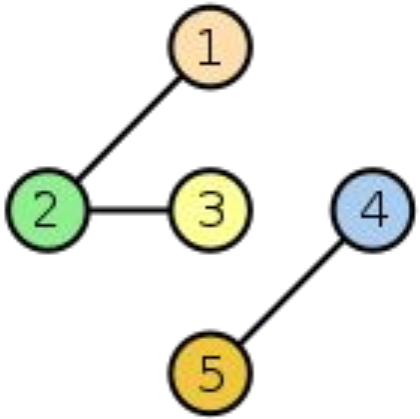
# Palestra di algoritmi

28/11/2023 - gruppo B

# Soluzioni esercizi

# Ponti

**Idea: devo trovare quante componenti connesse ci sono nel grafo. Il numero di ponti necessari sarà uguale al numero di componenti connesse meno uno.**




Esempio: 2 componenti connesse, basta 1 ponte per collegarle

Si fa partire una visita del grafo dal nodo 0, andando a segnare come visitati tutti i nodi che raggiungiamo. Si mette il contatore a 1.

Passiamo l’array dei nodi visitati, se ne troviamo uno non visitato, facciamo partire un’altra visita da quel nodo e mettiamo il contatore a 2.

Così via fin quando non visitiamo tutti i nodi.

```
for(int i=0;i<n;i++) {
	visitato[i]=false;
}

visitati=0;
int cont=0;

while(visitati!=n) {
	int indice;
	for(int i=0;i<n;i++) {
		if(!visitato[i]) {
			indice=i;
			break;
		}
	}
	walk(indice);
	cont++;
}
```

# Strutture dati: coda

# Coda

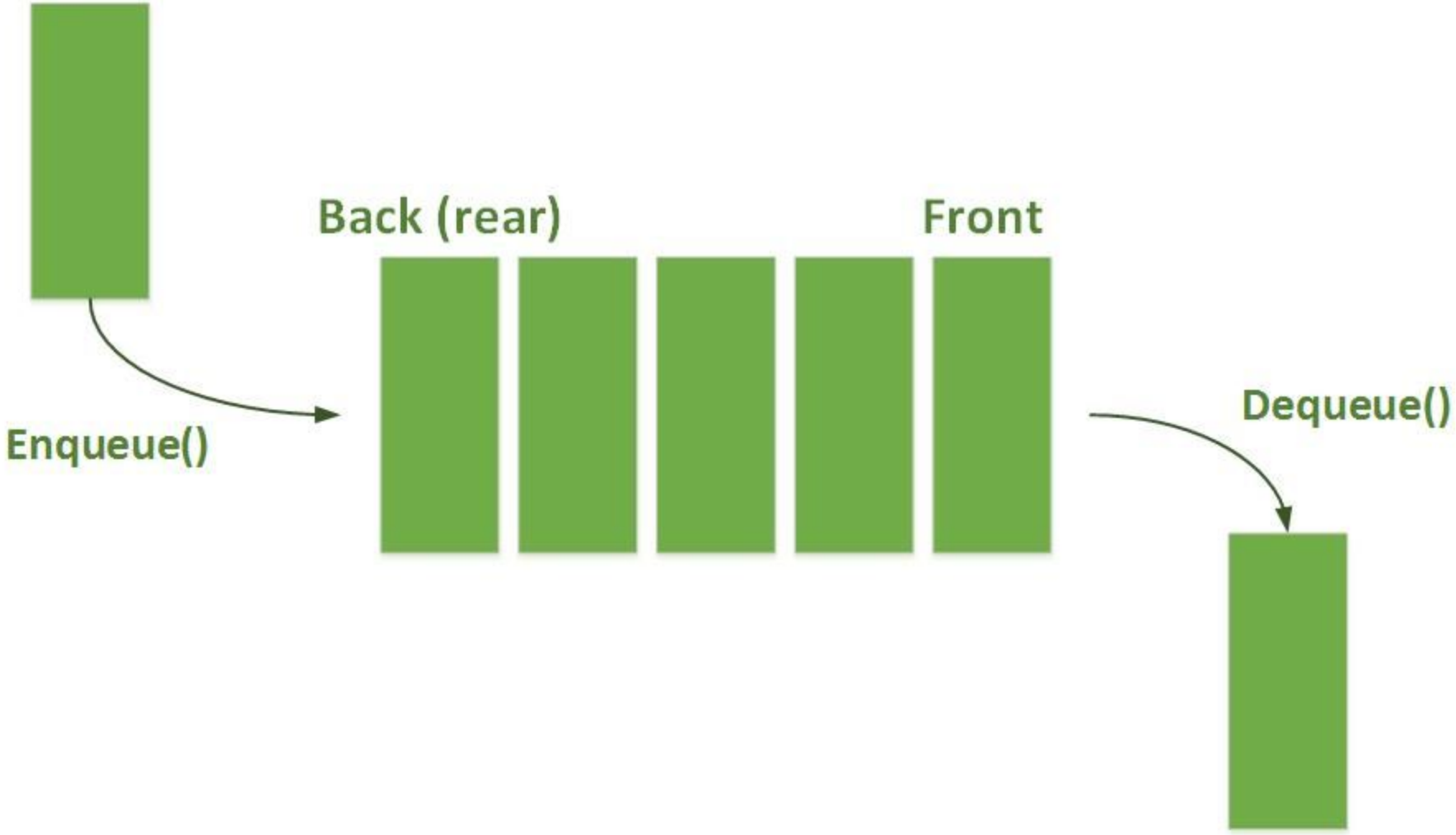
Le queue (code) sono un container di elementi.

Le code che useremo noi sono code FIFO = First In First Out.

Ciò significa che il primo elemento inserito nella coda sarà il primo ad uscirne.

Possiamo immaginare la coda come un tubo in cui si inseriscono gli elementi da un foro e poi si estraggono dall'altro.

Esistono anche strutture dati LIFO = Last In First Out, ossia in cui l'ultimo elemento inserito è l'ultimo estratto (immaginiamo di creare una torre di blocchi e poter aggiungere e togliere blocchi solo dalla cima).


Back (rear)
Front
Enqueue()
Dequeue()

# Queue in C++

```
queue<int> coda;
coda.push(10);
coda.push(20);
coda.push(30);

cout << “La coda contiene “ << coda.size() << “ elementi\n”;          //3
cout << “Il prossimo elemento estratto sarà: “ << coda.front() << “\n”;   //10
cout << “L’ultimo elemento inserito è: “ << coda.back() << “\n”;          //30

int elem = coda.pop();
cout << “Ho estratto l’elemento “ << elem << “\n”;                    //10
```

# BFS

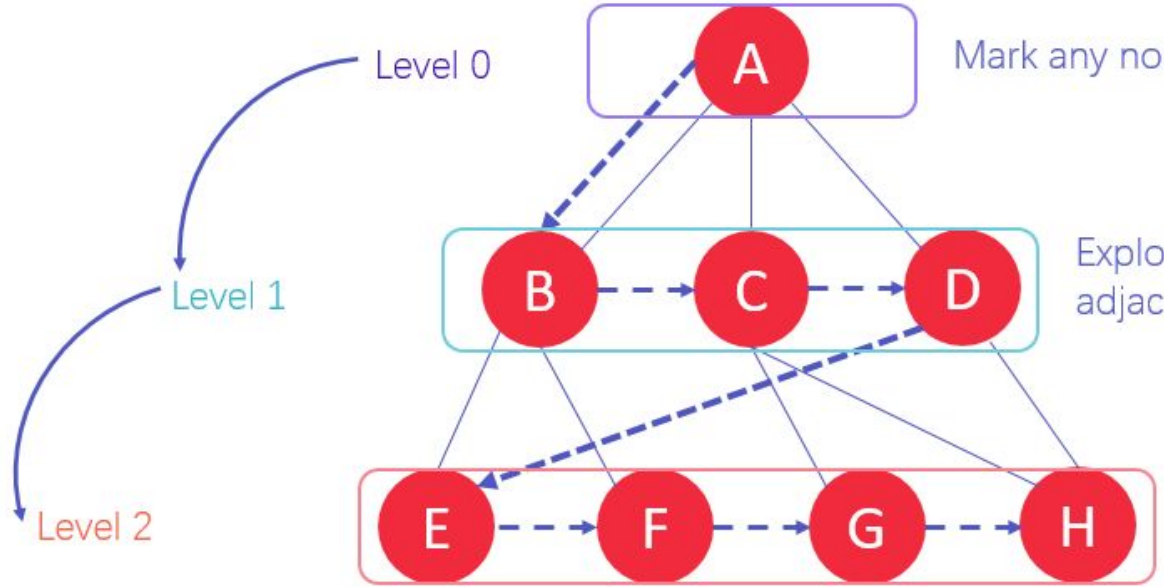
# BFS

BFS = Breadth First Search

La BFS è una visita “in ampiezza” di un grafo. Nel senso che partiamo da un nodo 0 e visitiamo tutti i suoi vicini diretti (a distanza di 1 arco dal nodo iniziale).

Quando abbiamo finito, visitiamo tutti i vicini dei vicini, ovvero i nodi a distanza di 2 archi.

# CONCEPT DIAGRAM



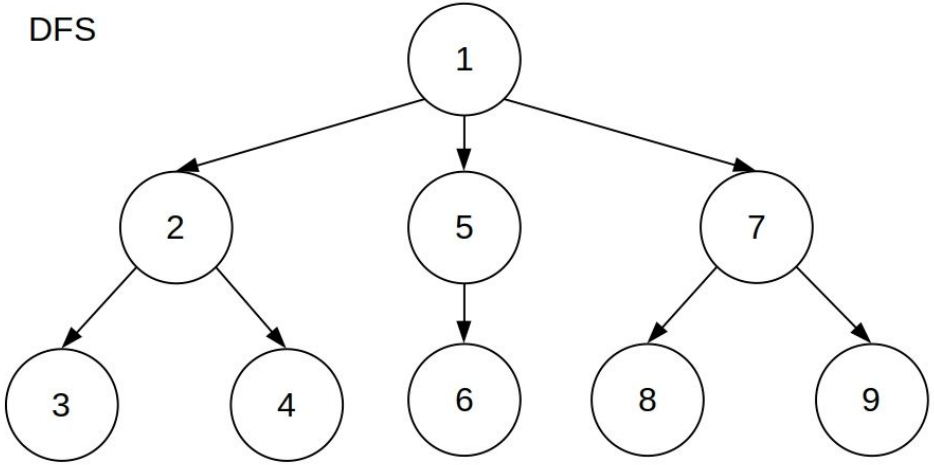

Mark any node as Starter or Initial 1

Explore and traverse un-visited nodes adjacent to starting node 2

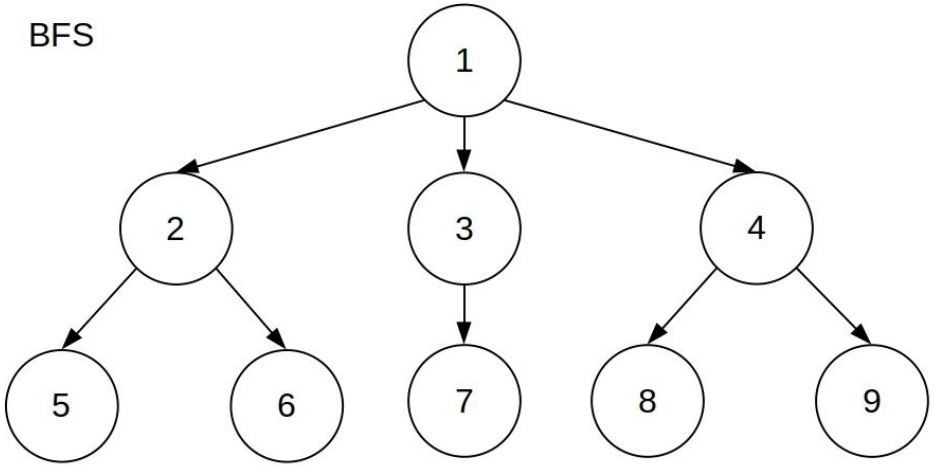
Mark Node as Completed and move to next adjacent and un-visited nodes 3

Guru99.com

DFS




BFS



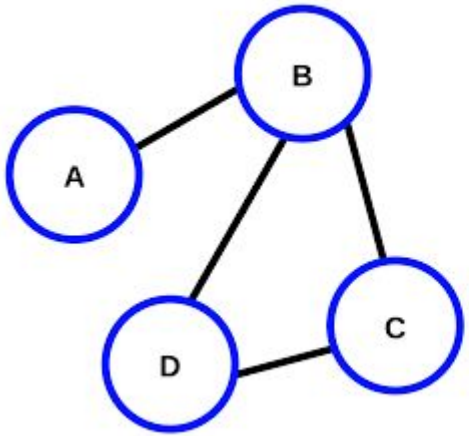
# In codice

```
queue<int> coda;
coda.push(nodoIniziale);

while(!coda.empty()) {              //finchè ci sono nodi da visitare in coda
        int nodoCorrente = coda.front();
        coda.pop();
        //visito il nodo corrente
        visitato[nodoCorrente] = true;
        for (int adiacente : grafo[nodoCorrente]) { //metto i suoi vicini in coda
                if (!visitato[adiacente]) {
                        coda.push(adiacente);
                }
        }
}
```

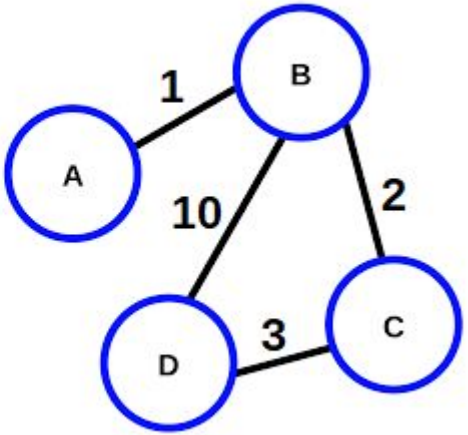
# Problema dei cammini minimi

Dato un grafo, trovare il cammino minimo da un nodo iniziale a un nodo finale.

Il cammino minimo può essere dato dal minimo numero di archi percorsi, in caso di grafo non pesato, oppure dalla somma minima dei pesi degli archi attraversati.



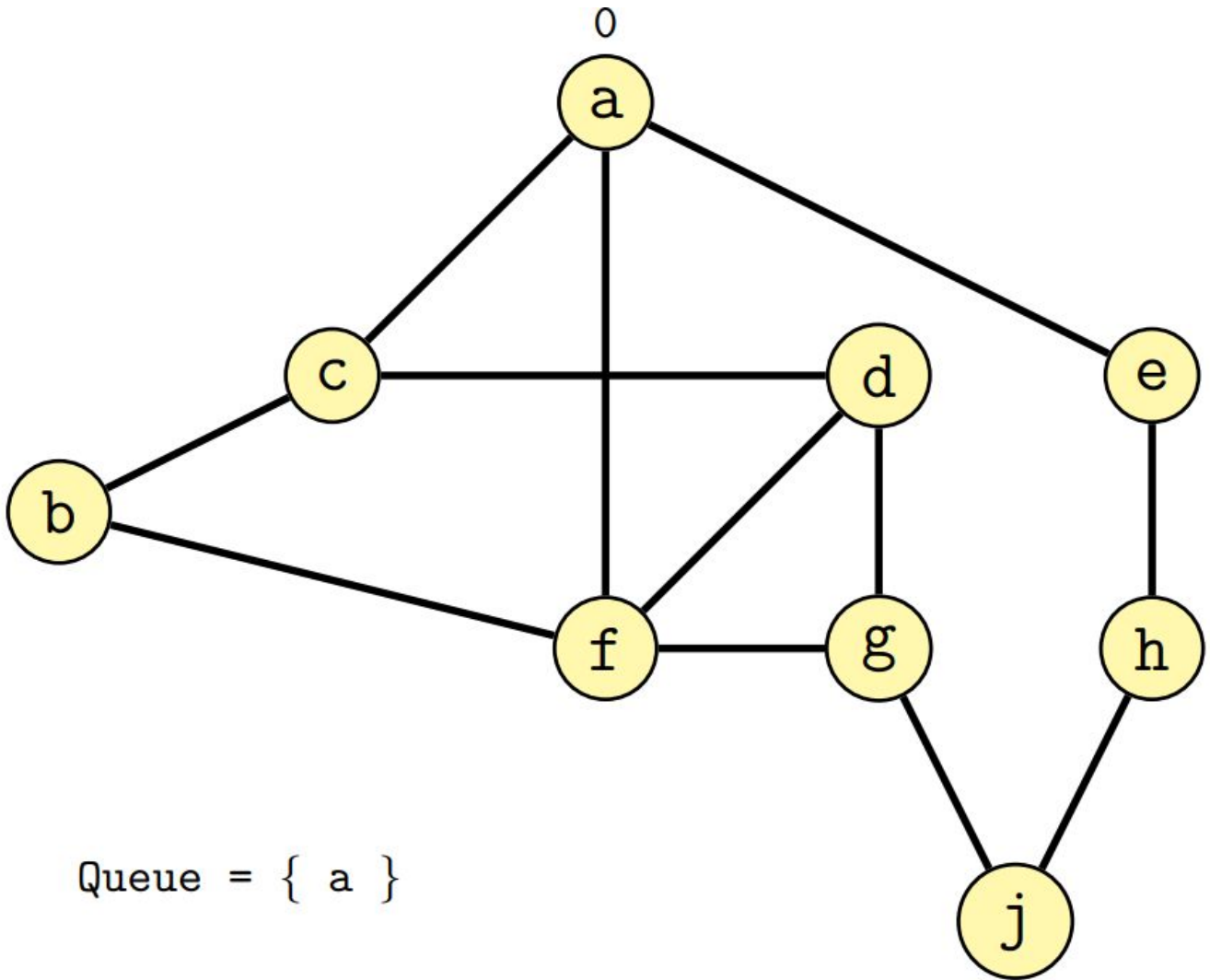

**A -> B -> D**



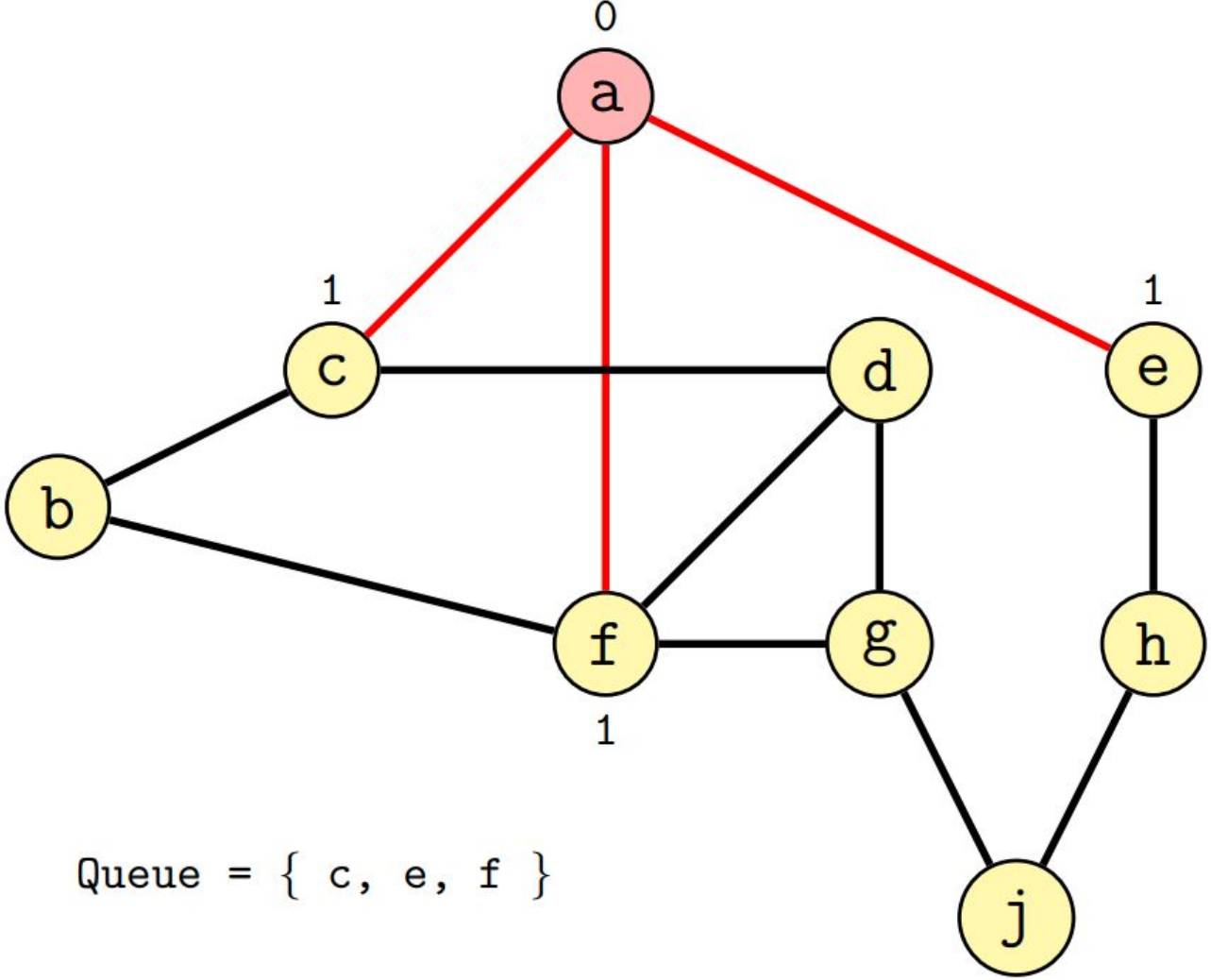

**A -> B -> C -> D**

# Algoritmo di risoluzione

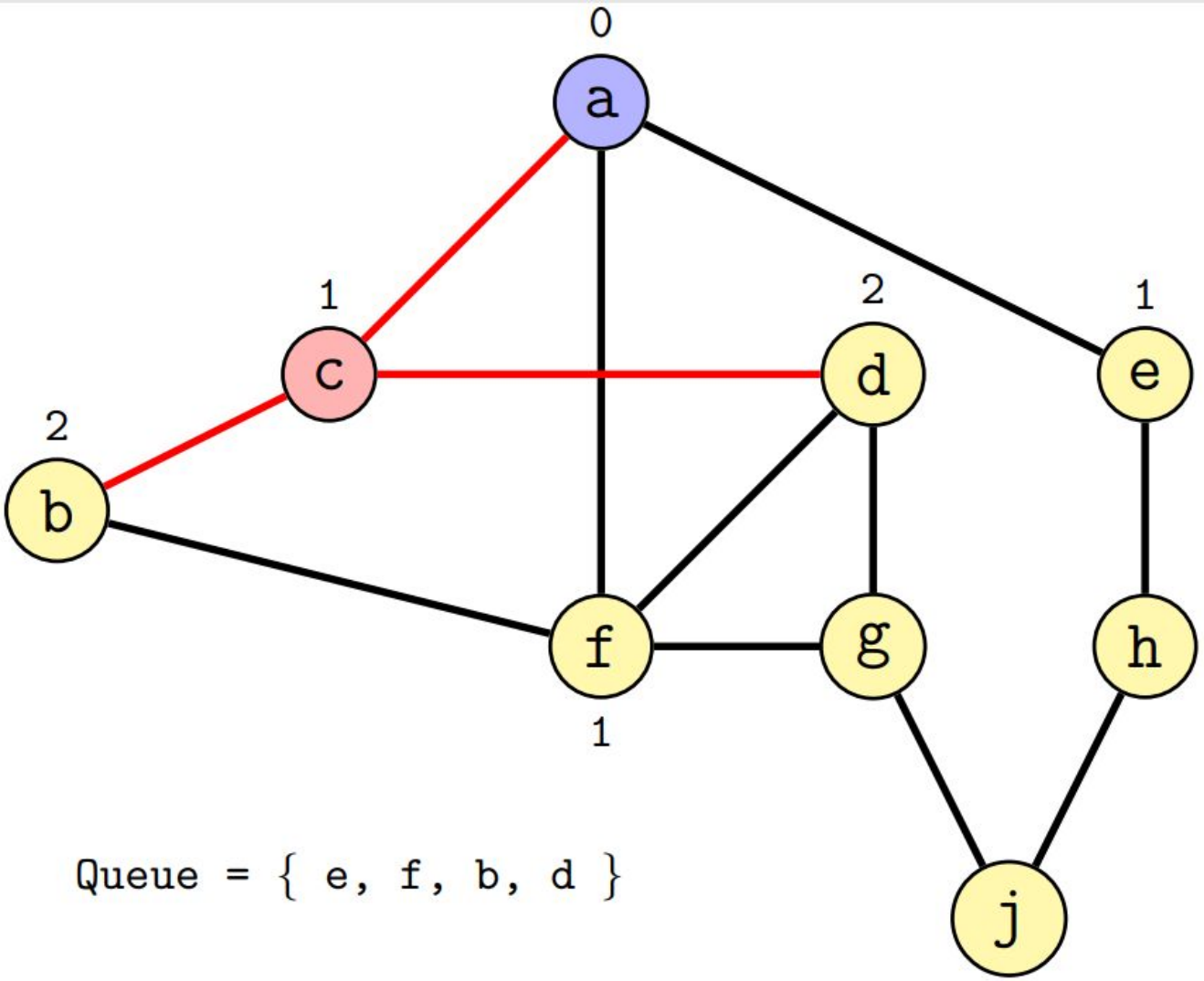
Si basa sul tenere un vettore delle distanze dal nodo di inizio, che aggiorniamo ogni volta che troviamo un modo di arrivarci con meno passi.



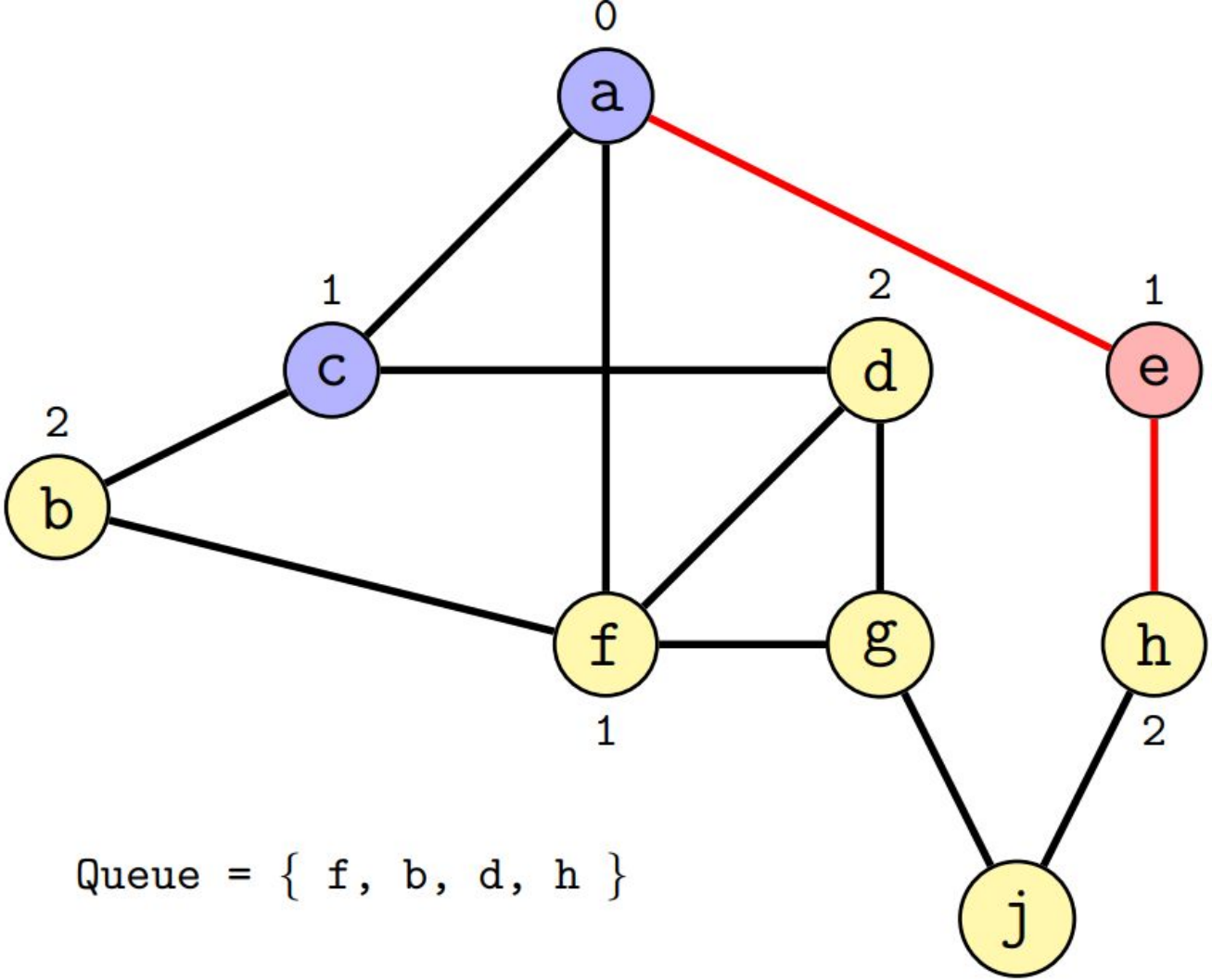

Queue = { a }



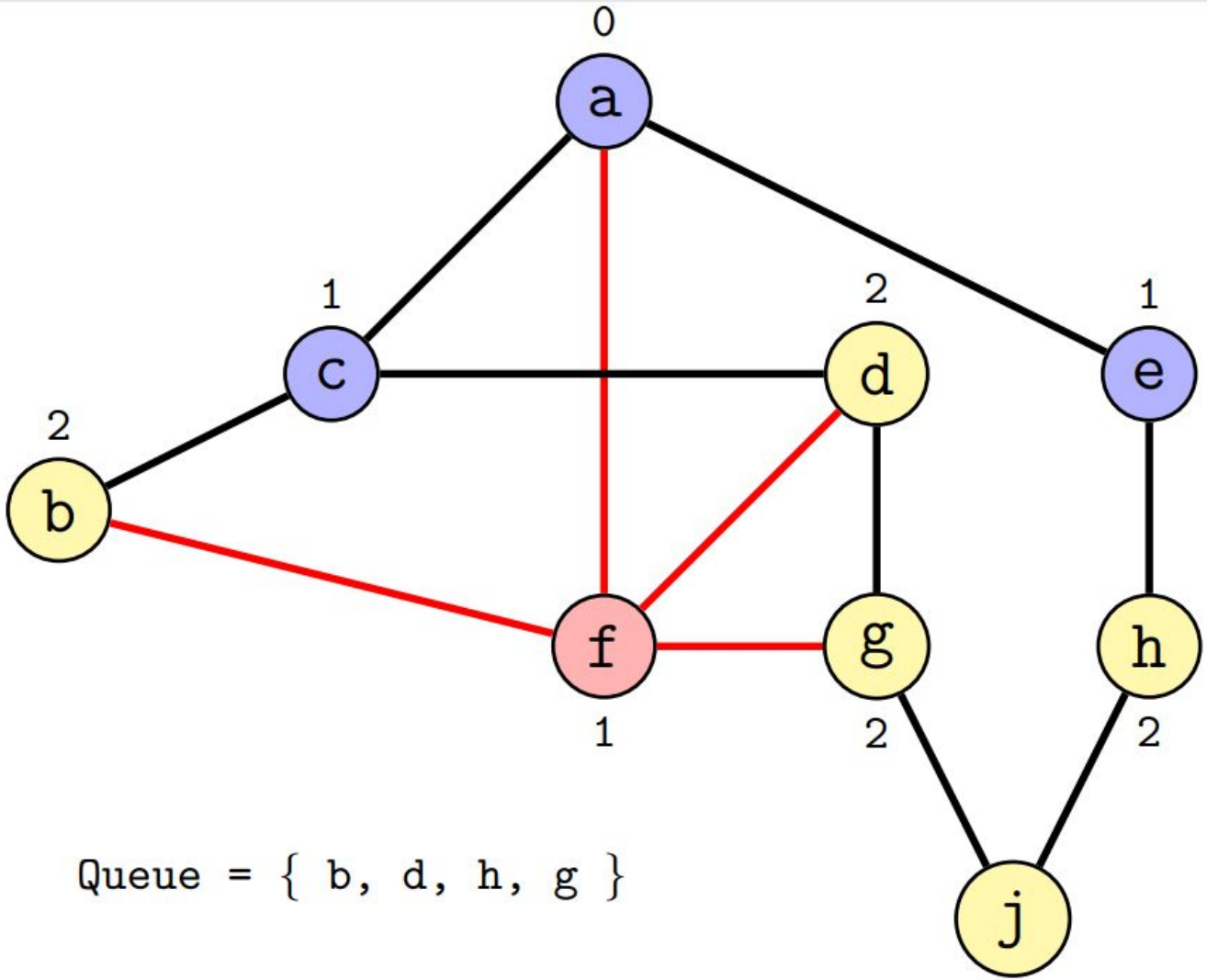

Queue = { c, e, f }



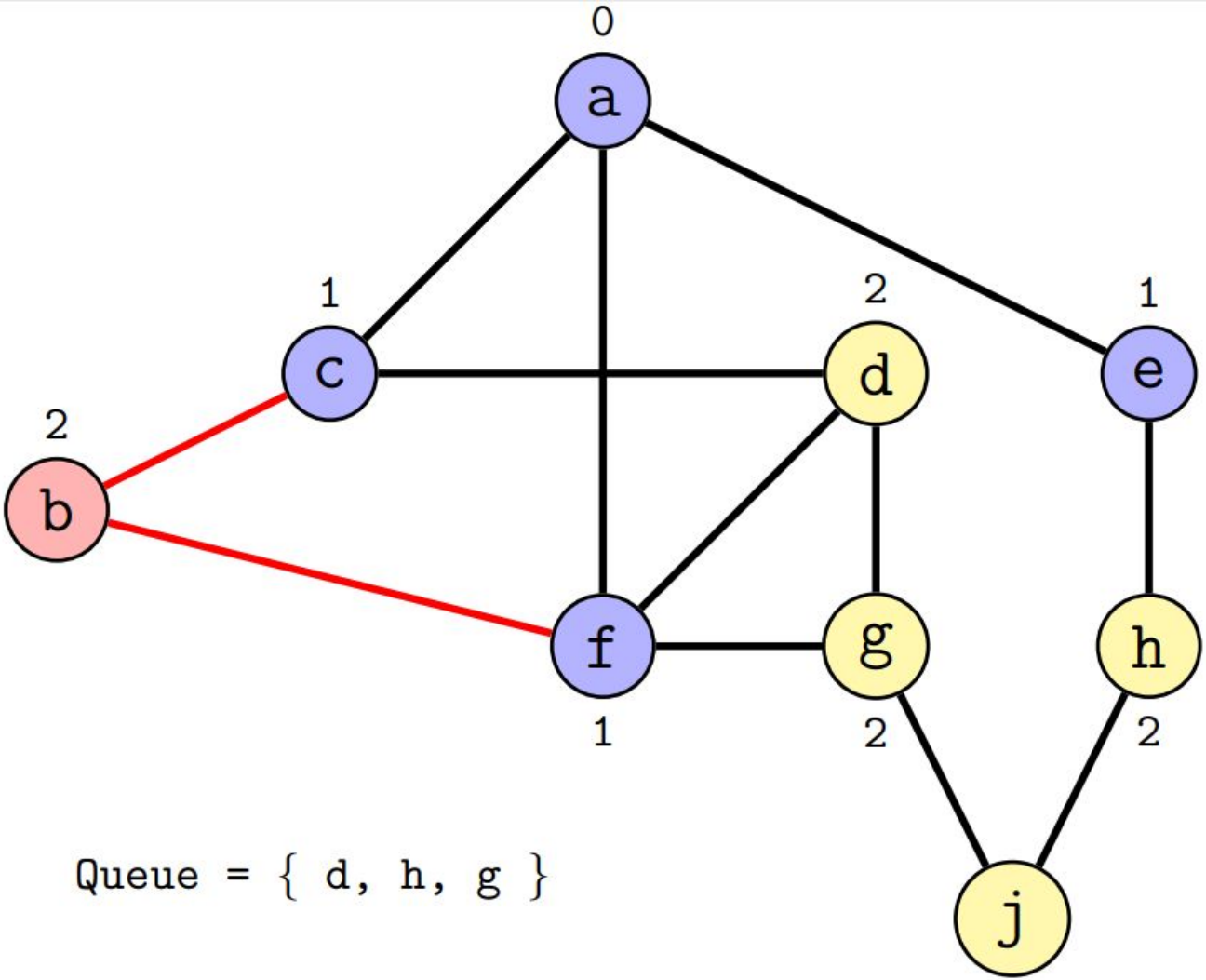

Queue = { e, f, b, d }



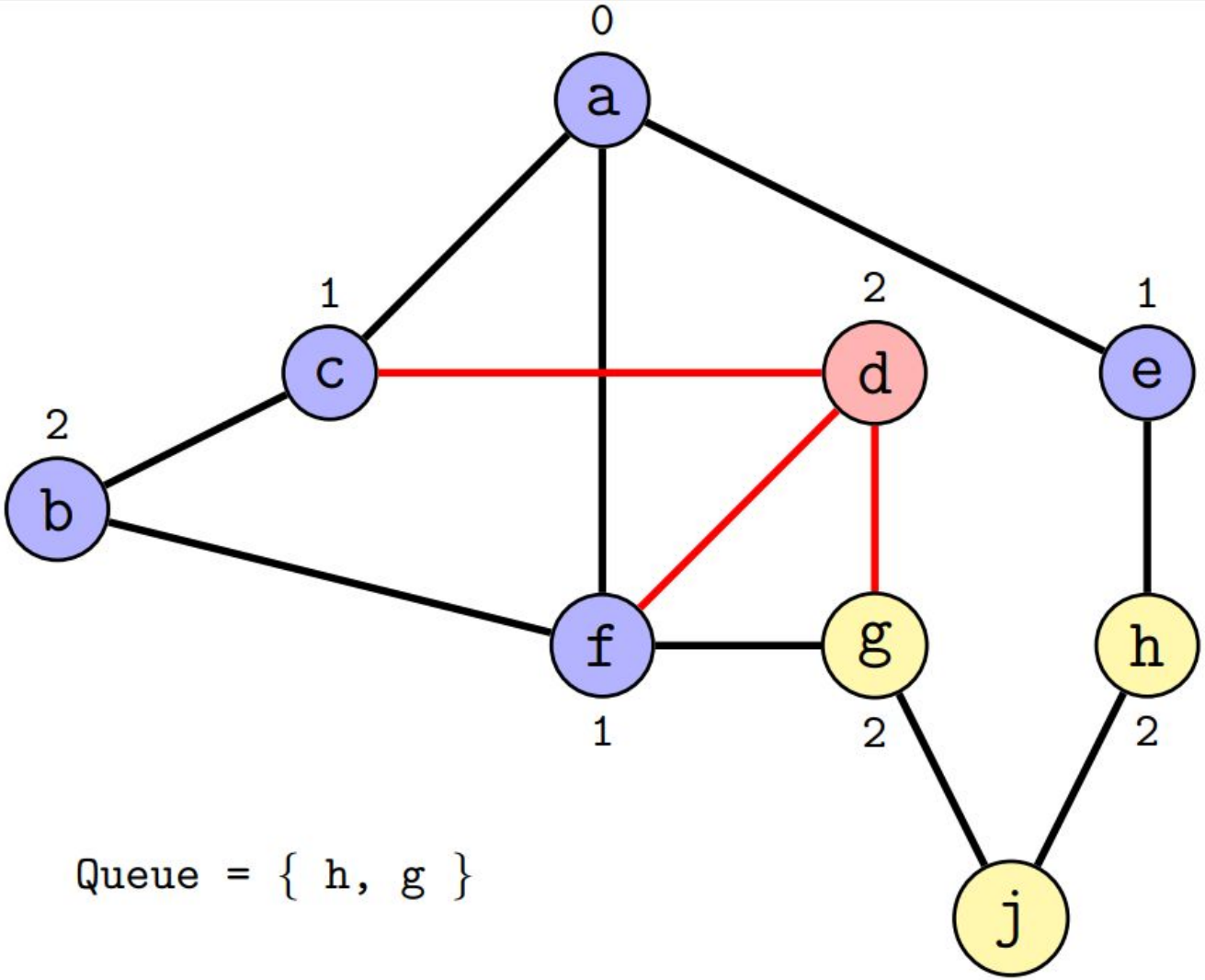

Queue = { f, b, d, h }



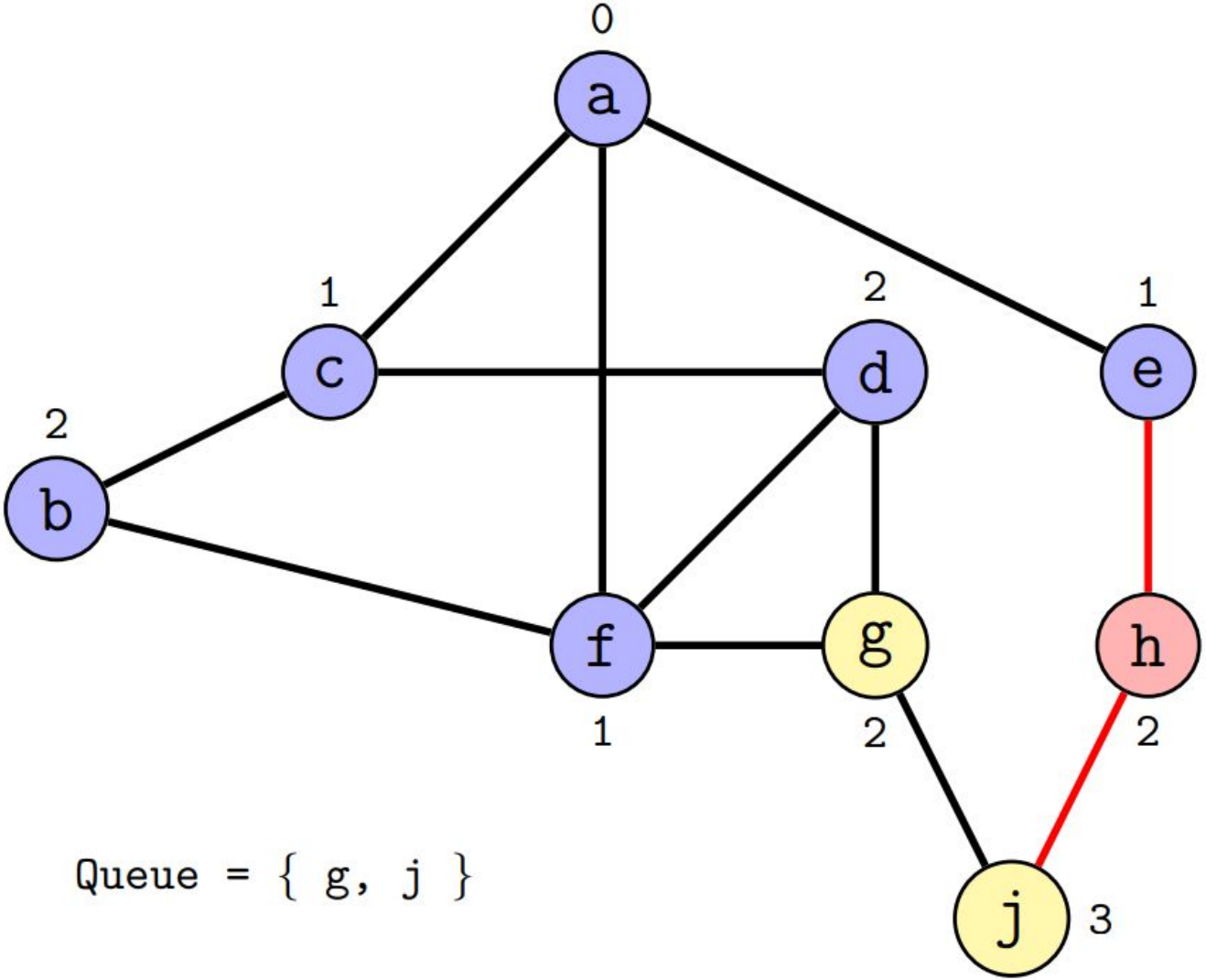

Queue = { b, d, h, g }

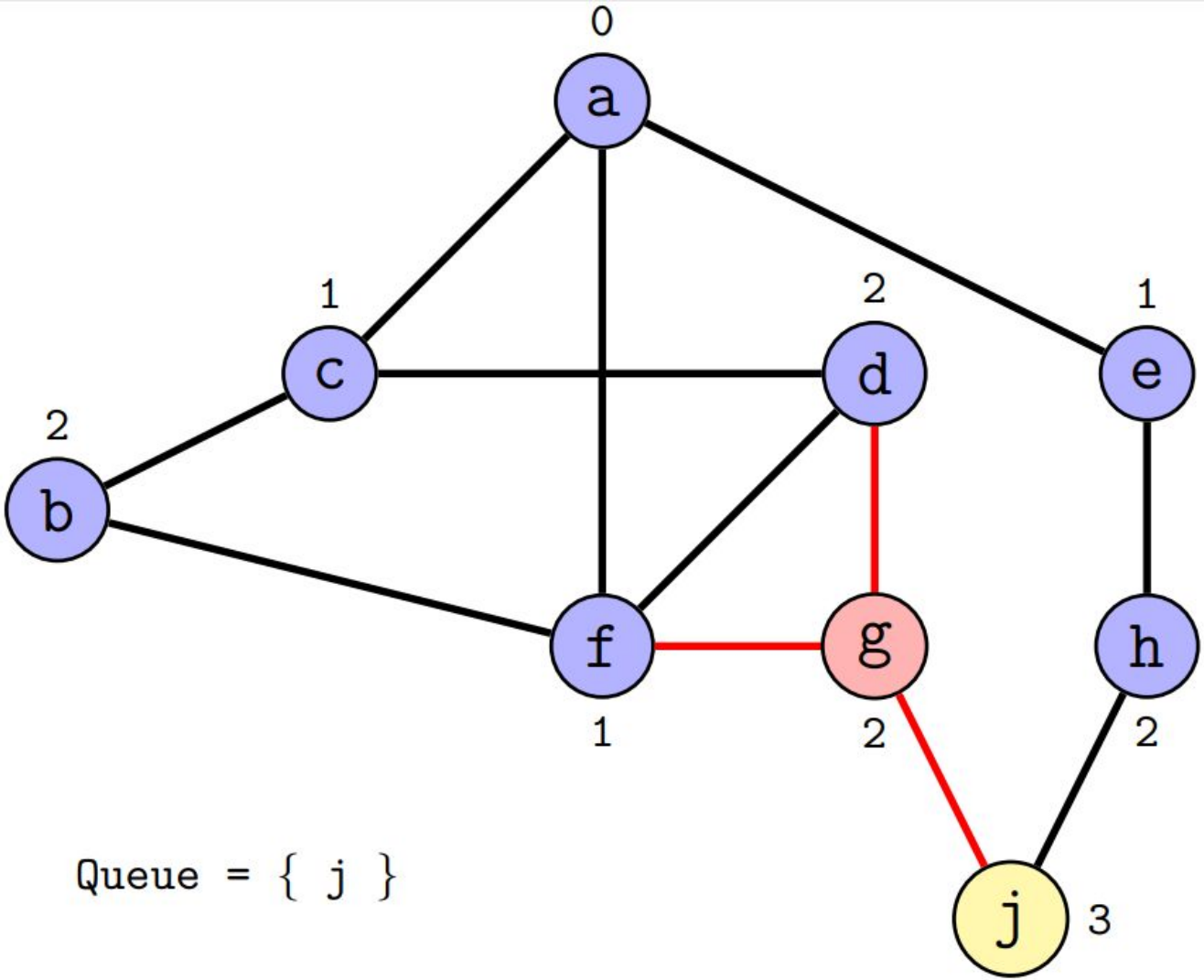


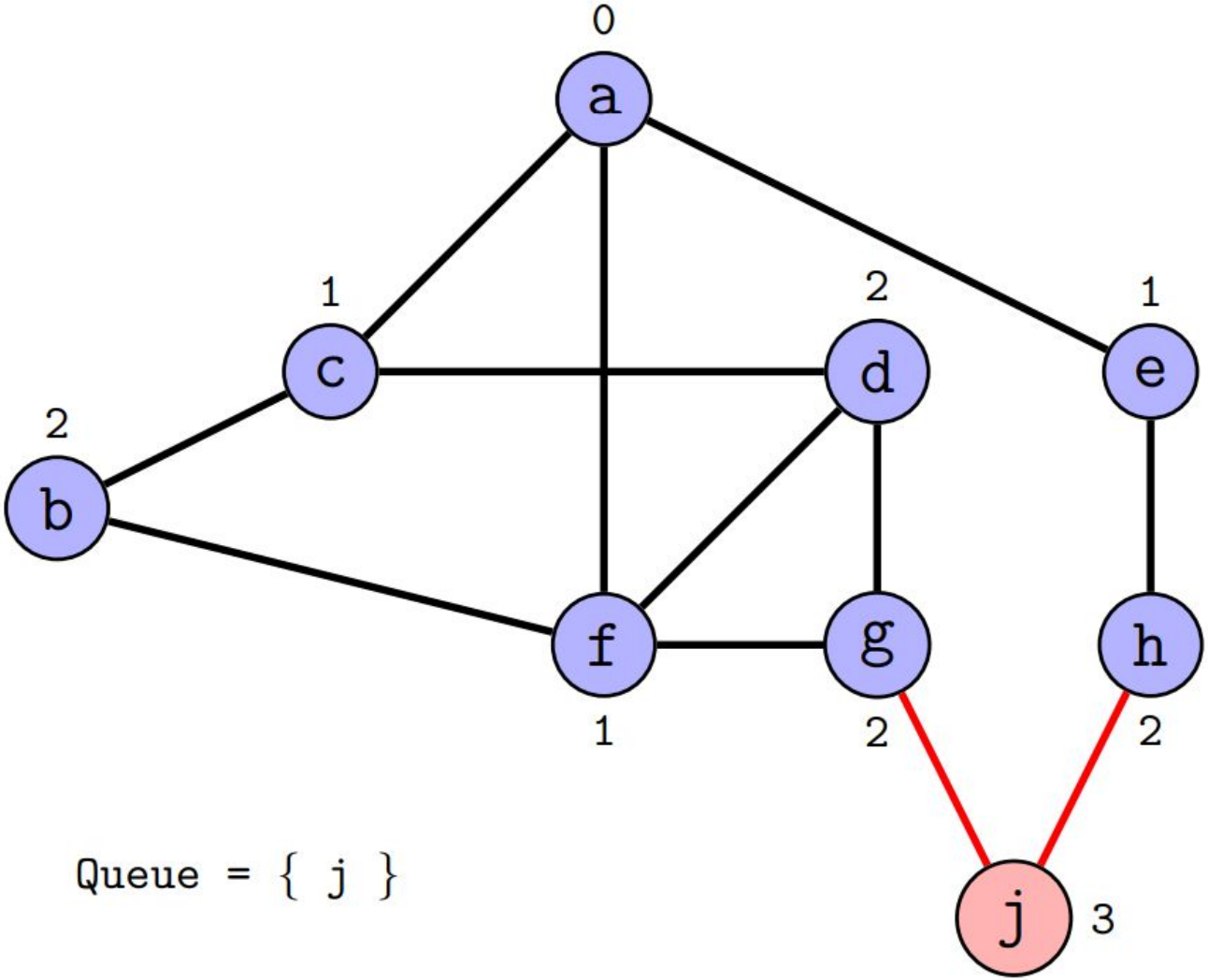

Queue = { d, h, g }

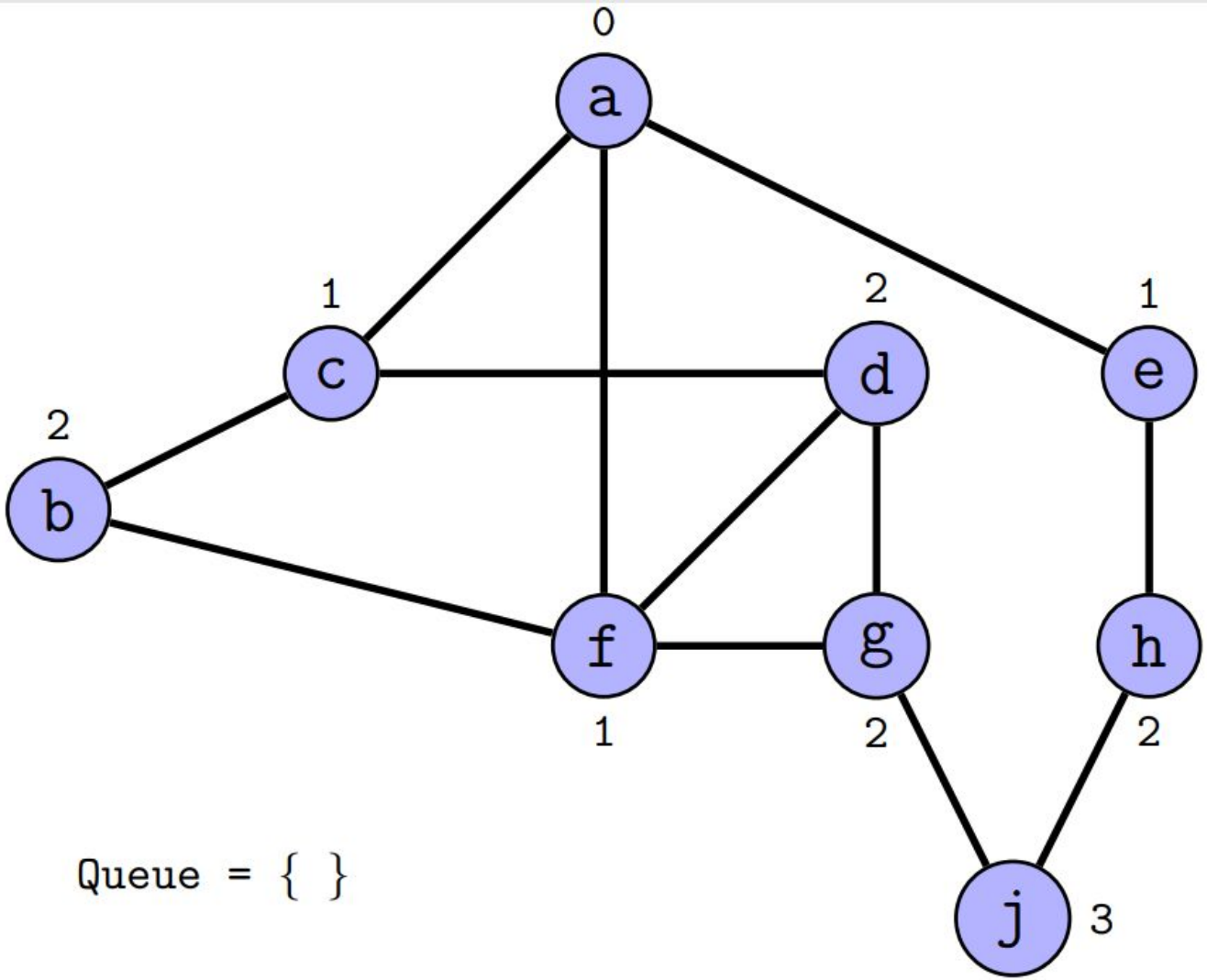



Queue = { h, g }


0
a
1
c
2
d
1
e
2
b
f
1
g
2
h
2
j
3
Queue = { g, j }


0
a
1
c
2
d
1
e
2
b
f
1
g
2
h
2
j
3
Queue = { j }


0
a
1
c
2
d
1
e
2
b
f
1
g
2
h
2
j
3
Queue = { j }


0
a
1
c
2
d
1
e
2
b
f
1
g
2
h
2
j
3
Queue = { }

# In codice

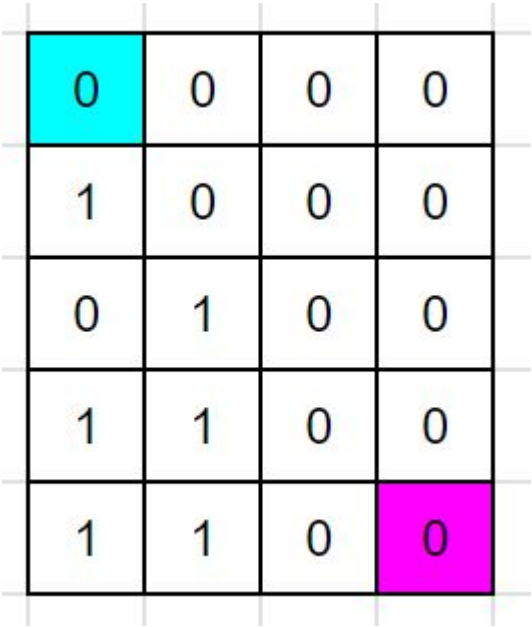
```
int N,M,inizio,fine;
vector <vector<int>> g;
int dist[MAX];                  //vettore con le distanze dal nodo iniziale
queue <int> q;

//inizializzo grafo…

for(int i=0;i<n;i++) { dist[i]=INT_MAX;}            //inizializziamo il vettore delle distanze

dist[inizio]=0; //il nodo iniziale dista 0 da sè stesso
q.push(inizio);

while(!q.empty()) {                     //facciamo una bfs
        int attuale=q.front();
        q.pop();
        for (const auto& vicino : g[attuale]) { //iteriamo sui vicini
                        if (dist[vicino] > dist[attuale] + 1) { //se ci arrivo con una distanza minore
                        dist[vicino] = dist[attuale] + 1;       //aggiorno la distanza
                        q.push(vicino);
                        }
        }
}

out << dist[fine];
```

# Esercizio

- Dijkstra (https://training.olinfo.it/#/task/dijkstra/submissions)

# Matrici come mappe

# Matrice come mappa



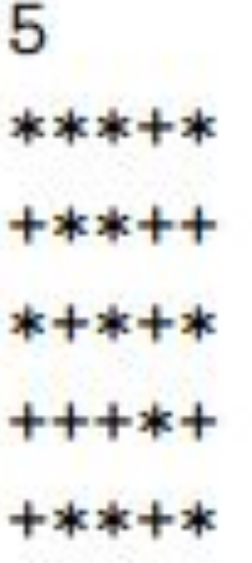

Alcuni problemi delle olimpiadi chiedono di vedere una matrice come una mappa.

In questo caso immaginiamo le celle della matrice come piastrelle e di poterci muovere solo sulle caselle con gli 0 (quelle con 1 hanno delle trappole).

Dobbiamo trovare se esiste un percorso dalla casella azzurra alla casella rosa.

# Input

NB: la matrice di solito è data come stringhe di asterischi (*) e più (+), come in questo esempio. Per prendere in input usare il seguente metodo:




```
in >> N;

for(int i=0; i < N; i++) {
        in.get(); // leggo il carattere 'a capo'
        for(int j=0; j < N; j++) {
            m[i][j] = in.get();
        }
}
```

# “Muoversi” sulla mappa

```
int mappa[MAX][MAX] m;
bool visitata[MAX][MAX] v;        //inizializzata a false

void walk(int r, int c) {         //riga e colonna (coordinate della cella in cui siamo)
        visitata[r][c] = true;
        //Questi 2 array rappresentano le 8 direzioni in cui posso andare
        int ar[] = {-1,-1,0,1,1,1,0,-1};      //Array righe (es. -1 significa vai nella riga sopra)
        int ac[] = {0,-1,-1,-1,0,1,1,1};      //Array colonne (es. 1 significa vai verso destra)

        for(int i=0; i < 8; i++) {            //Vai in una delle 8 direzioni
                int new_row = r + ar[i];
                int new_col = c + ac[i];

                if (isValid(new_row, new_col) && !visitata[new_row][new_col]) {   //Se è valida
                        walk(new_row, new_col);                                   //Visito la cella
                }
        }
}
```

**Nel main: walk(0,0)**

```
bool isValid(int r, int c) { //Ritorna true se la cella e' dentro la matrice
        if (r < 0 || c < 0 || r >= N || c >= N)
                return false;
        return true;
}
```

# Esercizio

- mappa (https://training.olinfo.it/#/task/mappa/statement)