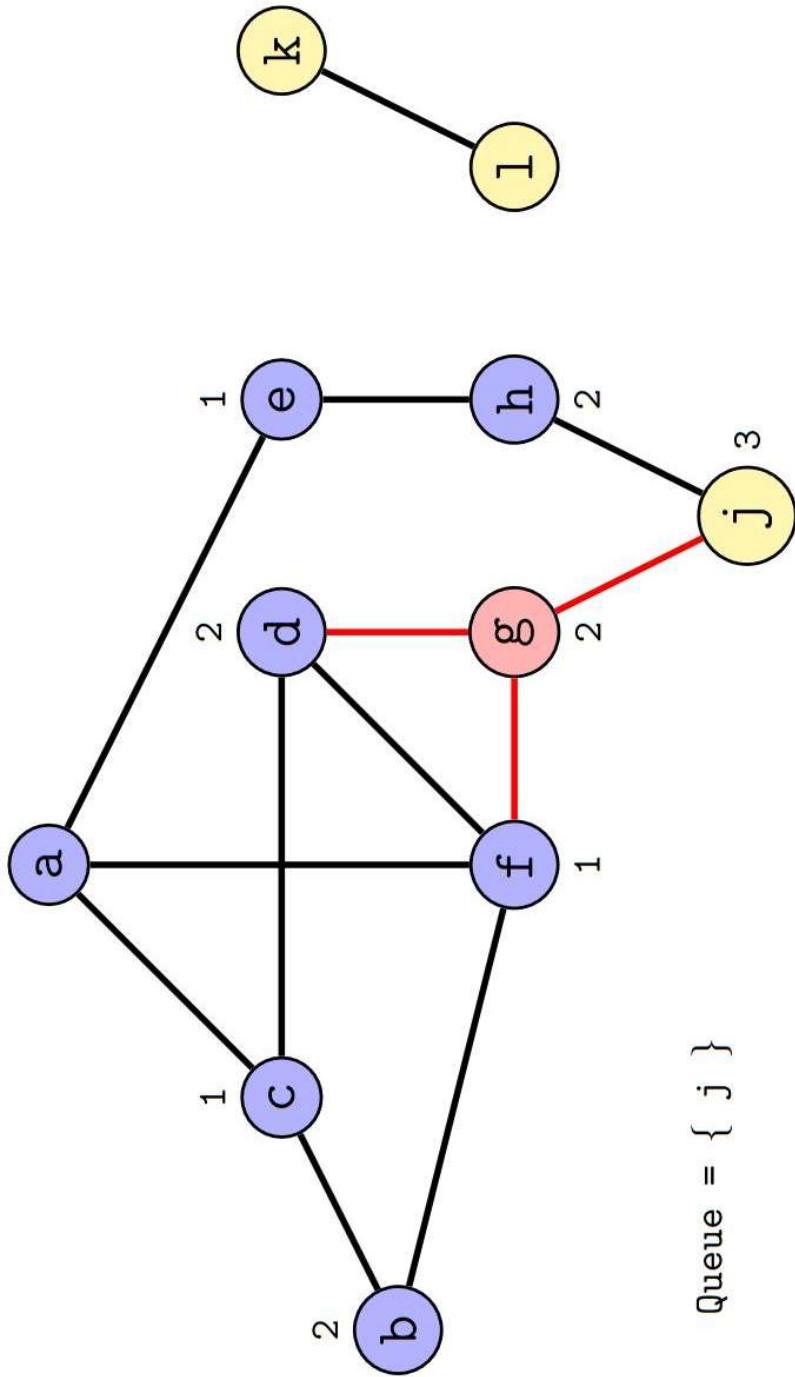
Queue = { h, g }

Esempio

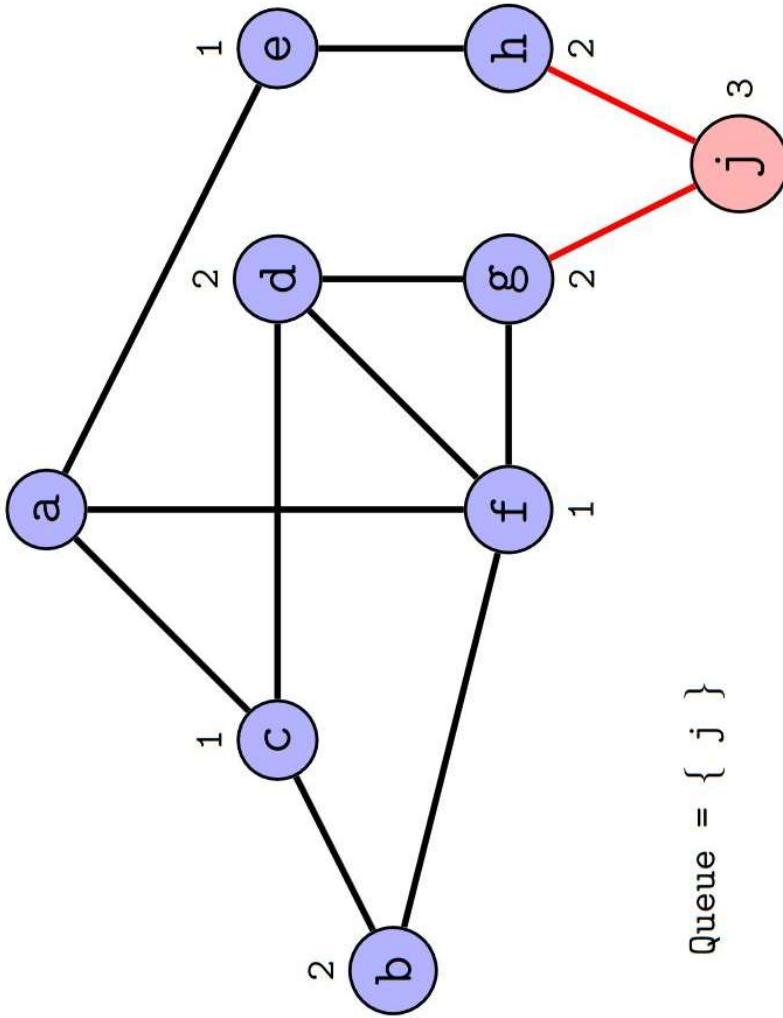


Queue = { g, j }

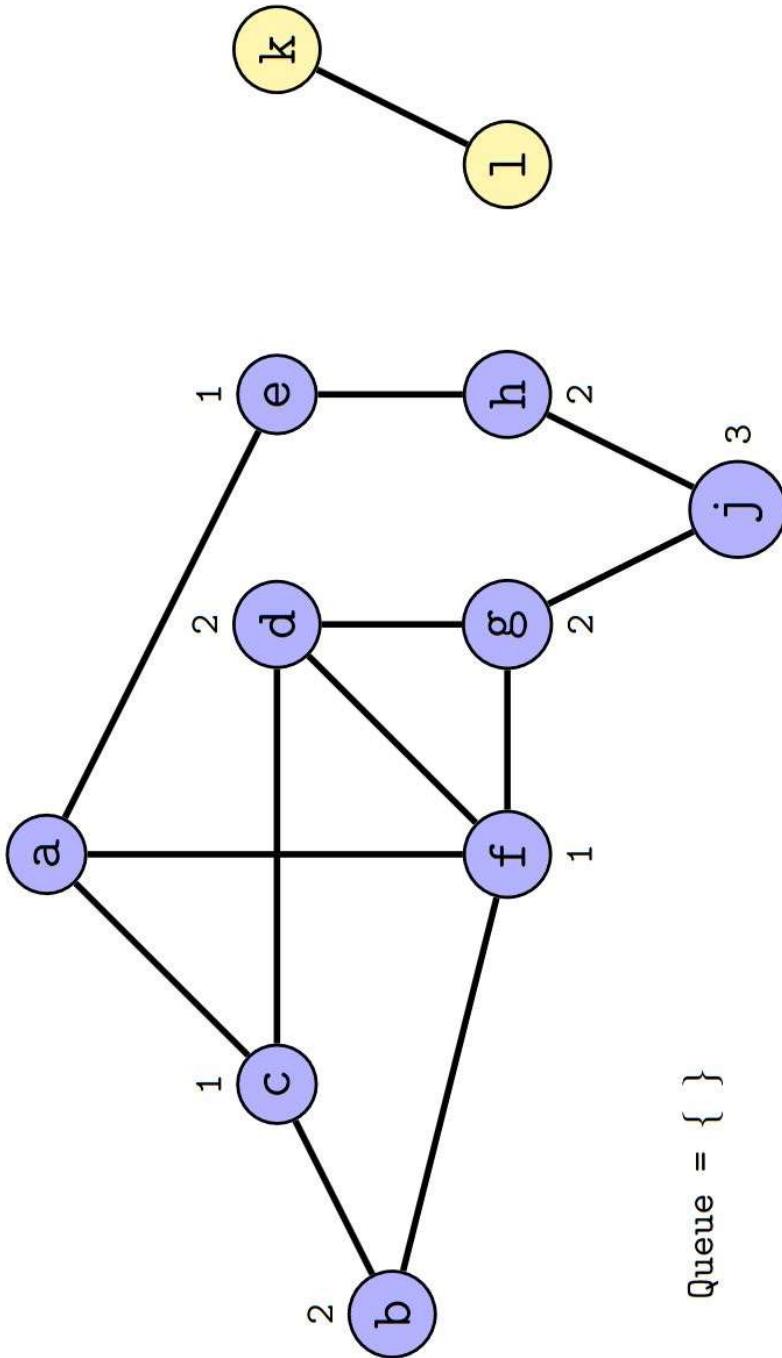
Esempio



Esempio



Esempio



3. Generare un albero breadth-first

La visita BFS può essere usata per ottenere il cammino più breve fra due nodi (misurato in numero di archi)

Possiamo creare un “Albero di copertura” con radice r, che memorizziamo in un vettore dei padri parent

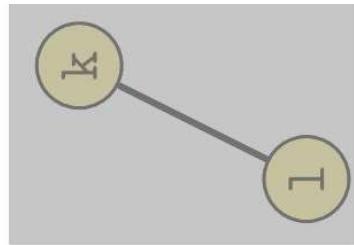
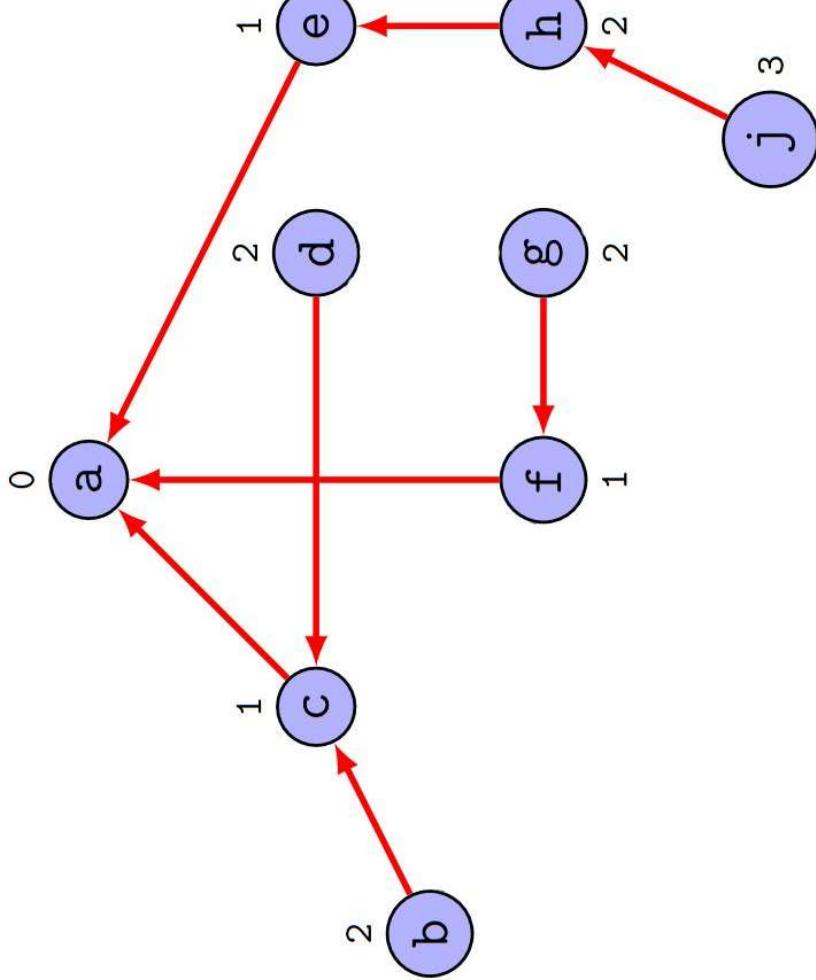
$\text{distance}([\dots], \text{NODE}[] \text{ parent})$

[...]

$\text{parent}[r] = \text{nil}$

```
while not  $S.\text{isEmpty}()$  do
    NODE  $u = S.\text{dequeue}()$ 
    foreach  $v \in G.\text{adj}(u)$  do
        if  $distance[v] == \infty$  then
             $distance[v] = distance[u] + 1$ 
             $\text{parent}[v] = u$ 
             $S.\text{enqueue}(v)$ 
```

L'albero di copertura dell'esempio

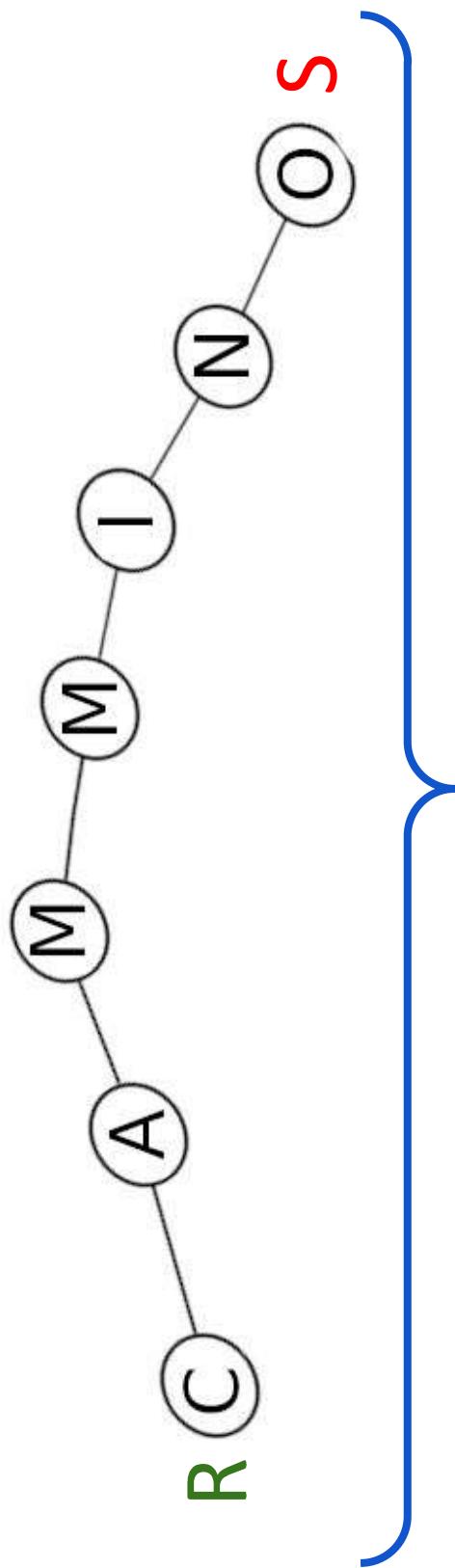


Come stampare un albero, se ho il genitore?

Ci serve una funzione che stampi il percorso da un nodo r a un nodo s : proviamo a ragionare ricorsivamente.

- **Caso base:** se ho un percorso composto da un solo nodo, lo stampo;
- **Passo induttivo:** se ho un percorso più lungo, posso stampare il percorso fino al nodo precedente a s , e poi stampare s .

Esempio



Come stampare un albero, lo pseudocodice

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

BFS, complessità computazionale

- Ognuno degli n nodi viene inserito nella coda al massimo una volta.
- Ogni volta che un nodo viene estratto, tutti i suoi archi vengono analizzati una volta sola.
- Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} d_{out}(u)$$

dove d_{out} è l'out-degree del nodo u .

- La complessità è quindi $O(m+n)$

Esercizi

- Implementare la BFS
- Dato un grafo non orientato, trovare il cammino minimo di lunghezza massima tra ogni coppia di nodi del grafo, ovvero il più lungo percorso minimo tra due nodi.
Suggerimento: bisogna trovare il massimo tra tutti i percorsi tra tutte le coppie di nodi.



- In un grafo orientato G , dati due nodi s e v , si dice che:
 - v è **raggiungibile** da s se esiste un cammino da s a v ;
 - la **distanza** di v da s è **la lunghezza del più breve cammino** da s a v (misurato in numero di archi), oppure $+\infty$ se v non è raggiungibile da s
- Scrivere un algoritmo che prenda in input un grafo orientato $G = (V, E)$ e due nodi $s_1, s_2 \in V$, che restituisca il numero di nodi in V tali che:
 - siano raggiungibili sia da s_1 che da s_2 , e
 - si trovino alla stessa distanza da s_1 e da s_2 .
- Discutere la complessità dell'algoritmo proposto.

Ripasso delle differenze tra BFS e DFS

Breadth First Search (BFS)

- Visita per livelli, ne abbiamo appena discusso
- Richiede una queue

Depth-First Search (DFS)

- Per visitare un grafo si visitano ricorsivamente i nodi adiacenti
- Due (Tre) varianti: pre/post visita (pre/post order)
- Richiede uno stack di appoggio

Depth-First Search

66

A differenza di BFS, che viene usata “da sola” per trovare un percorso minimo tra due nodi, DFS è spesso utilizzata come subroutine della soluzione di altri problemi (cioè, il problema che sto cercando di risolvere richiede l’uso di una DFS).

Inoltre, DFS è utilizzata per esplorare un intero grafo, non solo i nodi raggiungibili da una singola sorgente.

DFS, ricorsiva, con stack implicito, pre-order

67

```
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: $O(m+n)$

Problema: Se il grafo è composto da molti nodi e da molti archi, lo stack delle chiamate ricorsive potrebbe esaurirsi.

Il problema dello stack

Eseguire una DFS basata su chiamate ricorsive e stack implicito, 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.

DFS, iterativa, con stack esplicito, pre-order

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

Problema: Un nodo può essere

inserito nella pila più volte, perché
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

Esercizi

- Dato un grafo orientato e un nodo sorgente, trovare il numero di nodi raggiungibili da quella sorgente.

