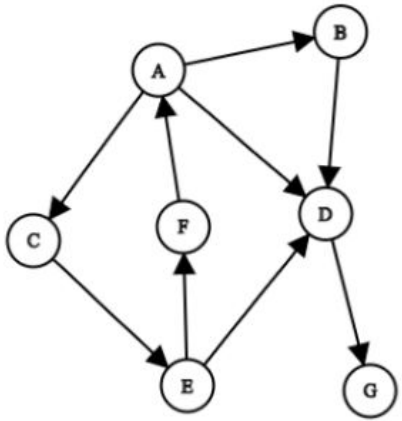
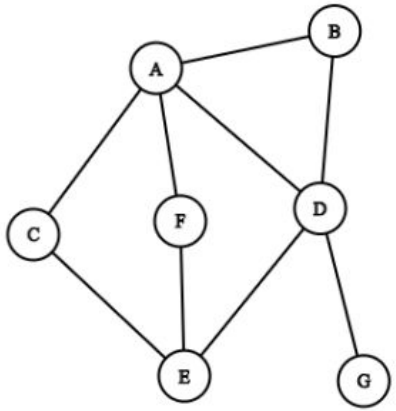
Teoría de Grafos 2880

Definición

Un grafo es un conjunto de objetos llamados vértices o nodos, unidos por enlaces llamados aristas o arcos.



Grafo dirigido



Grafo no dirigido

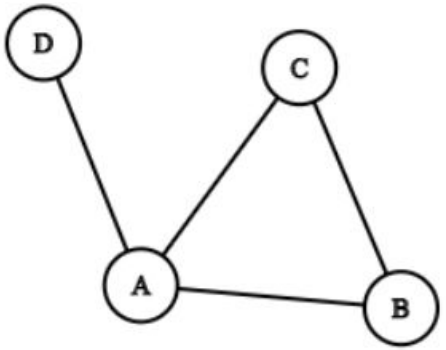
Ejemplos

- En Facebook, dos usuarios, pueden o no ser amigos. Esta relación es mutua, es decir que puede pensarse como un grafo no dirigido, en donde cada usuario es un nodo y cada relación de amistad es una arista.
- En Instagram, un usuario puede seguir a otro, sin necesidad que ese otro usuario lo siga a uno. Es decir que puede pensarse como un grafo dirigido, en donde cada usuario es un nodo y cada relación seguidor-seguido es una arista.

En general, dada cualquier situación en la que tenemos pares de cosas relacionadas, probablemente sirve verla como un grafo.

Matriz de adyacencia

La matriz de adyacencia es una matriz de n × n donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j, o 0 (o false) en caso contrario. Uso de memoria: O(n^2)



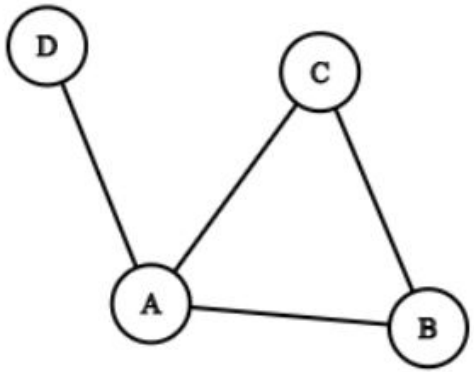
	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	0
D	1	0	0	0

Matriz de adyacencia

```
int n,m;
cin>>n>>m;
vector<vector<int>> g(n, vector<int>(a));
for(int i=0;i<m;i++){
    int a,b;
    cin>>a>>b;
    g[a][b] = 1;
    g[b][a] = 1;
}
```

Listas de adyacencia

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Uso de memoria: O(m), donde m es la cantidad de aristas.



A :	{B,C,D}
B :	{A,C}
C :	{A,B}
D :	{A}

Listas de adyacencia

```
vector<vector<int>> g;
void lista_adyacencia()
{
    int n,m;
    cin>>n>>m;
    g.resize(n);
    for(int i=0;i<m;i++){
        int a,b;
        cin>>a>>b;
        g[a].pb(b);
        g[b].pb(a);
    }
}
```

Algunas definiciones

Vecino y grado: Dada una arista que conecta dos vértices u y v, decimos que u es vecino de v (y que v es vecino de u). Además, a la cantidad de vecinos de u se le llama el grado de u.

Distancia: Definimos la distancia de u a v como el menor n tal que hay un camino de largo n de u a v.

BFS

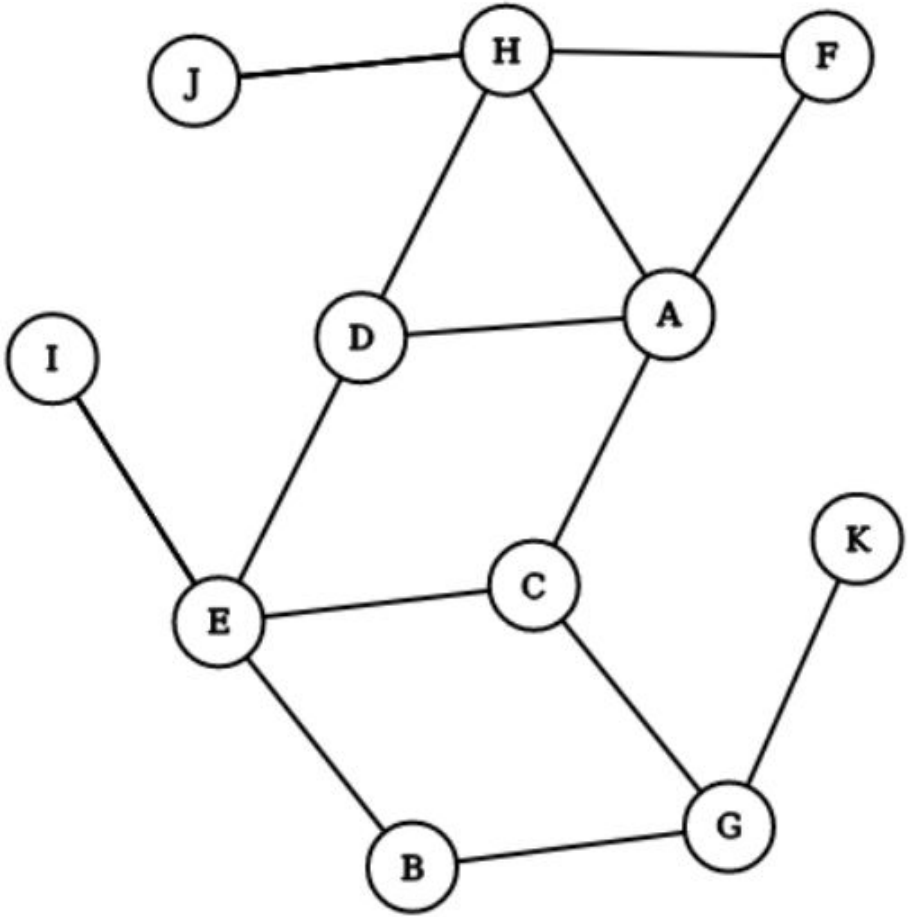
BFS (siglas de breadth-first-search o “búsqueda por amplitud”) es un algoritmo para recorrer grafos, que a su vez sirve para calcular la distancia mínima desde un nodo v a cada uno de los otros.

- Inicialmente se procesa v, que tiene distancia a sí mismo 0, por lo que seteamos d[v] = 0. Además es el único a distancia 0.
- Los vecinos de v tienen sí o sí distancia 1, por lo que seteamos d[y] = 1 para todo y vecino de v. Además, estos son los únicos a distancia 1.

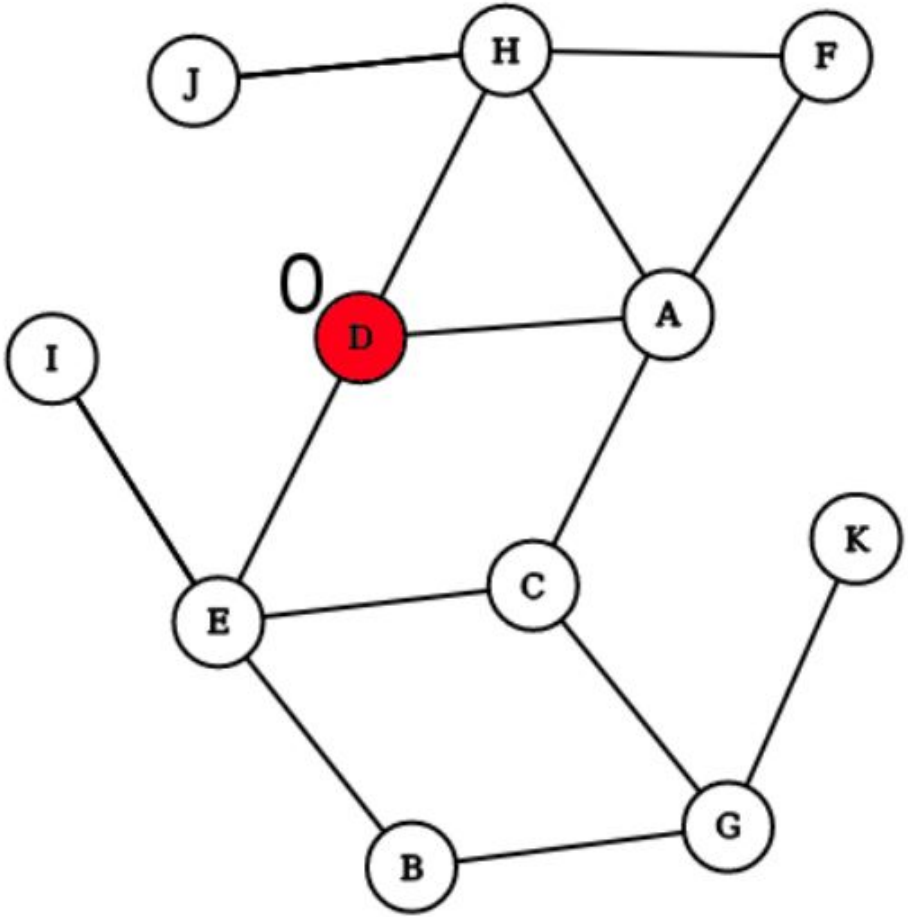
BFS

- Ahora tomamos los nodos a distancia 1, y para cada uno nos fijamos en sus vecinos cuya distancia aún no calculamos. Estos son los nodos a distancia 2.
- Así sucesivamente, para saber cuáles son los nodos a distancia k + 1, tomamos los vecinos de los nodos a distancia k que no hayan sido visitados aún.

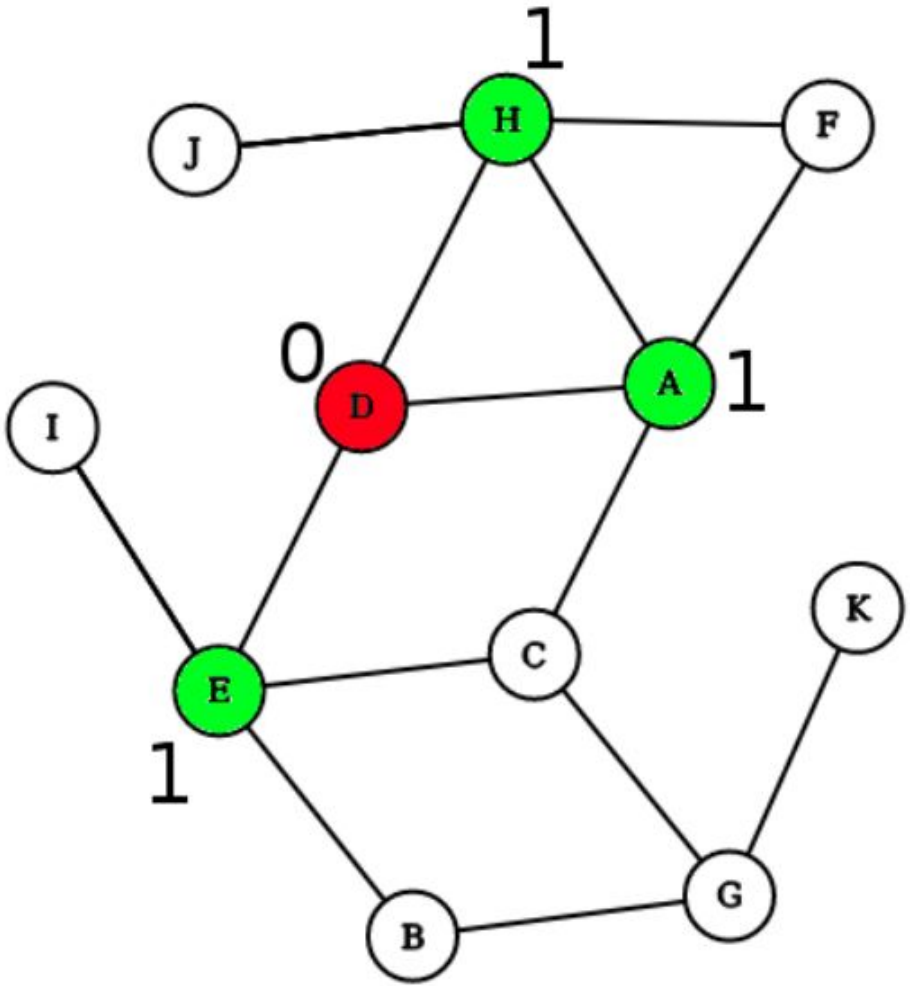
BFS



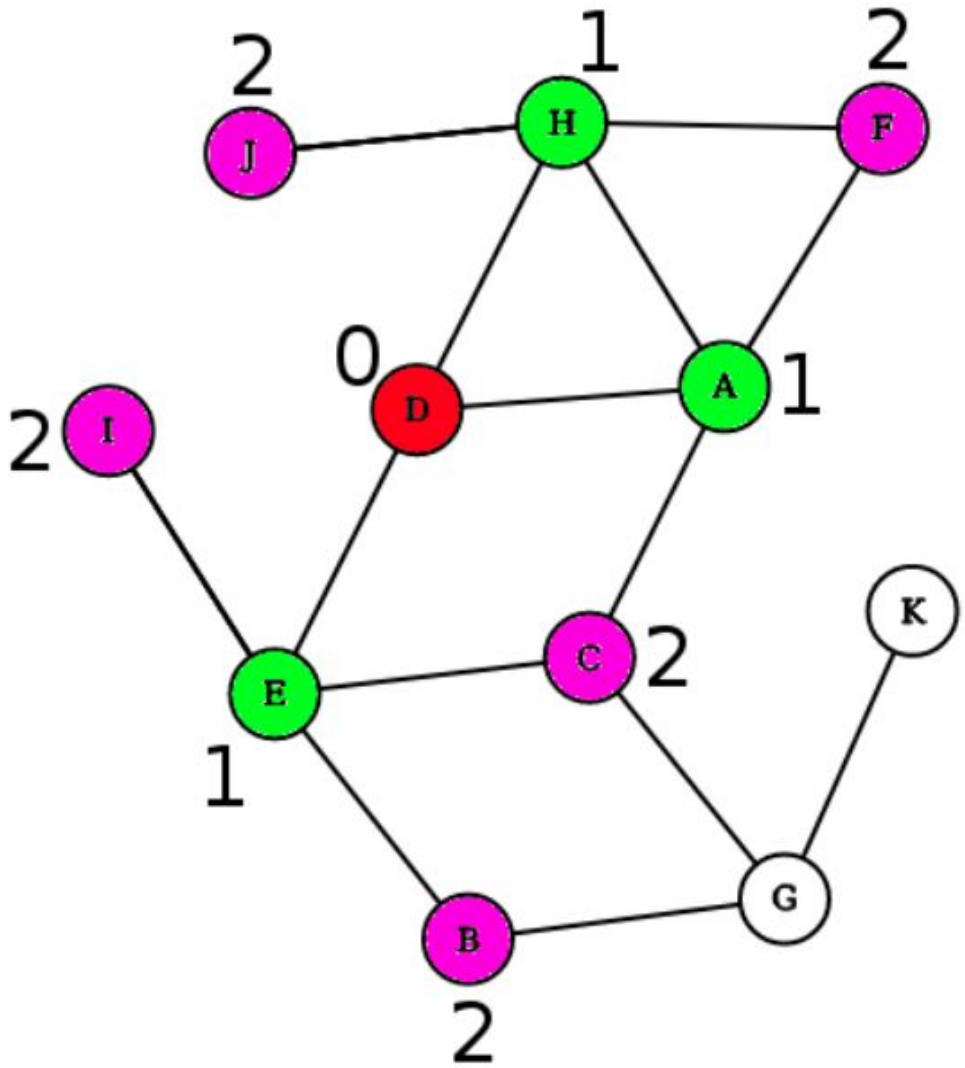
BFS



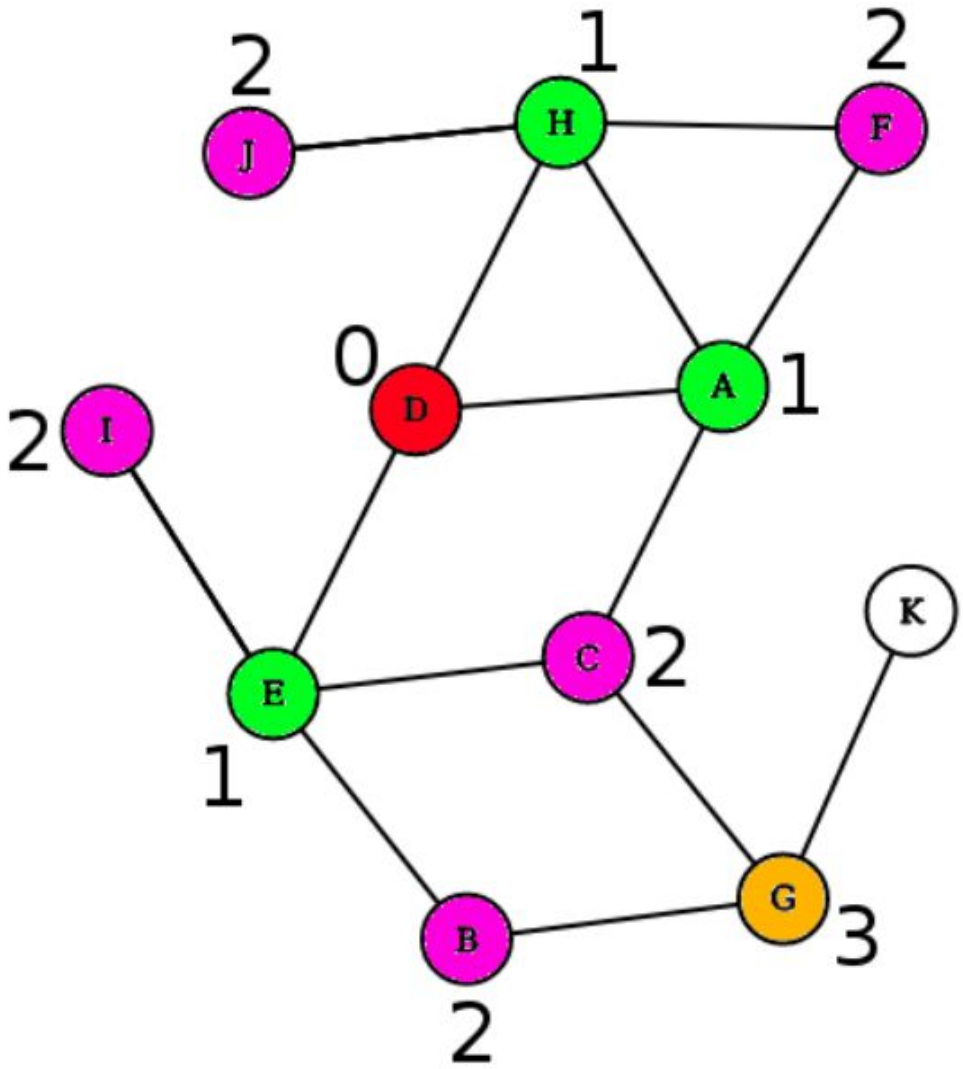
BFS



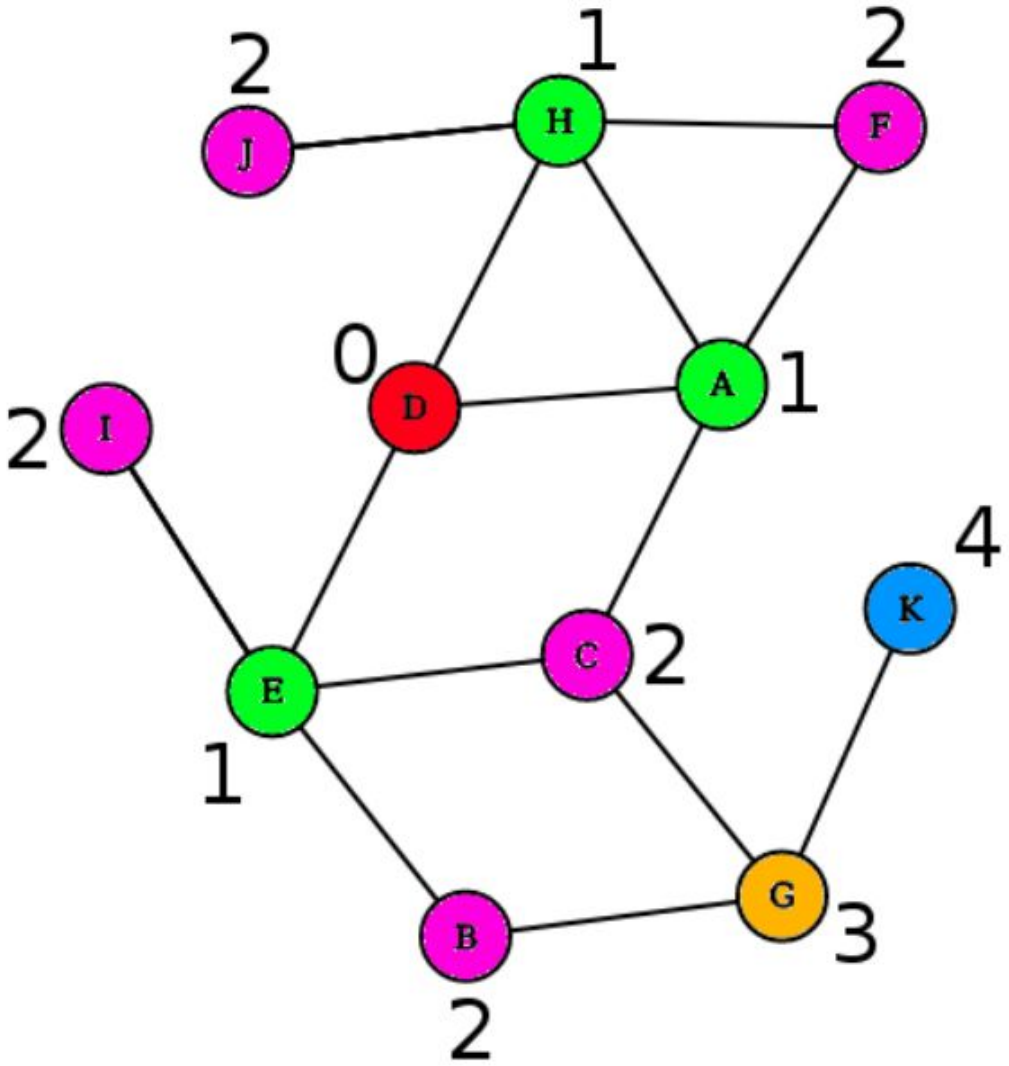
BFS



BFS



BFS



BFS CÓDIGO

```
vector<int> bfs(vector<vector<int>>& g , int v) {
    vector<int> dis(g.size(), -1); // vector de distancias
    queue<int> q;
    dis[v] = 0;
    q.push(v);
    while(!q.empty()) { // mientras haya nodos por procesar
        int node= q.front();
        q.pop();
        for(int x : g[node]) { // para cada y vecino de x
            if(dis[x] == -1) { // si x no fue encolado aun
                dis[x] = dis[node] + 1;
                q.push(x);
            }
        }
    }
    return dis;
}
```

BFS complejidad

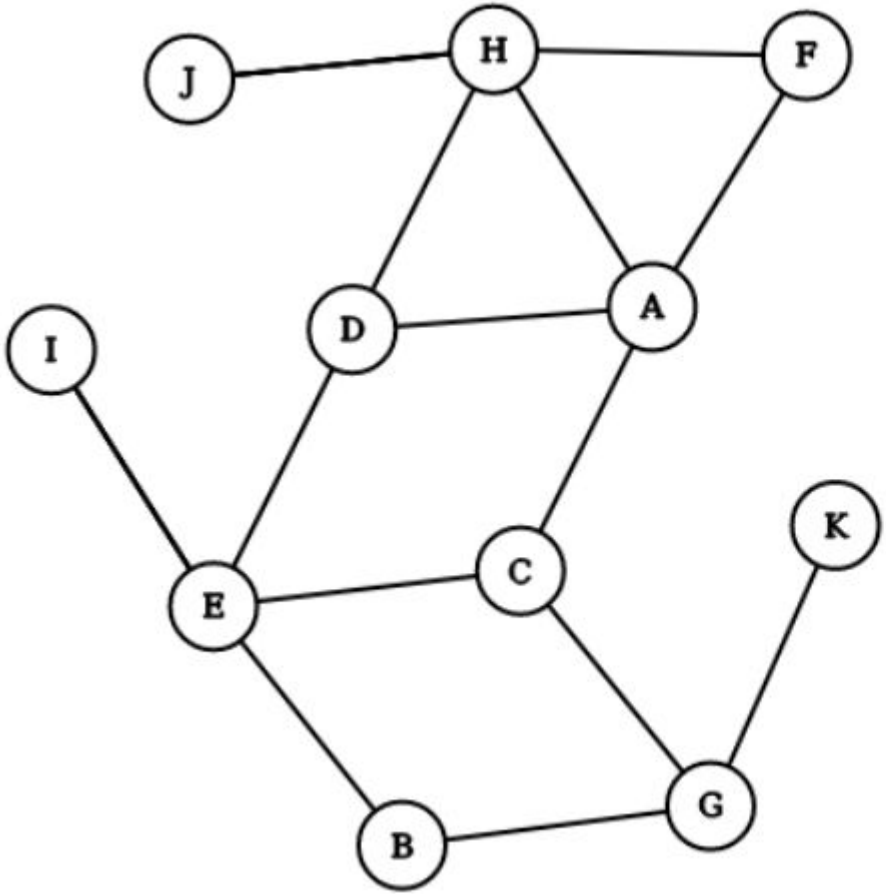
- Cada nodo lo procesamos una sóla vez, porque cuando lo agregamos a la cola, también inicializamos su distancia, por lo que no volverá a ser agregado.
- Como cada nodo es procesado una sóla vez, entonces cada arista es procesada una sóla vez (o una vez en cada sentido para no-dirigidos).
- Entonces, la complejidad de BFS es O(m), donde m es la cantidad de aristas.

DFS

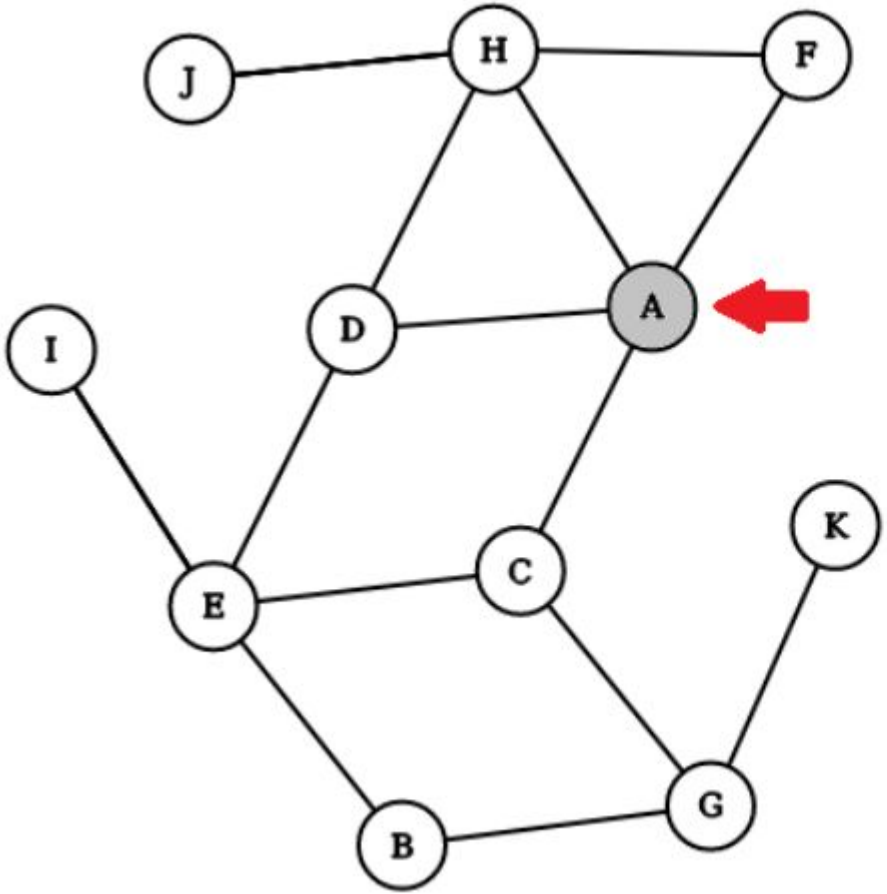
Otra manera de recorrer un grafo es con DFS (depth-first-search o “búsqueda por profundidad”):

- Arrancamos de cierto nodo y lo marcamos como visitado.
- Luego, evaluamos sus vecinos de a uno, y cada vez que encontramos uno que no esté marcado como visitado, seguimos procesando a partir de él. Este procedimiento es recursivo, por lo que se puede implementar así.
- Cuando no quedan vecinos por visitar salgo para atrás (vuelvo en la recursión).

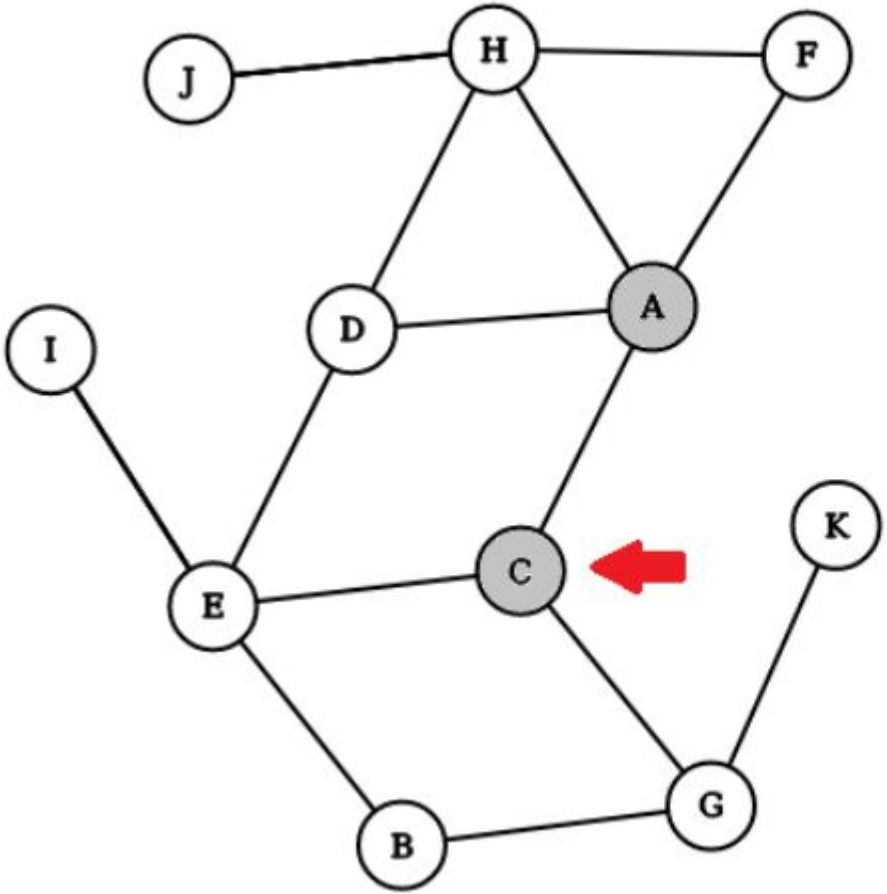
DFS



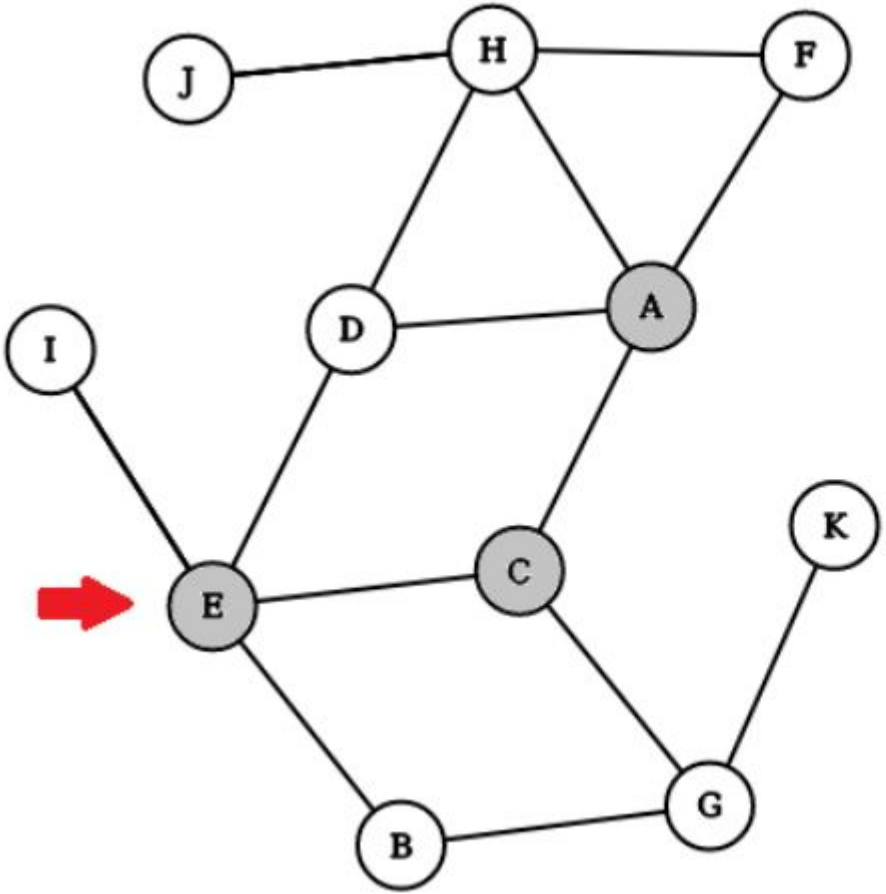
DFS



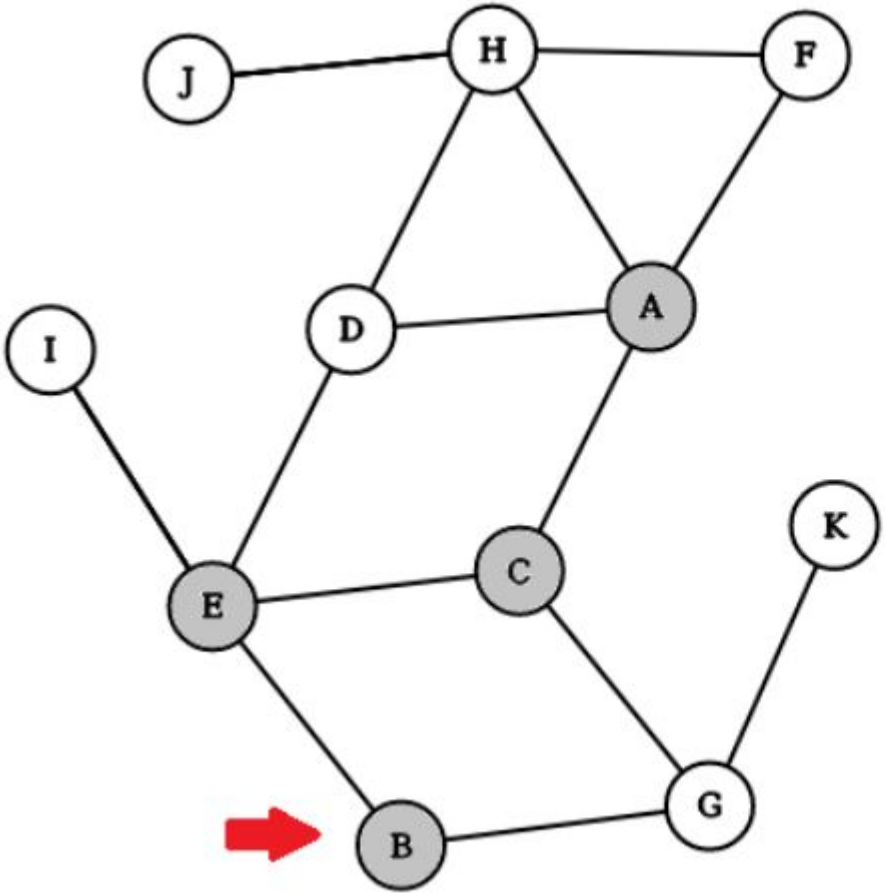
DFS



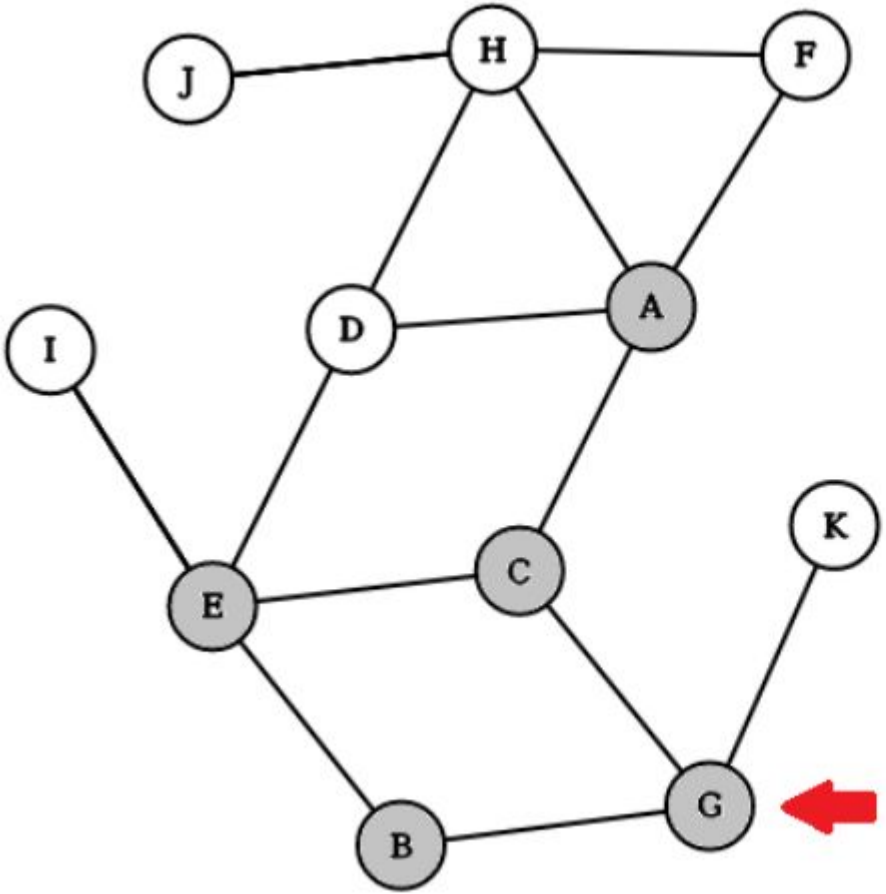
DFS



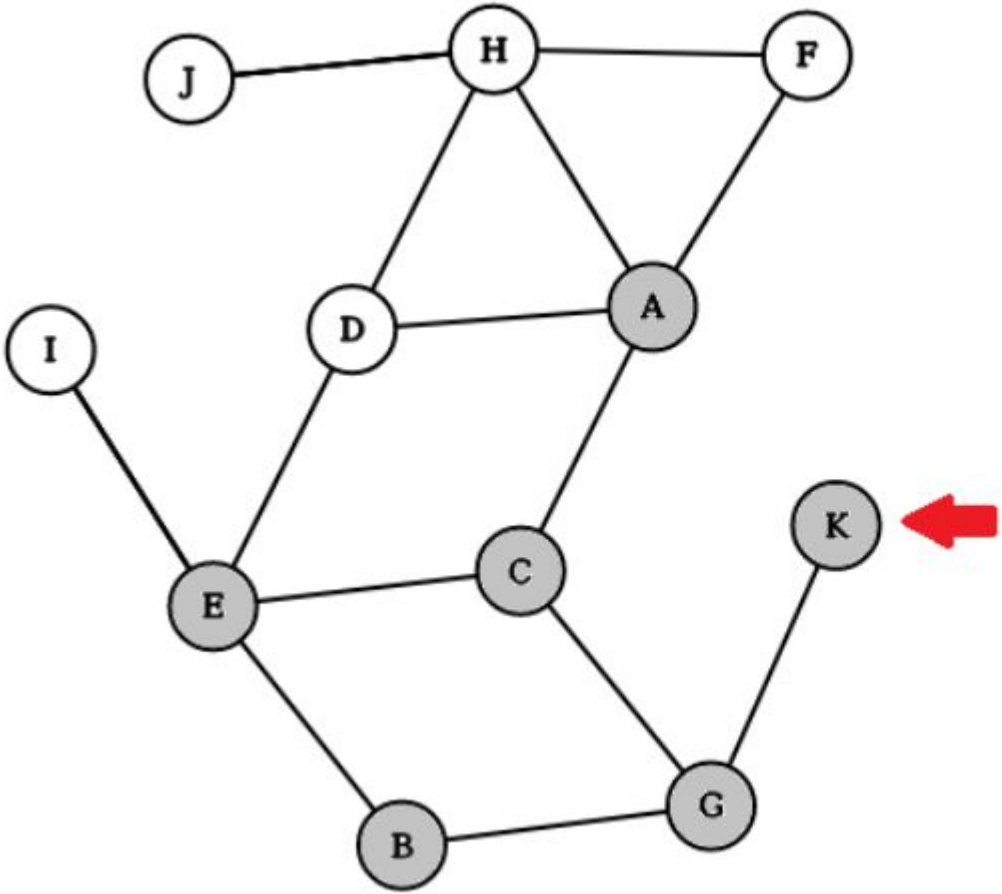
DFS



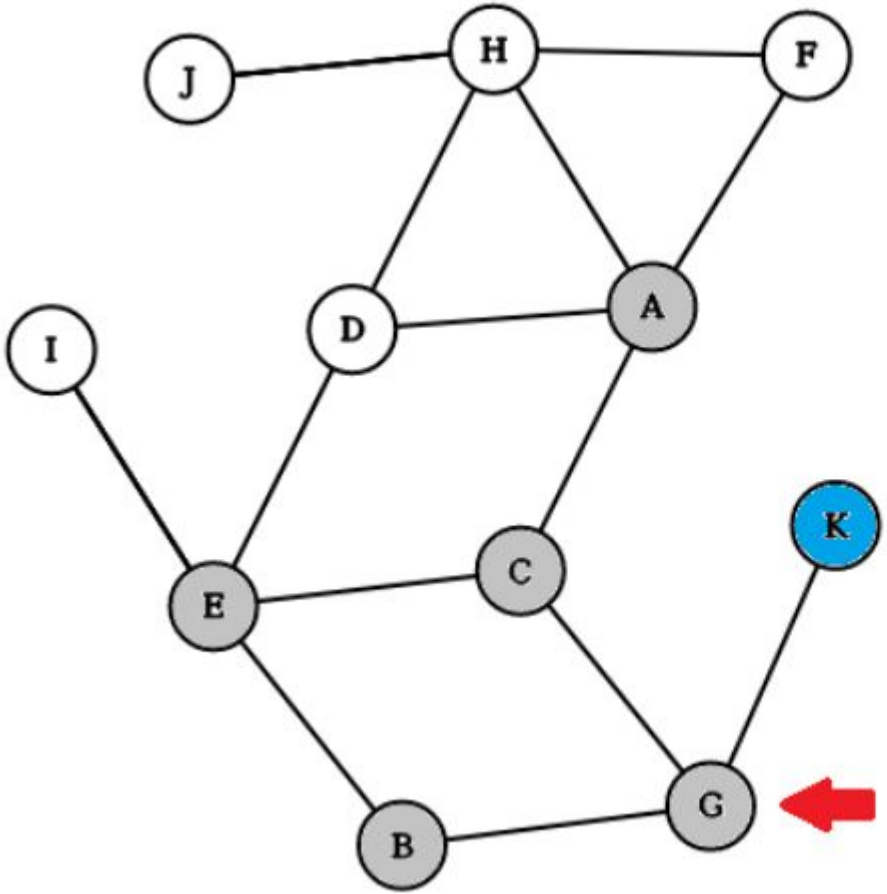
DFS



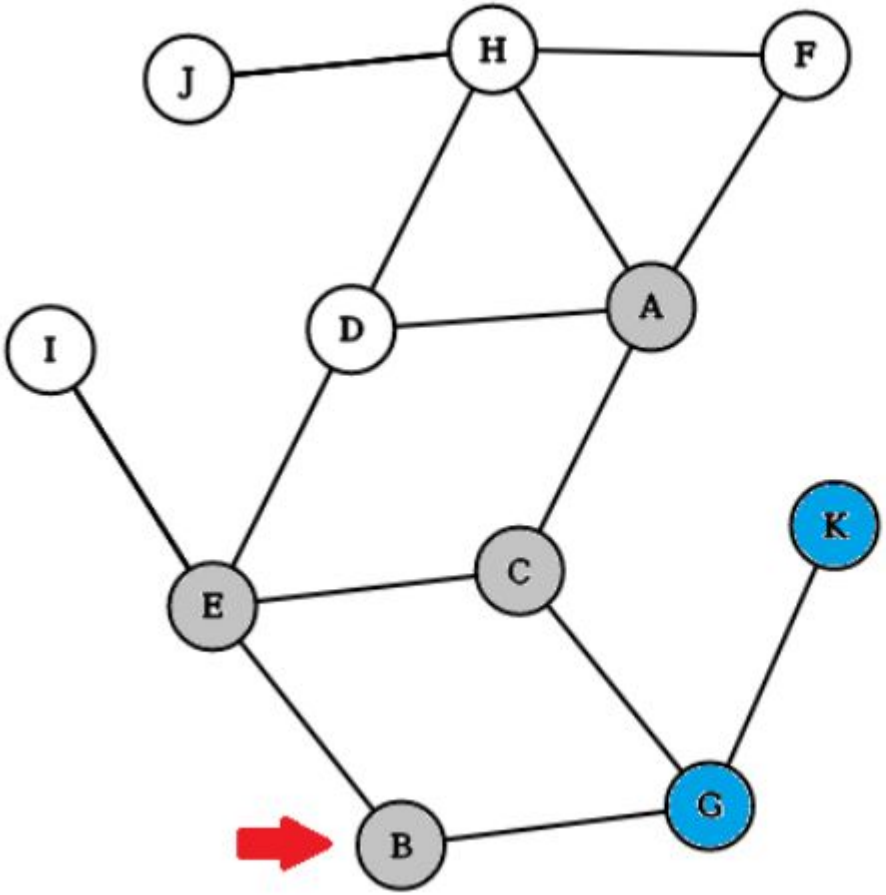
DFS



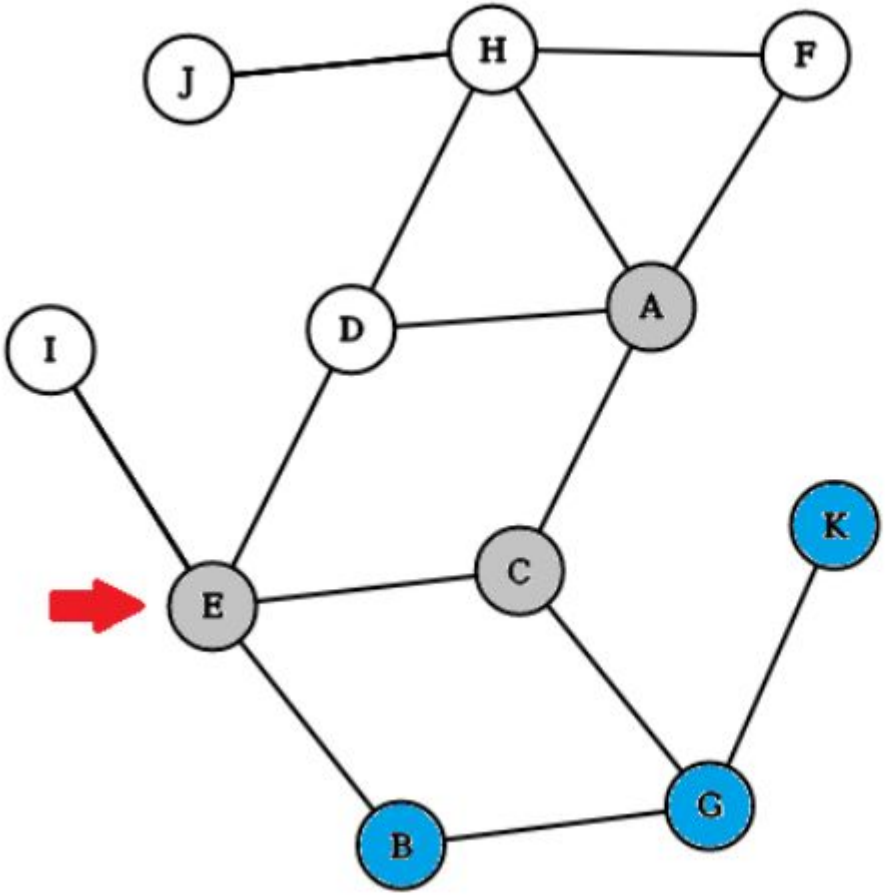
DFS



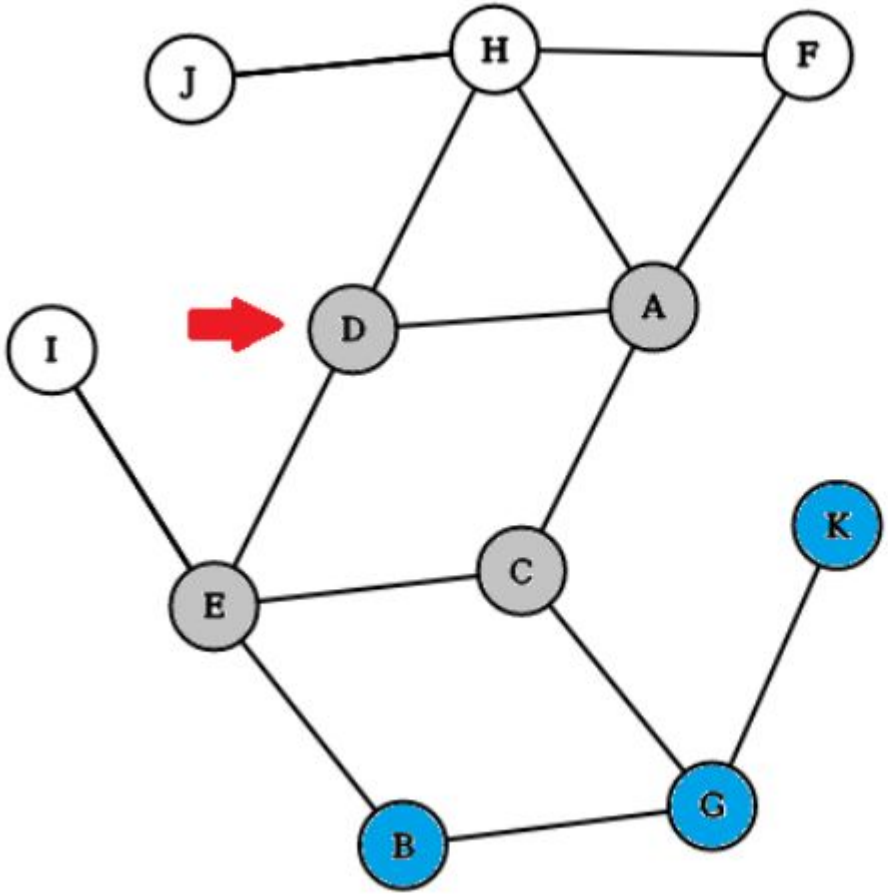
DFS



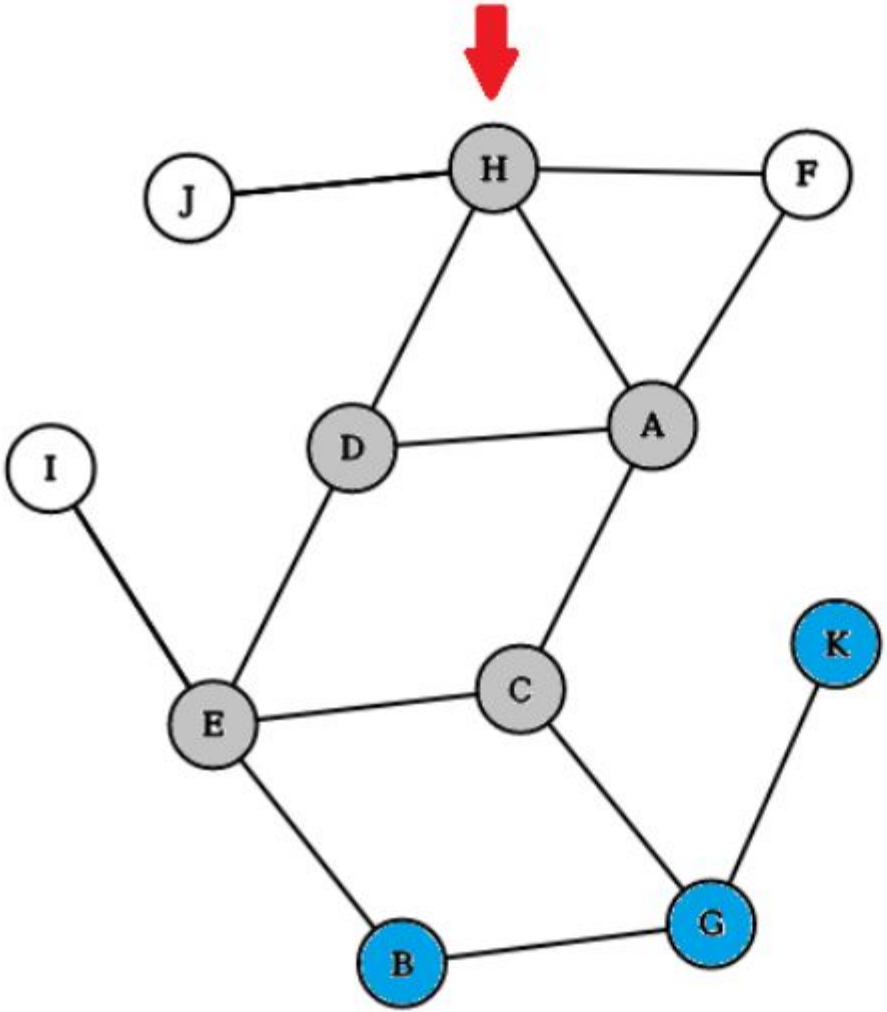
DFS



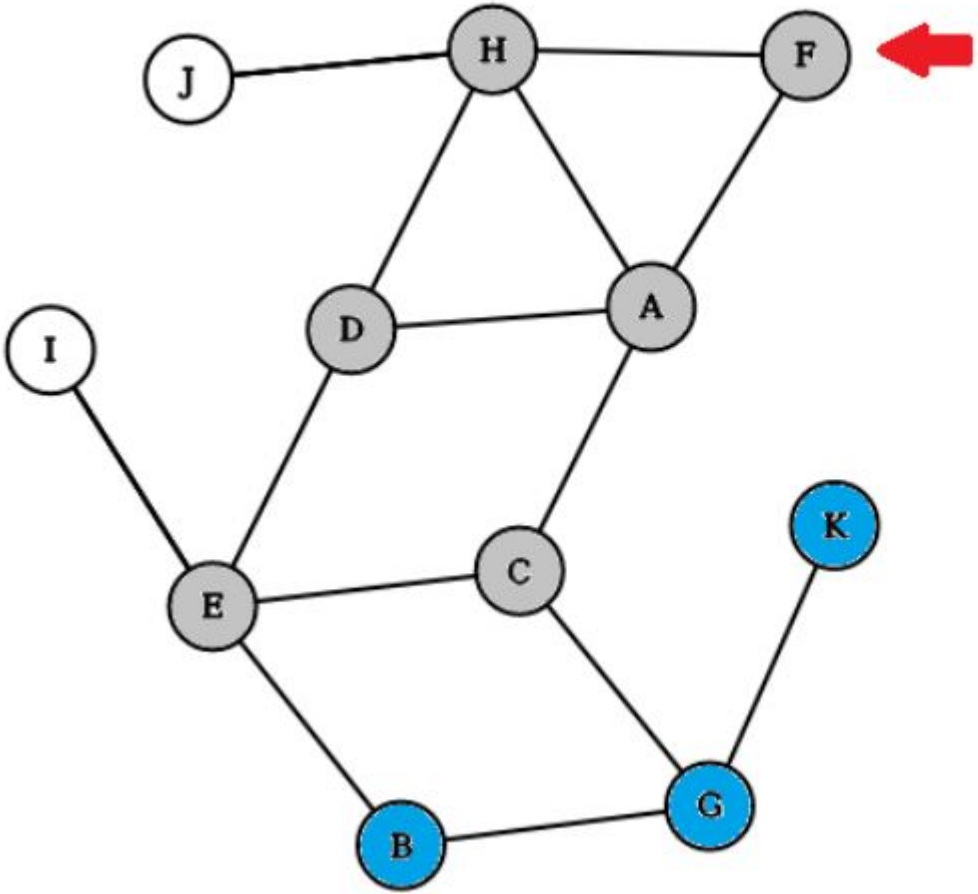
DFS



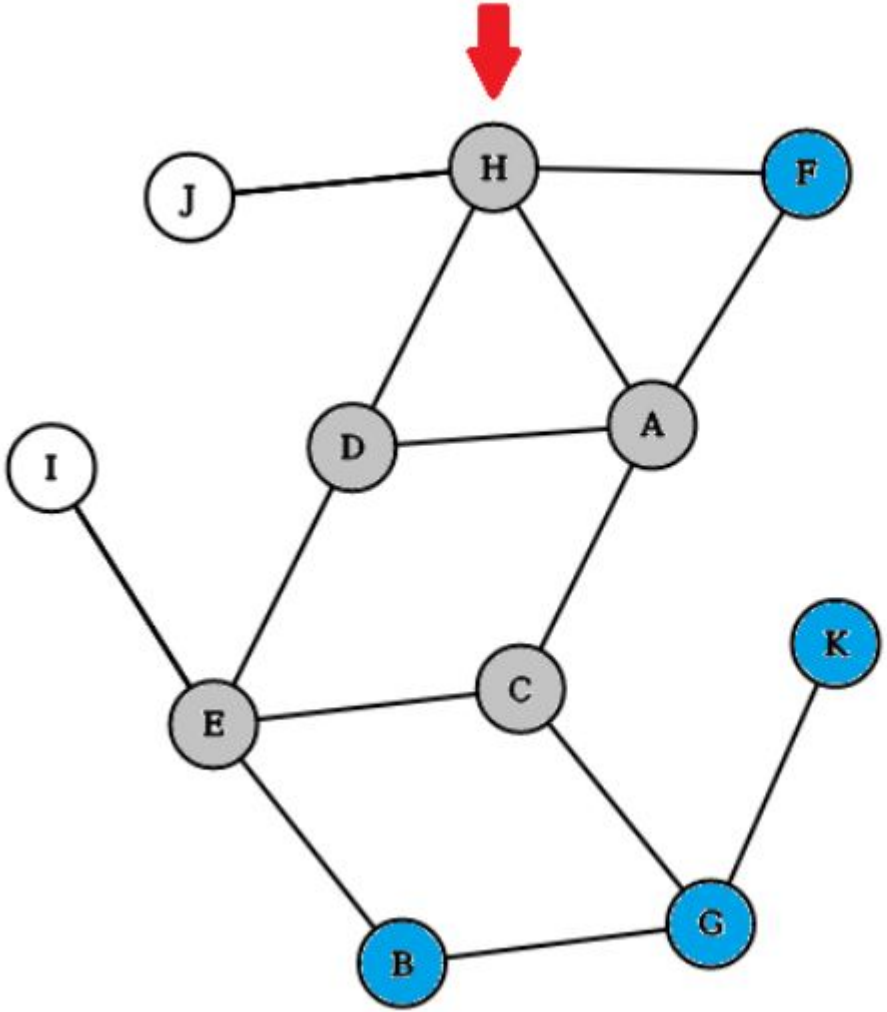
DFS



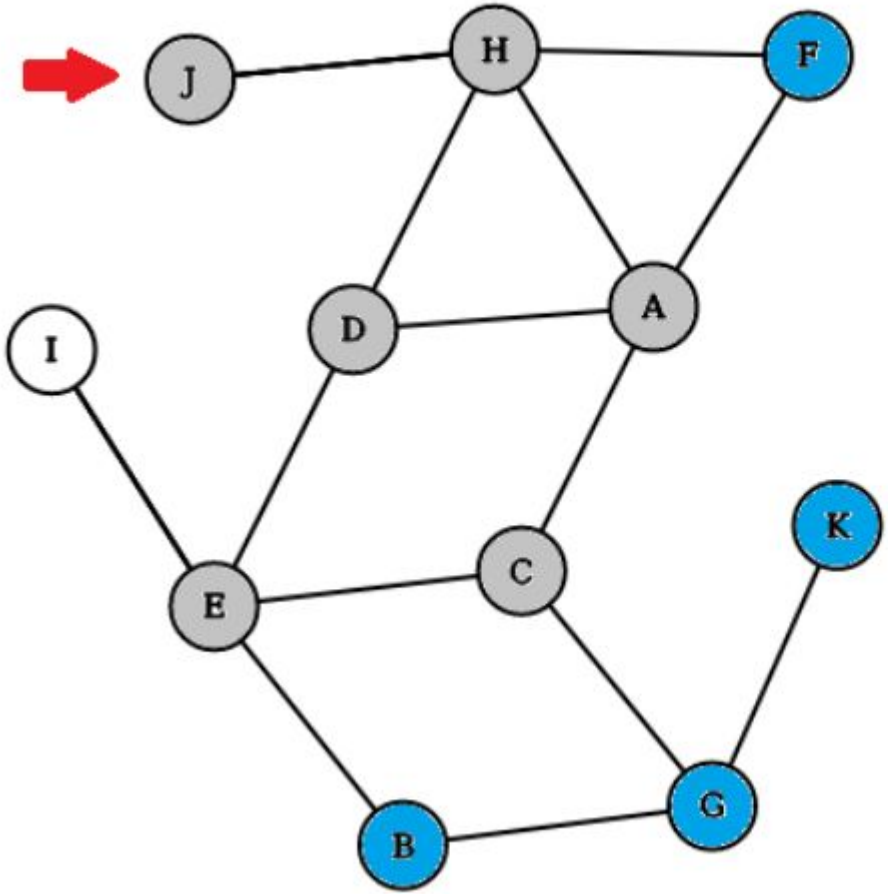
DFS



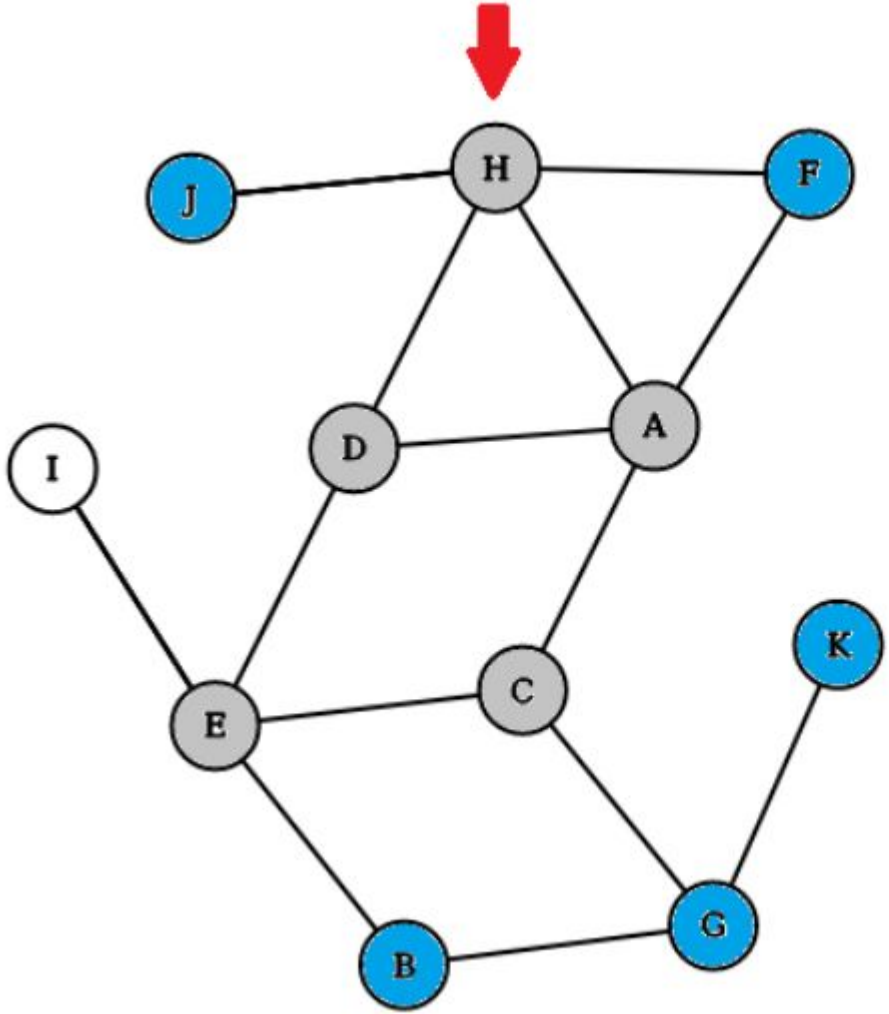
DFS



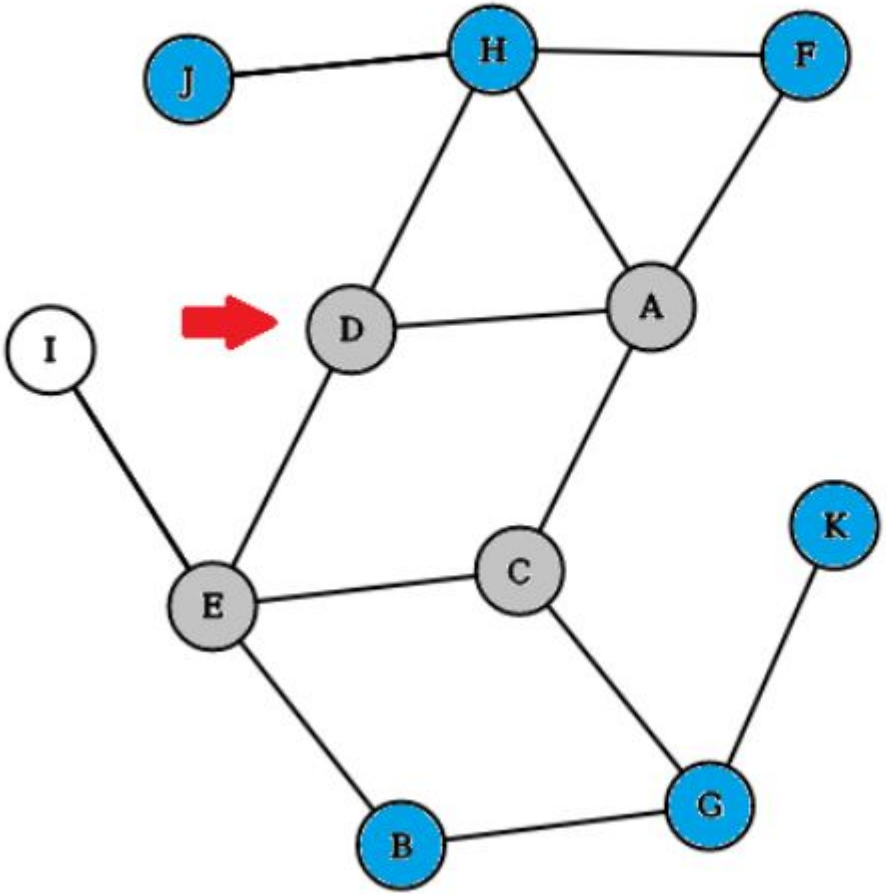
DFS



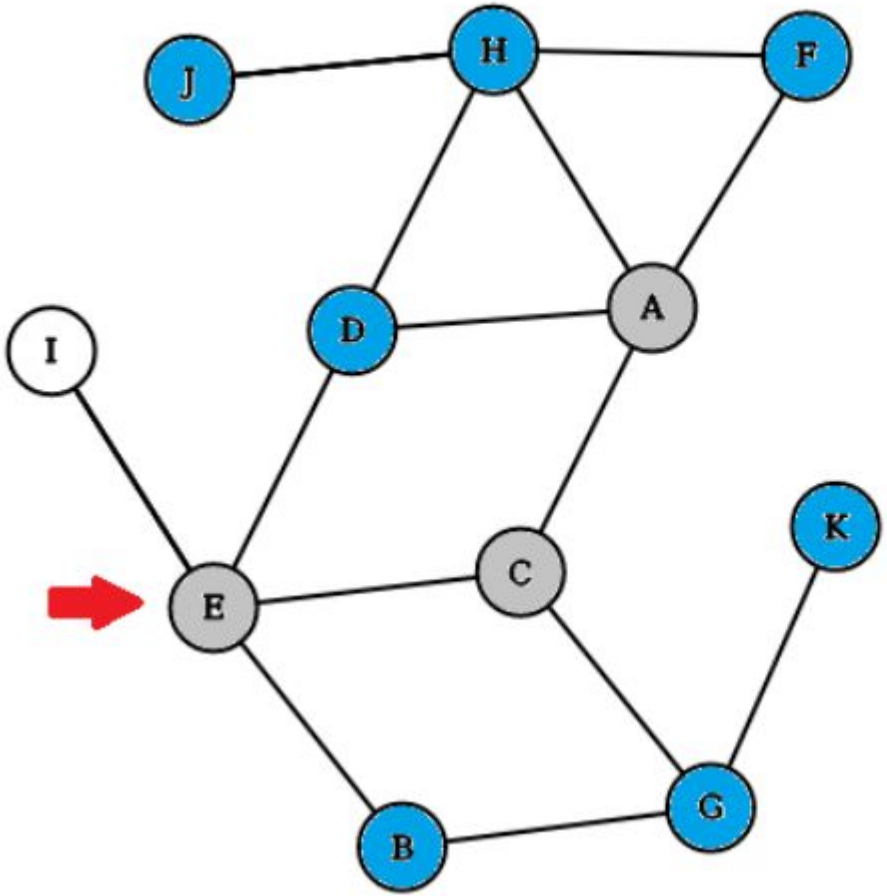
DFS



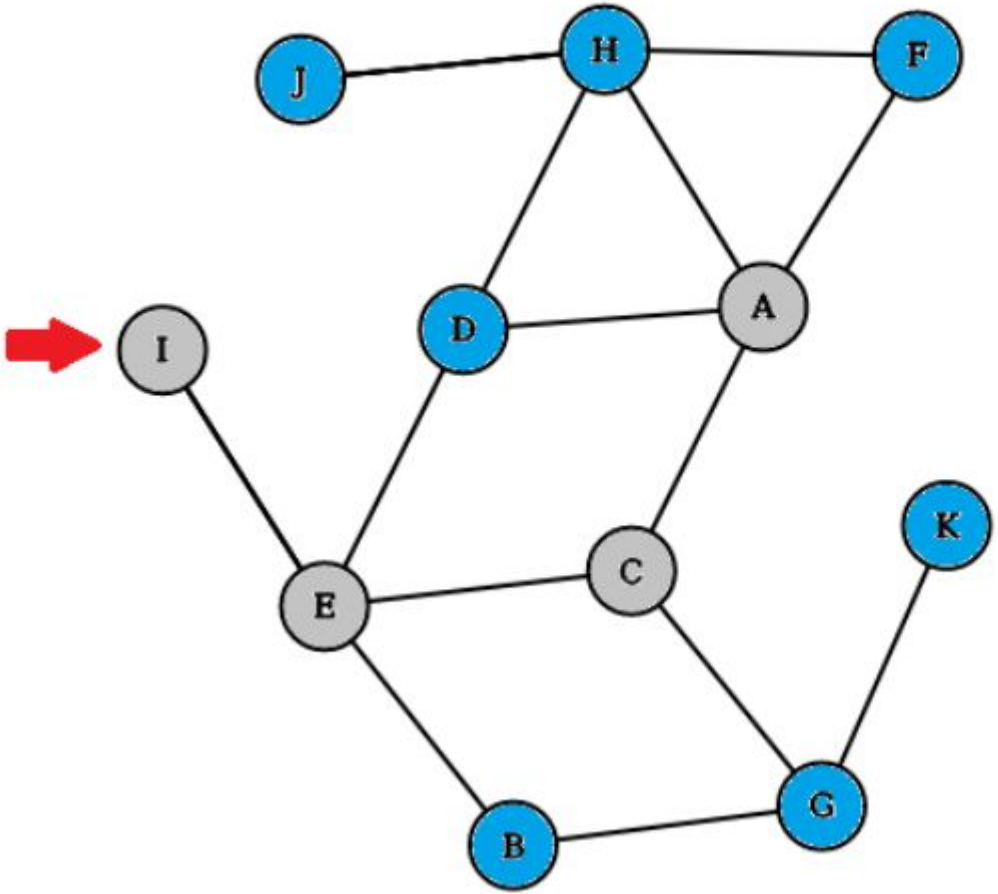
DFS



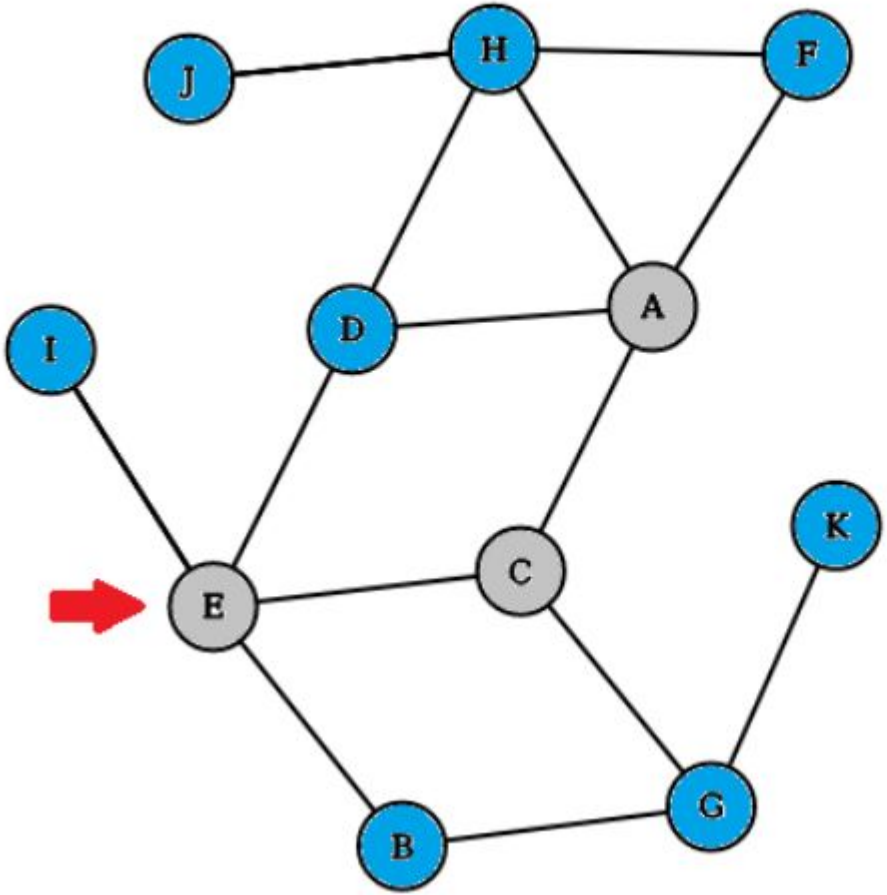
DFS



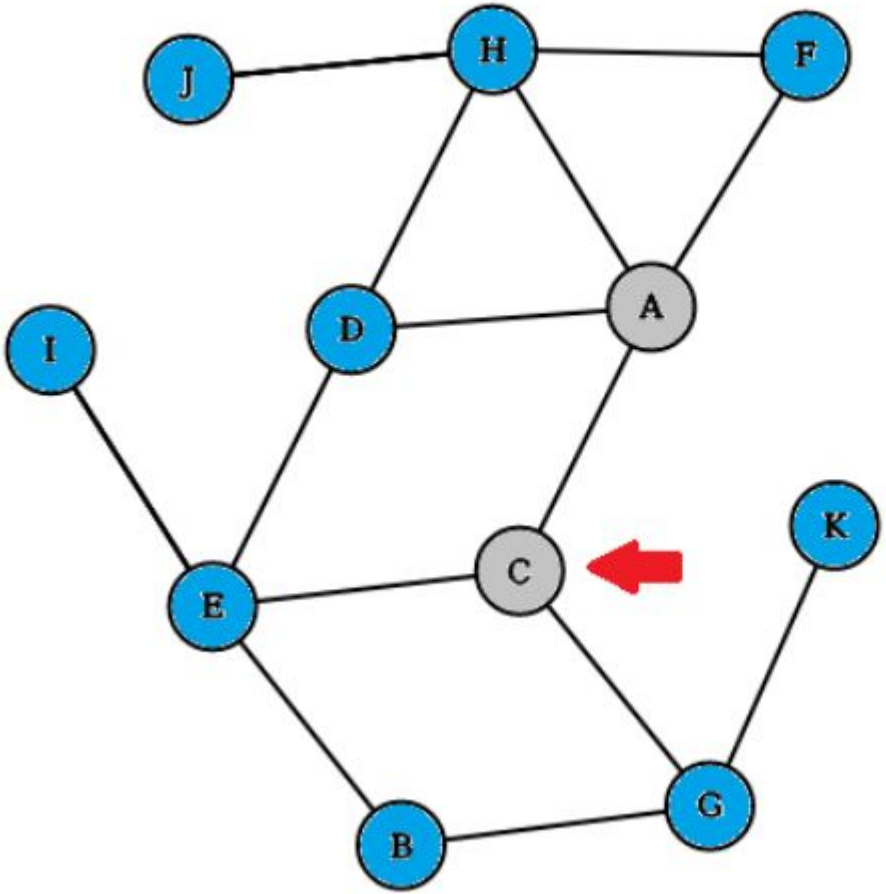
DFS



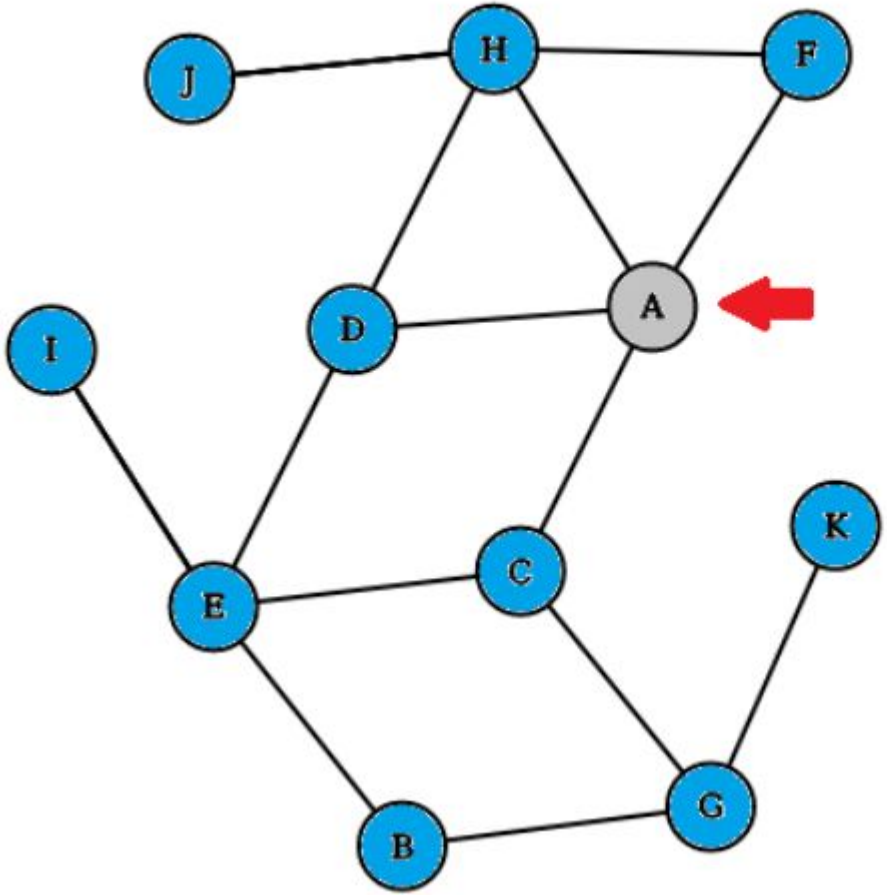
DFS



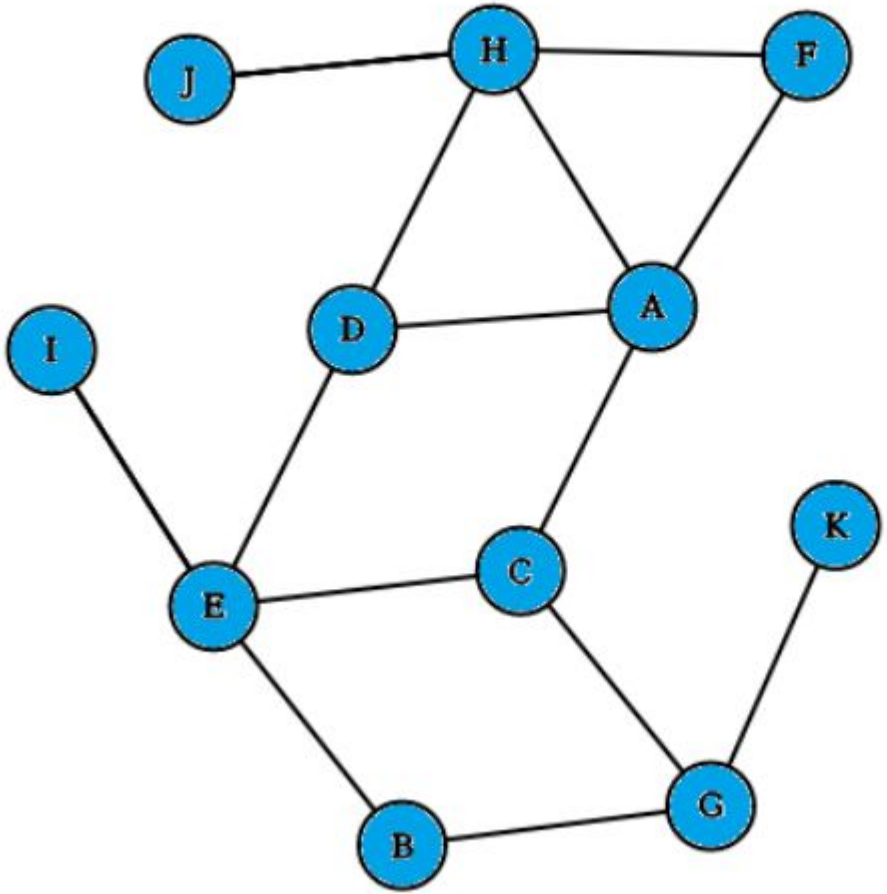
DFS



DFS



DFS



DFS Implementación

```
vector<bool> vis(tam);
void dfs(int node)
{
    vis[node] = 1;
    for(int x : g[node])
        if(!vis[x])
            dfs(x);
}
```

DFS

DFS también es O(m). BFS y DFS son similares: ambos procesan todos los nodos alcanzables desde cierto nodo, pero lo hacen en distinto orden. Muchos problemas salen con ambas técnicas, por lo que podemos usar la que nos quede más cómodo.

Algunos problemas sólo se resuelven con una de ellas:

BFS: distancias mínimas.

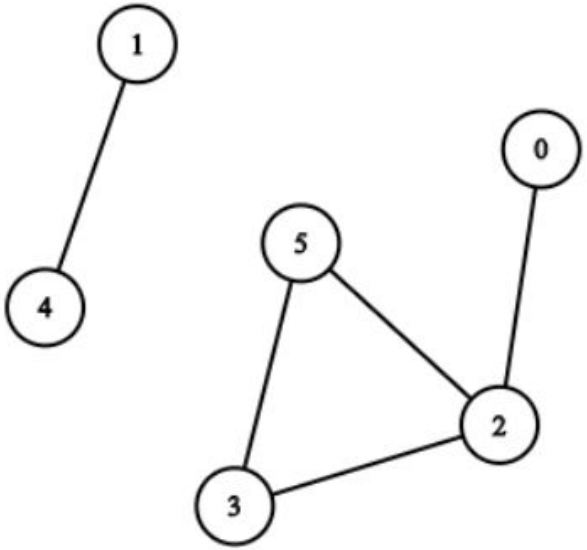
DFS: algunos problemas que tienen que ver con la estructura del grafo, como por ejemplo encontrar puentes y puntos de articulación.

Componentes Conexas

Decimos que dos vértices u y v están conectados si hay un camino (secuencia de aristas con vértice en común) que los une.

- Para todo grafo no-dirigido, podemos particionar el conjunto de nodos en varios subconjuntos, tales que dos nodos están conectados si y sólo si pertenecen al mismo subconjunto.
- Estos subconjuntos se denominan componentes conexas del grafo.

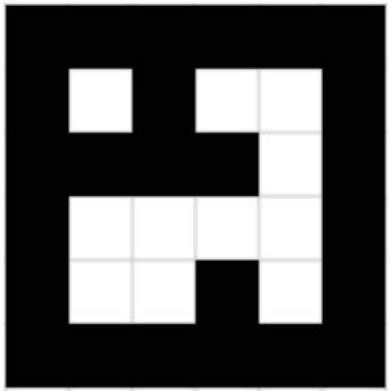
Componentes Conexas

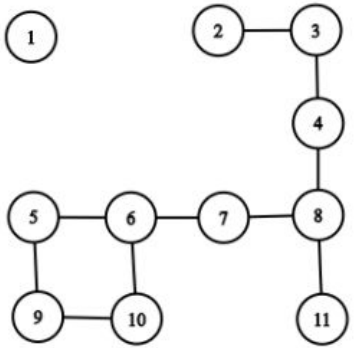


Las componentes conexas son $\{1, 4\}$ y $\{0, 2, 3, 5\}$

Flood fill

Dado un tablero donde un 0 representa el color negro y un 1 el color blanco. Contar la cantidad de componentes blancas, y cuantas celdas tiene cada una.





Se puede usar la matriz como grafo implícito

Flood fill

```
int n, m; // dimensiones del tablero
int dir[2][4] = {{0,0,1,-1}, {1,-1,0,0}}; // direcciones
vector<vector<int>> tab, visi;
int floodfill(int x, int y) {
    if(x < 0 || y < 0 || x >= n || y >= m || visi[x][y] || tab[x][y] == 0)
        return;
    visi[x][y] = 1;
    int ret = 1;
    for(int i = 0; i < 4; i++)
        ret += floodfill(x + dir[0][i], y + dir[1][i]);
    return ret;
}
```