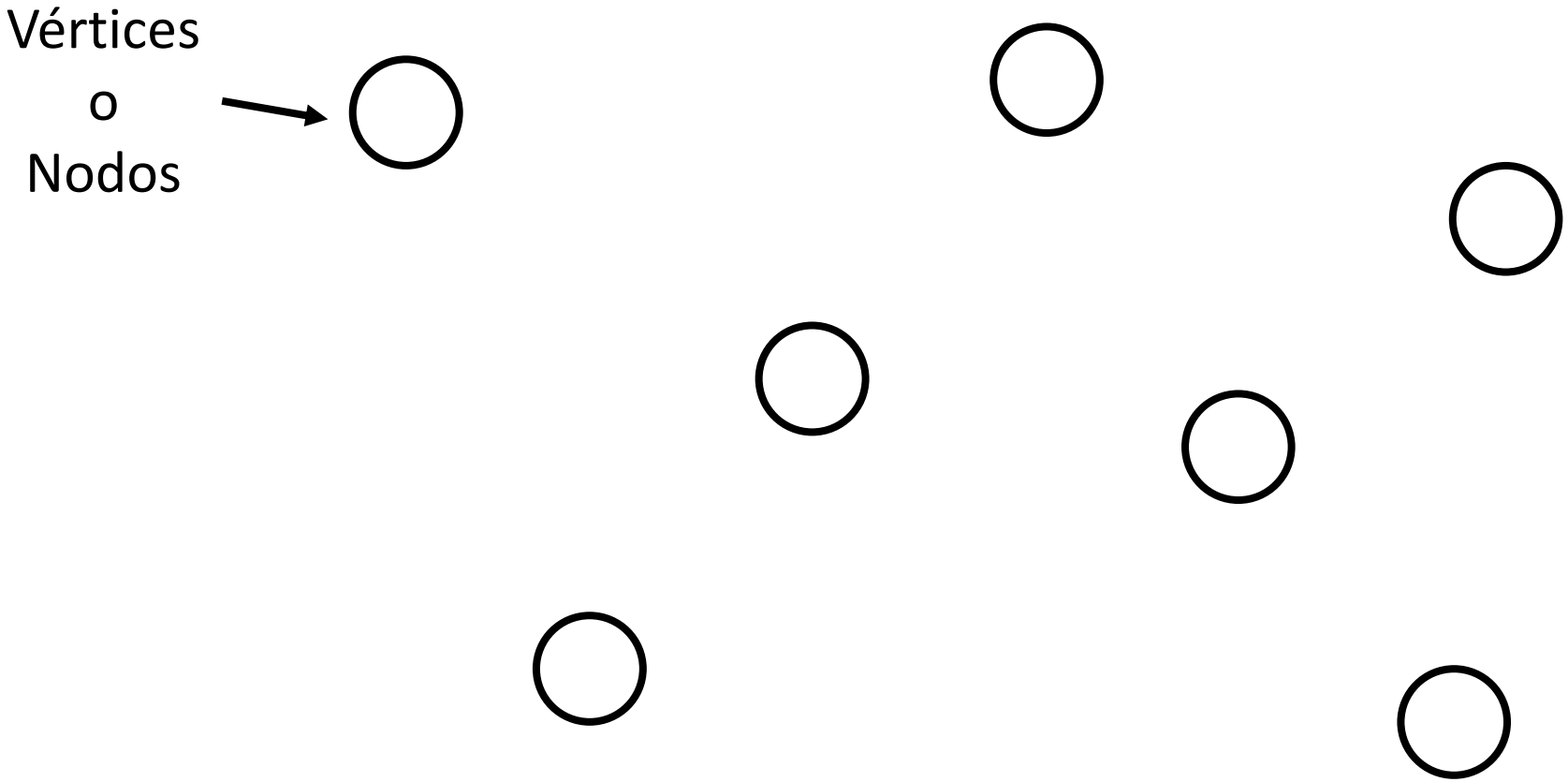
Introducción a grafos, DFS, BFS y sus aplicaciones

Miguel Ortiz

Programación competitiva para ICPC

¿Qué es un grafo?

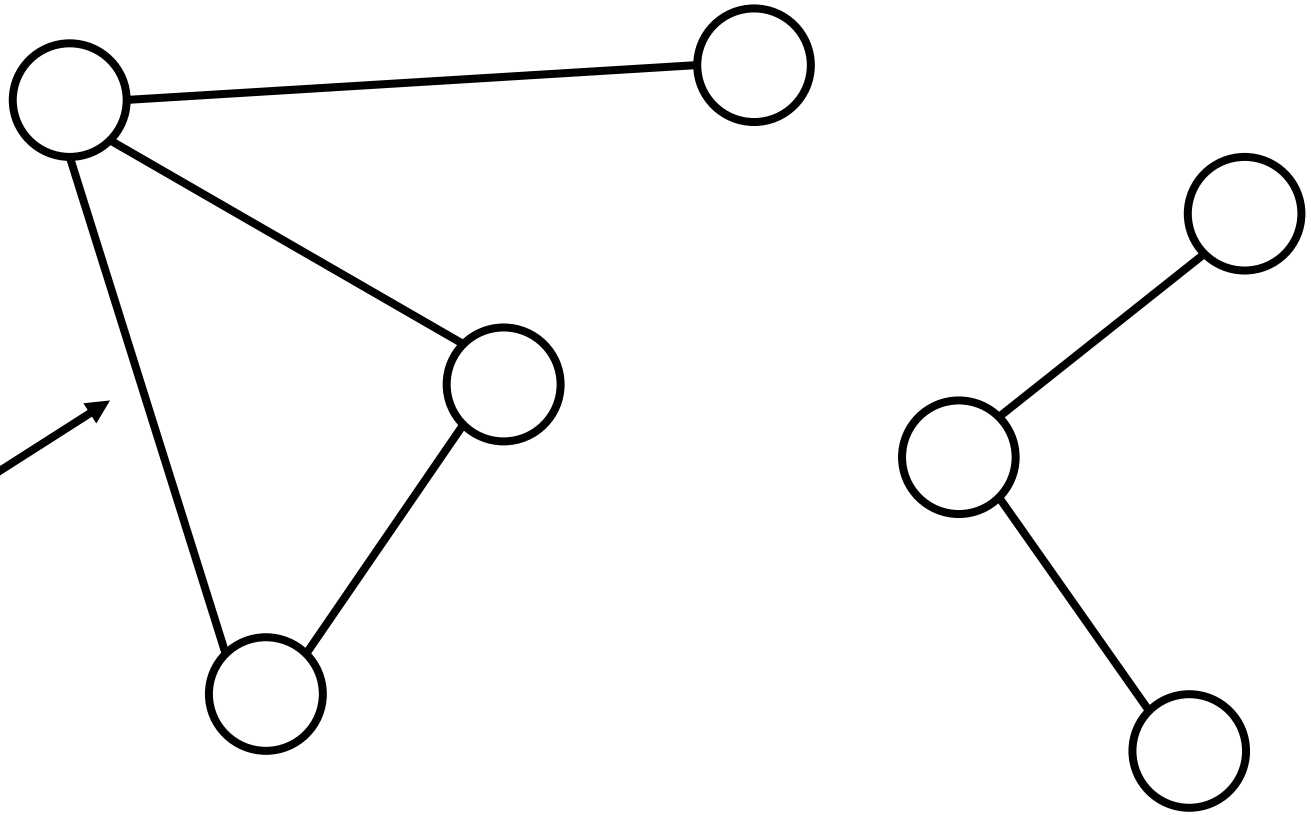
¿Qué es un grafo?



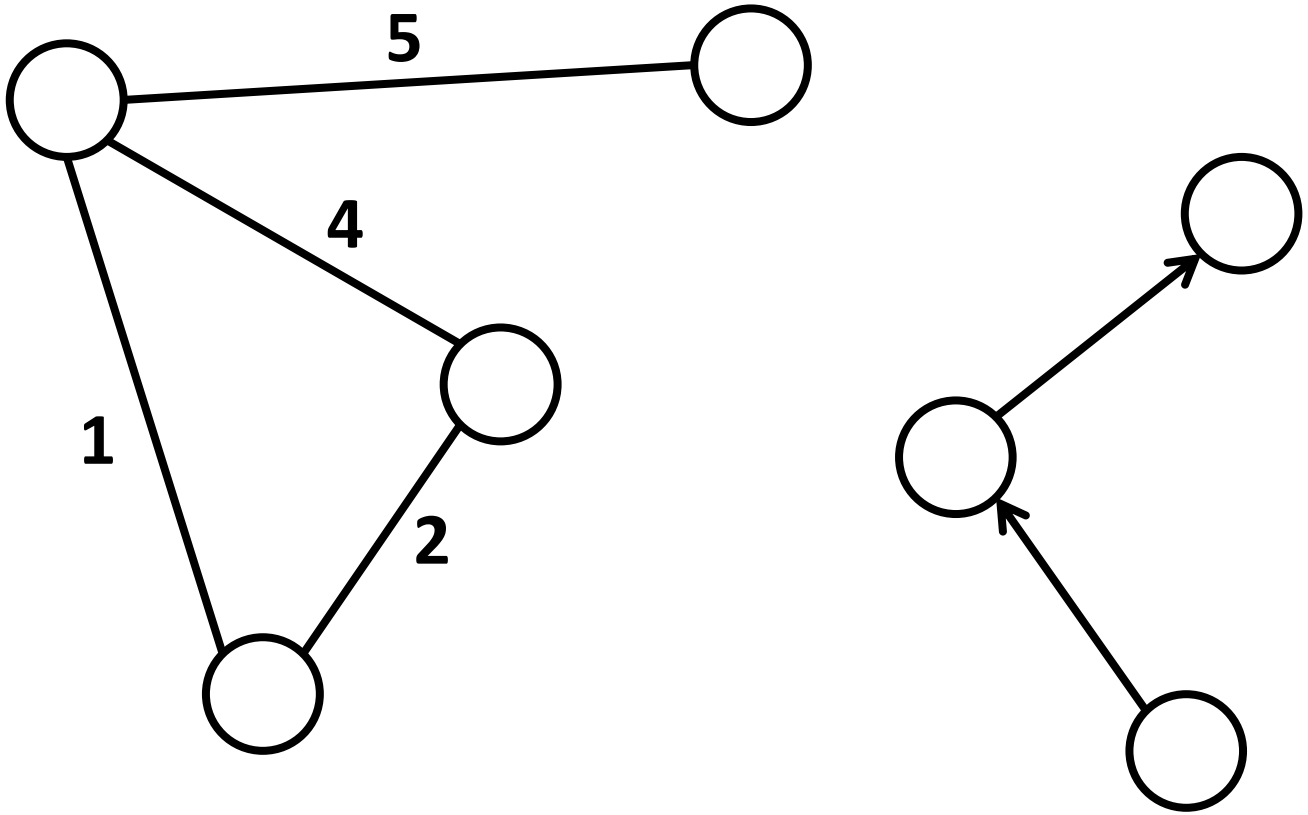
¿Qué es un grafo?

Vértices
o
Nodos

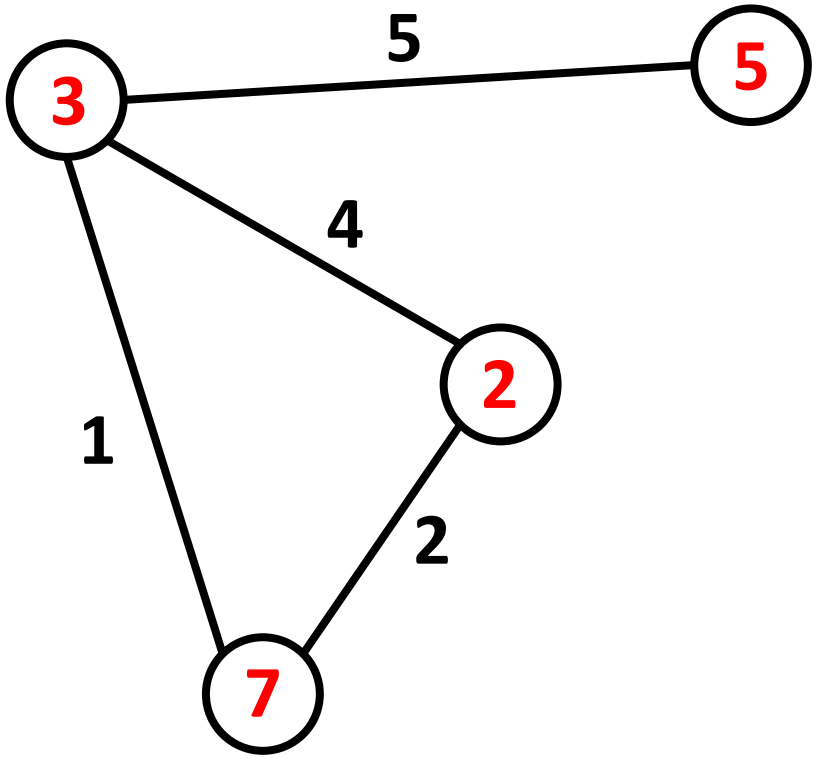
Aristas

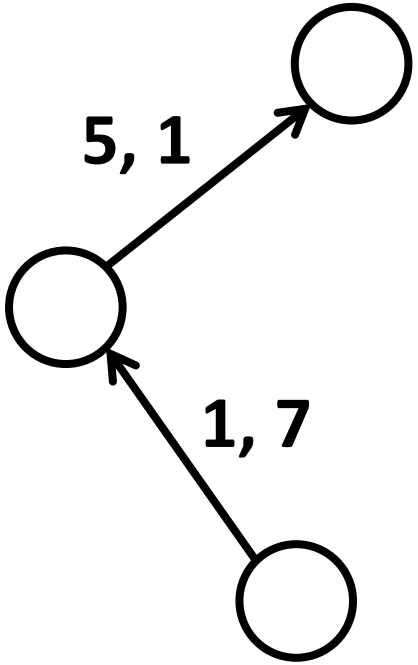


¿Qué es un grafo?

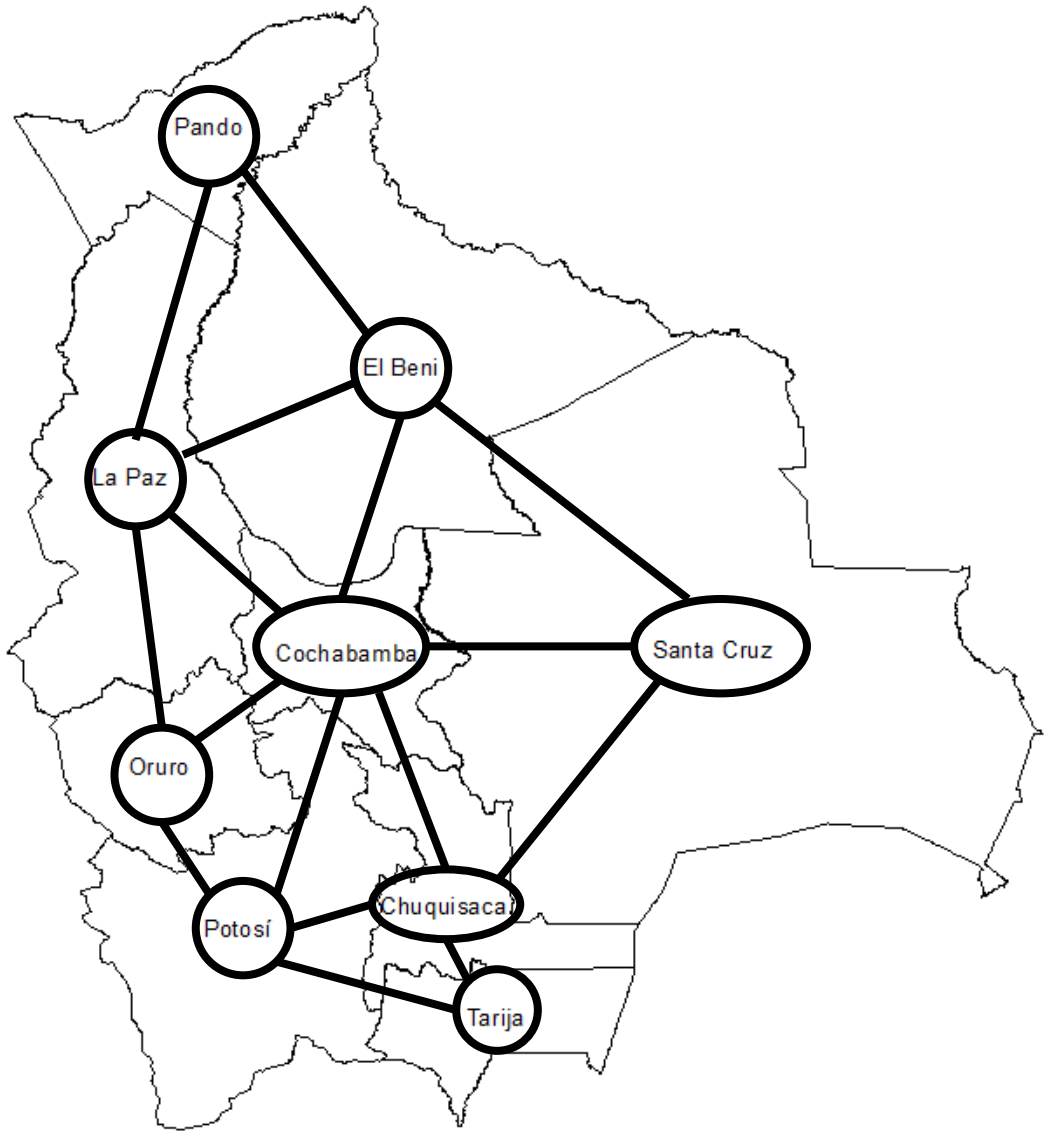


¿Qué es un grafo?

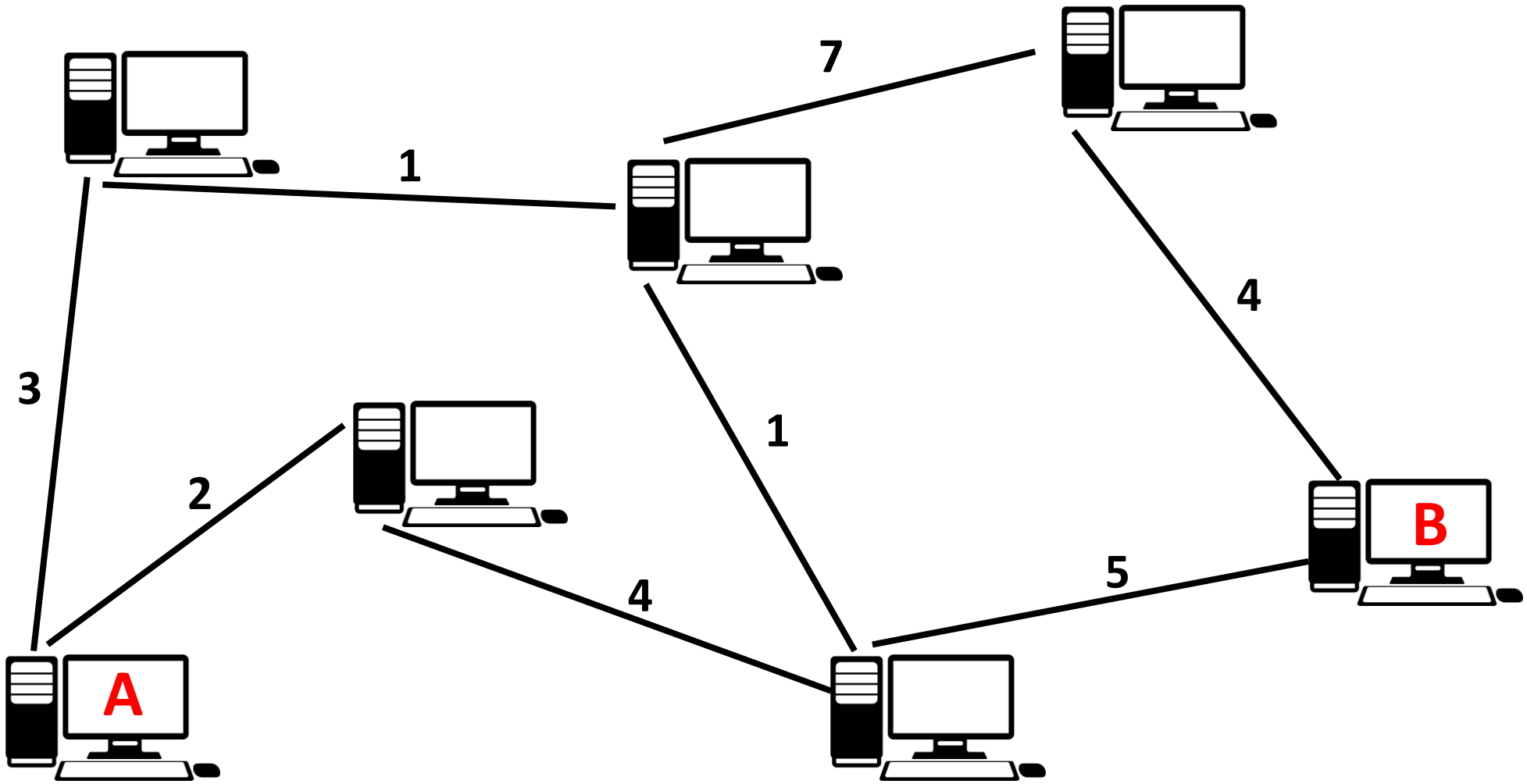




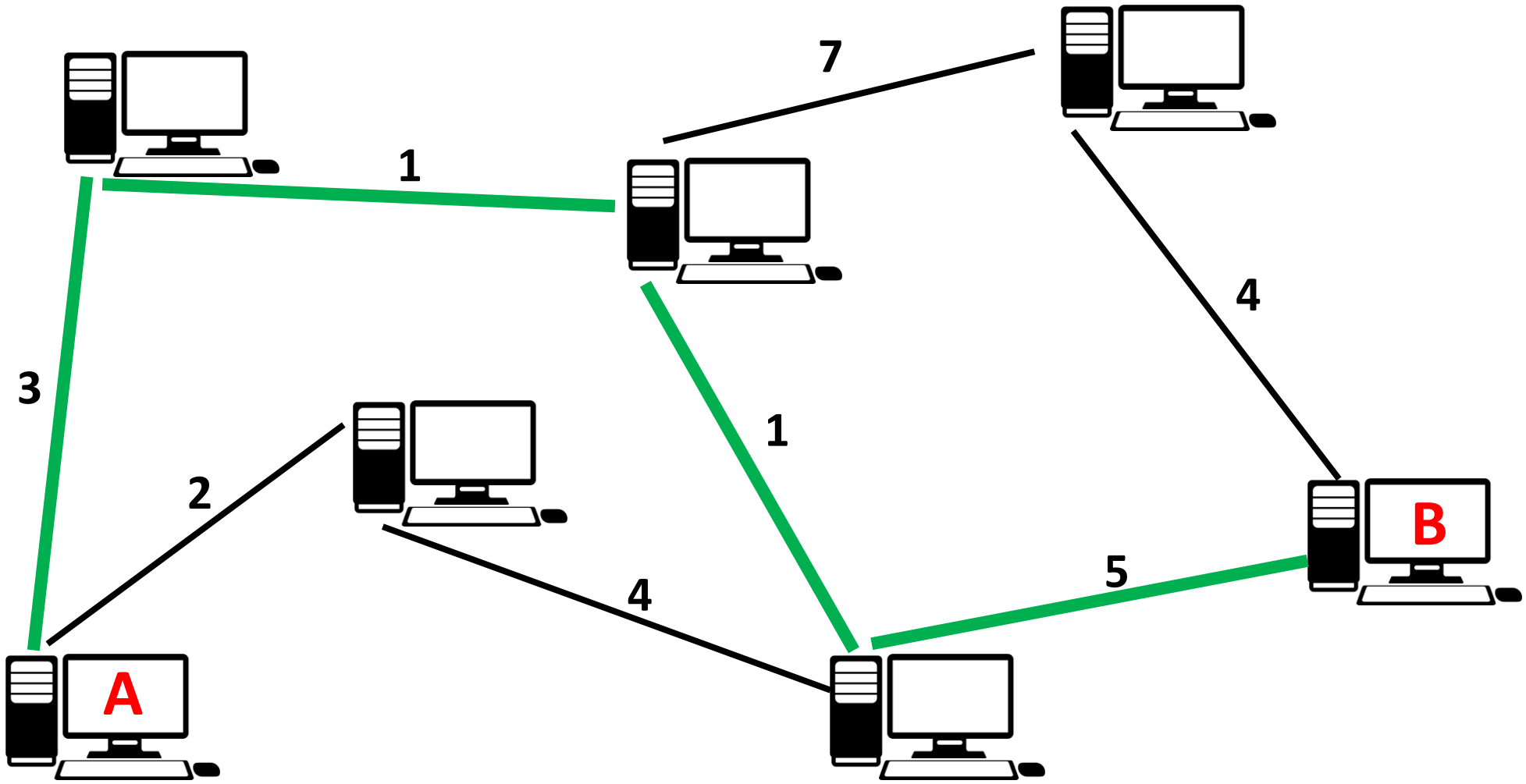
Ejemplos


Pando
El Beni
La Paz
Cochabamba
Santa Cruz
Oruro
Chuquisaca
Potosí
Tarija

Ejemplos



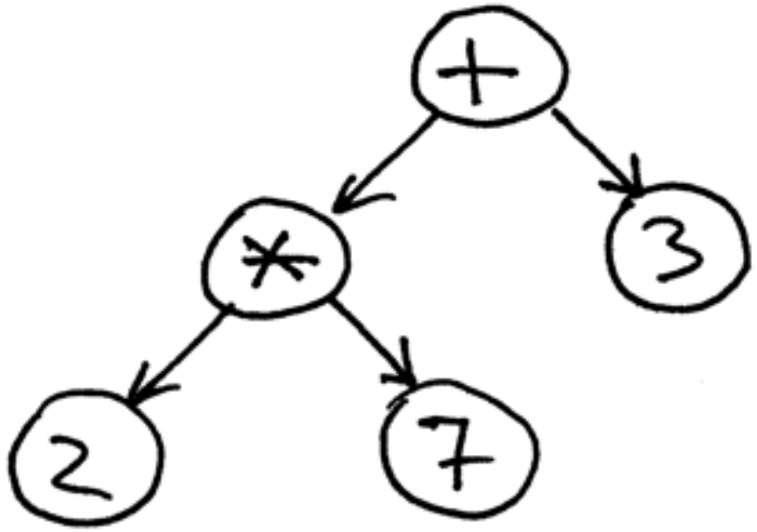
Ejemplos



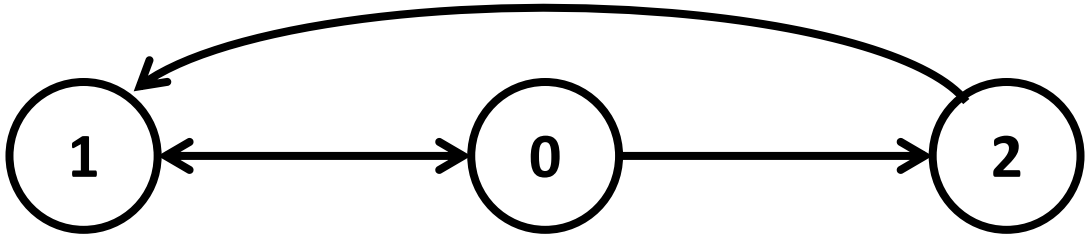
Ejemplos

2 * 7 + 3

AST



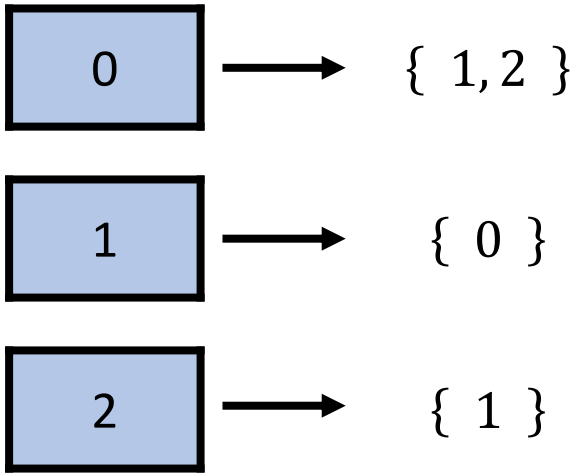
¿Cómo se representan?



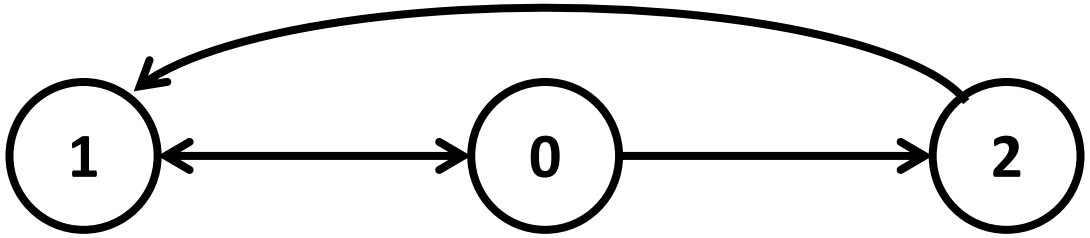
Matriz de adyacencia

Nodo de salida \ Nodo de llegada	0	1	2
0	0	1	1
1	1	0	0
2	0	1	0

Listas de adyacencia



¿Cómo se representan?



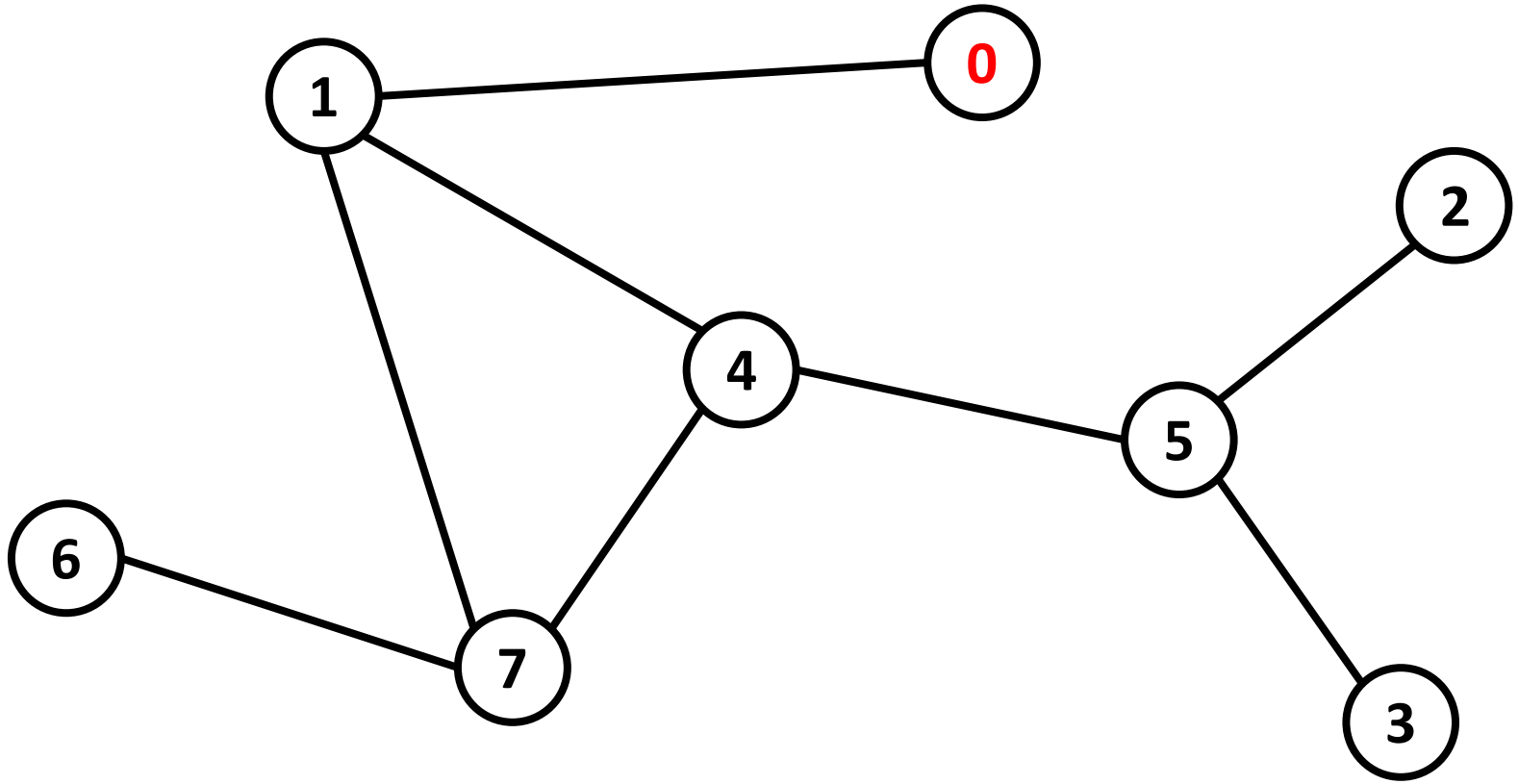

```
// Matriz de adyacencia
int g[n][n];
for (i = 0 -> n-1) {
  for (j = 0 -> n-1) {
    g[i][j] = 0;
  }
}
// Llenado
g[0][1] = 1;
g[0][2] = 1;
g[1][0] = 1;
g[2][1] = 1;
```

```
//Lista de adyacencia
vector<int> g[n];
// vector<vector<int>> g;
// Llenado
g[0].push_back(1);
g[0].push_back(2);
g[1].push_back(0);
g[2].push_back(1);
```

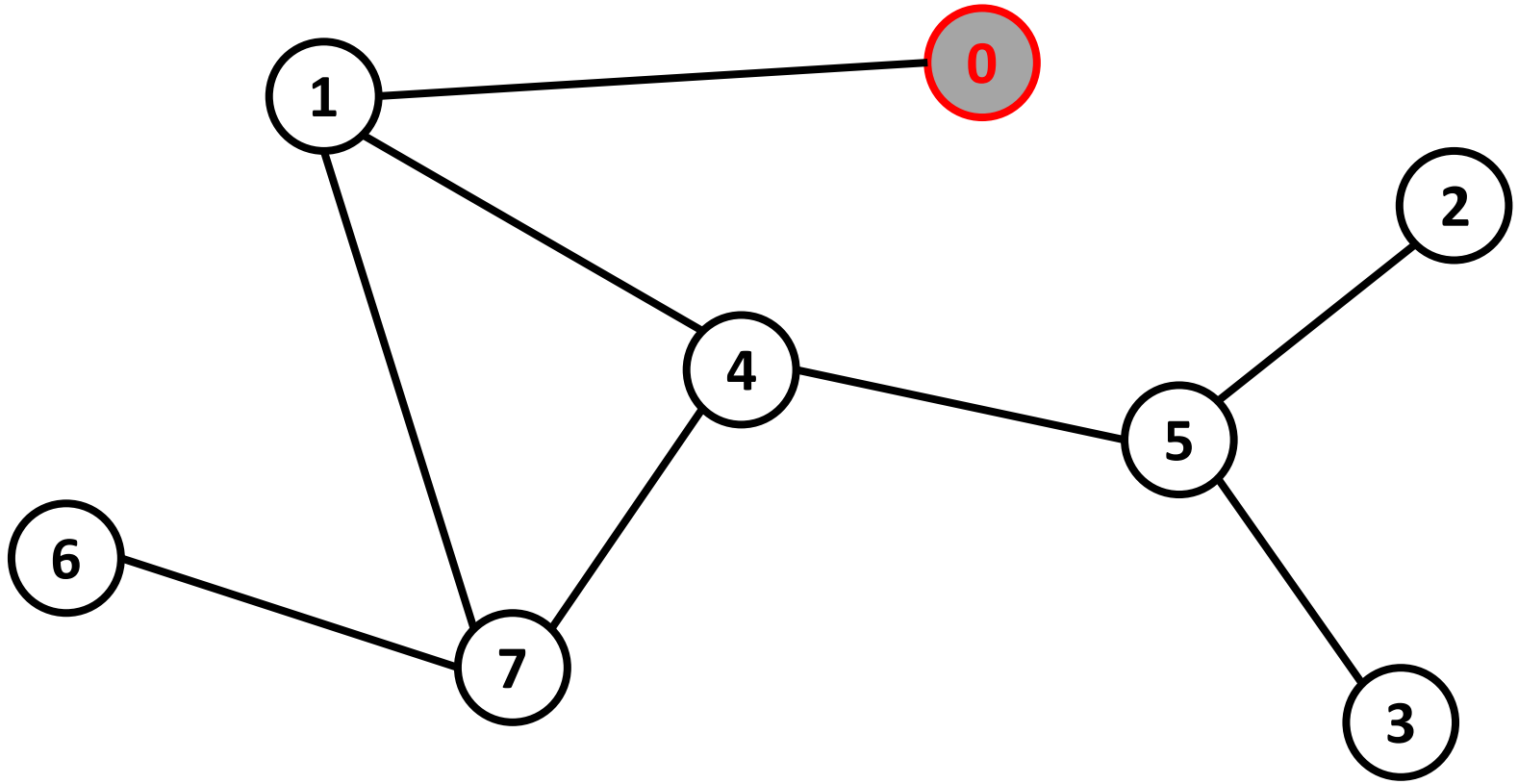
DFS

- Algoritmo de recorrido de grafos
- Empieza en un nodo, lo marca como visitado y se mueve a sus vecinos que aun no han sido visitados
  - Si hay varias opciones para moverse, se mueve a cualquiera
- Cuando no hay vecinos sin marca, vuelve por donde vino
- Hace el mismo procedimiento en cada paso hasta haber visitado todos los nodos posibles
- Sigue un solo camino por el grafo, como caminar por un laberinto con una mano siempre en la pared

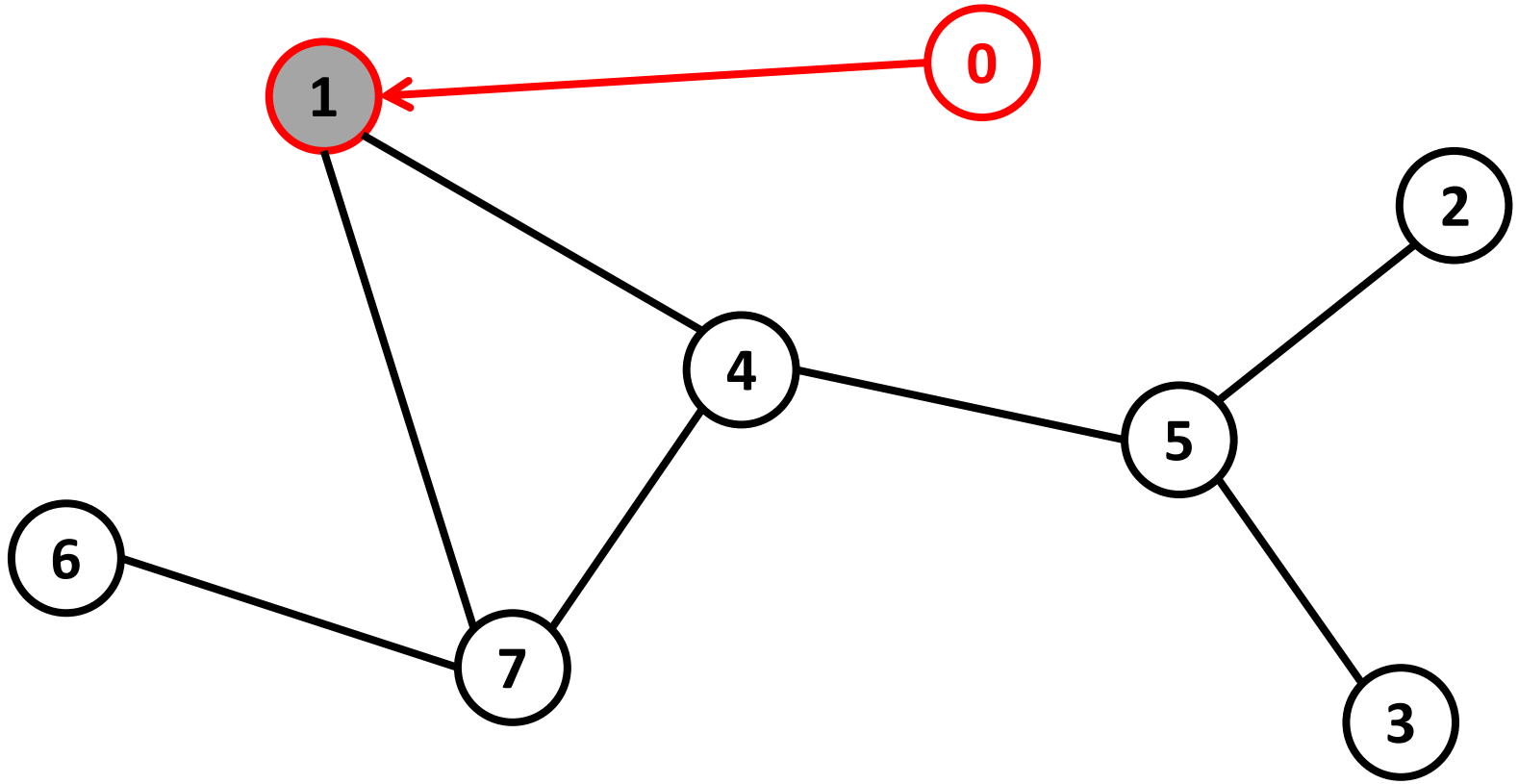
DFS



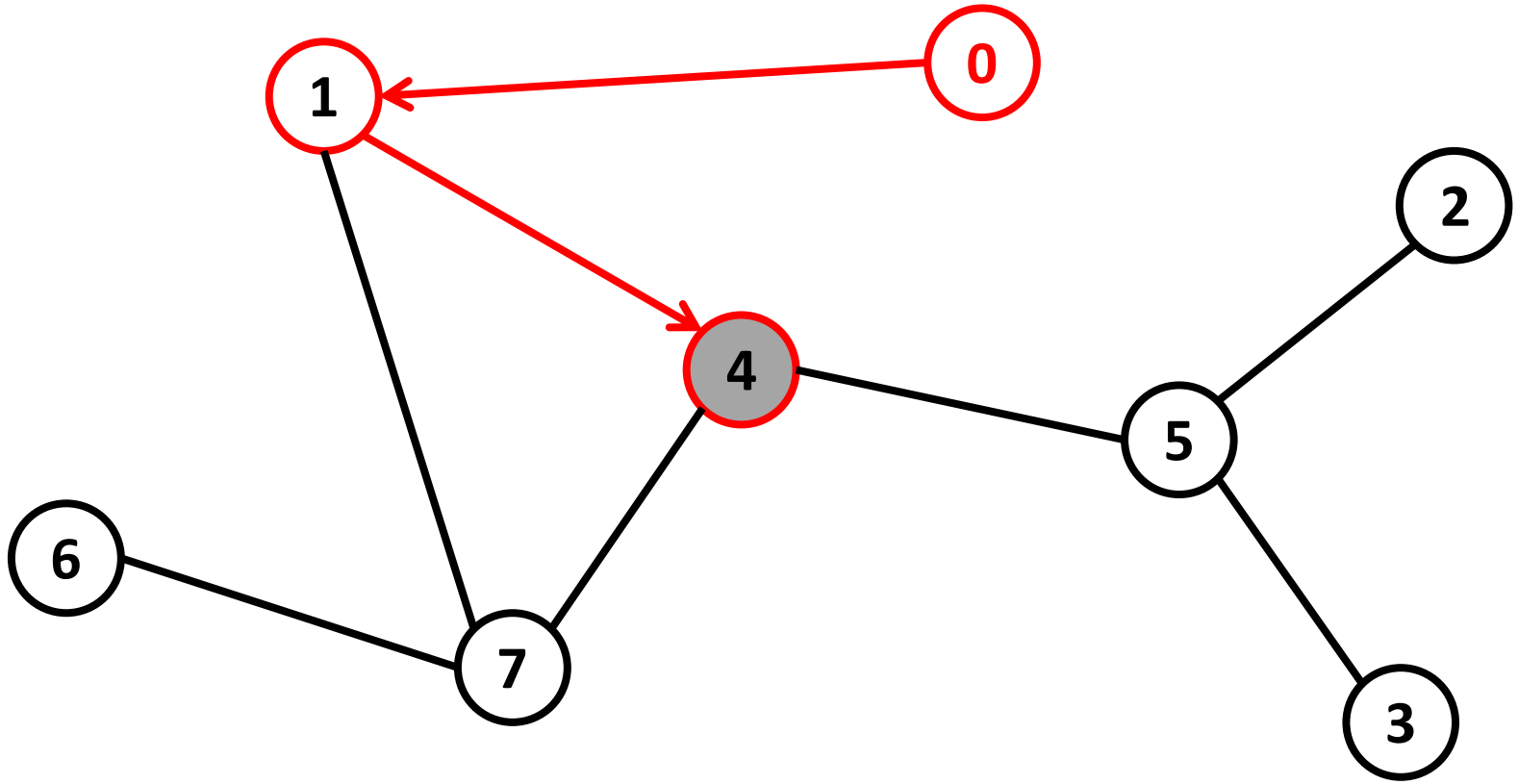
DFS



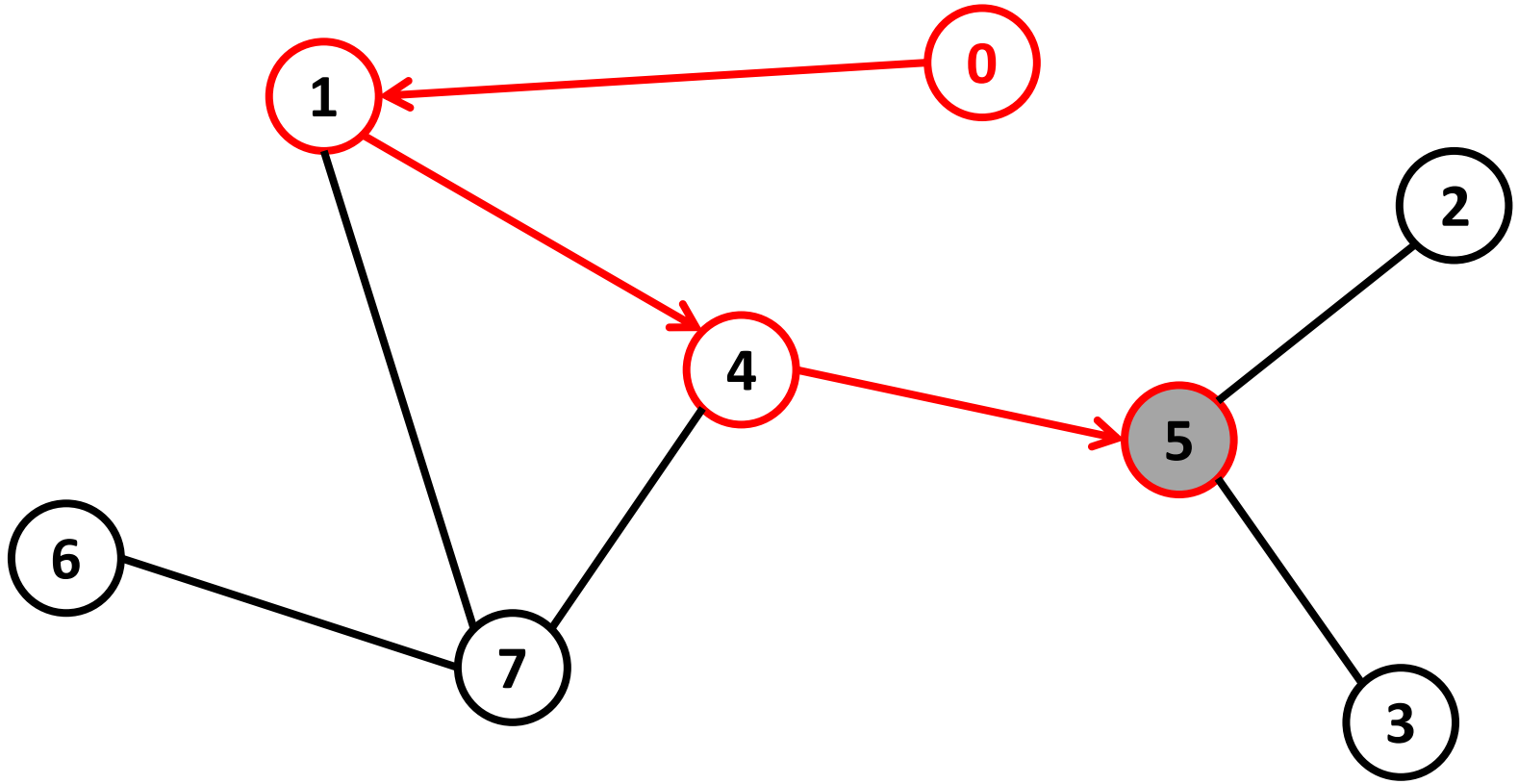
DFS



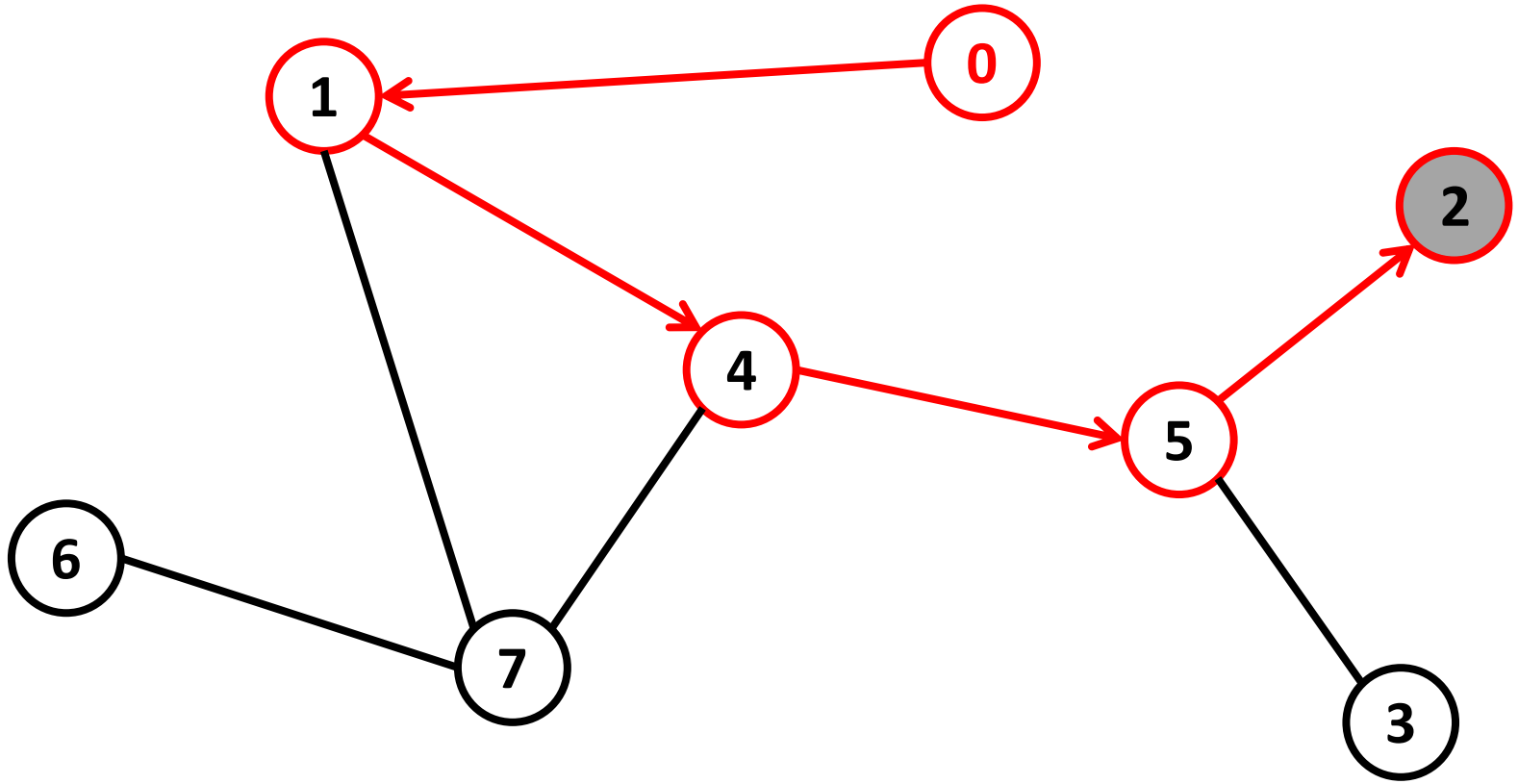
DFS



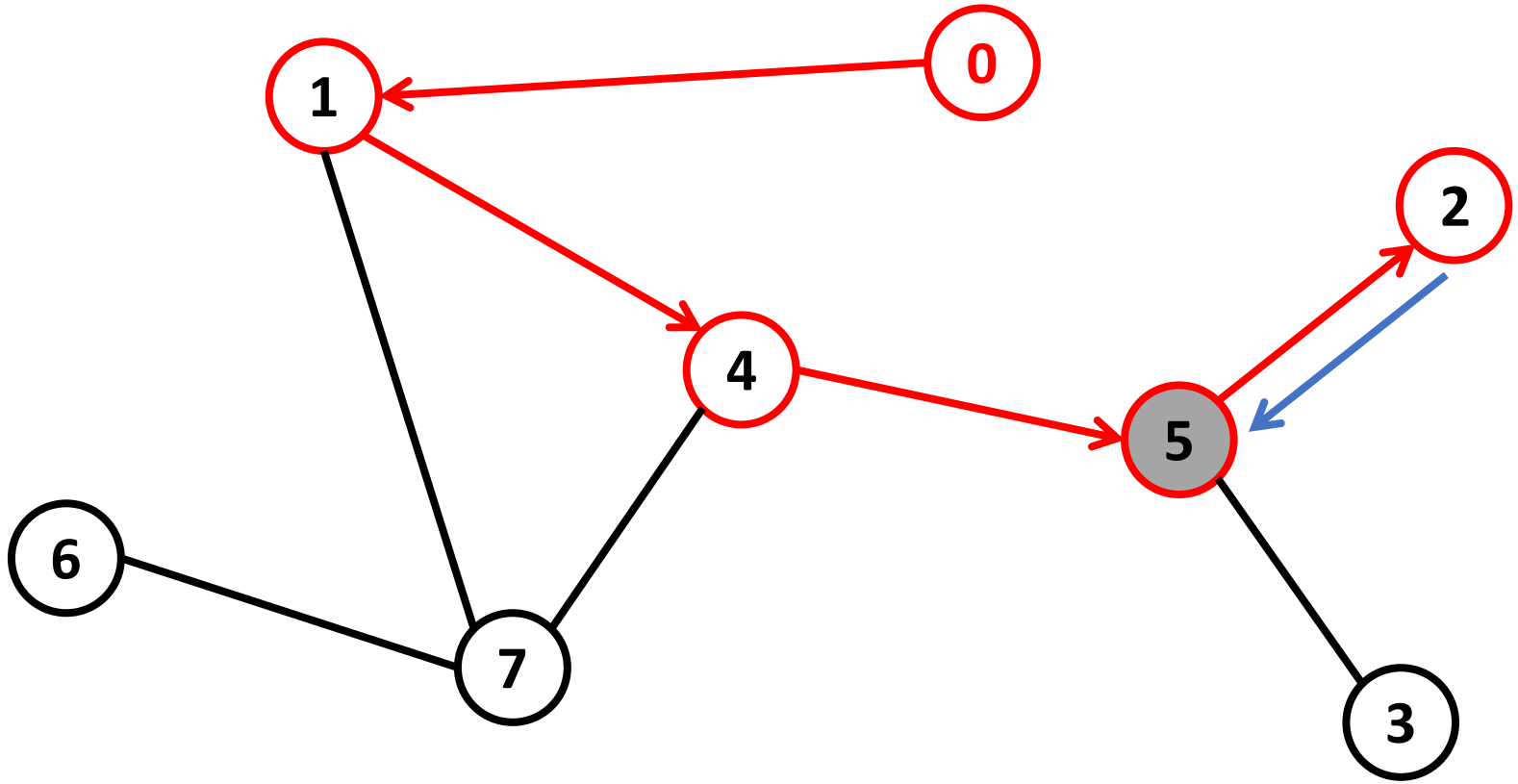
DFS



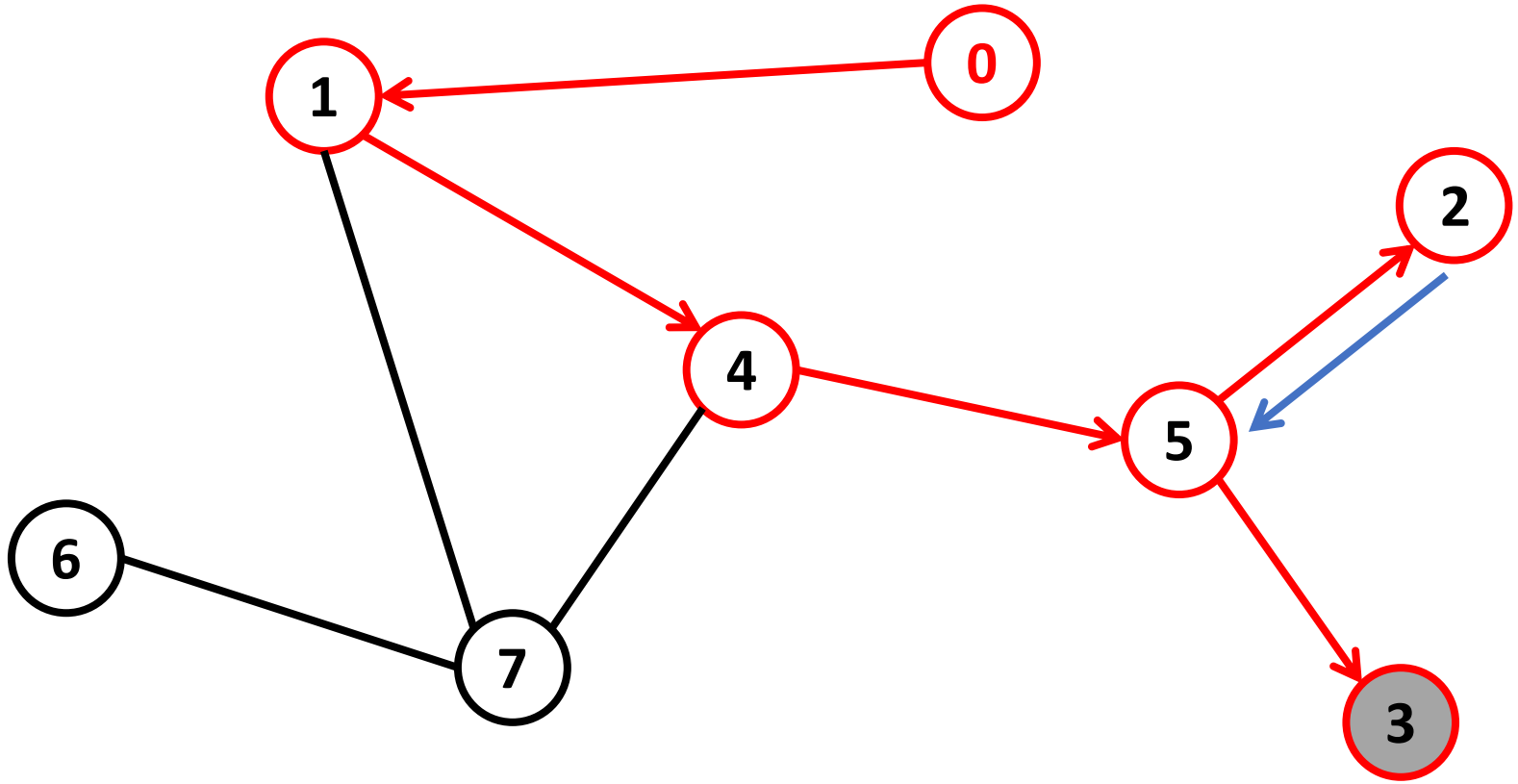
DFS



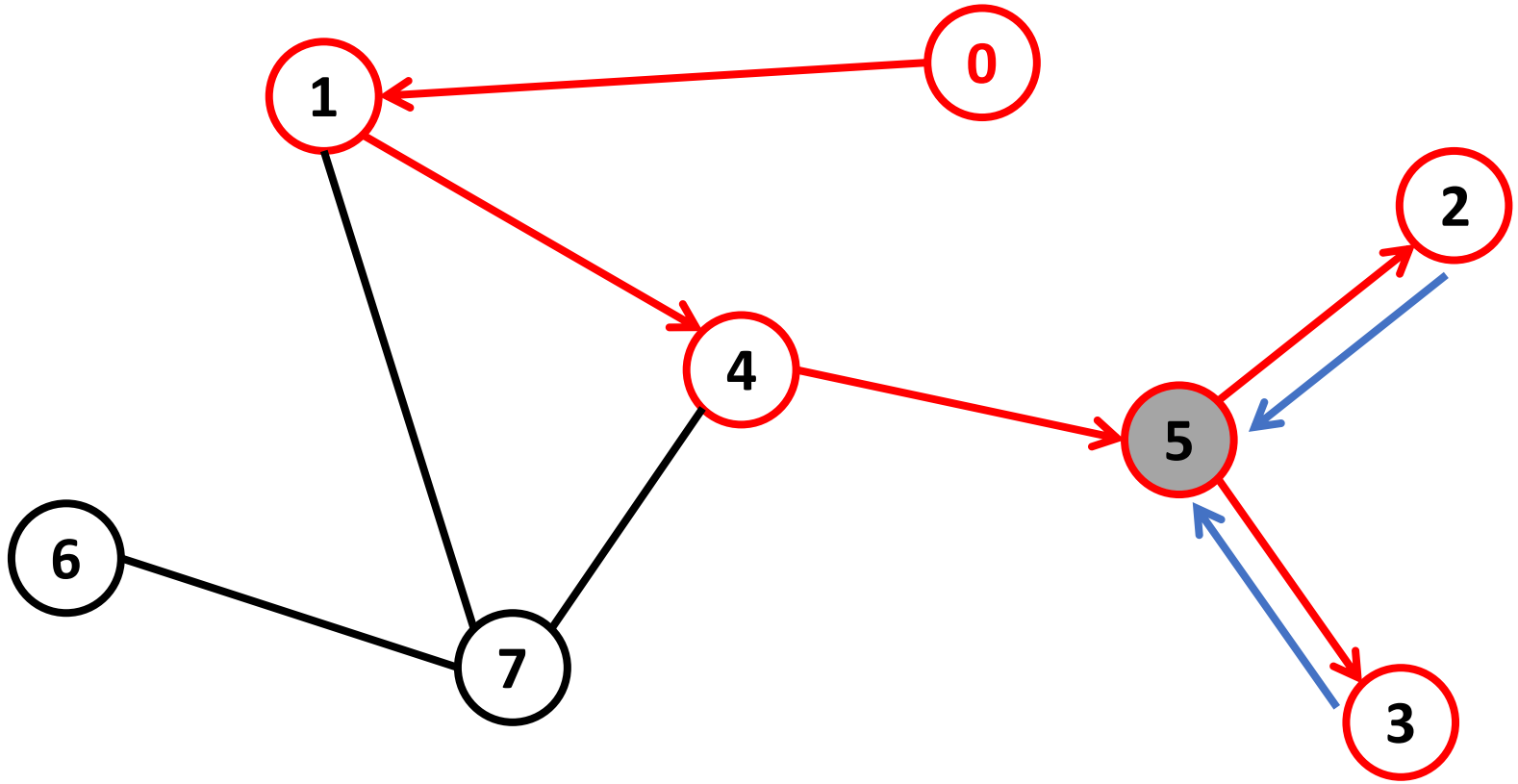
DFS



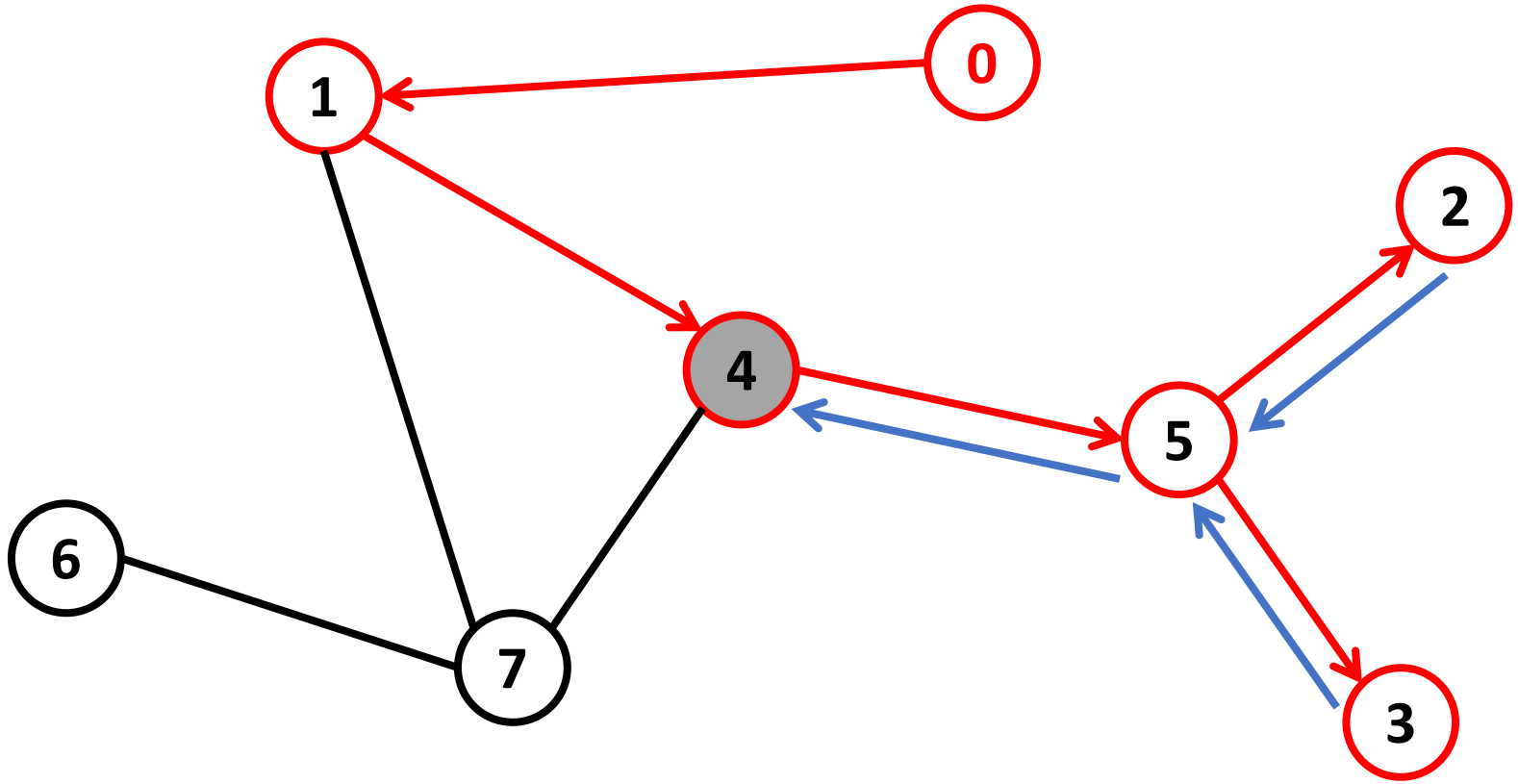
DFS



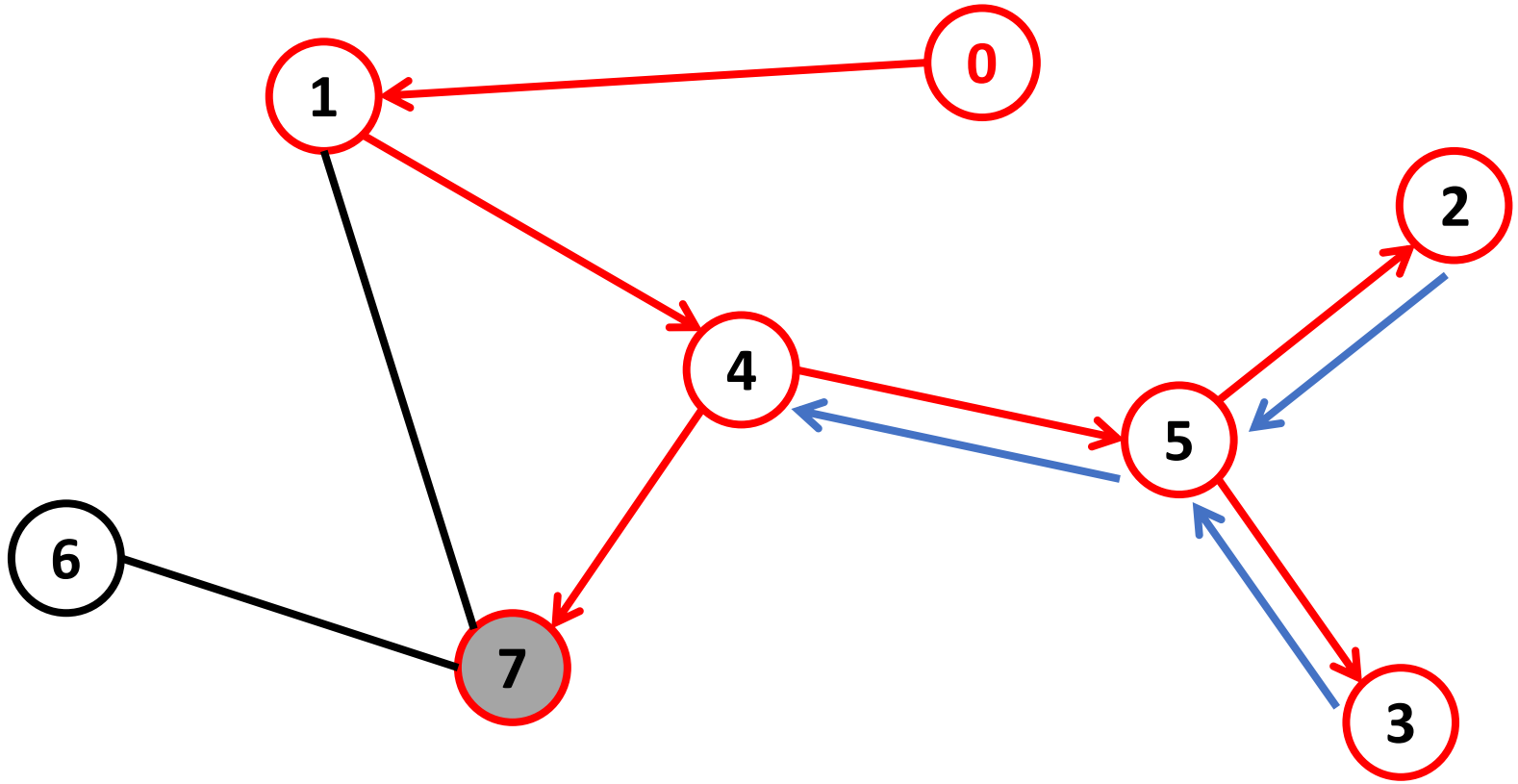
DFS



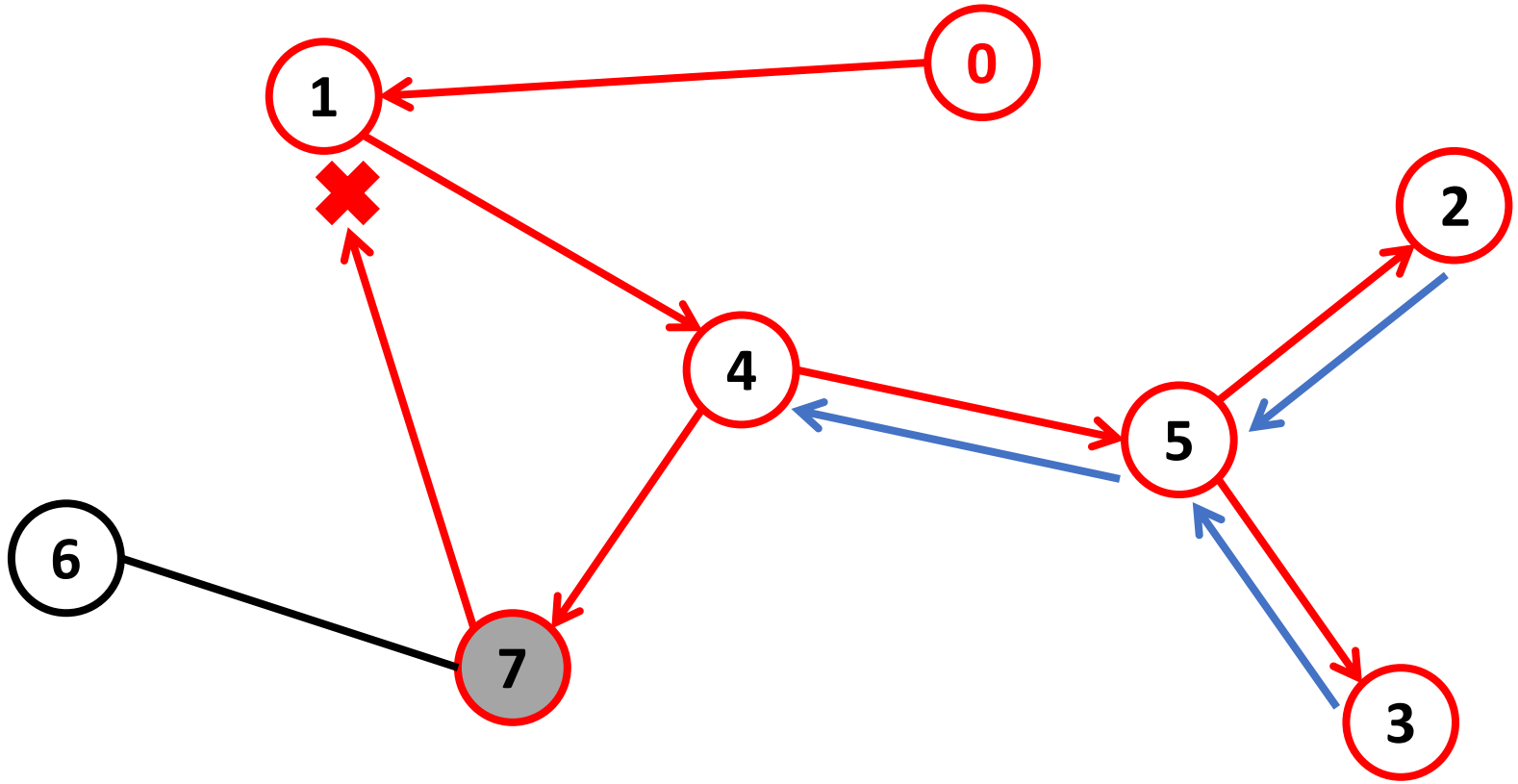
DFS



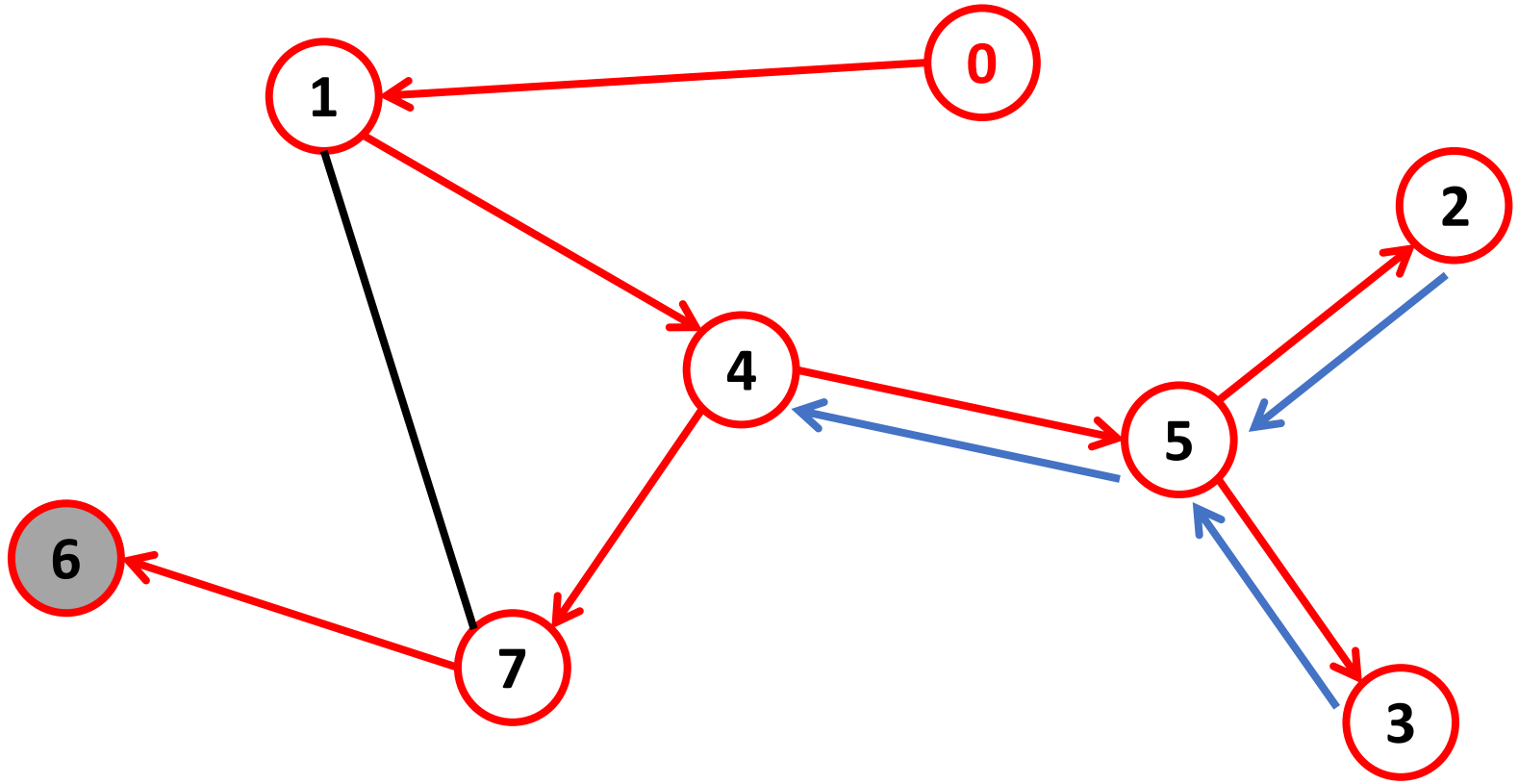
DFS



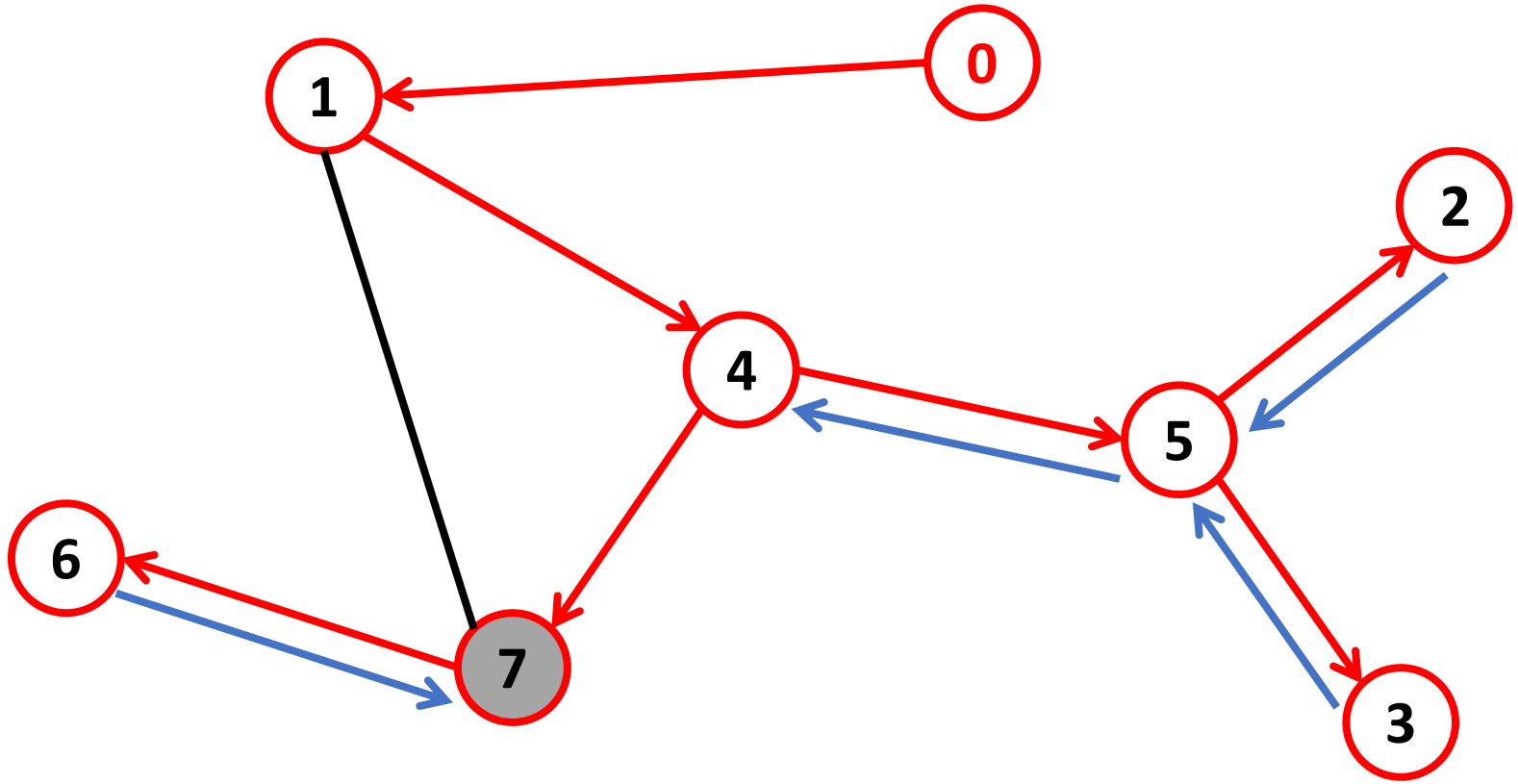
DFS



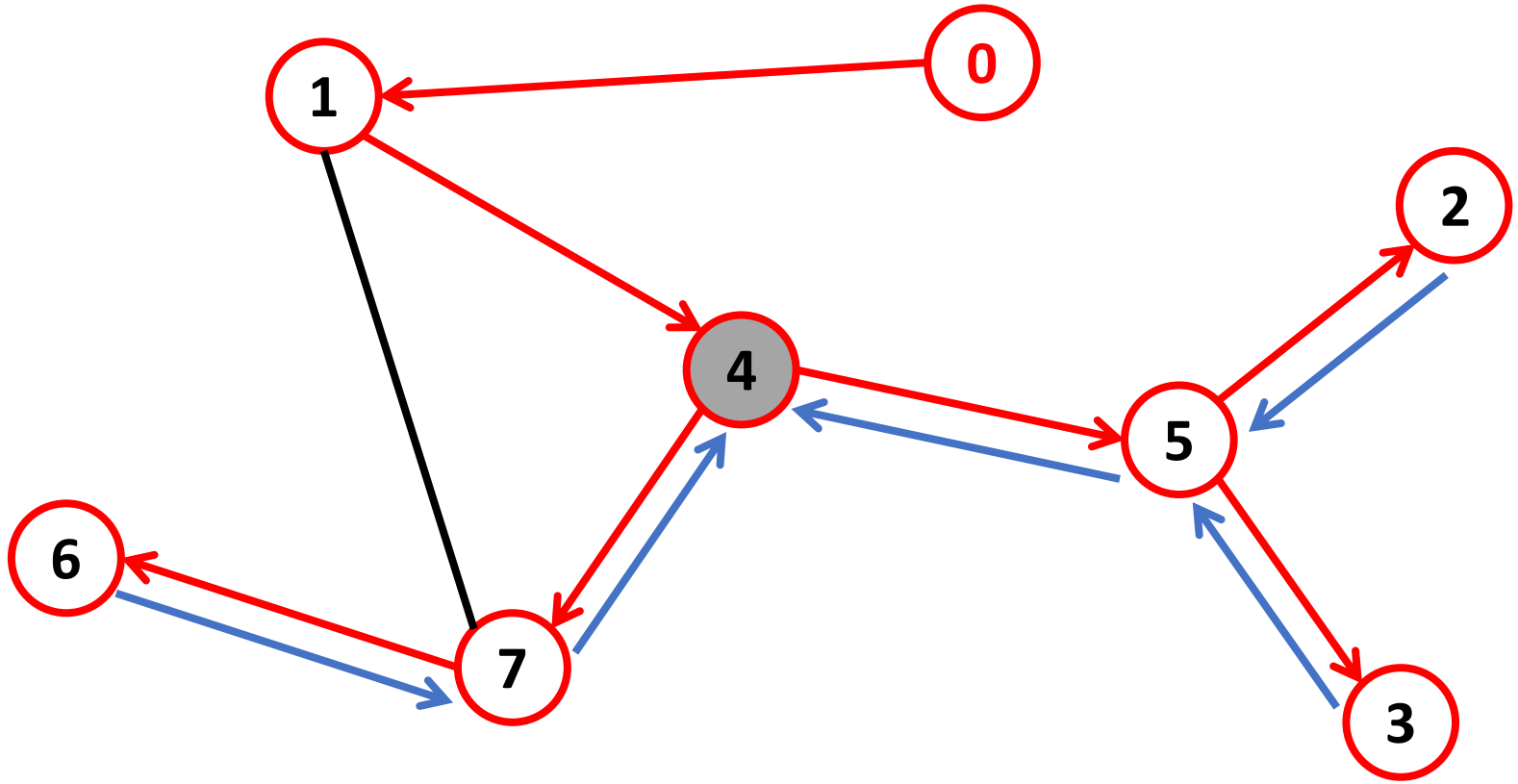
DFS



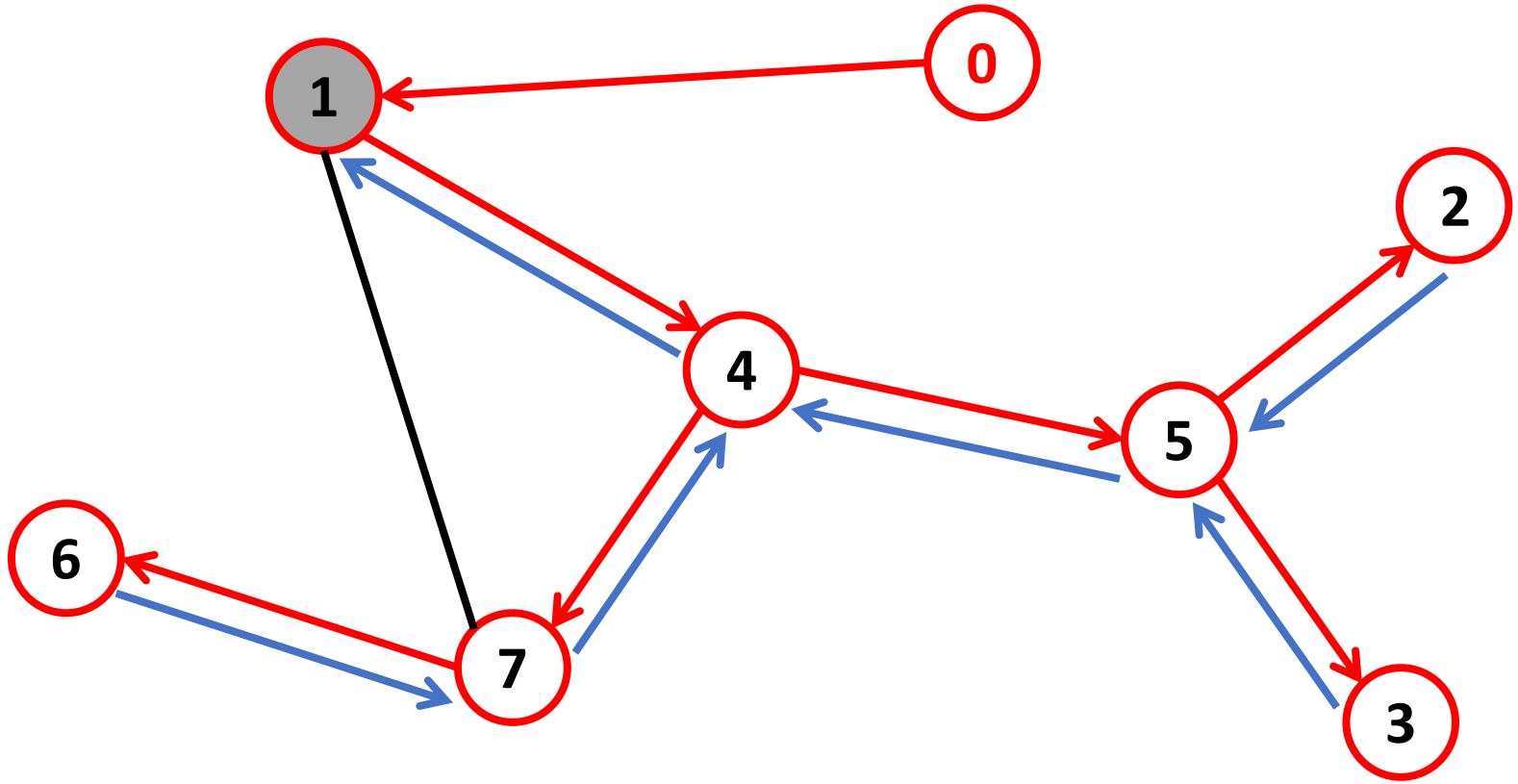
DFS



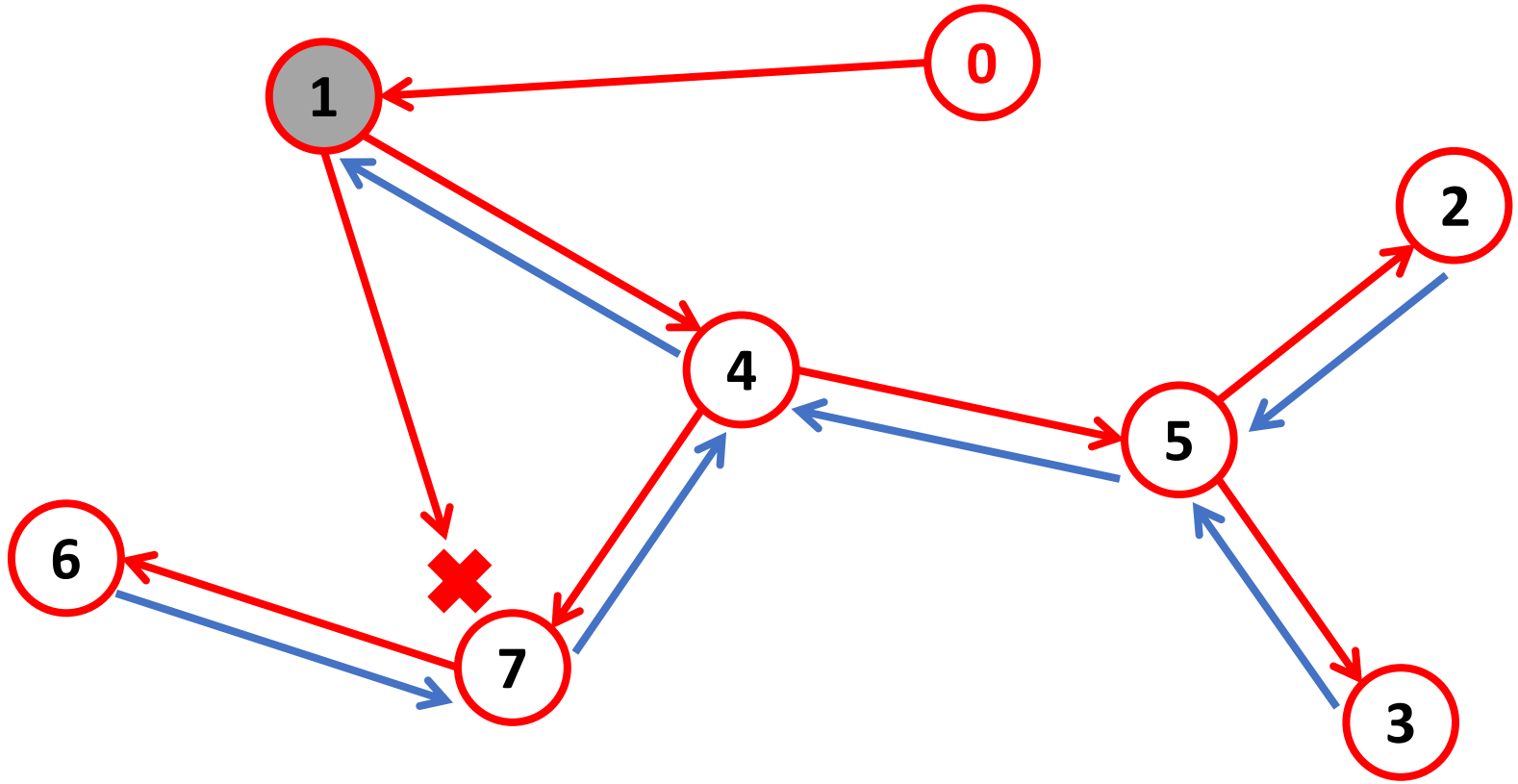
DFS



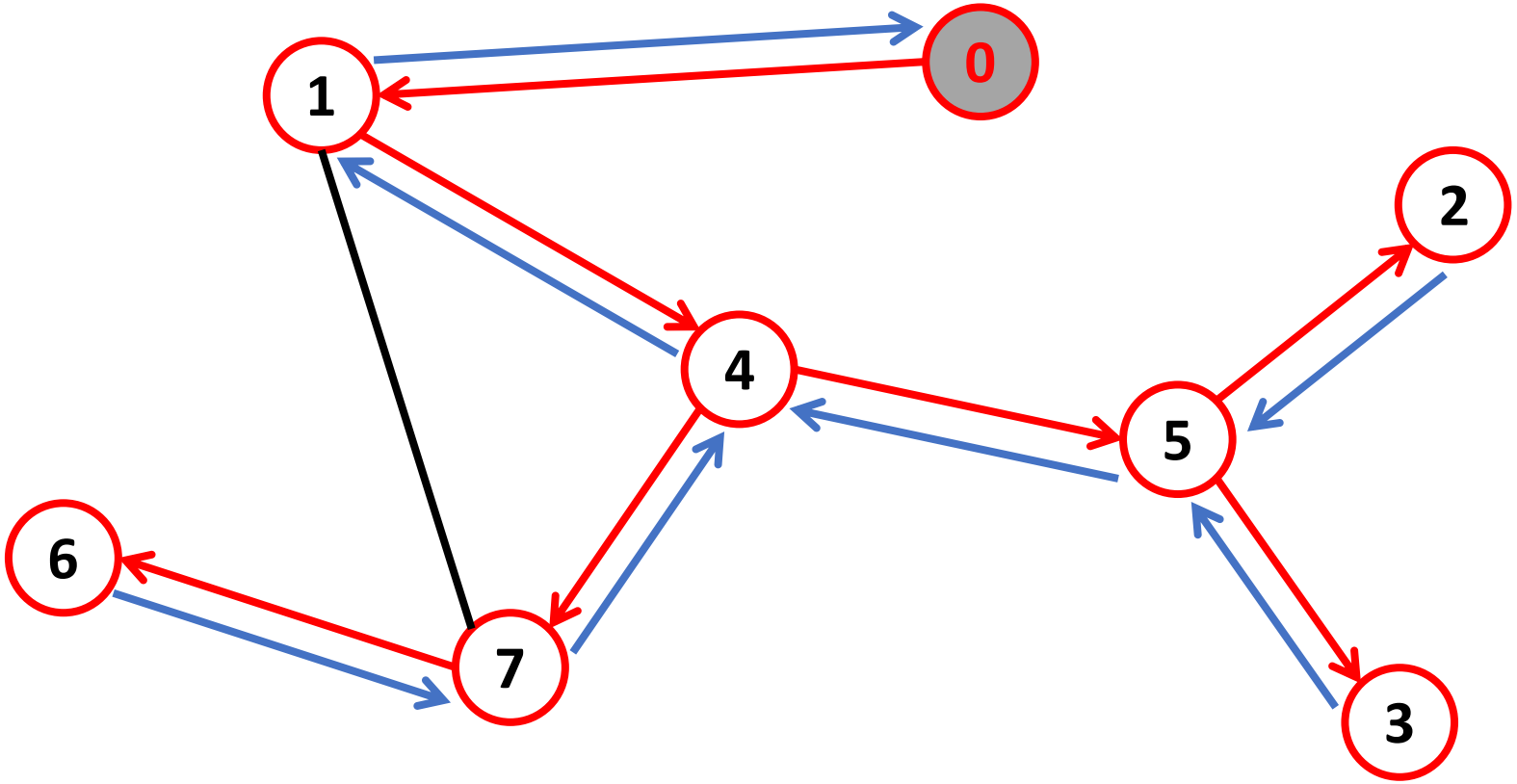
DFS



DFS



DFS



Código

Necesitamos:

```
vector<int> g[n];

bool vis[n];
```

Código

Necesitamos:

```
vector<int> g[n];

bool vis[n];
```

```
void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      dfs(v);
    }
  }
}
```

Código

Iterativo:

```
stack<int> st;
vector<int> vis(n, false);
st.push(u);
while (st.size() > 0) {
  int u = st.top();
  st.pop();
  vis[u] = true;
  for (int v : adj[u]) {
    if (!vis[v]) {
      st.push(v);
    }
  }
}
```

Recursivo:

```
void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      dfs(v);
    }
  }
}
```

Código

Iterativo:

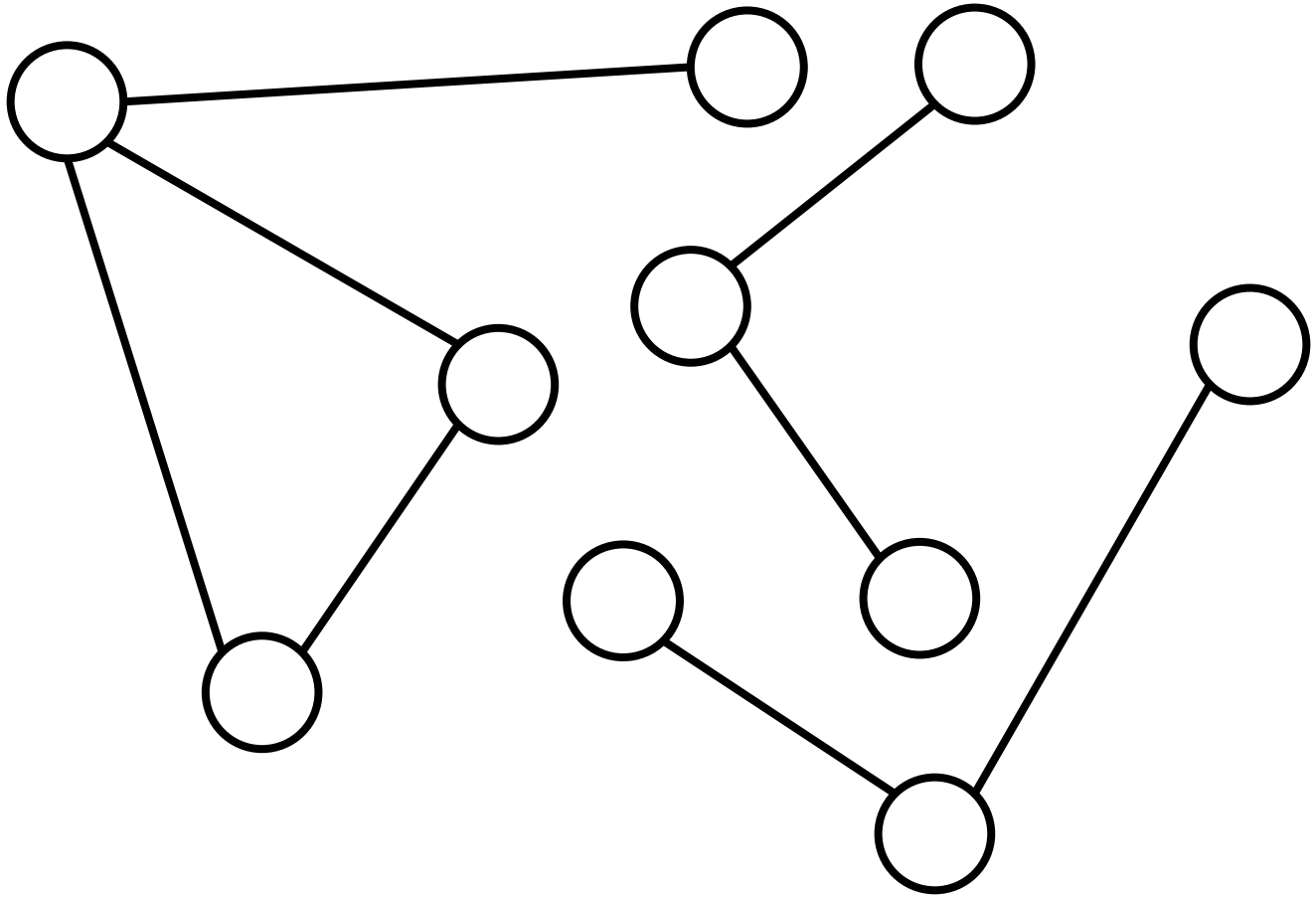
```
stack<int> st;
vector<int> vis(n, false);
st.push(u);
while (st.size() > 0) {
  int u = st.top();
  st.pop();
  vis[u] = true;
  for (int v : adj[u]) {
    if (!vis[v]) {
      st.push(v);
    }
  }
}
```

Recursivo:

```
void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      dfs(v);
    }
  }
}
```

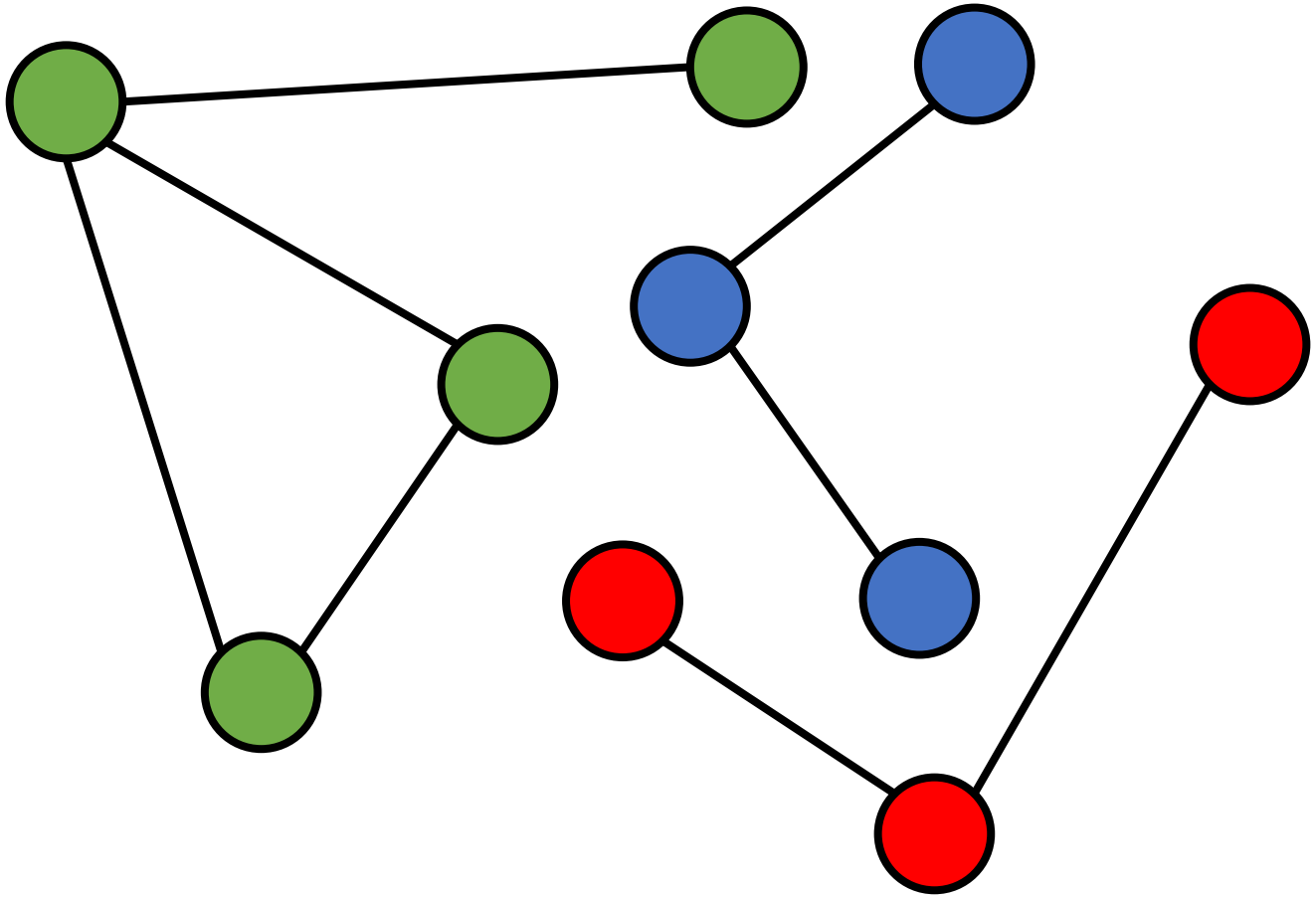
Aplicación – Verificar conectividad

Componentes Conexos



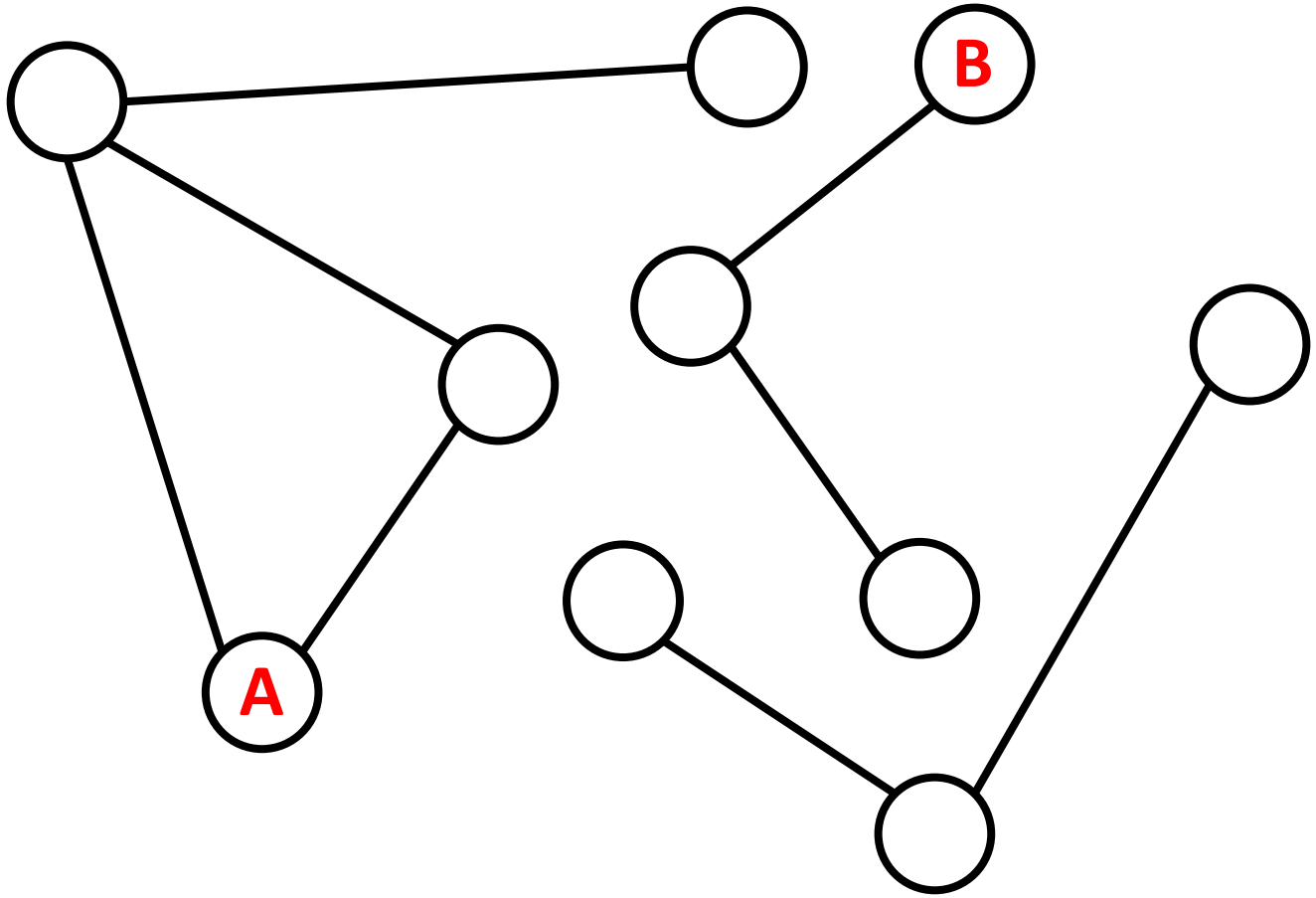
Aplicación – Verificar conectividad

Componentes Conexos



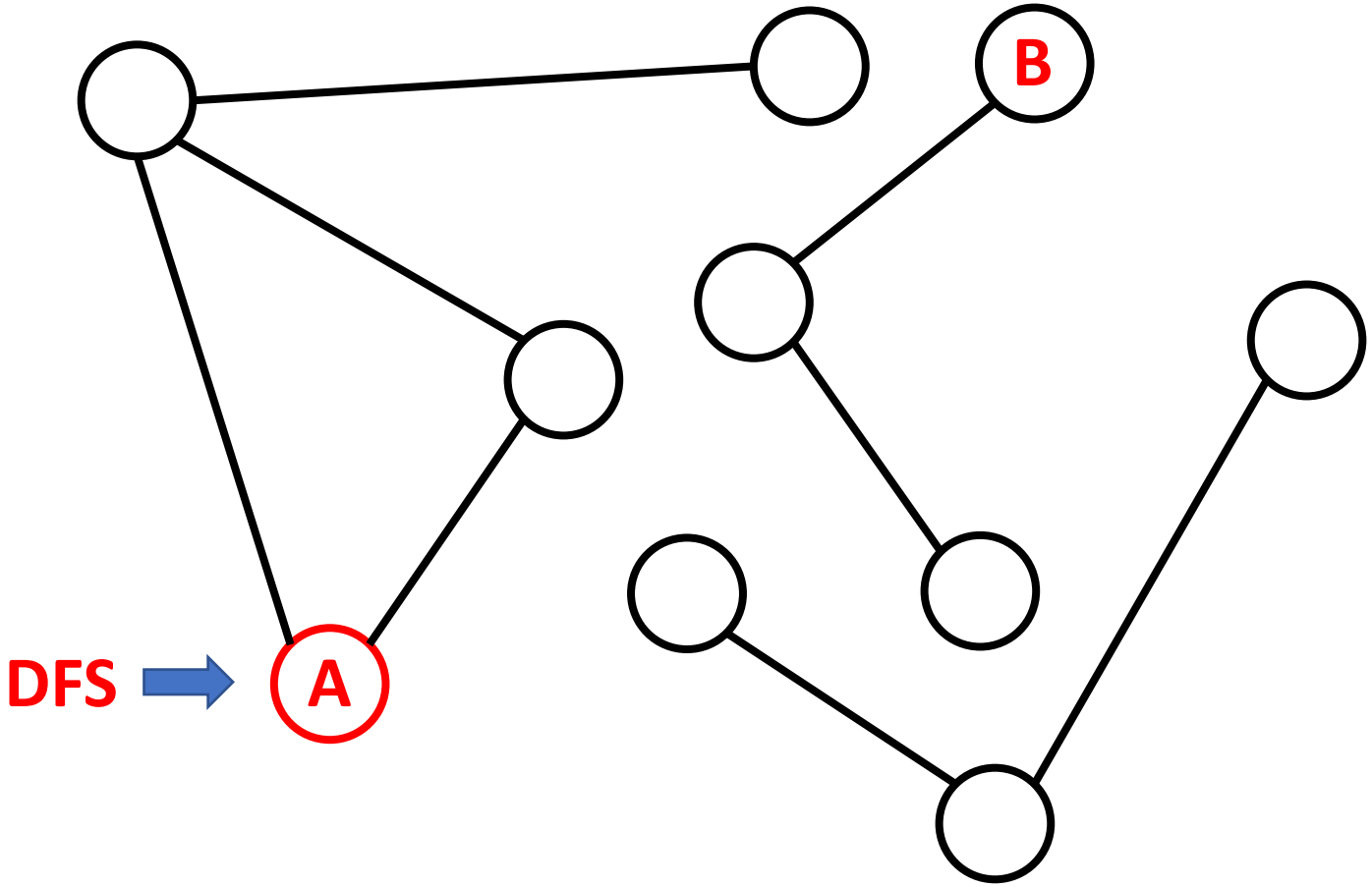
Aplicación – Verificar conectividad

Componentes Conexos



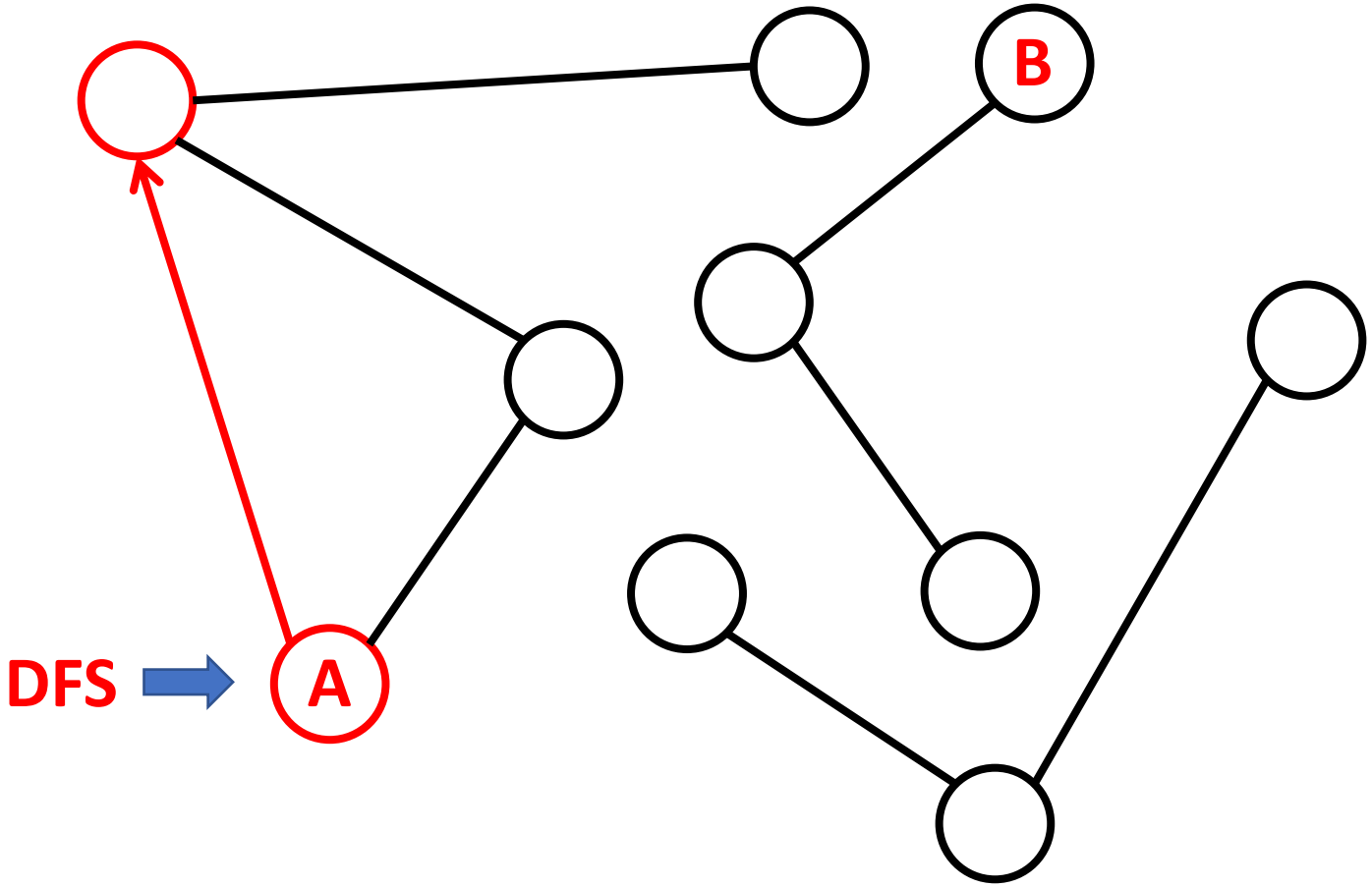
Aplicación – Verificar conectividad

Componentes Conexos



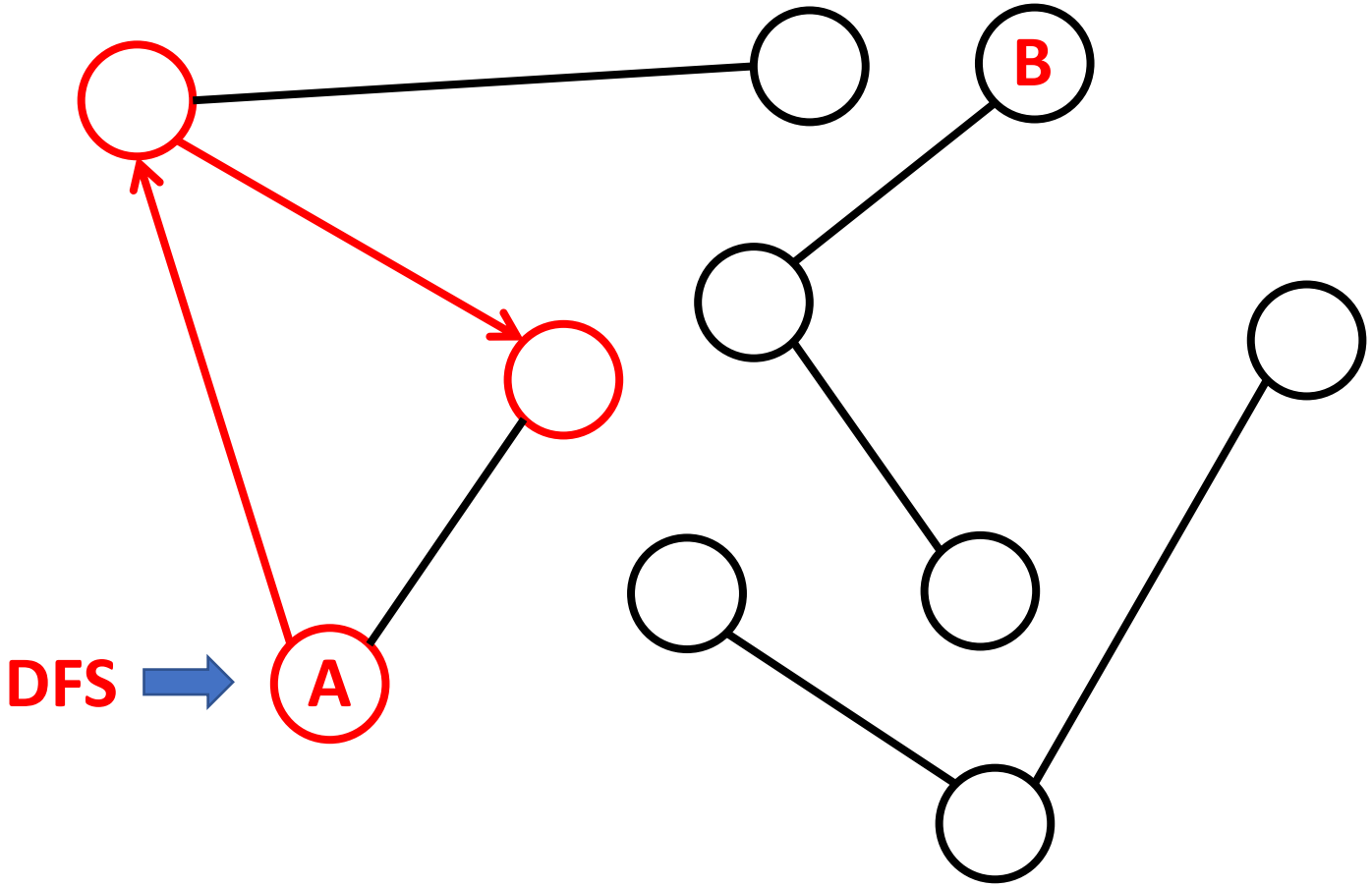
Aplicación – Verificar conectividad

Componentes Conexos



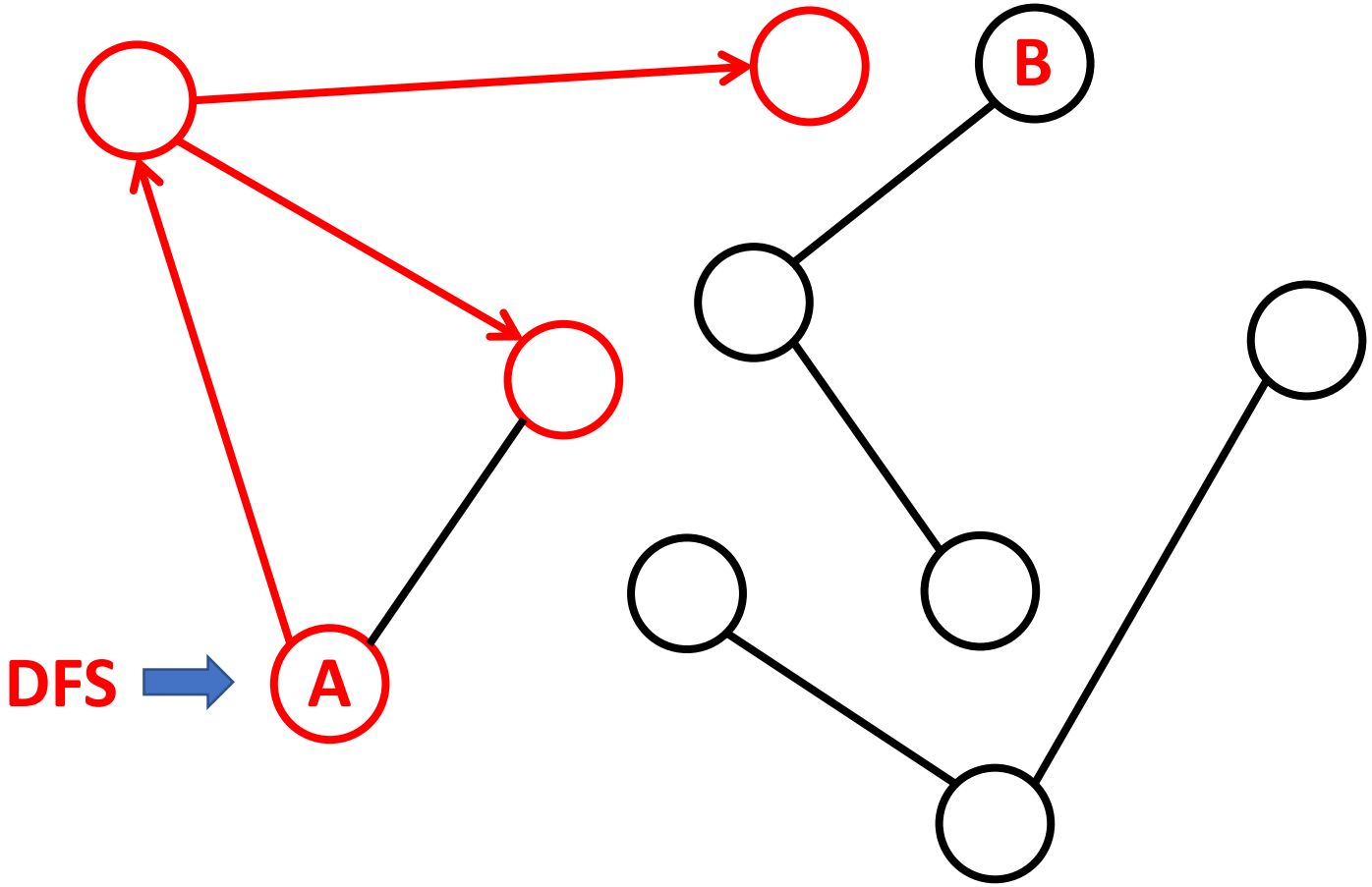
Aplicación – Verificar conectividad

Componentes Conexos



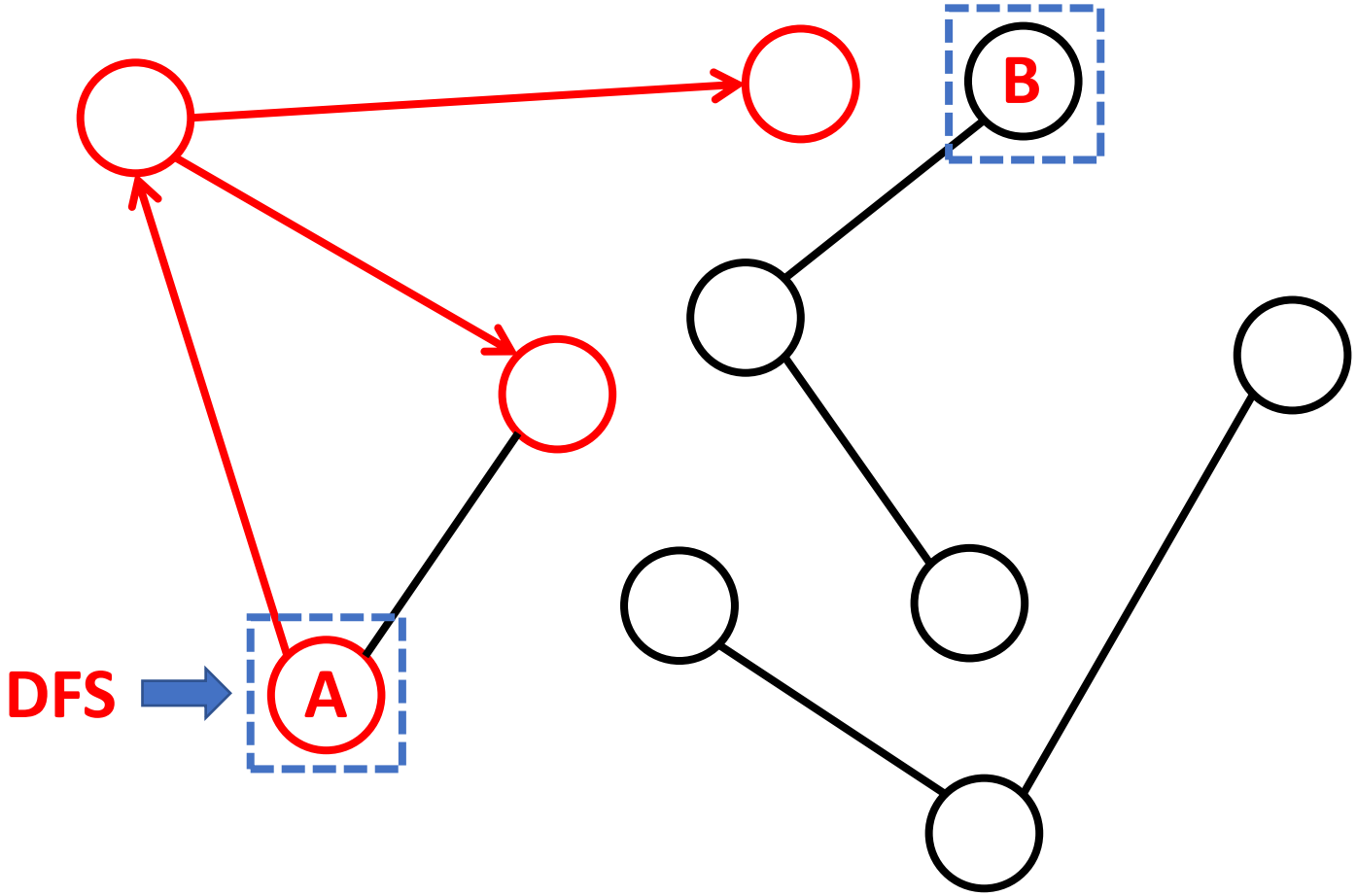
Aplicación – Verificar conectividad

Componentes Conexos

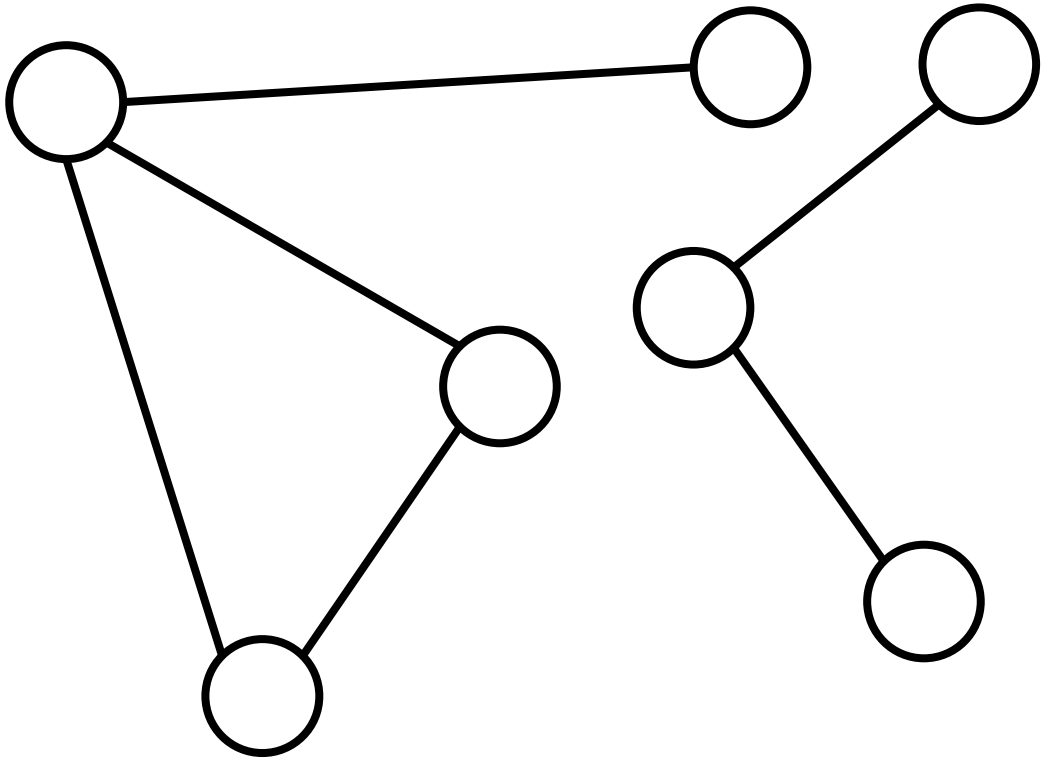


Aplicación – Verificar conectividad

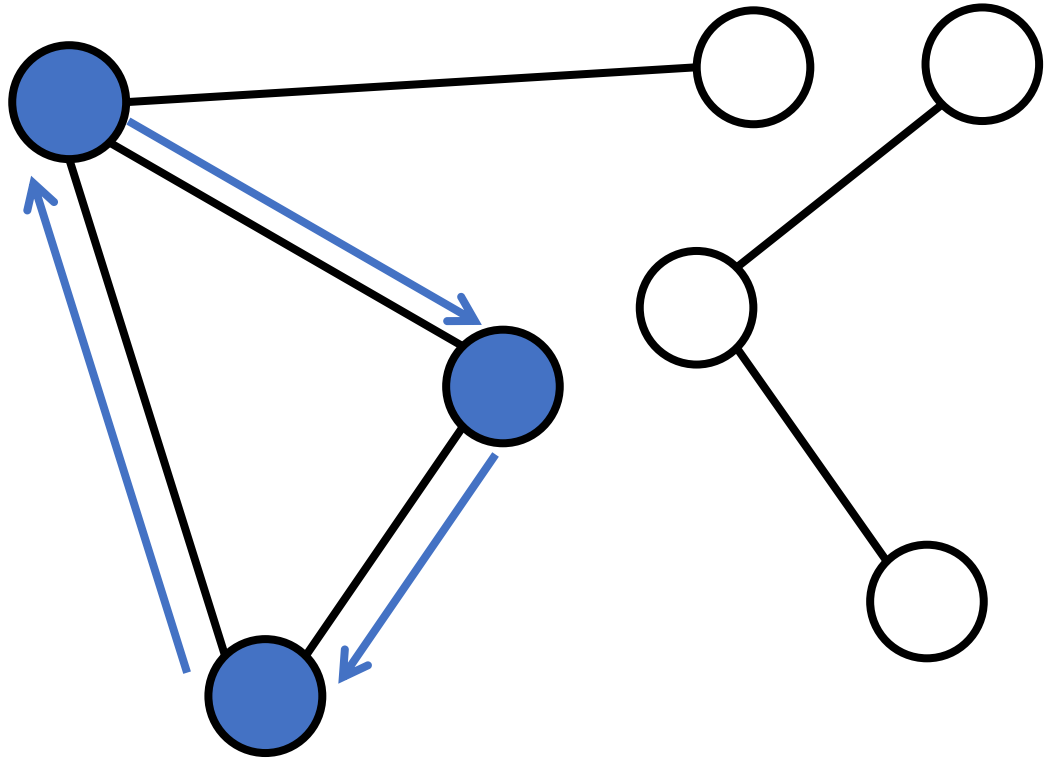
Componentes Conexos



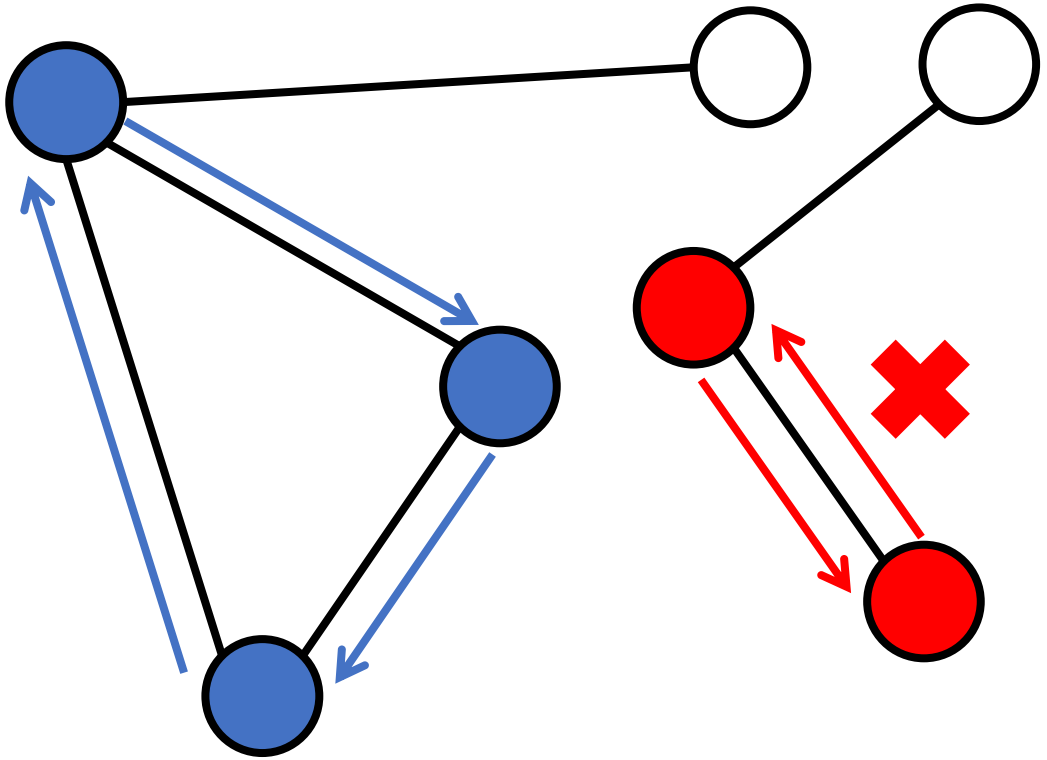
Aplicación – Detectar Ciclos



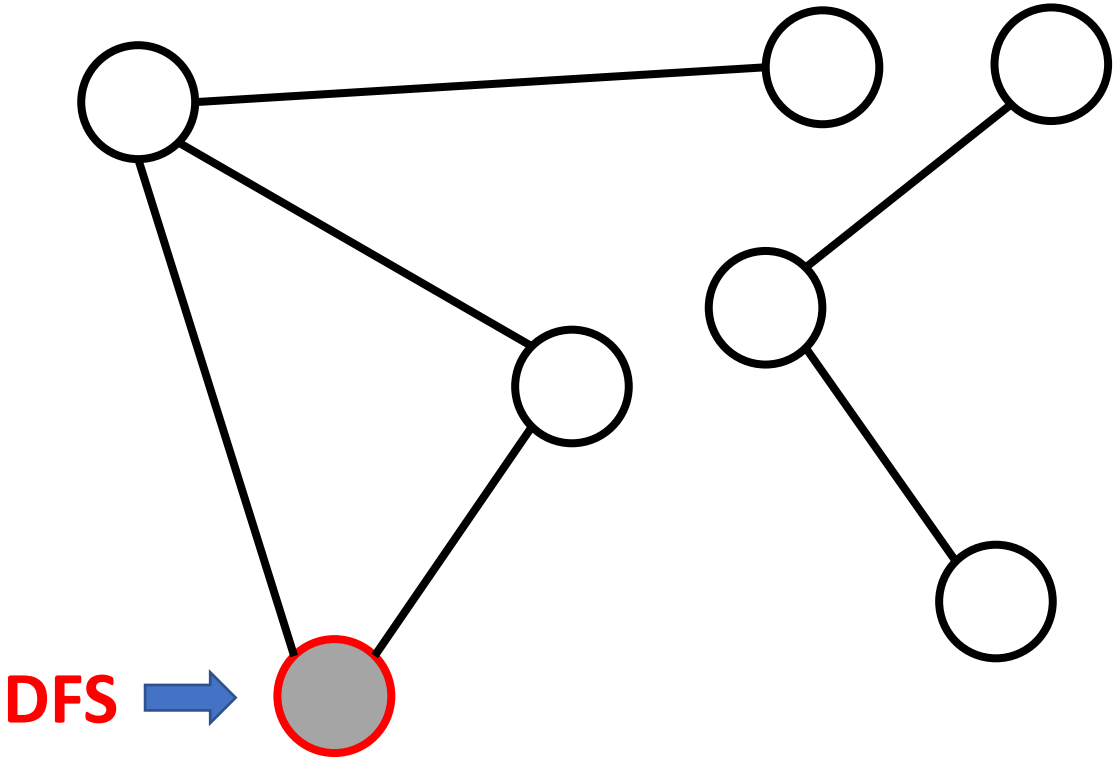
Aplicación – Detectar Ciclos



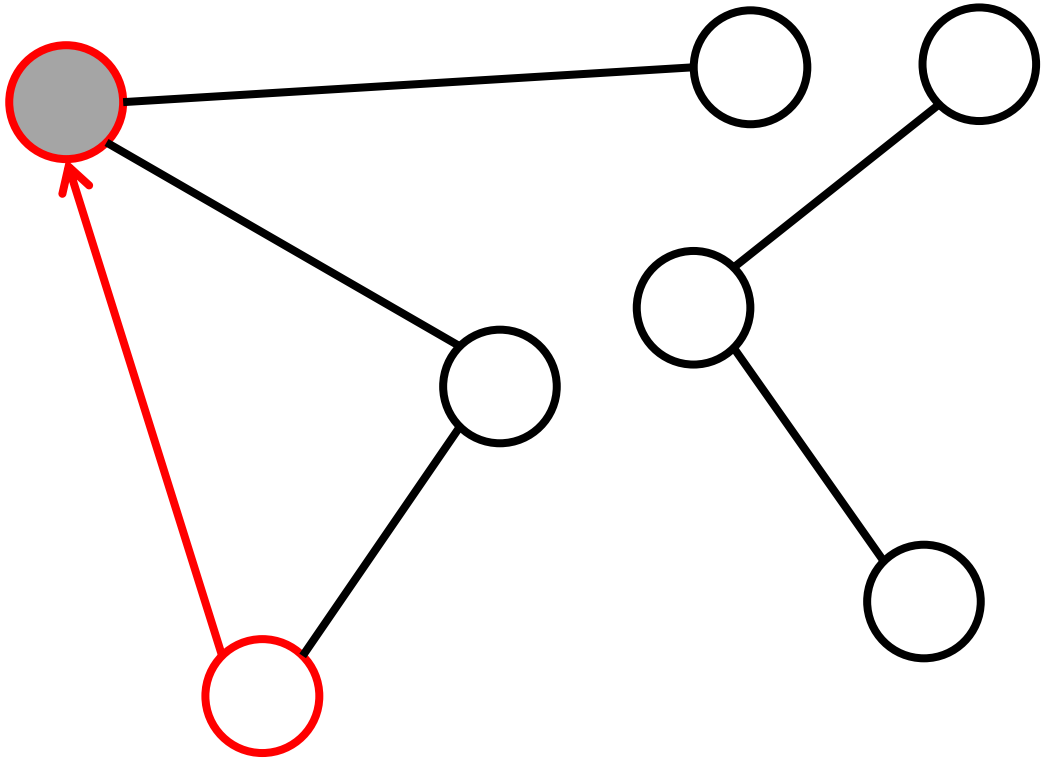
Aplicación – Detectar Ciclos



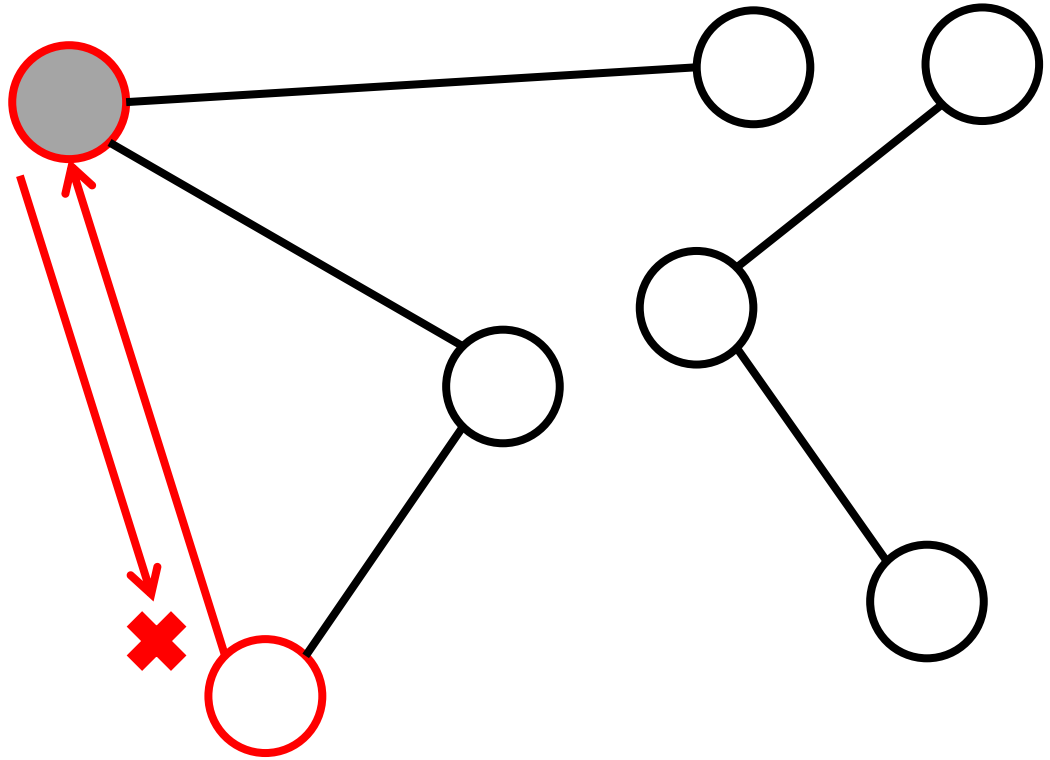
Aplicación – Detectar Ciclos



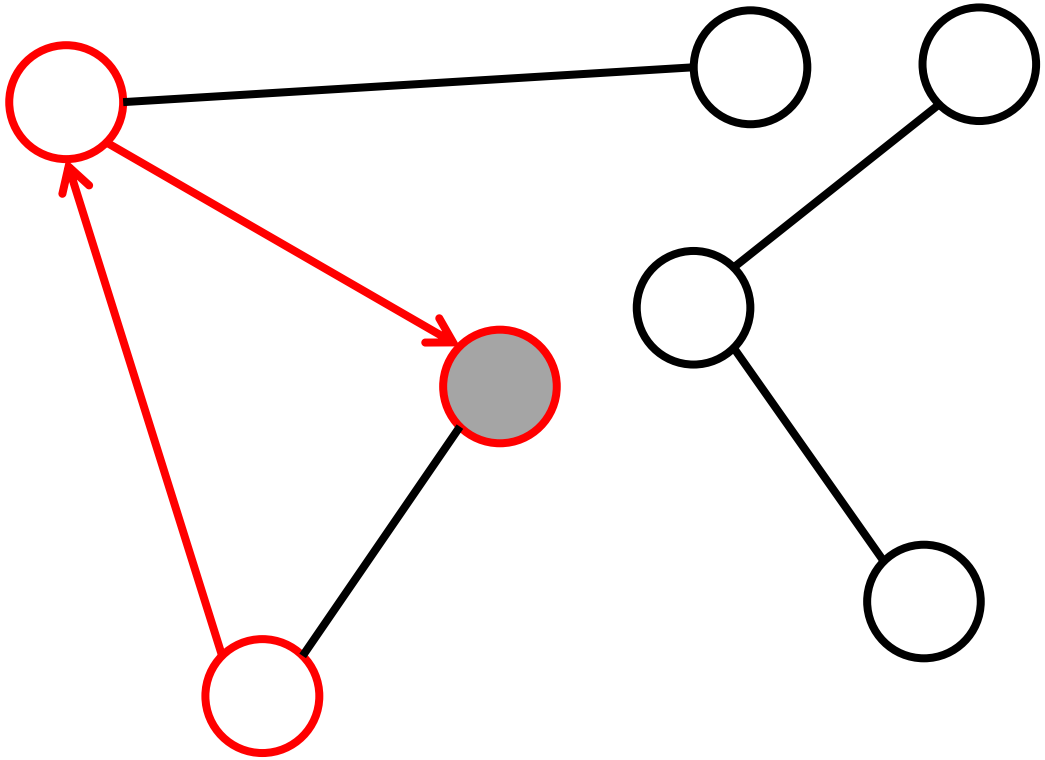
Aplicación – Detectar Ciclos



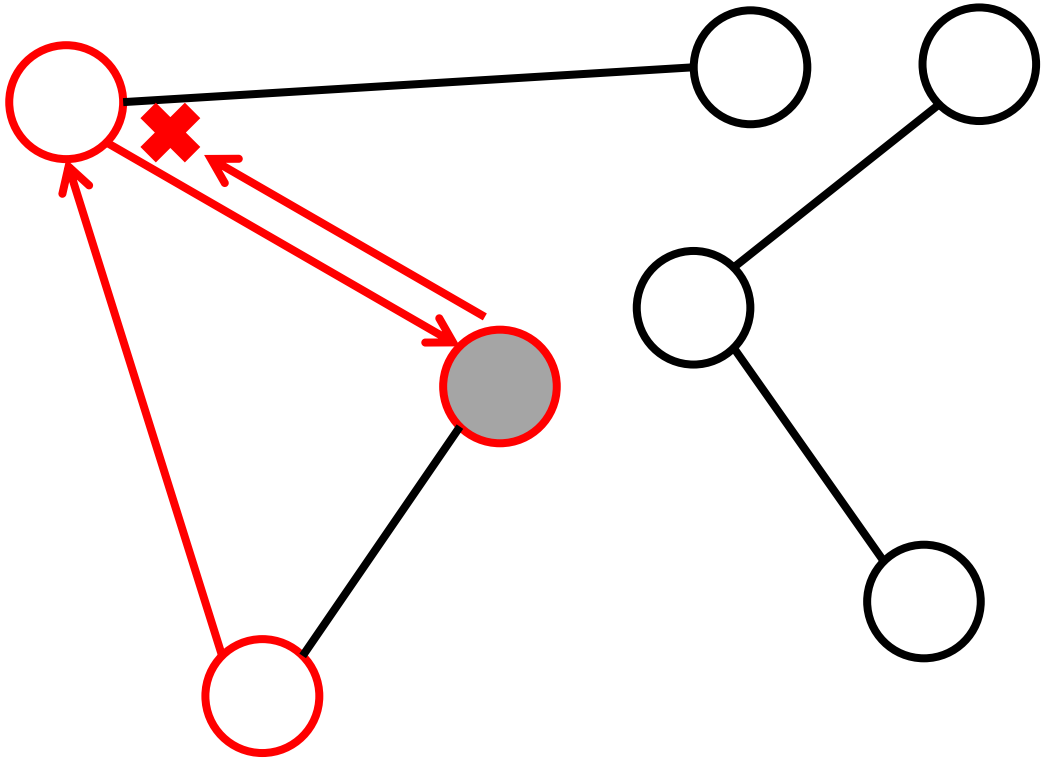
Aplicación – Detectar Ciclos



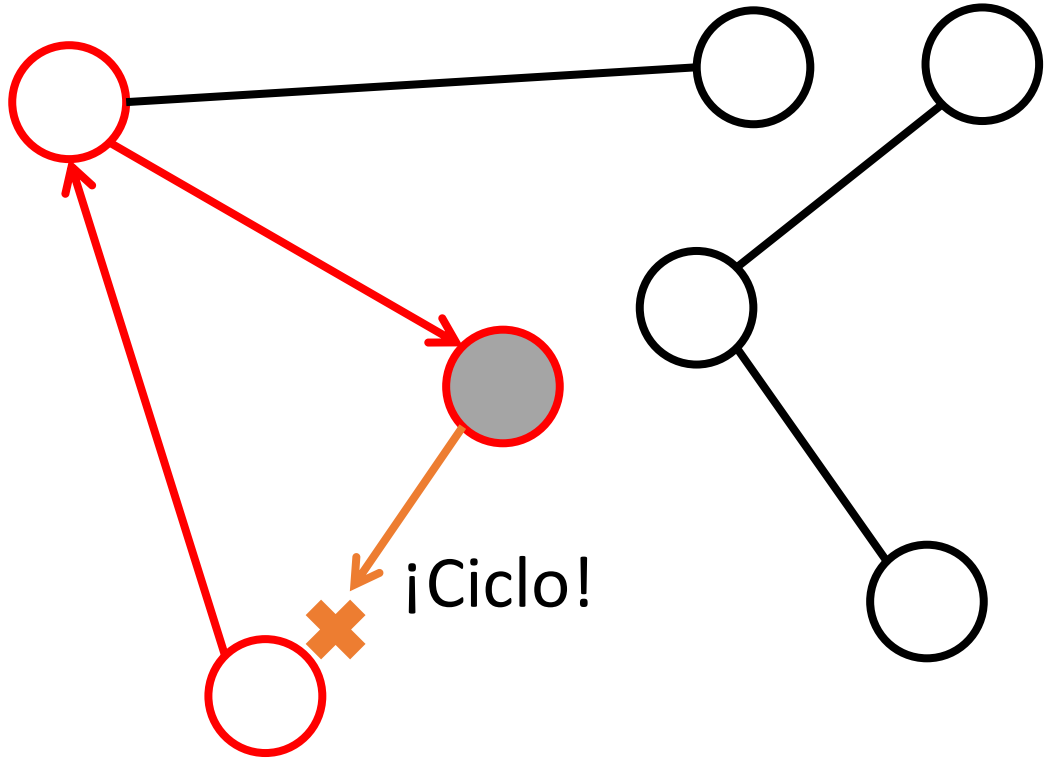
Aplicación – Detectar Ciclos



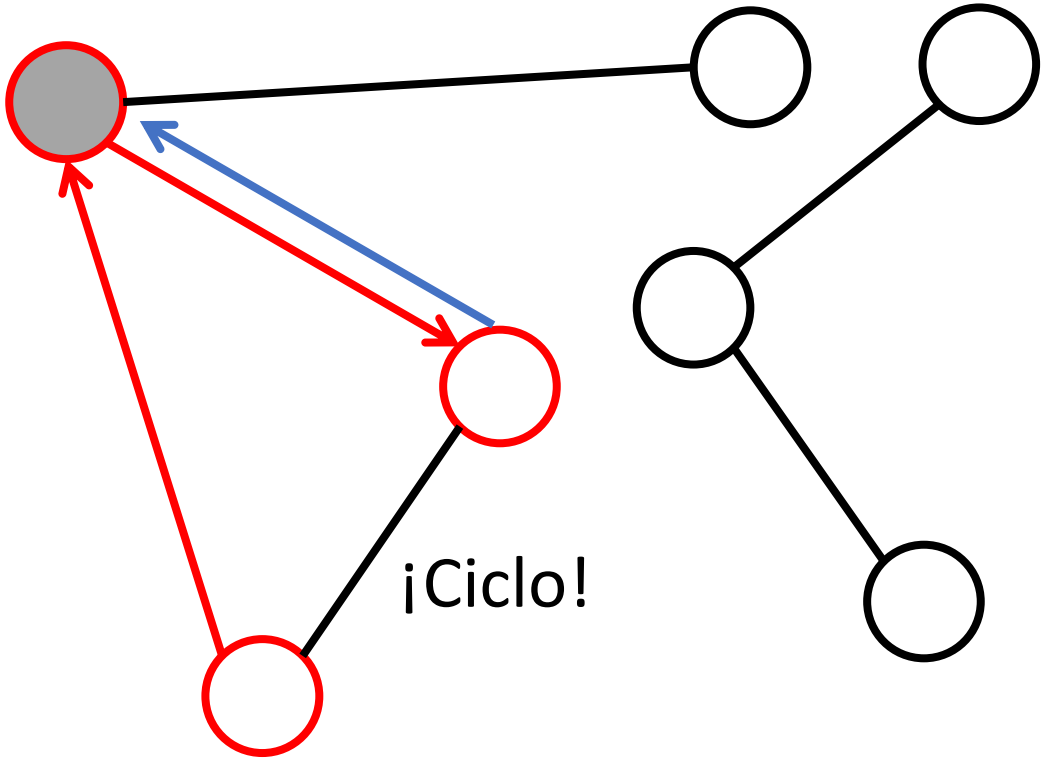
Aplicación – Detectar Ciclos



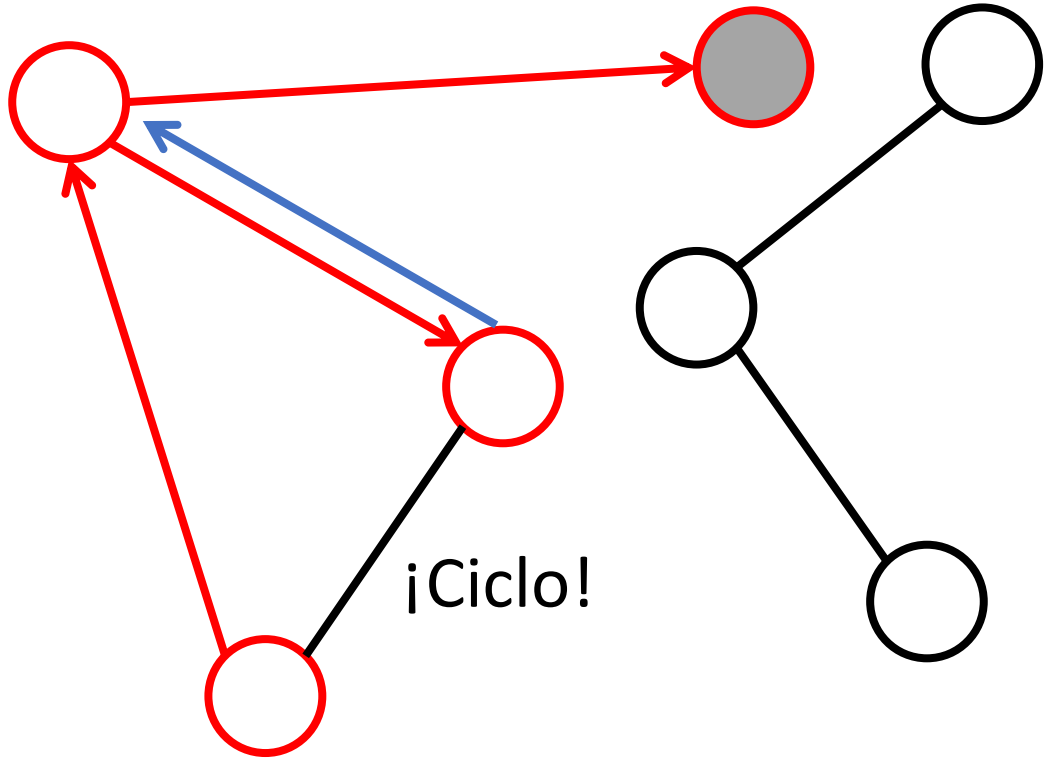
Aplicación – Detectar Ciclos



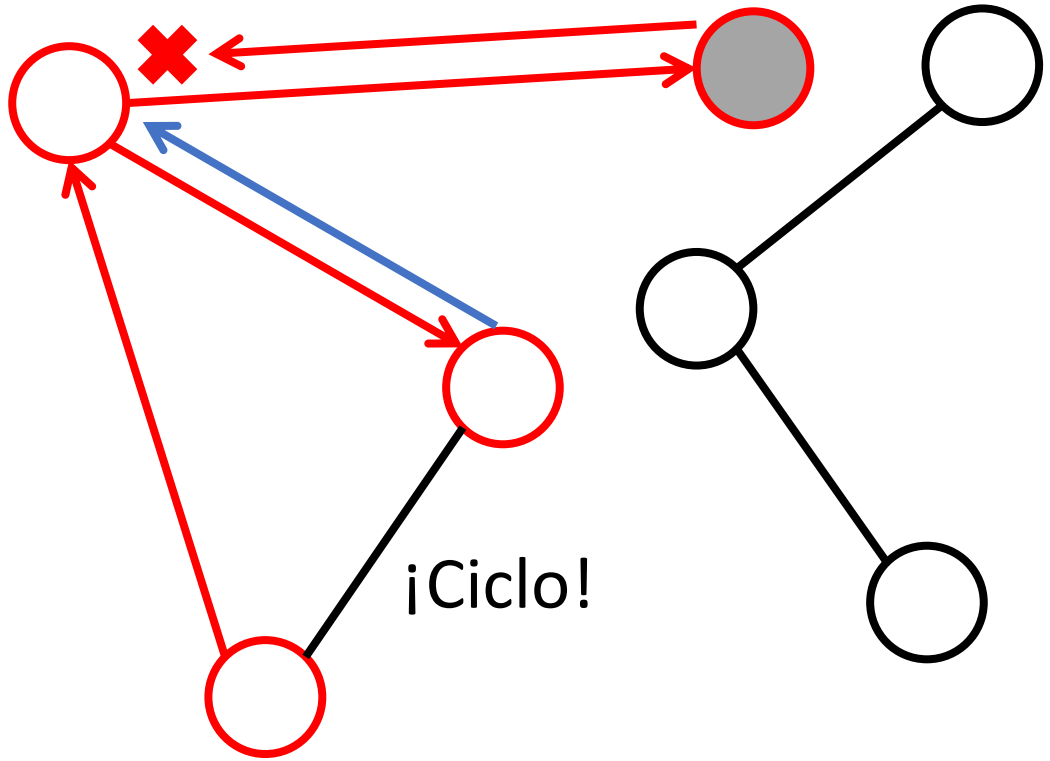
Aplicación – Detectar Ciclos



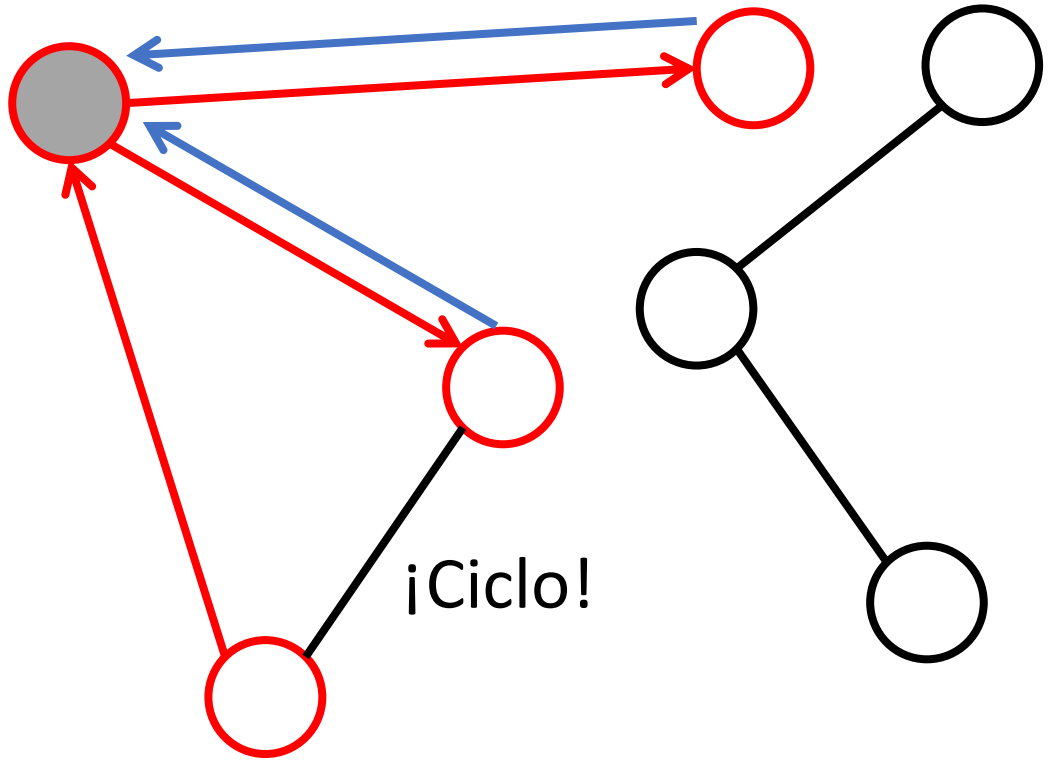
Aplicación – Detectar Ciclos



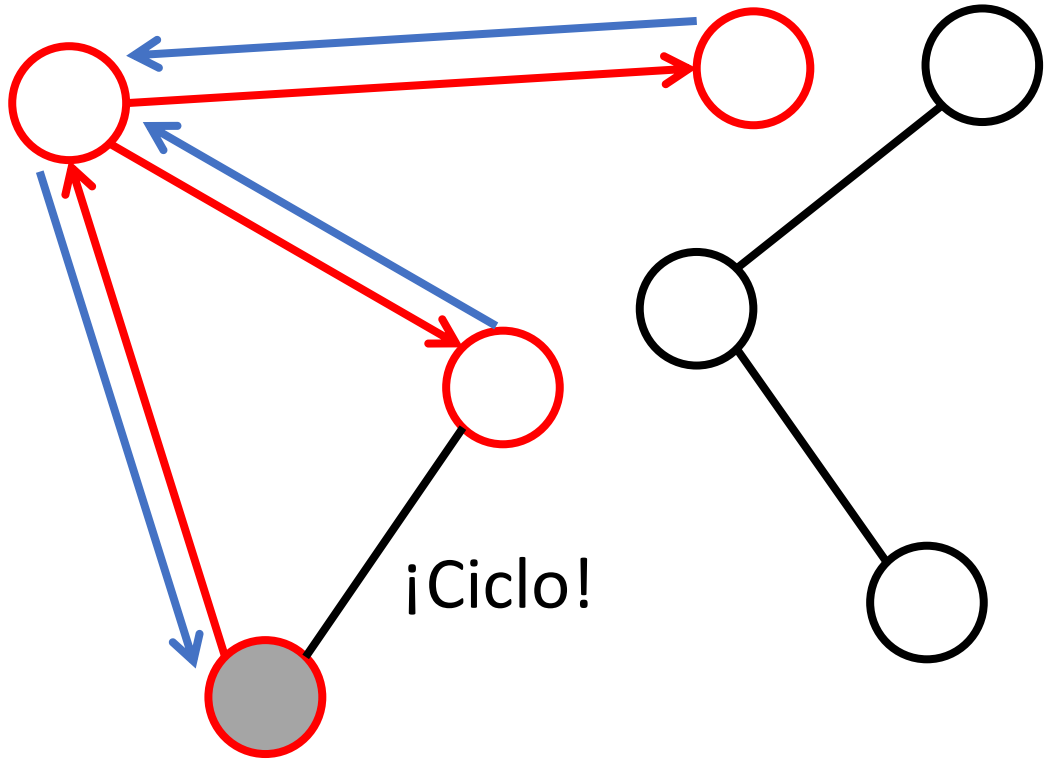
Aplicación – Detectar Ciclos



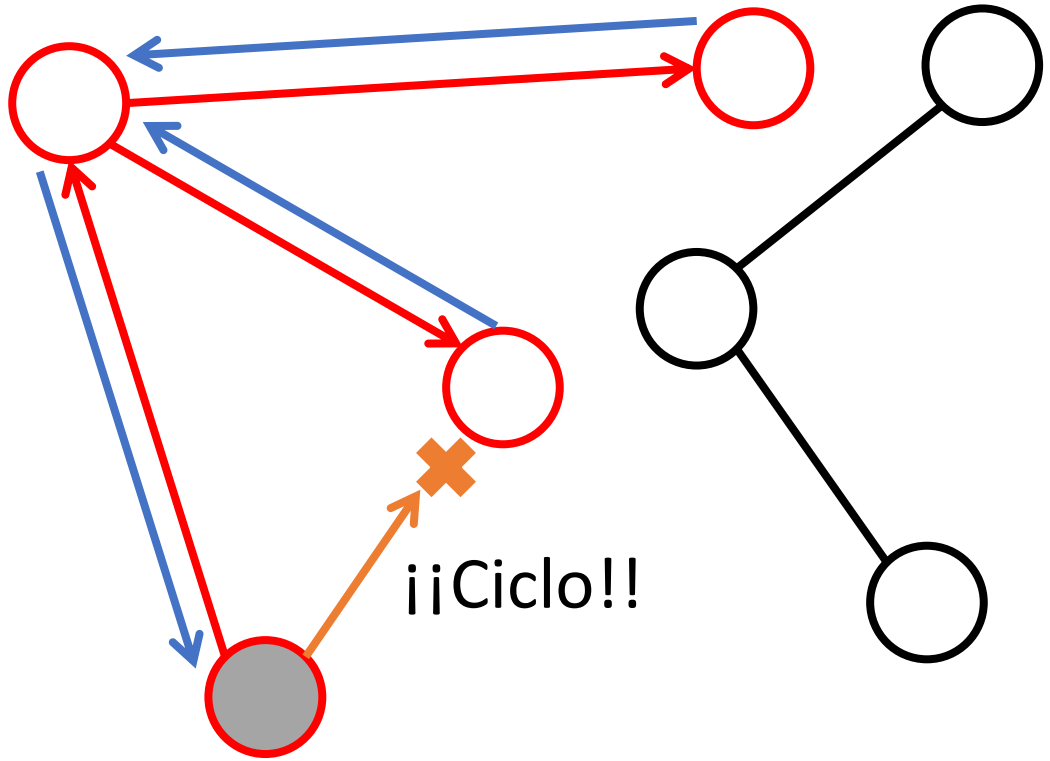
Aplicación – Detectar Ciclos



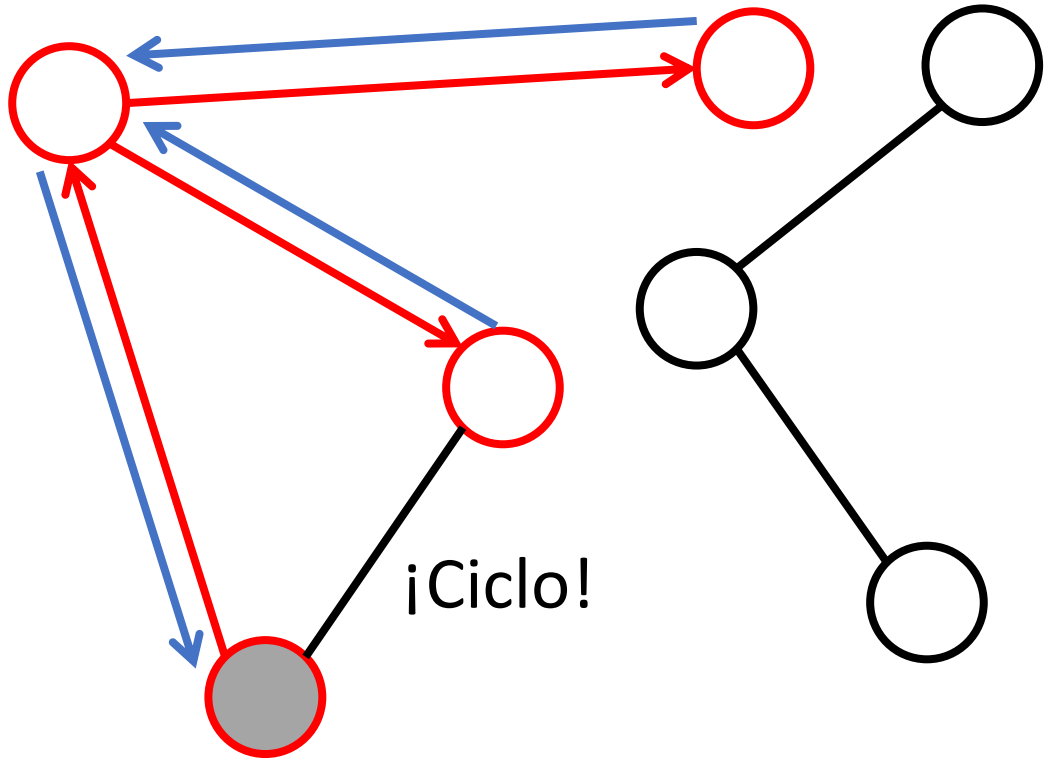
Aplicación – Detectar Ciclos



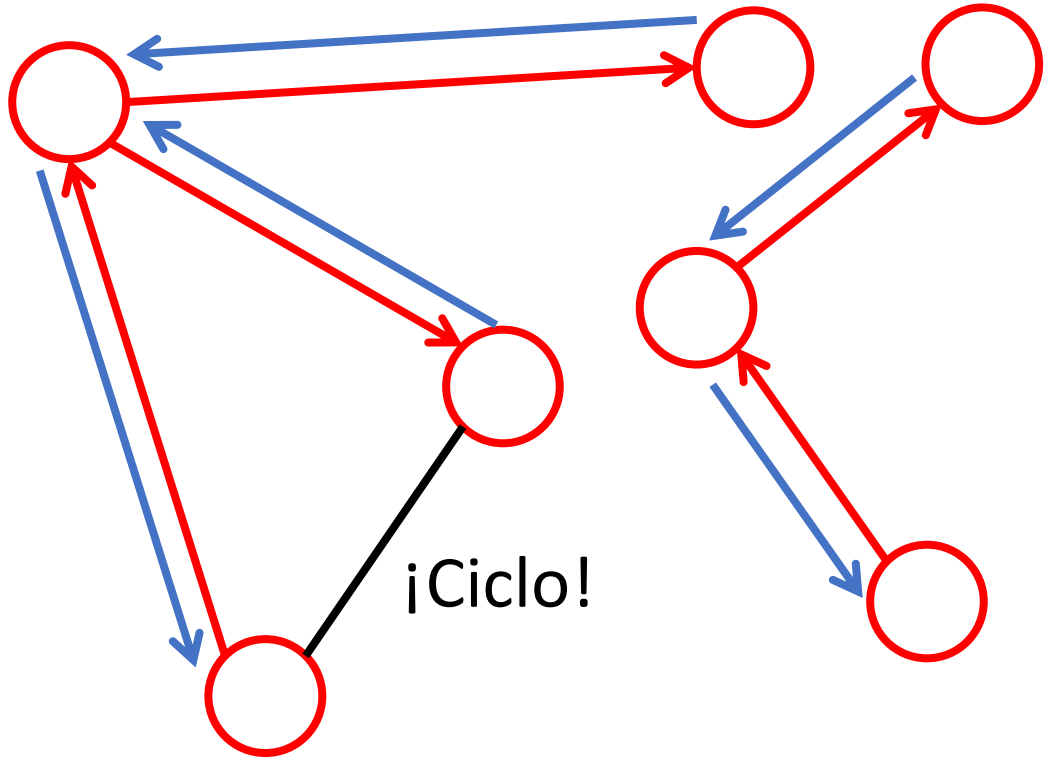
Aplicación – Detectar Ciclos



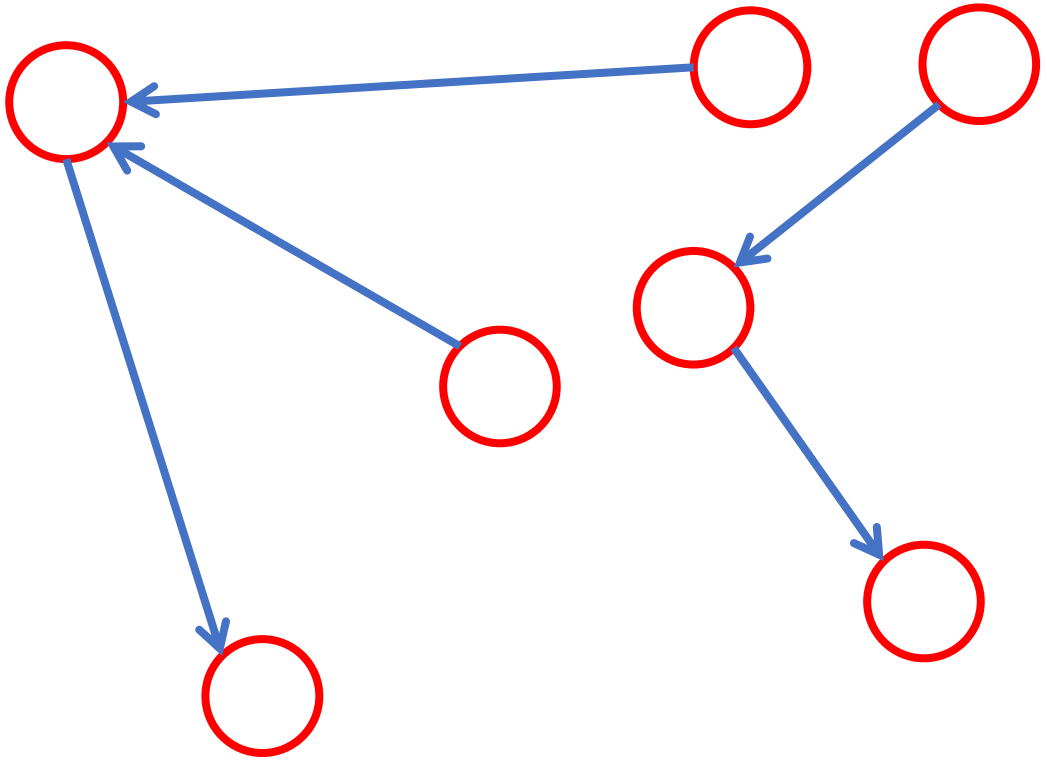
Aplicación – Detectar Ciclos



Aplicación – Detectar Ciclos



Aplicación – Detectar Ciclos



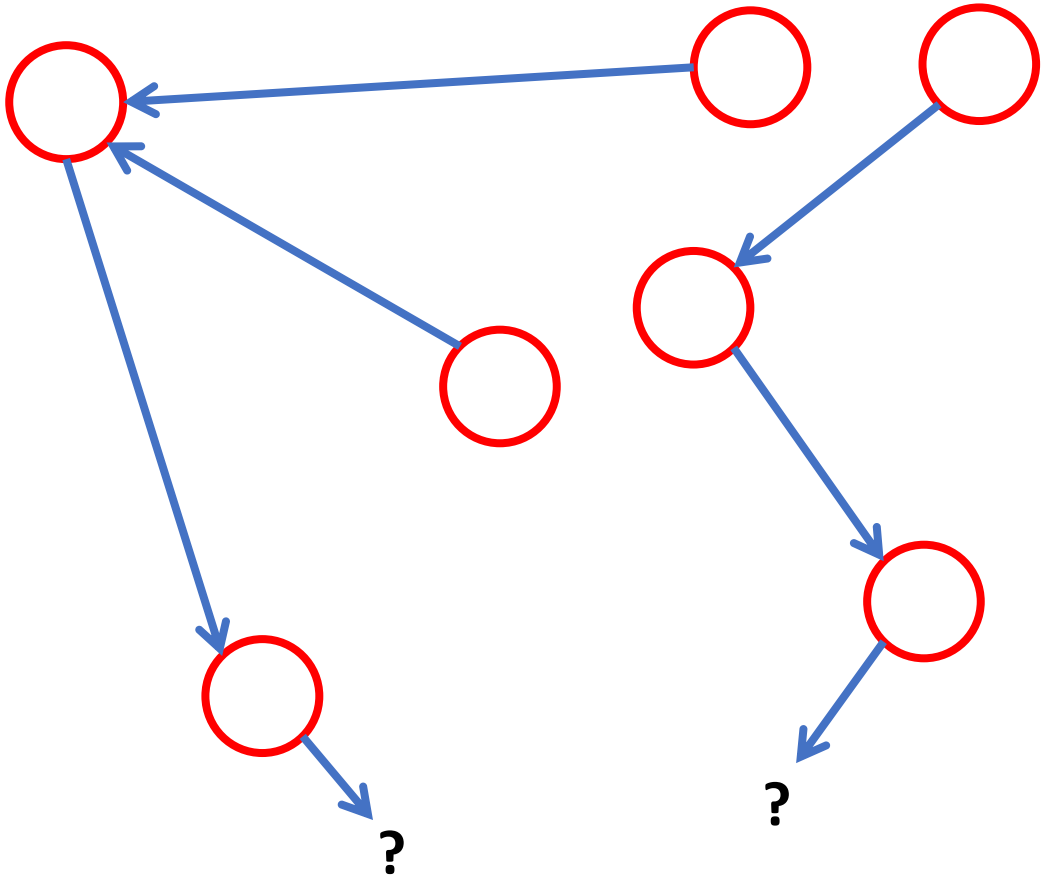
Aplicación – Detectar Ciclos

```
vector<int> g[n];
bool vis[n];


void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v
      // aqui se define el padre
      dfs(v);
    }
  }
}
```

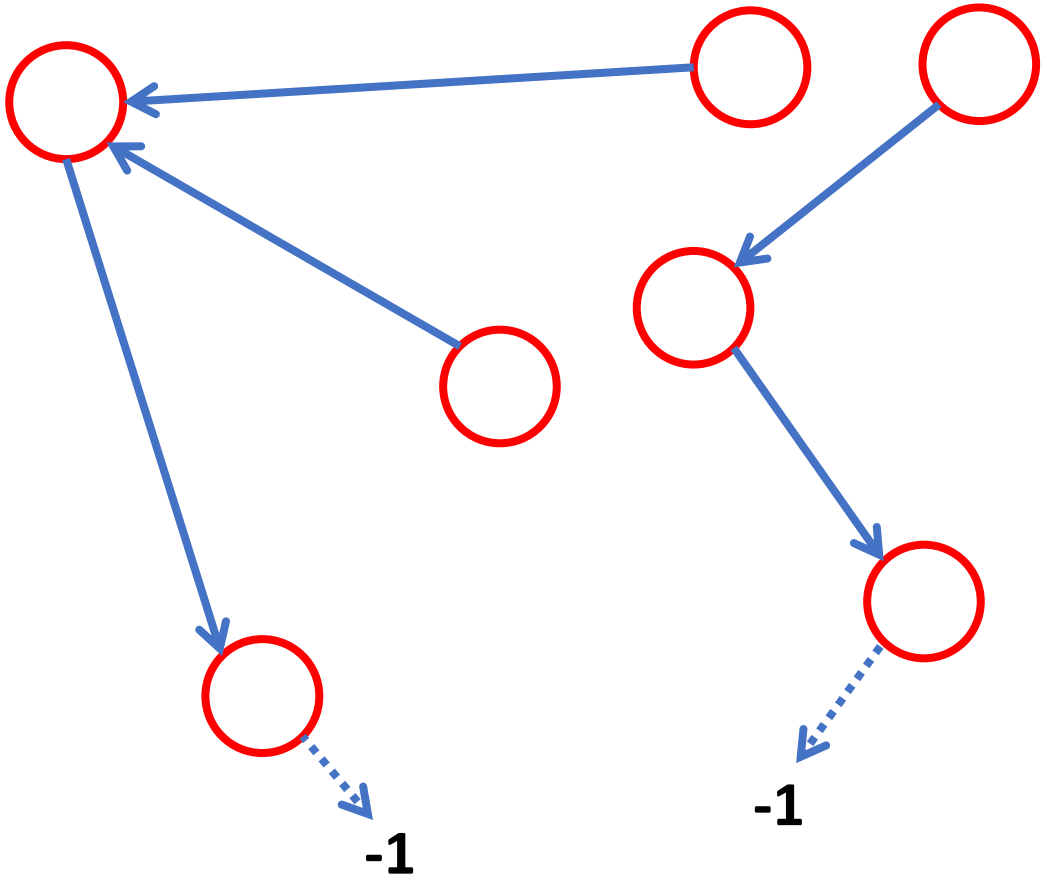
Aplicación – Detectar Ciclos

```cpp
vector<int> g[n];
bool vis[n];
int padre[n];

void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v
      padre[v] = u;
      dfs(v);
    }
  }
}
```

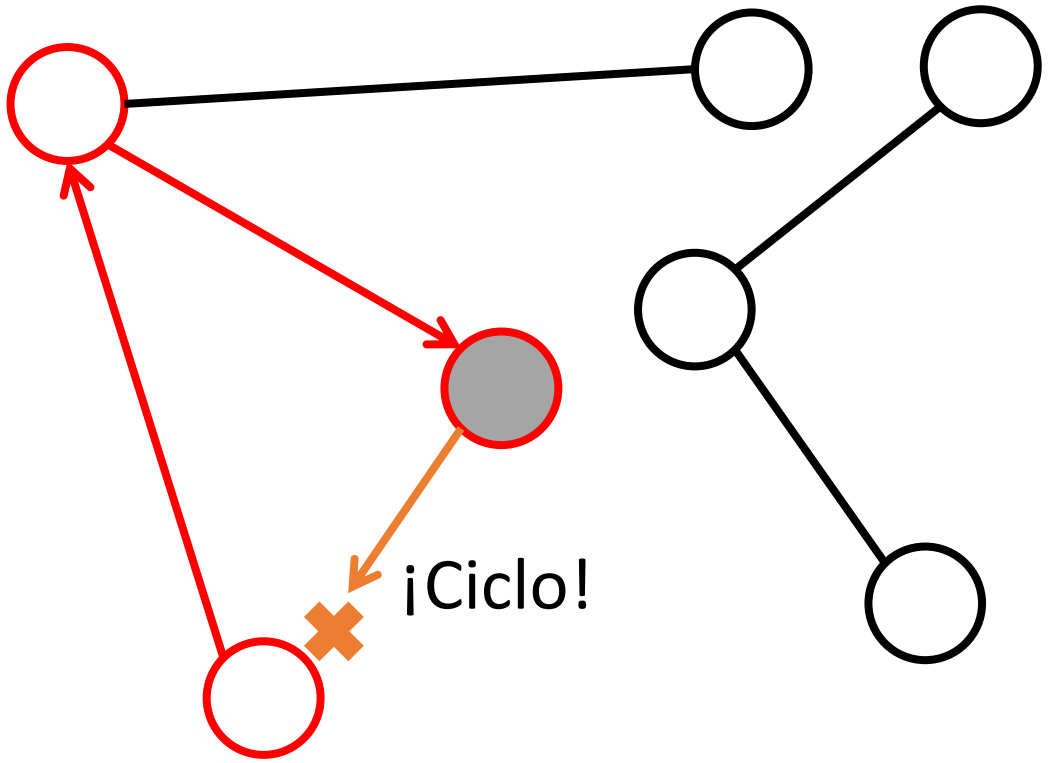
Aplicación – Detectar Ciclos



Aplicación – Detectar Ciclos

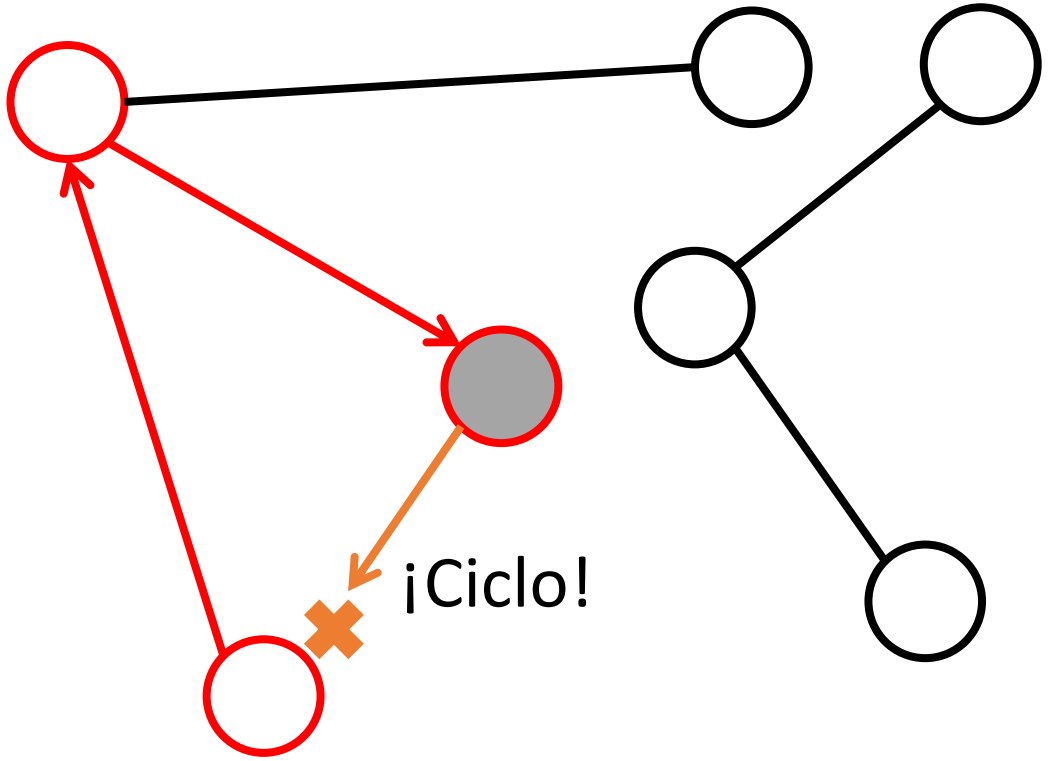


Aplicación – Detectar Ciclos




```
vector<int> g[n];
bool vis[n];
int padre[n];

void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v
      padre[v] = u;
      dfs(v);
    }
  }
}
```

Aplicación – Detectar Ciclos




```cpp
vector<int> g[n];
bool vis[n];
int padre[n];
bool hayCiclo = false;

void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v
      padre[v] = u;
      dfs(v);
    }
    else if (v != padre[u]) {
      hayCiclo = true;
    }
  }
}
```

Aplicación – Orden Topológico

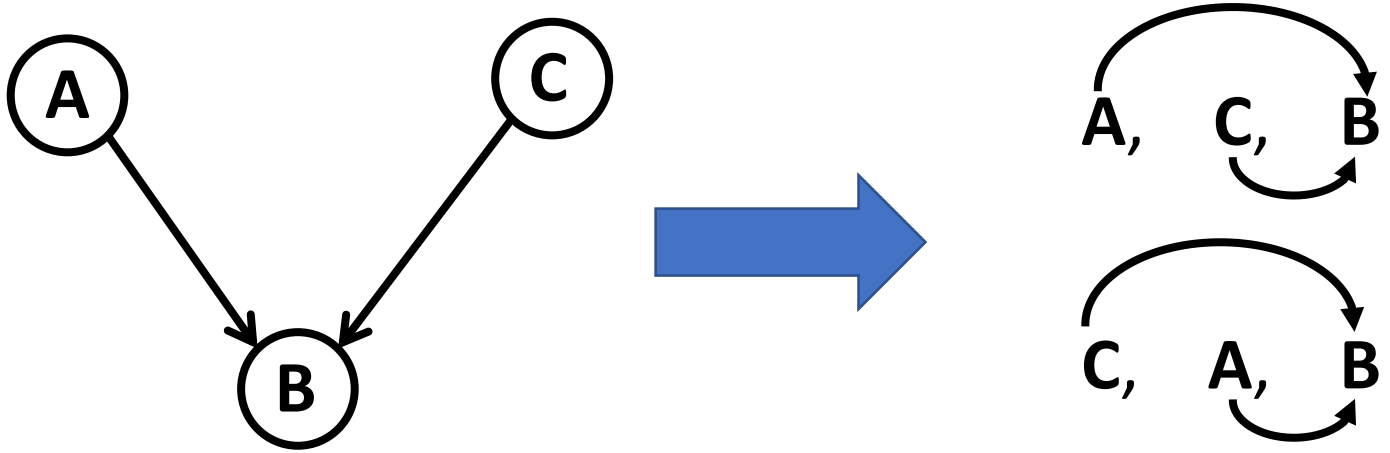
Es un concepto aplicado a grafos dirigidos que no contienen ciclos.

Orden lineal de los vértices del grafo de tal modo que por cada arista dirigida $u \rightarrow v$, u va antes que v en el orden.

Aplicación – Orden Topológico

Es un concepto aplicado a grafos dirigidos que no contienen ciclos.

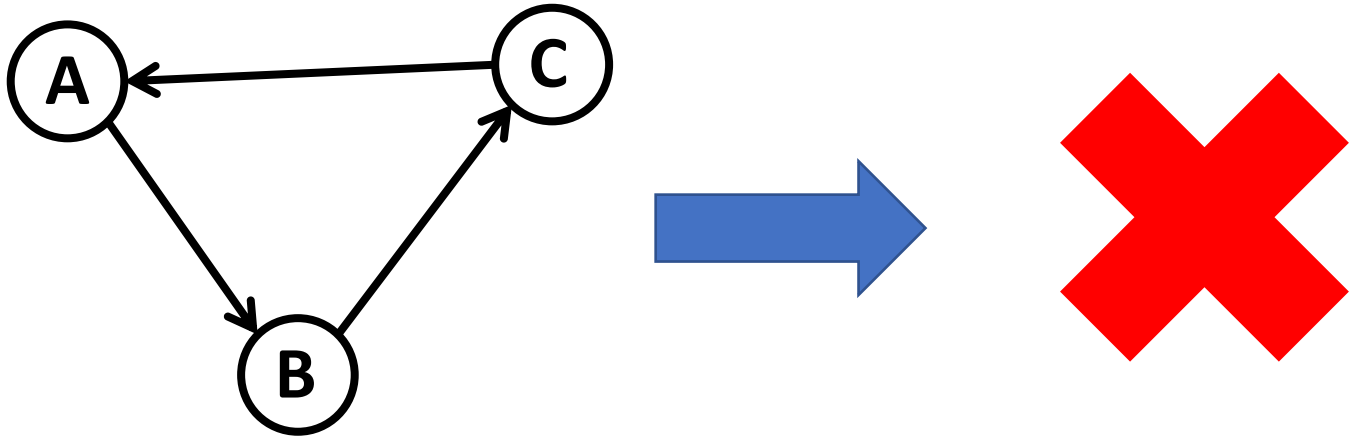
Orden lineal de los vértices del grafo de tal modo que por cada arista dirigida $u \rightarrow v$, u va antes que v en el orden.



Aplicación – Orden Topológico

Es un concepto aplicado a grafos dirigidos **que no contienen ciclos**.

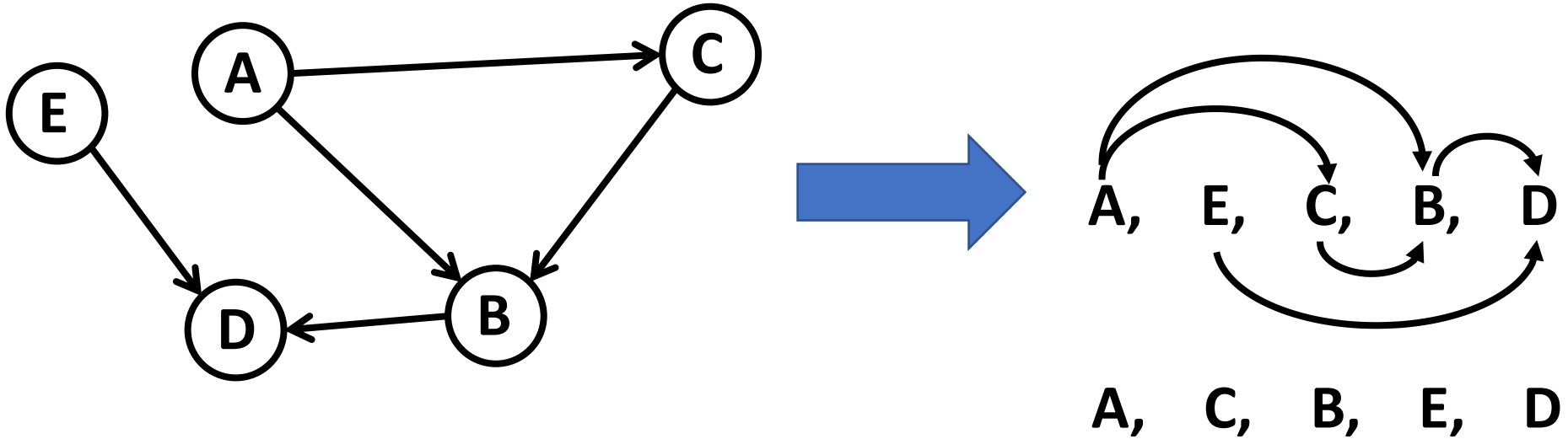
Orden lineal de los vértices del grafo de tal modo que por cada arista dirigida $u \rightarrow v$, u va antes que v en el orden.



Aplicación – Orden Topológico

Es un concepto aplicado a grafos dirigidos que no contienen ciclos.

Orden lineal de los vértices del grafo de tal modo que por cada arista dirigida $u \rightarrow v$, u va antes que v en el orden.

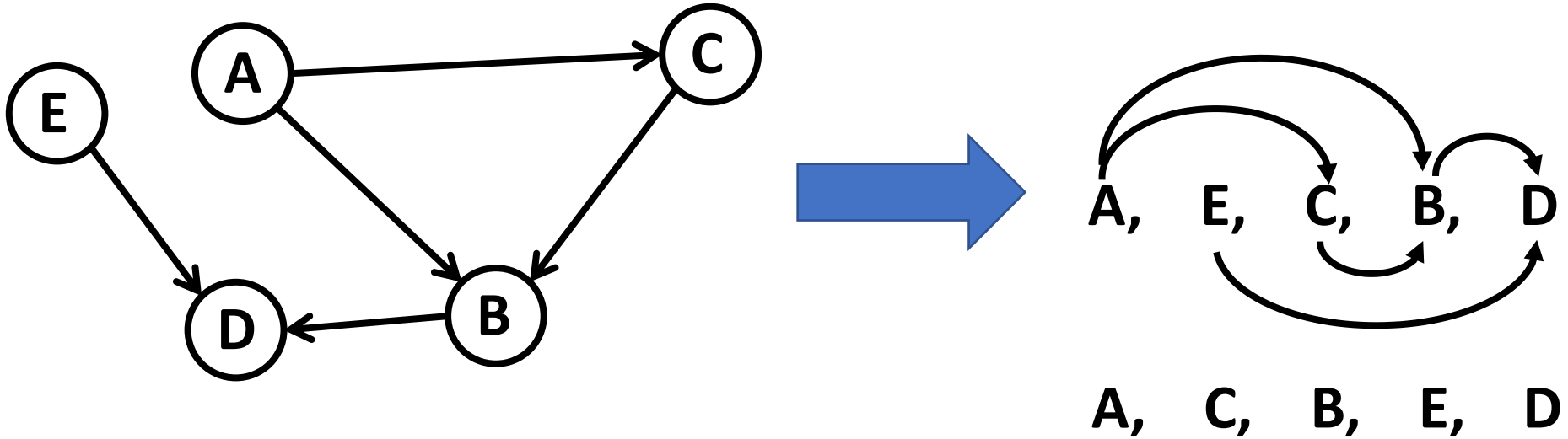


Aplicación – Orden Topológico

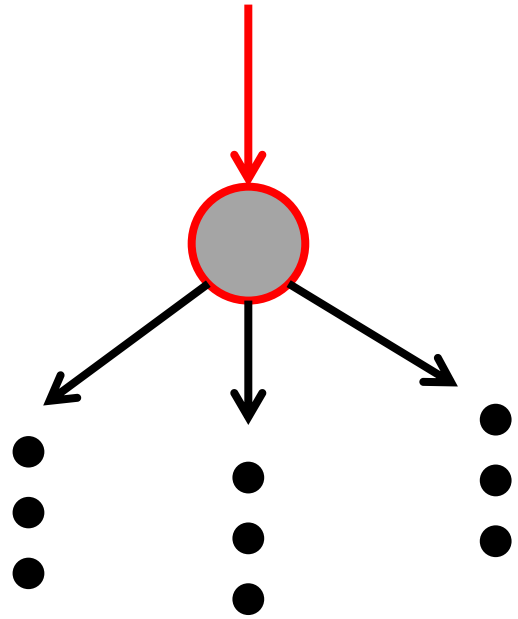
Es un concepto aplicado a grafos dirigidos que no contienen ciclos.

Orden lineal de los vértices del grafo de tal modo que por cada arista dirigida $u \rightarrow v$, u va antes que v en el orden.

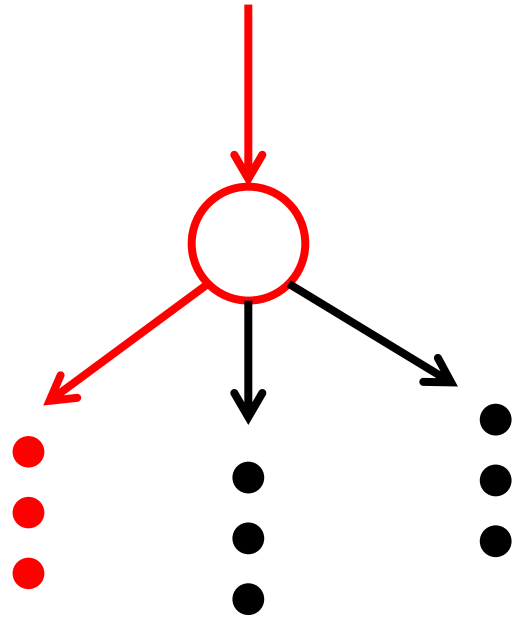
Aprovechamos una propiedad del recorrido DFS para calcular esto.



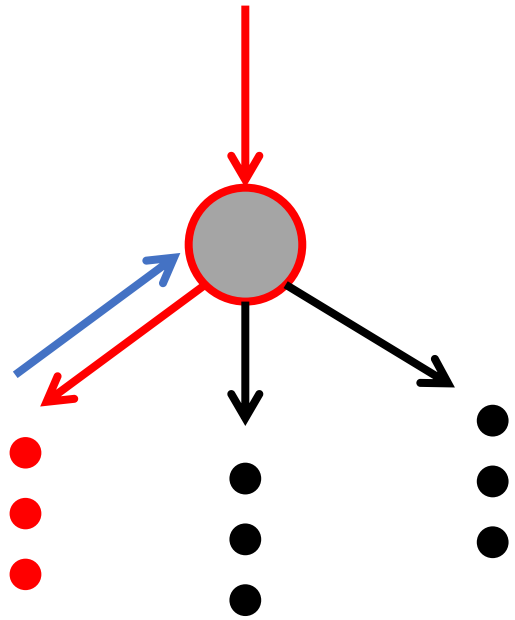
Aplicación – Orden Topológico



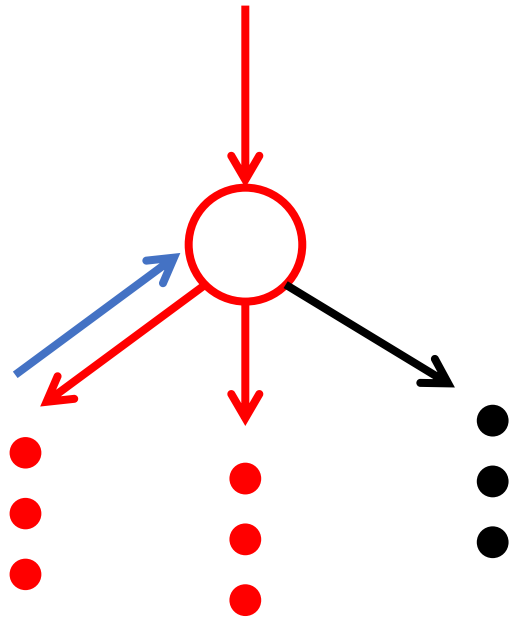
Aplicación – Orden Topológico



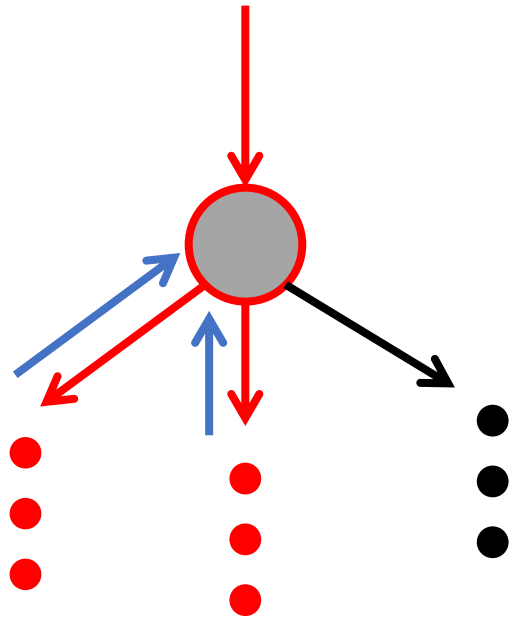
Aplicación – Orden Topológico



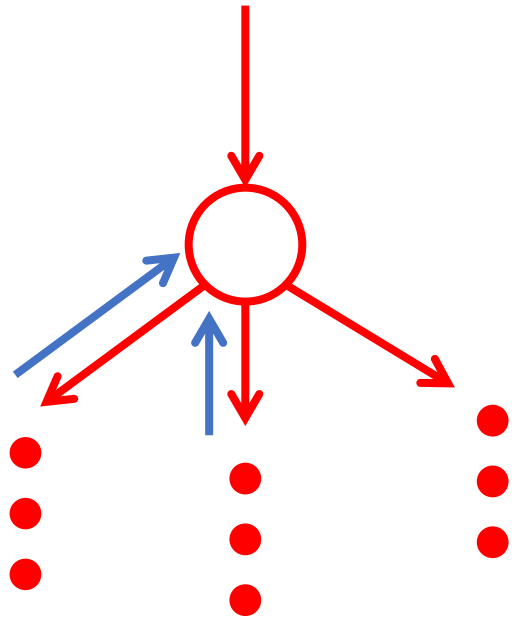
Aplicación – Orden Topológico



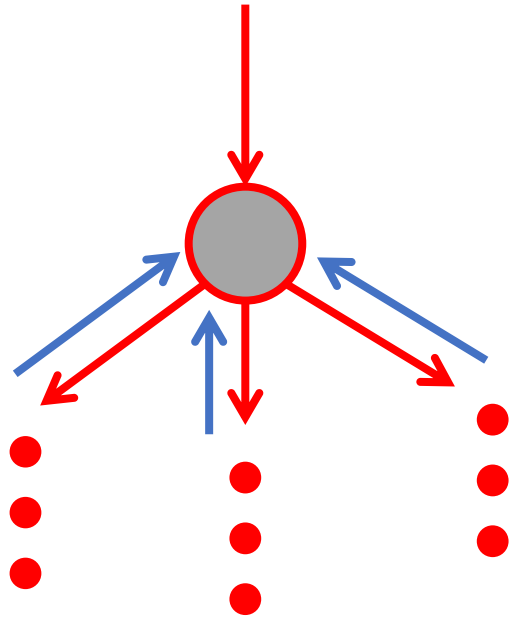
Aplicación – Orden Topológico



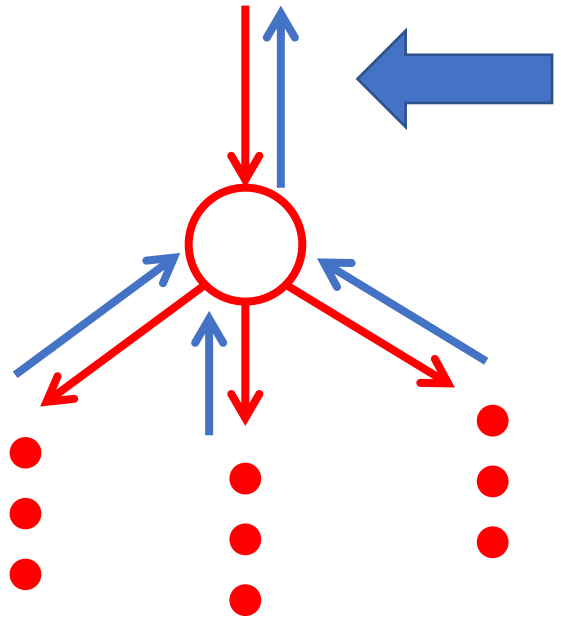
Aplicación – Orden Topológico



Aplicación – Orden Topológico

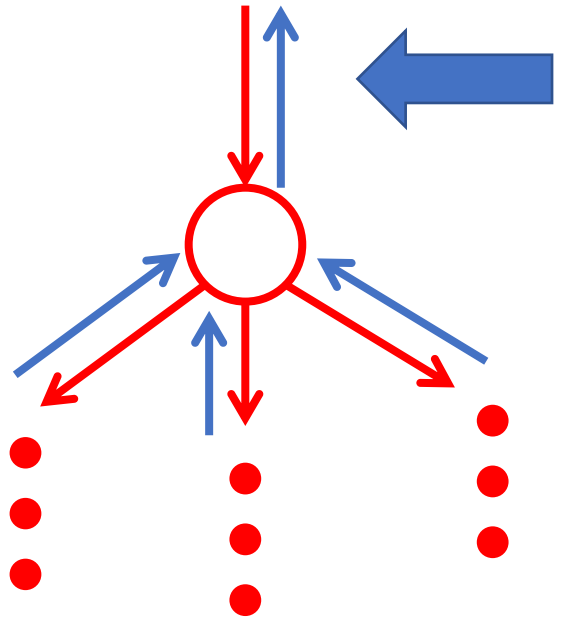


Aplicación – Orden Topológico



Regresar (backtracking) significa que ya se termino de procesar a **todos** sus vecinos.

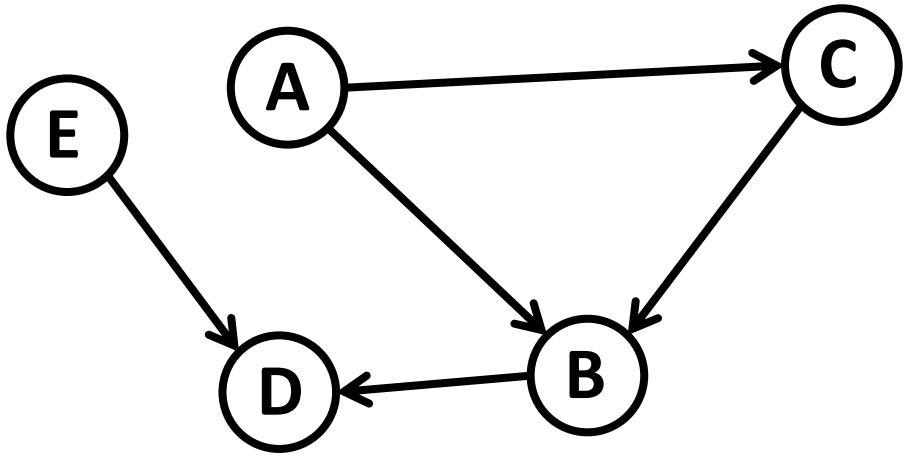
Aplicación – Orden Topológico



Regresar (backtracking) significa que ya se termino de procesar a **todos** sus vecinos.

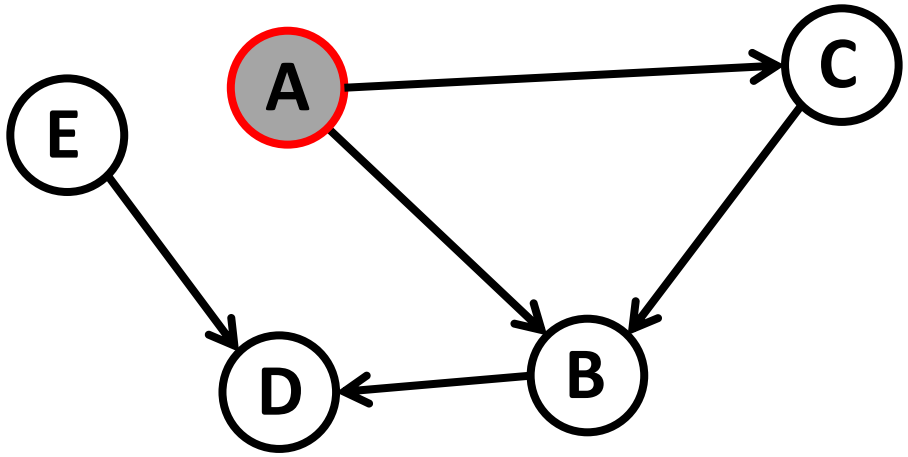
Entonces, podemos poner el nodo **antes** de sus vecinos en el orden topológico.

Aplicación – Orden Topológico



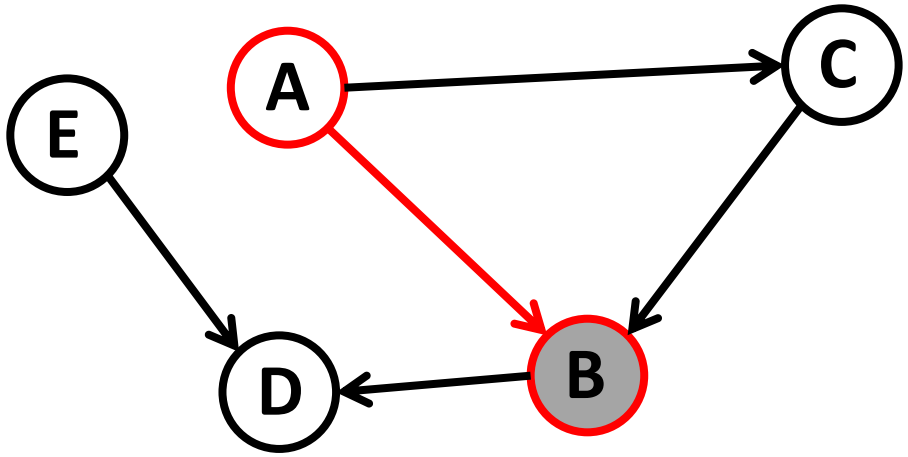
Orden topológico:

Aplicación – Orden Topológico



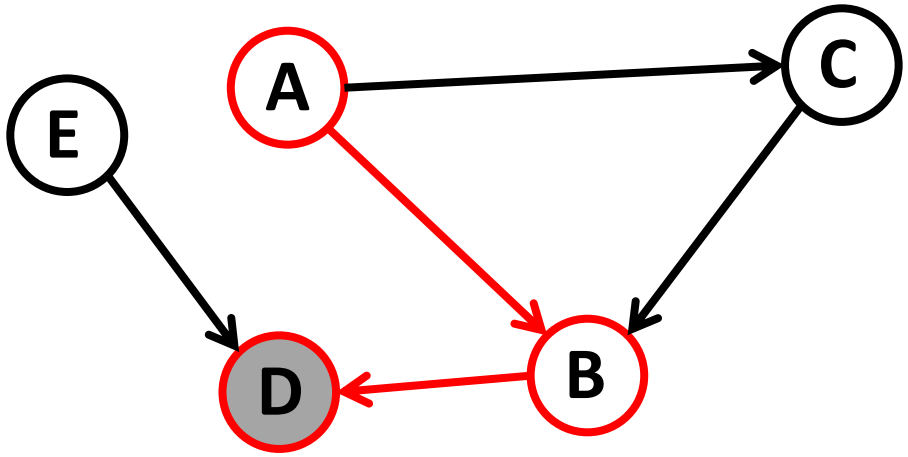
Orden topológico:

Aplicación – Orden Topológico



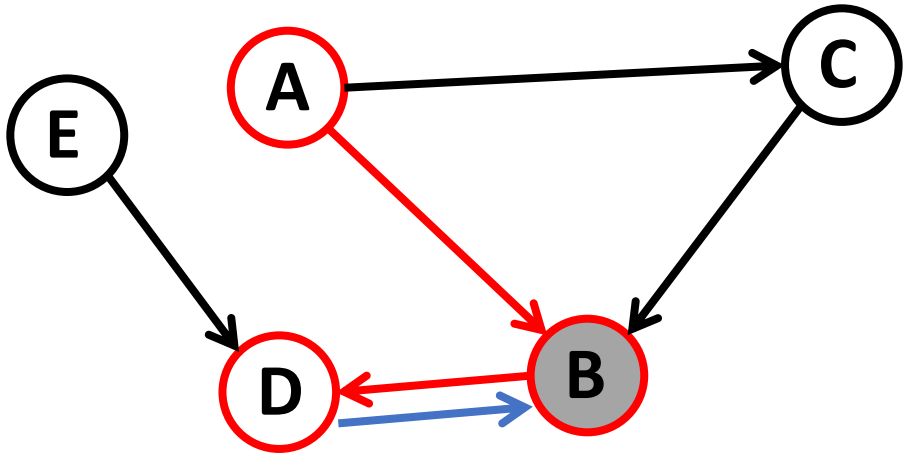
Orden topológico:

Aplicación – Orden Topológico



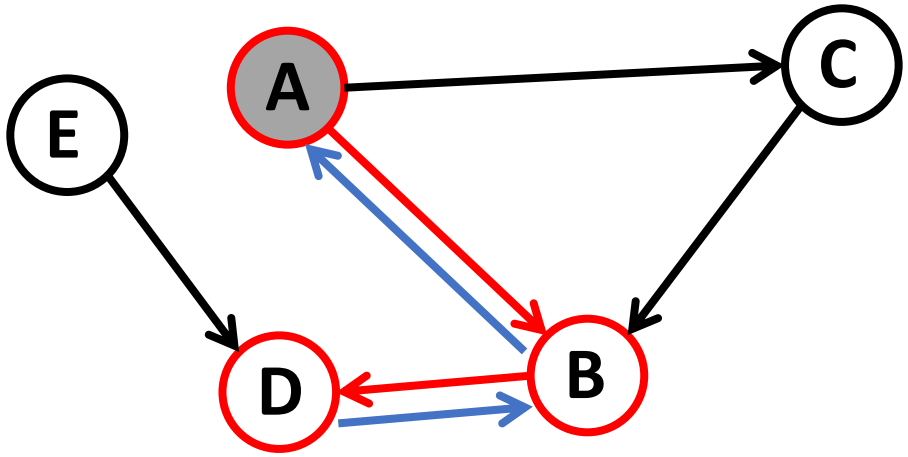
Orden topológico:

Aplicación – Orden Topológico



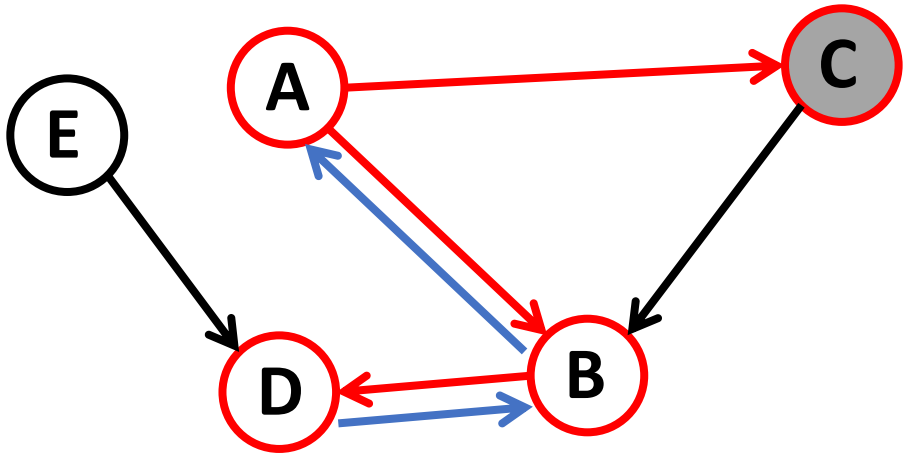
Orden topológico: **D**

Aplicación – Orden Topológico



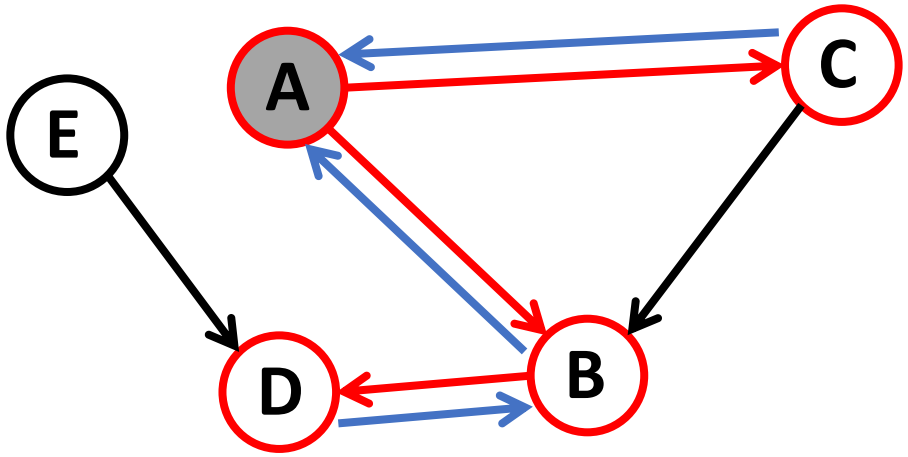
Orden topológico: **B D**

Aplicación – Orden Topológico



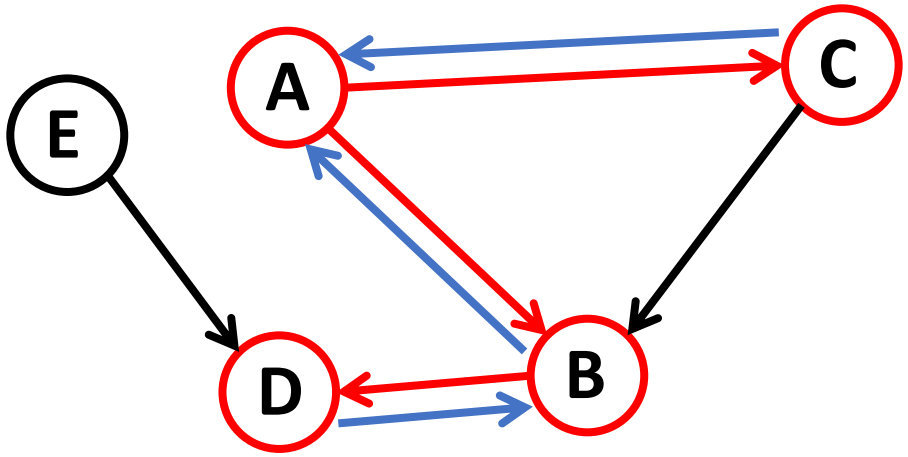
Orden topológico: **B D**

Aplicación – Orden Topológico



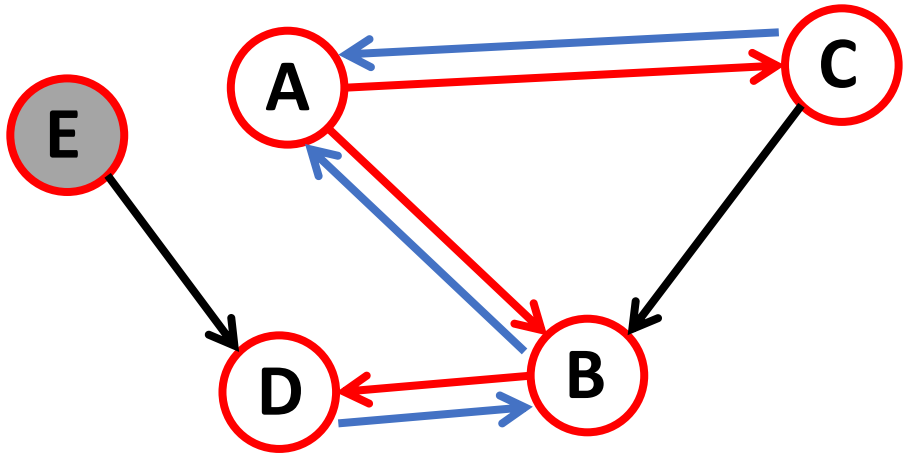
Orden topológico: **C B D**

Aplicación – Orden Topológico



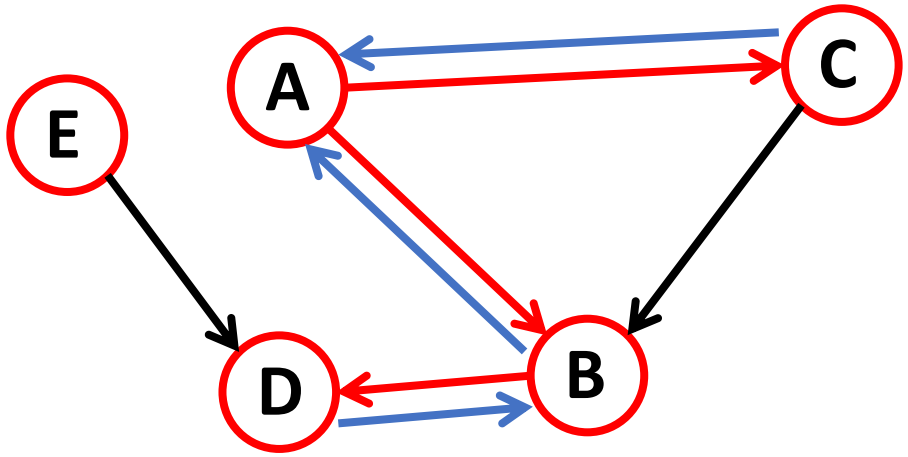
Orden topológico: **A C B D**

Aplicación – Orden Topológico



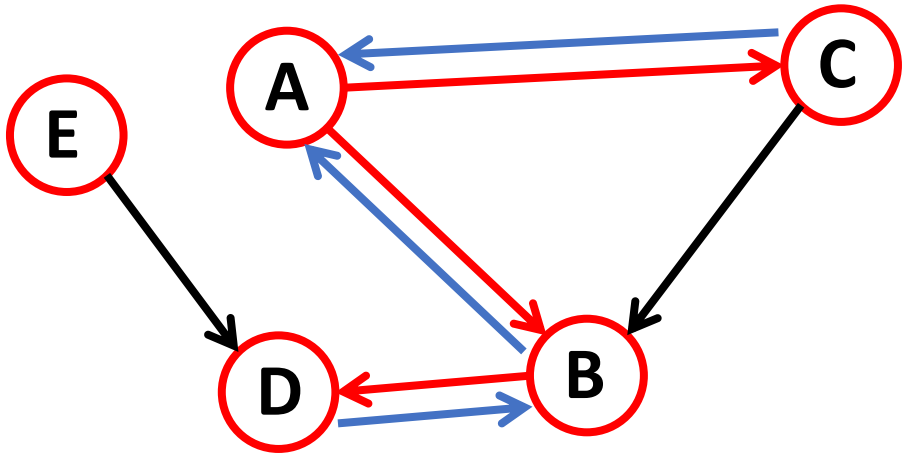
Orden topológico: **A C B D**

Aplicación – Orden Topológico



Orden topológico: **E A C B D**

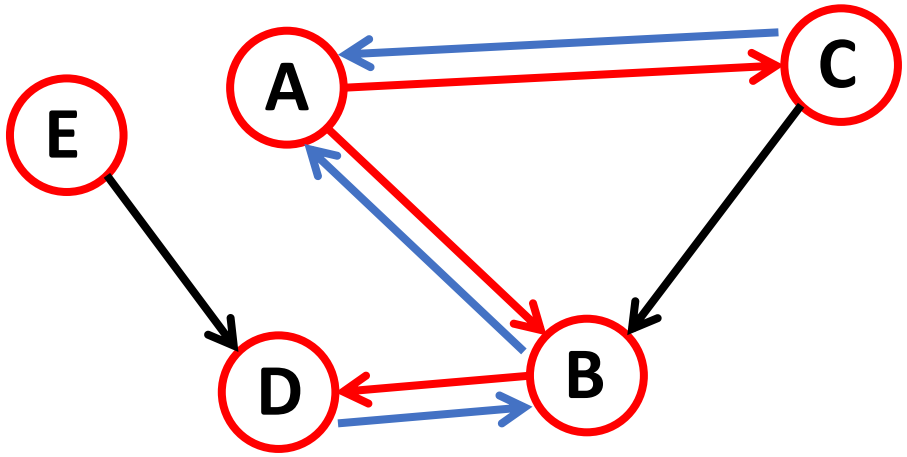
Aplicación – Orden Topológico




```cpp
vector<int> g[n];
bool vis[n];


void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v (ida)
      dfs(v);
      // v <- u (vuelta)
    }
  }

  // justo antes de salir del nodo (backtracking)
}
```

Orden topológico: **E A C B D**

Aplicación – Orden Topológico




```
vector<int> g[n];
bool vis[n];
vector<int> invTopSort;

void dfs(int u) {
  vis[u] = true;
  for (int v : g[u]) {
    if (!vis[v]) {
      // u -> v (ida)
      dfs(v);
      // v <- u (vuelta)
    }
  }
  invTopSort.push_back(u);
  // justo antes de salir del nodo (backtracking)
}
```

Orden topológico: **E A C B D**

invTopSort : **D B C A E**

El nodo se pone al final de la lista en vez de al inicio, solo hace falta trabajar con la lista invertida.

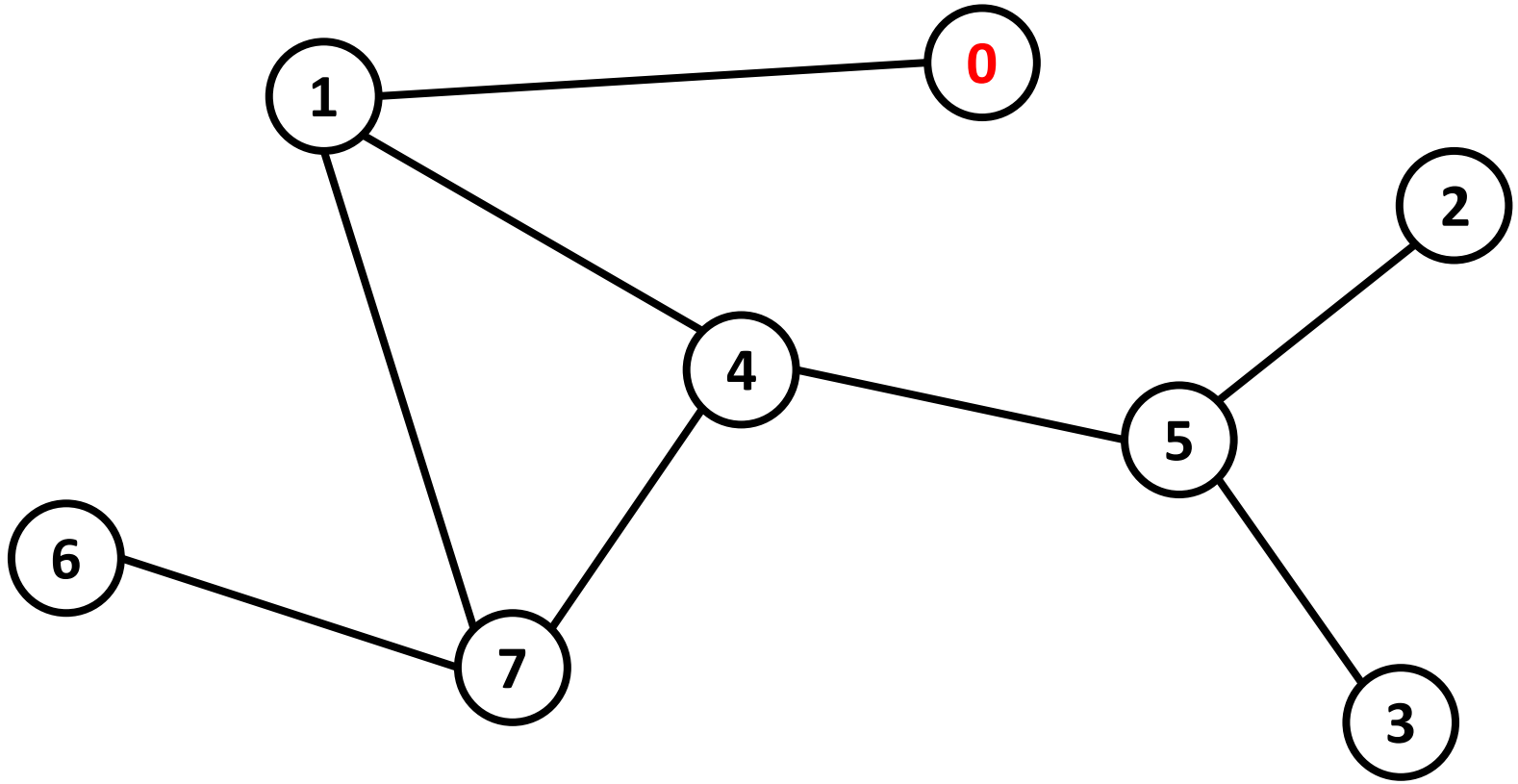
Otros Algoritmos con DFS

- Flood Fill (DFS en matrices)
- Detectar ciclos en grafos dirigidos
- Puentes y Puntos de Articulación
- Componentes Fuertemente Conexos
  - Problema 2-SAT
- Heavy-Light Decomposition de arboles
- Euler Tour en arboles

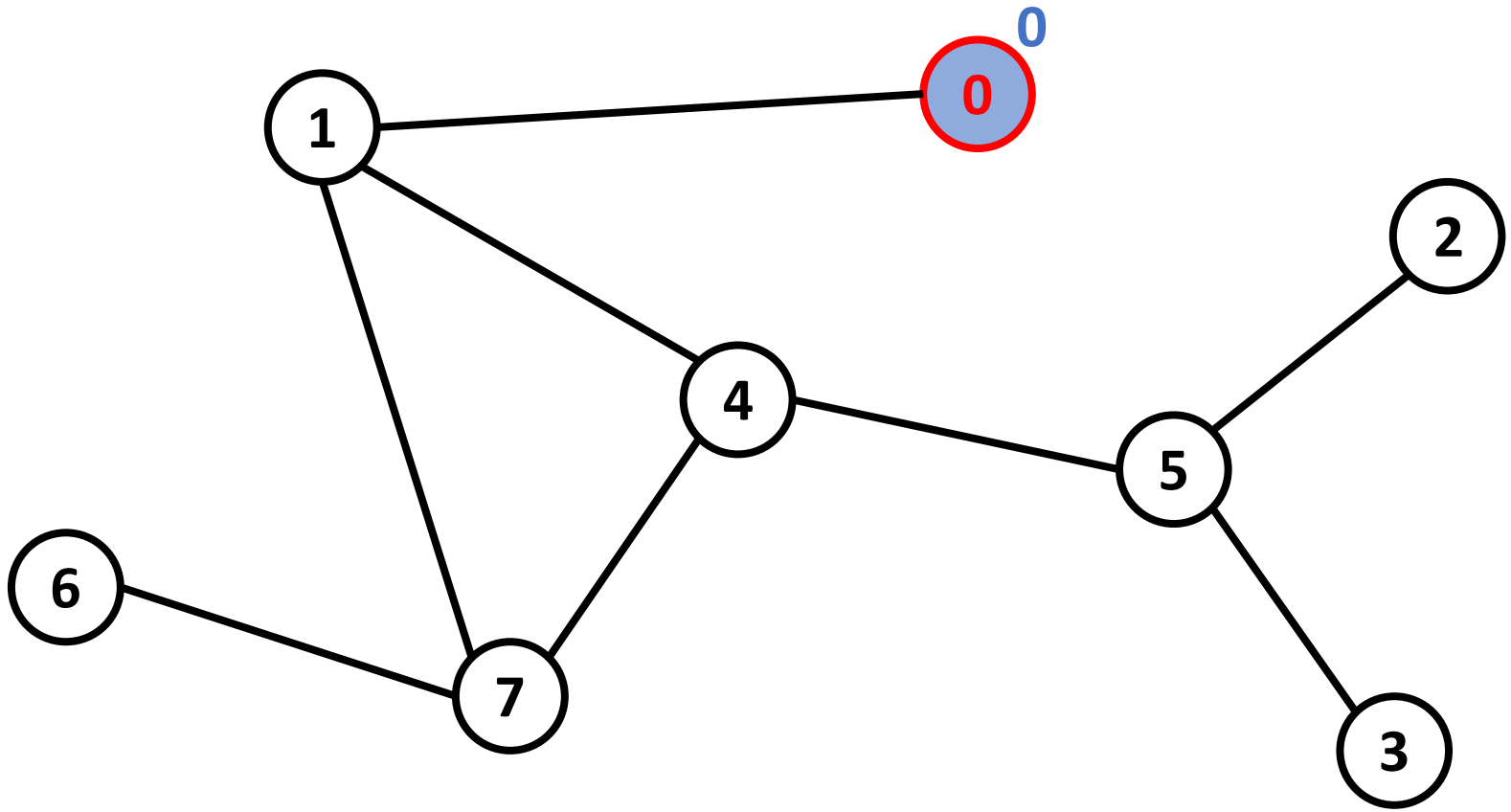
BFS

- Algoritmo de recorrido de grafos
- Empieza en un nodo y visita el resto en orden creciente segun la distancia al nodo inicial
- Es un poco mas dificil de implementar que DFS

BFS

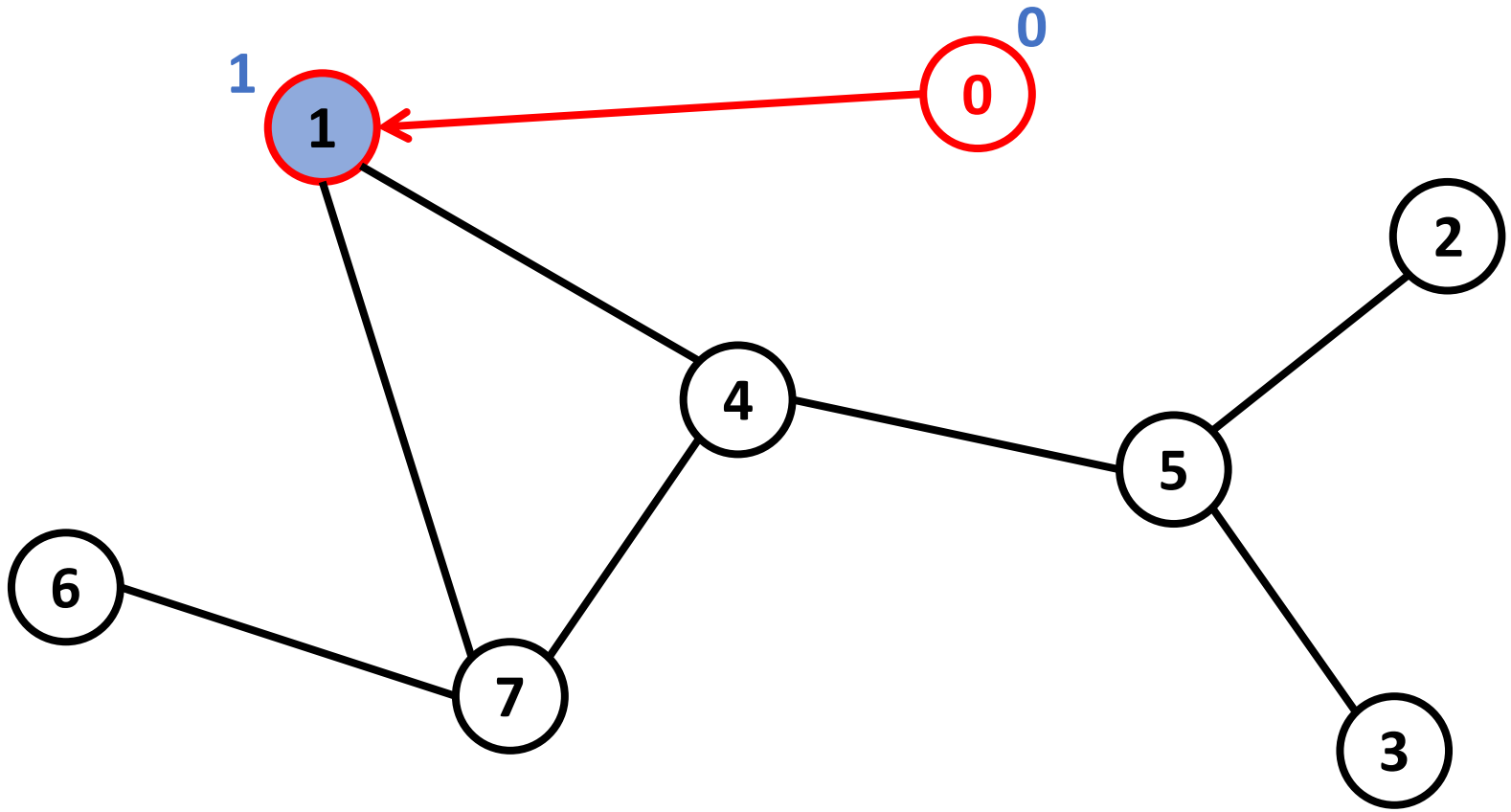


BFS



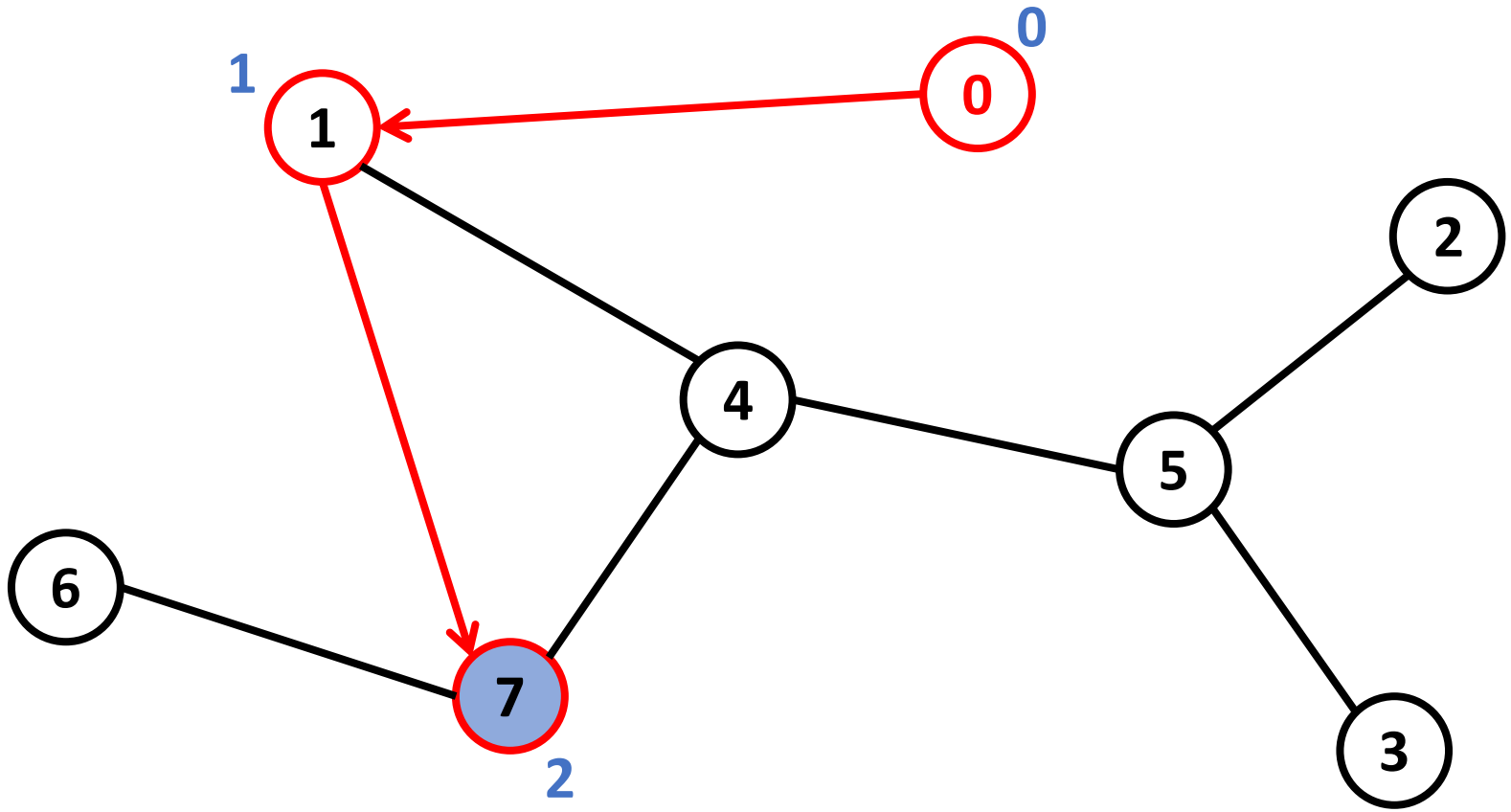
Orden de recorrido: 0

BFS



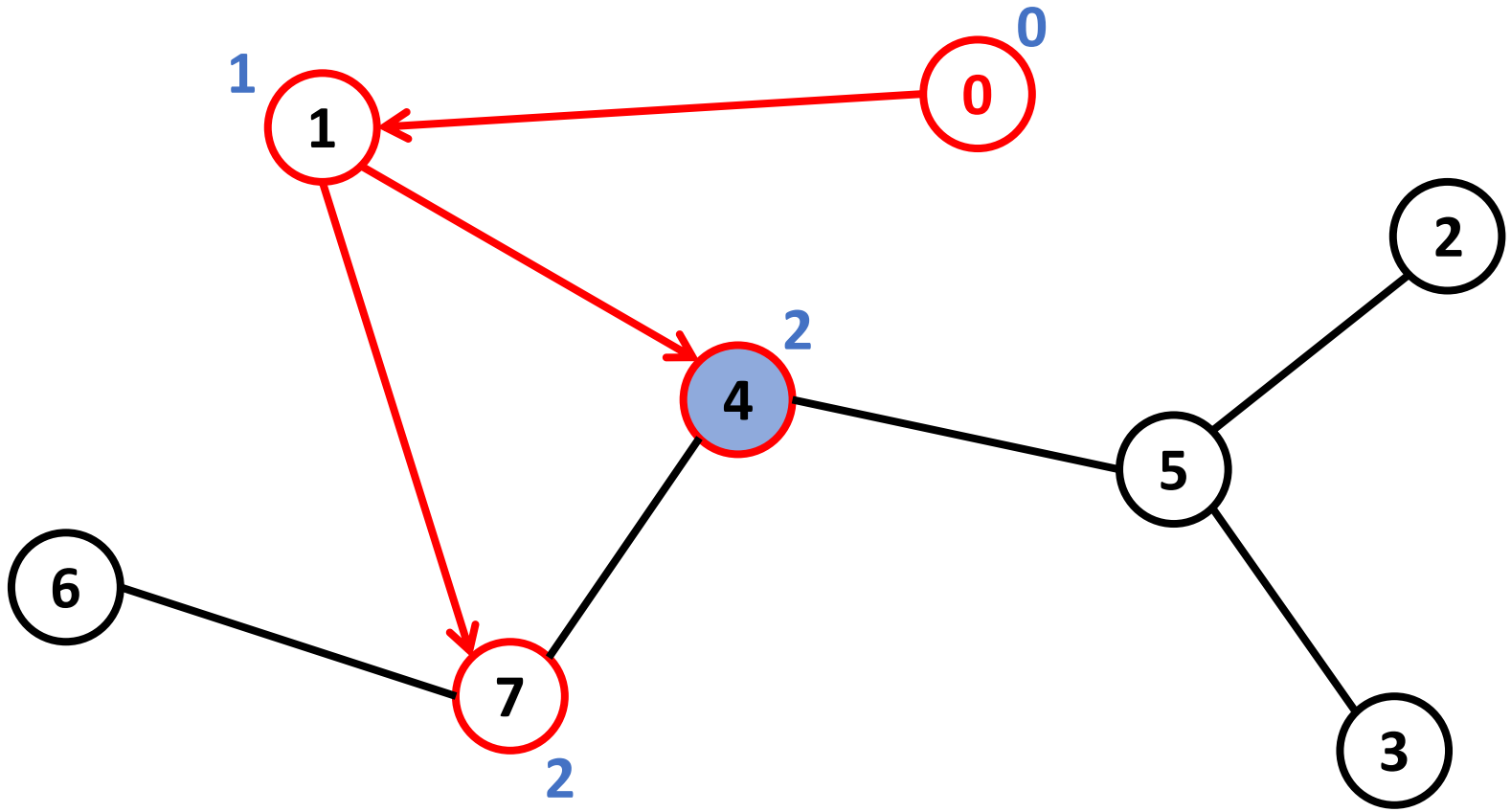
Orden de recorrido: 0, 1

BFS



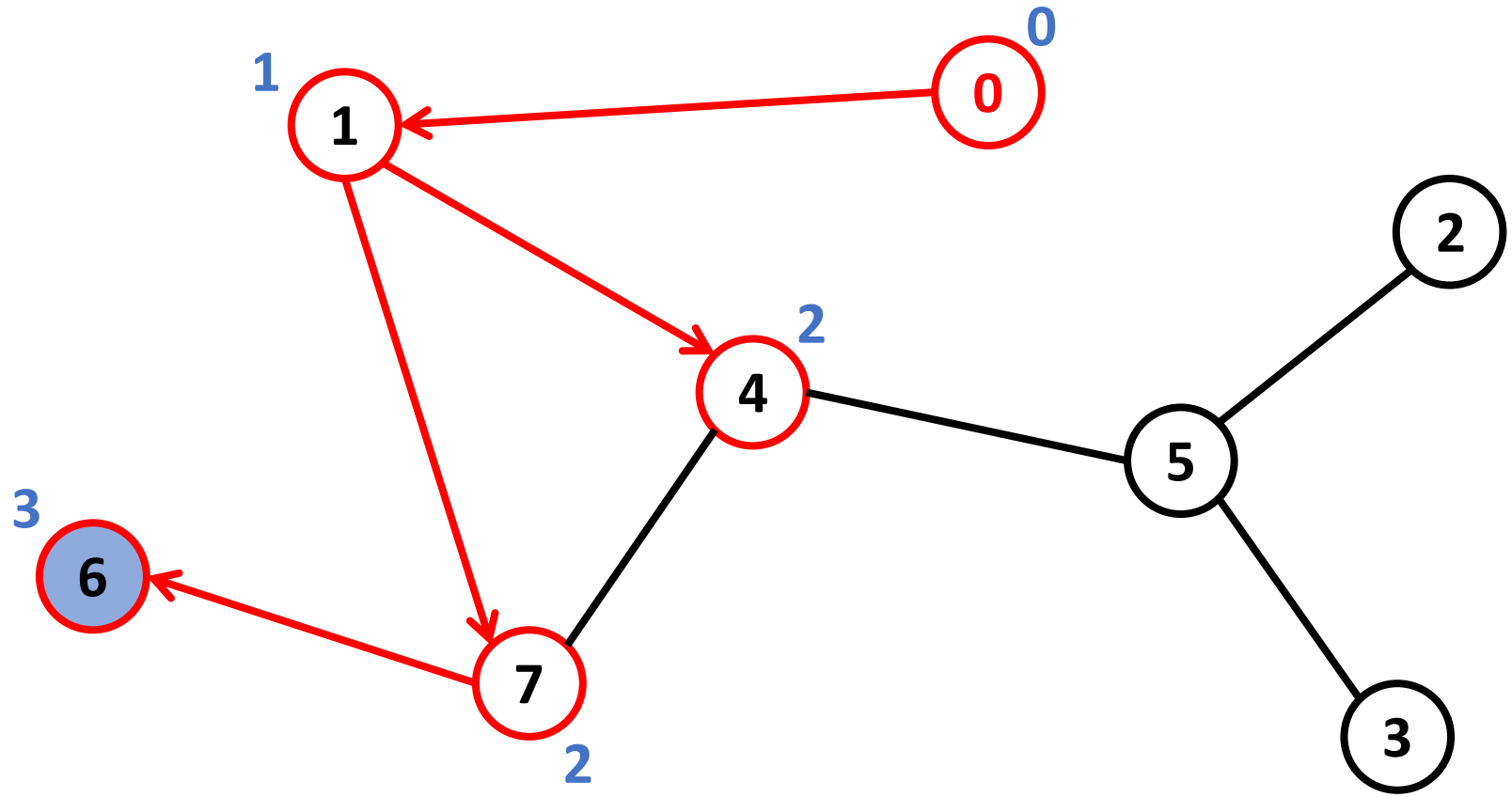
Orden de recorrido: 0, 1, 7

BFS



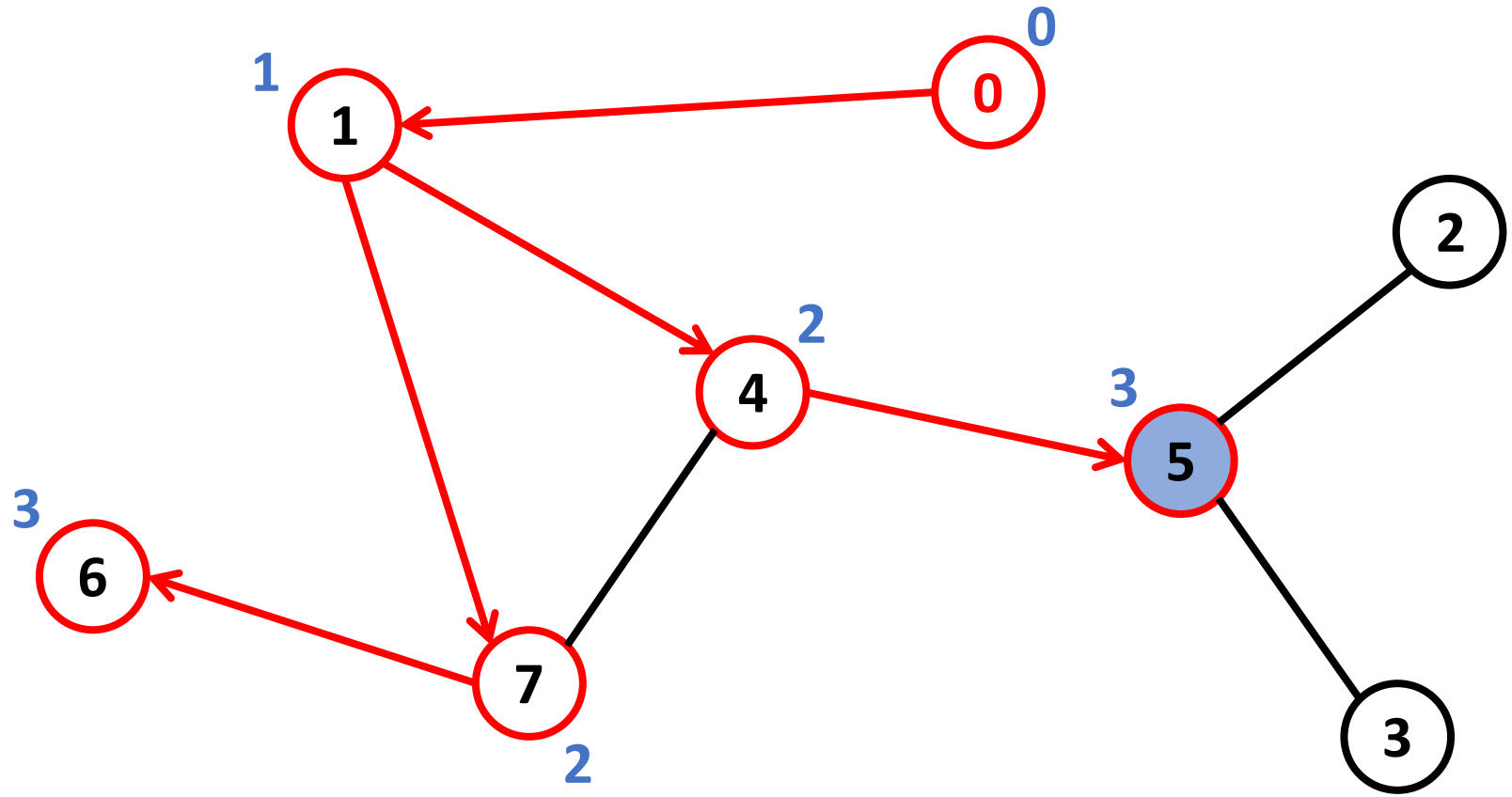
Orden de recorrido: 0, 1, 7, 4

BFS



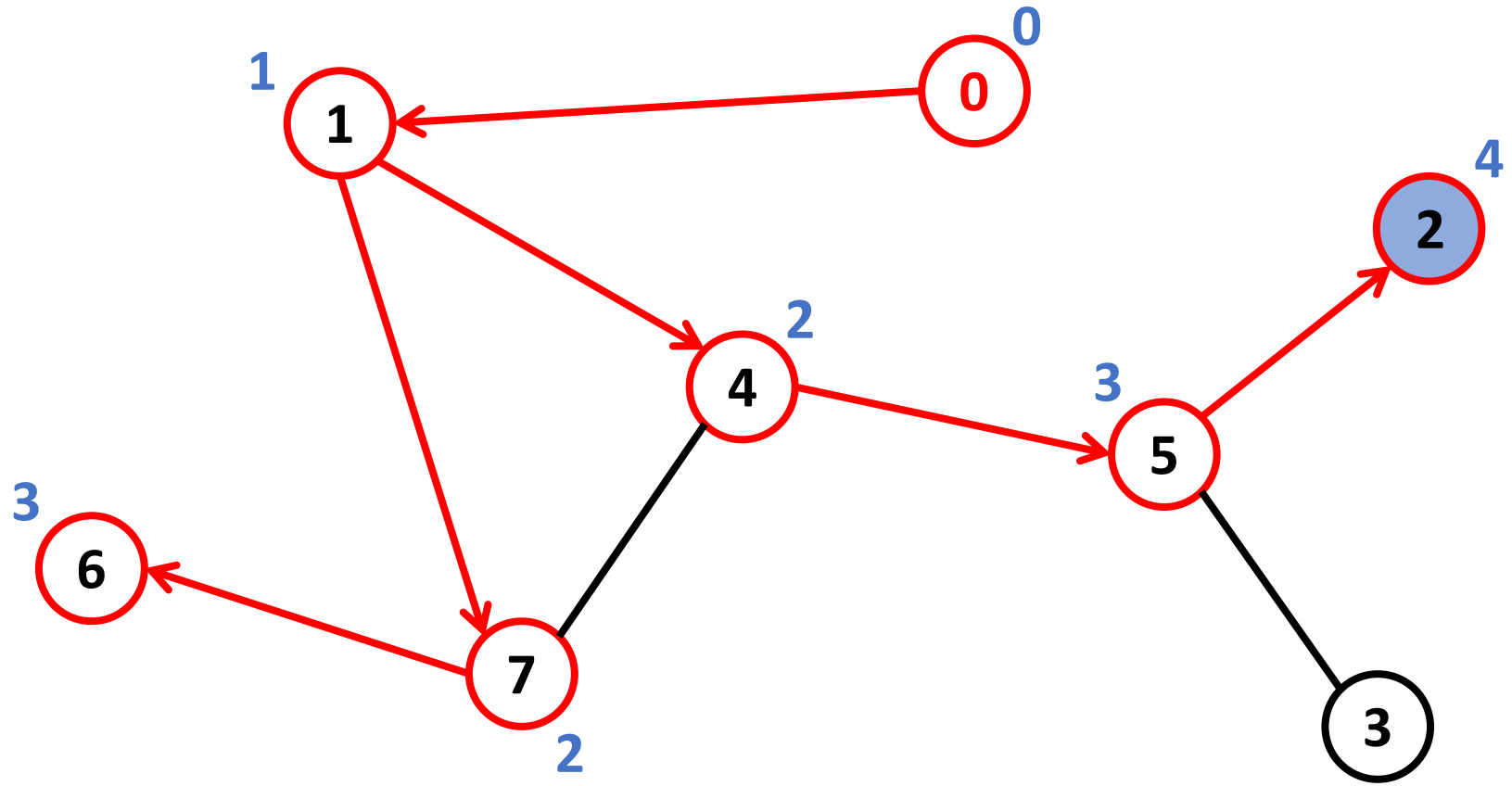
Orden de recorrido: 0, 1, 7, 4, 6

BFS



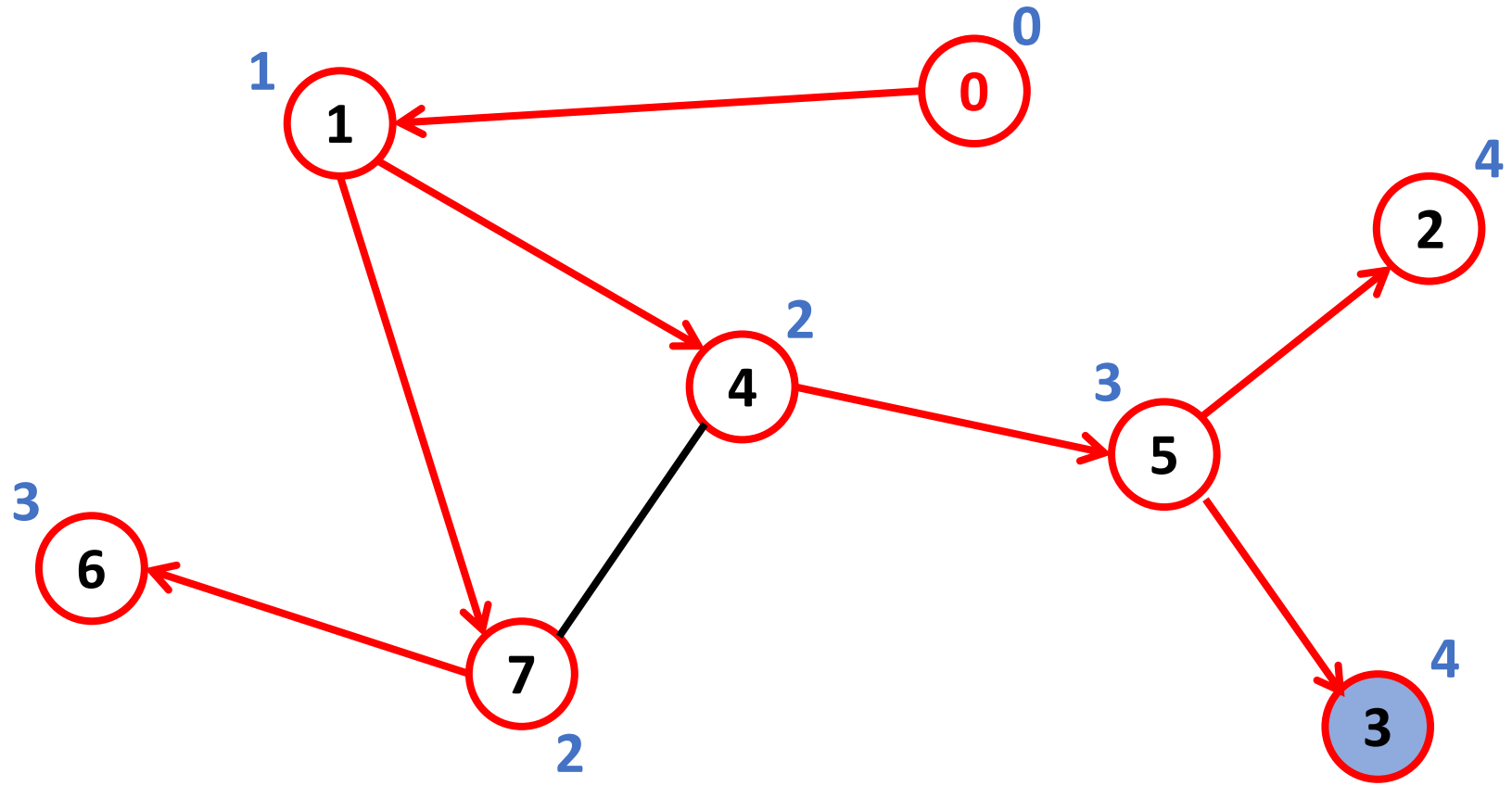
Orden de recorrido: 0, 1, 7, 4, 6, 5

BFS



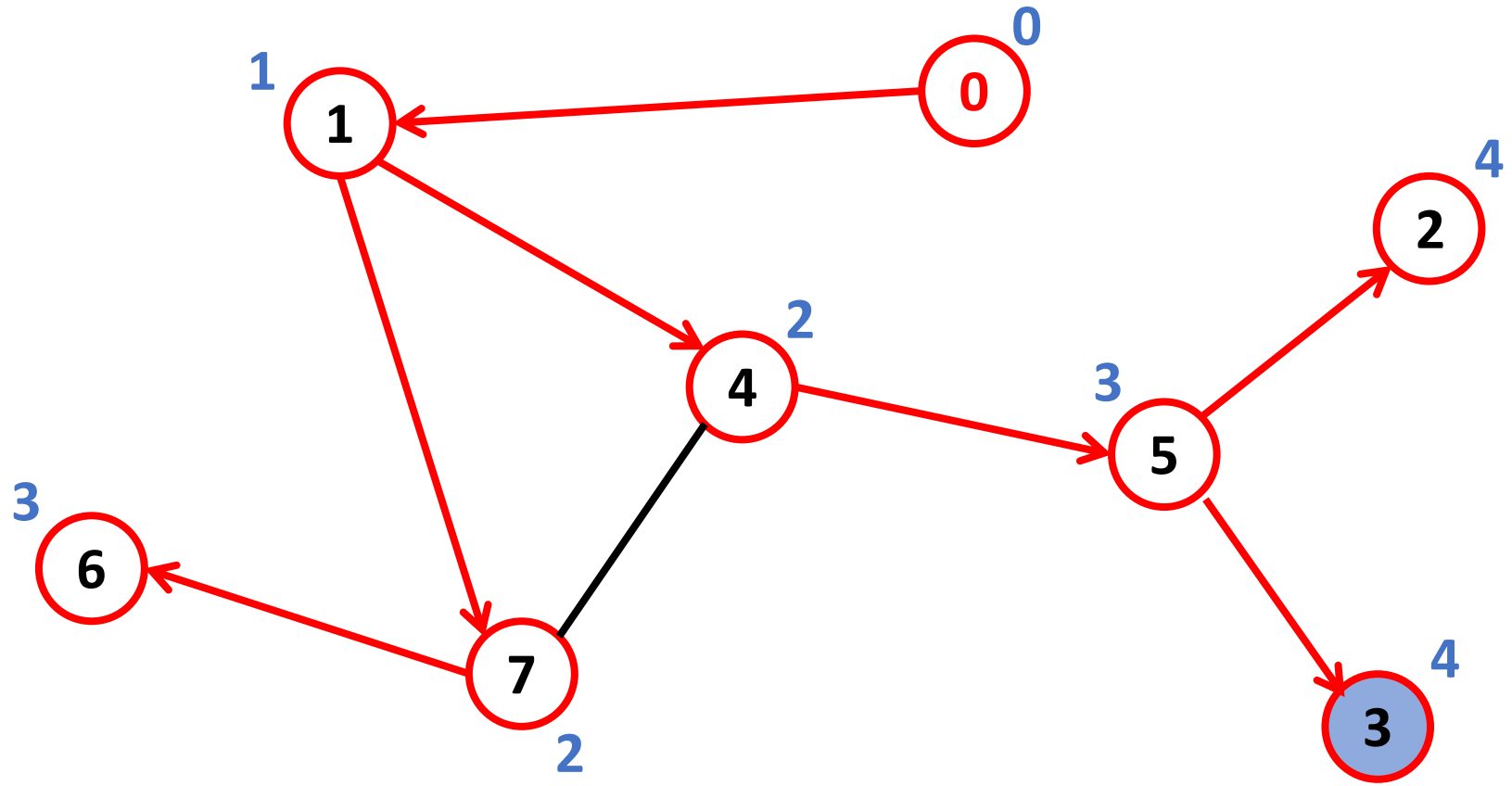
Orden de recorrido: 0, 1, 7, 4, 6, 5, 2

BFS



Orden de recorrido: 0, 1, 7, 4, 6, 5, 2, 3

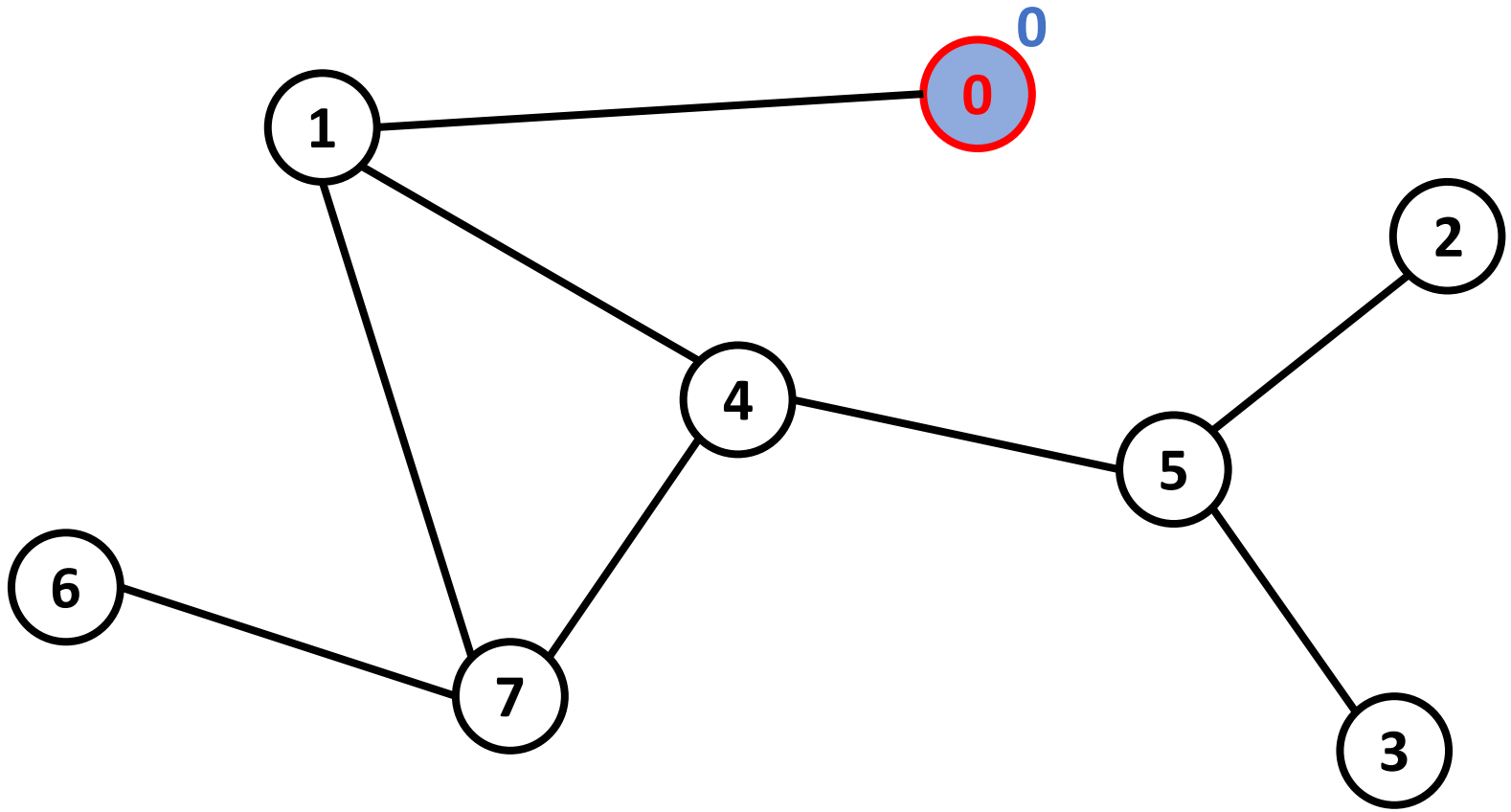
BFS



Orden de recorrido: 0, 1, 7, 4, 6, 5, 2, 3

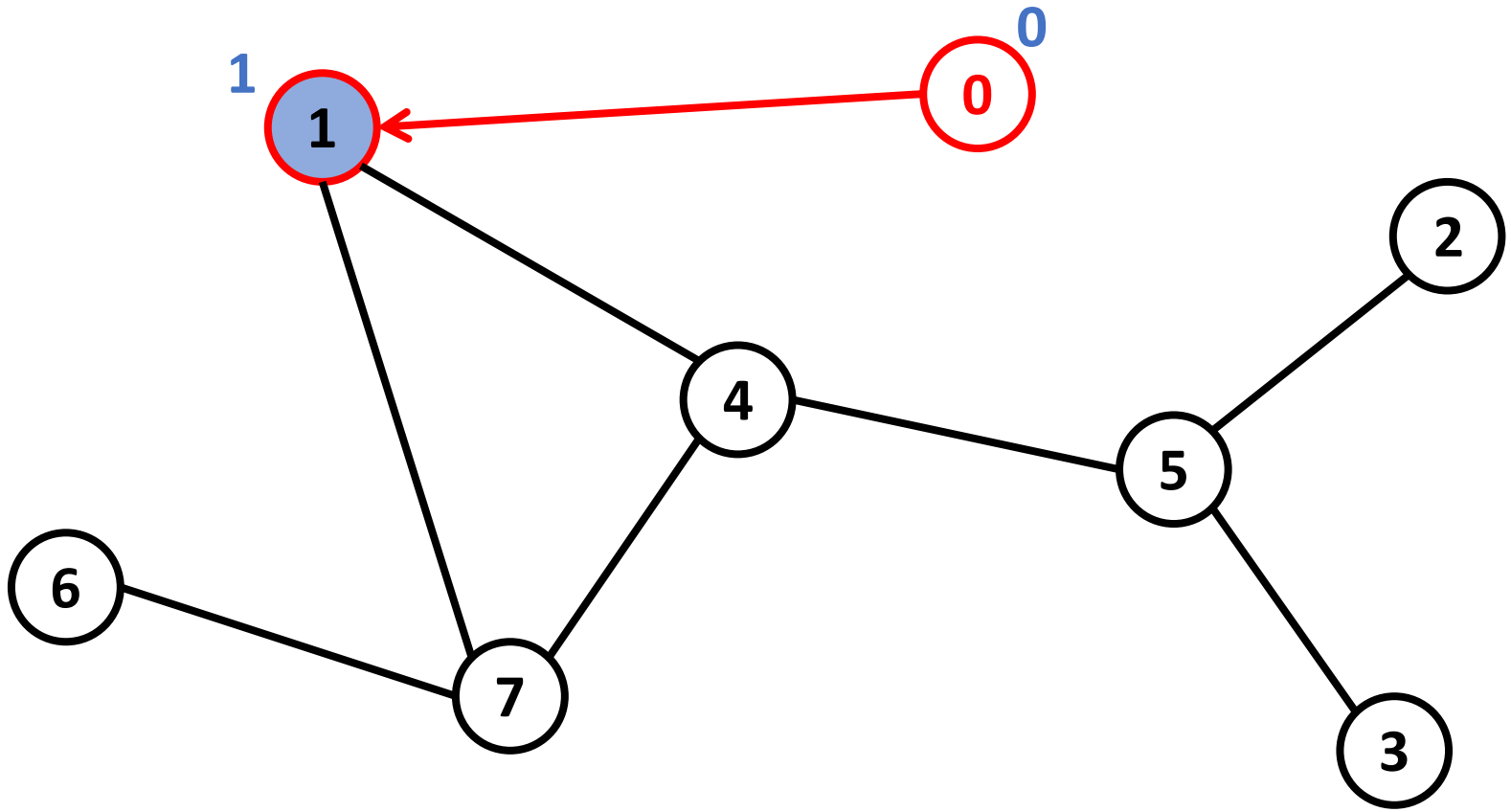
Distancia : 0 1 2 2 3 3 4 4

BFS



Cola: {1}
Nodo: 0

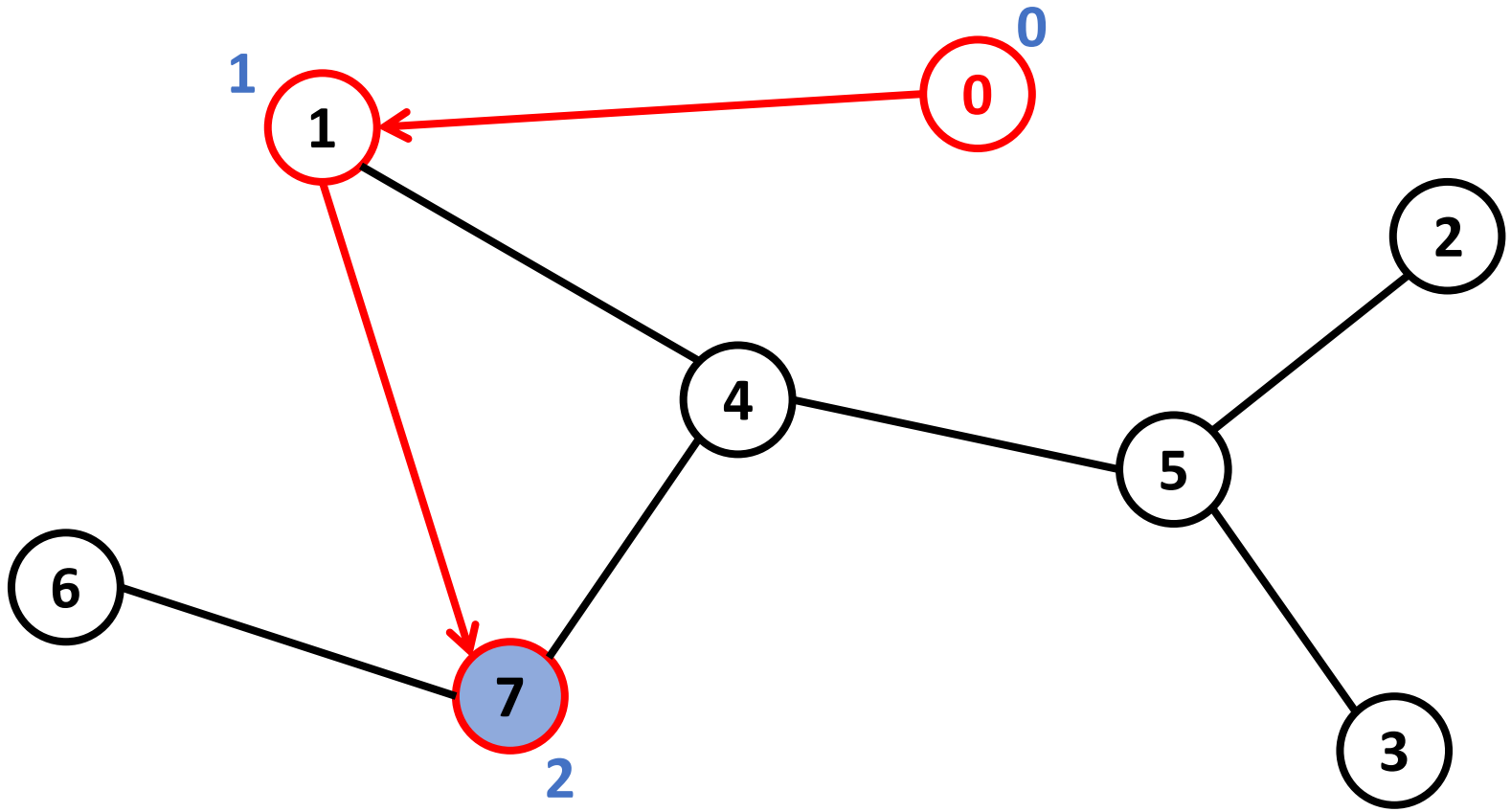
BFS



Cola: {~~1,~~ 7, 4}

Nodo: 1

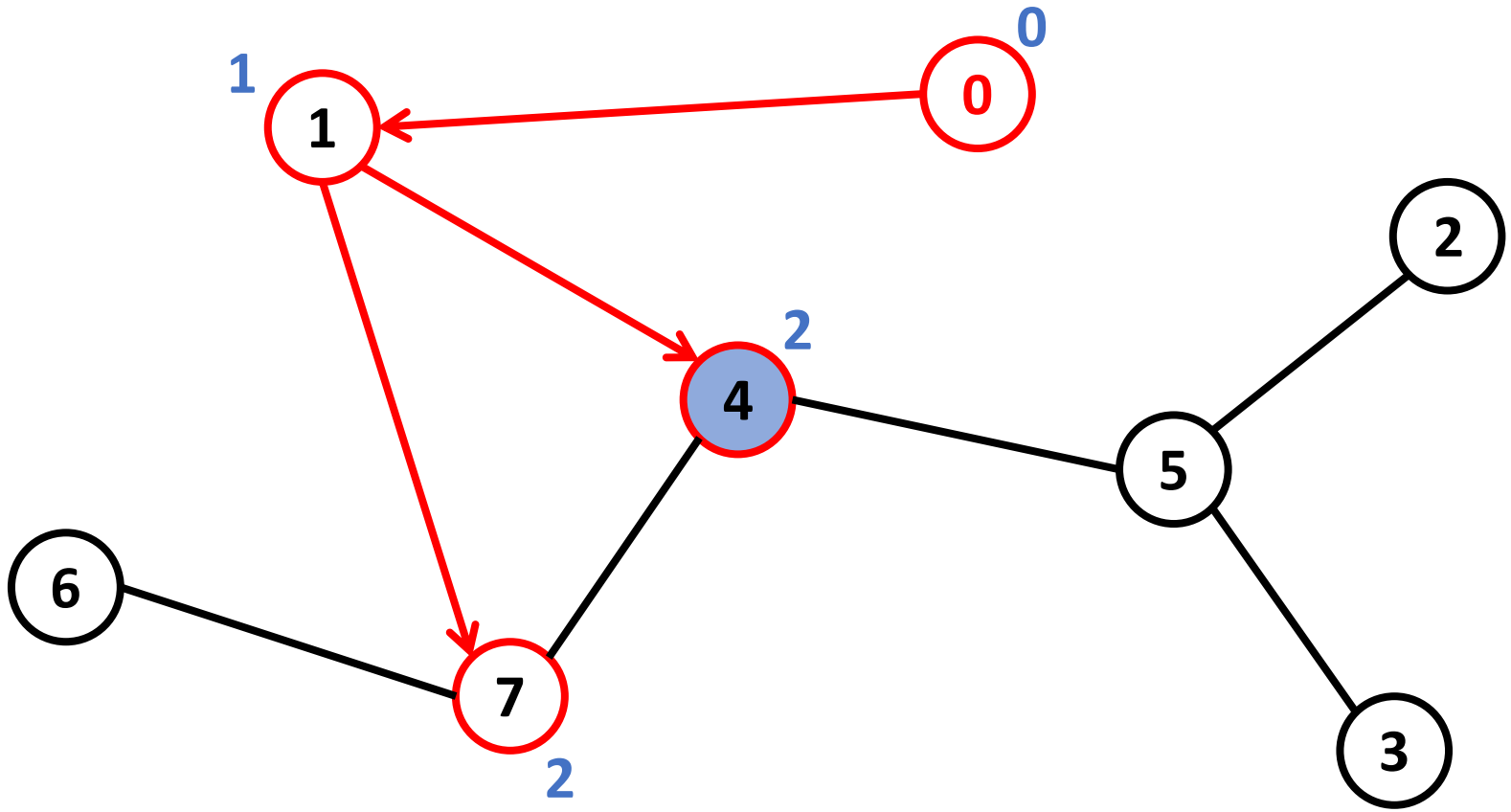
BFS



Cola: {~~1, 7,~~ 4, 6}

Nodo: 7

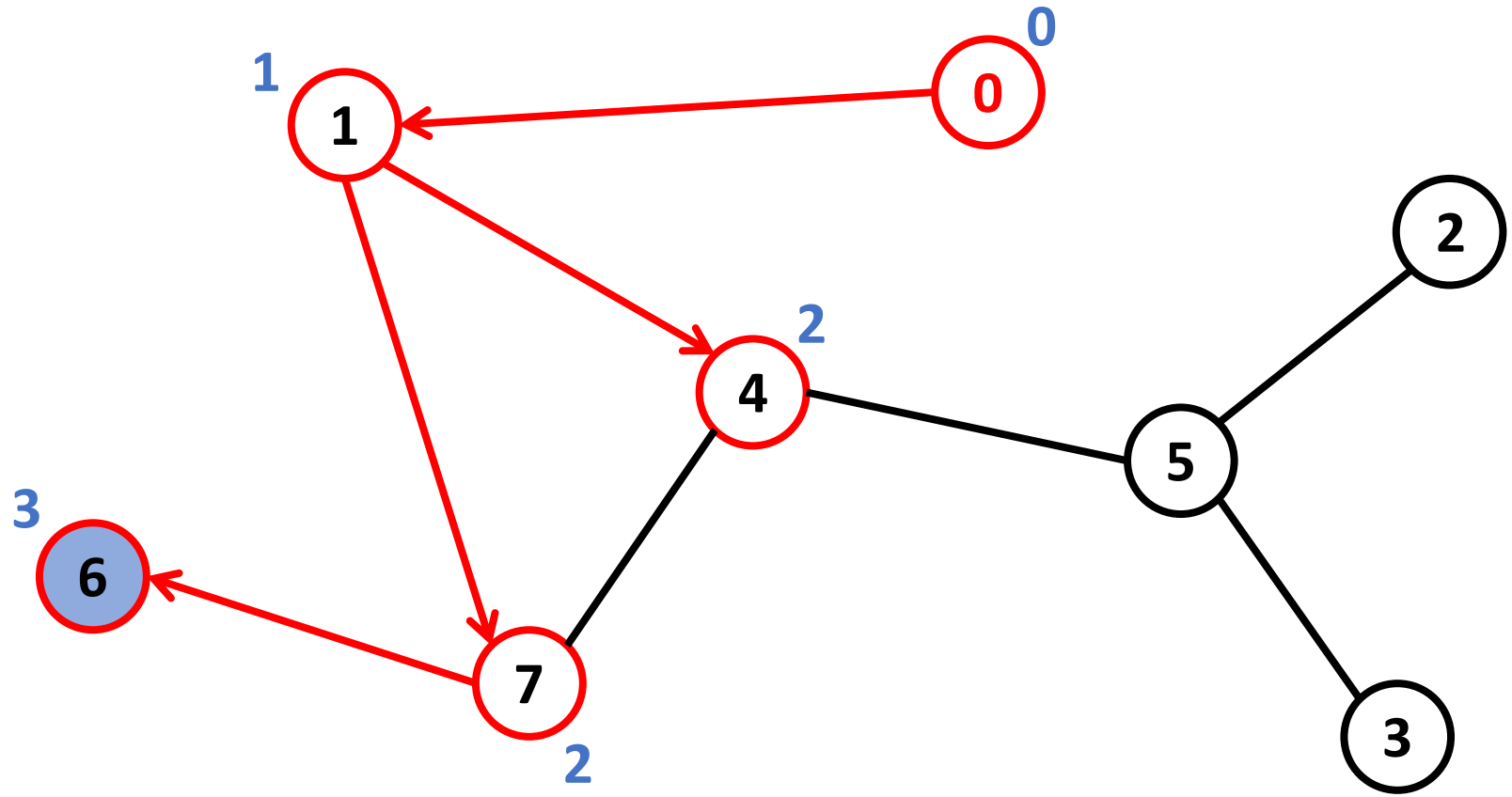
BFS



Cola: {~~1, 7, 4,~~ 6, 5}

Nodo: 4

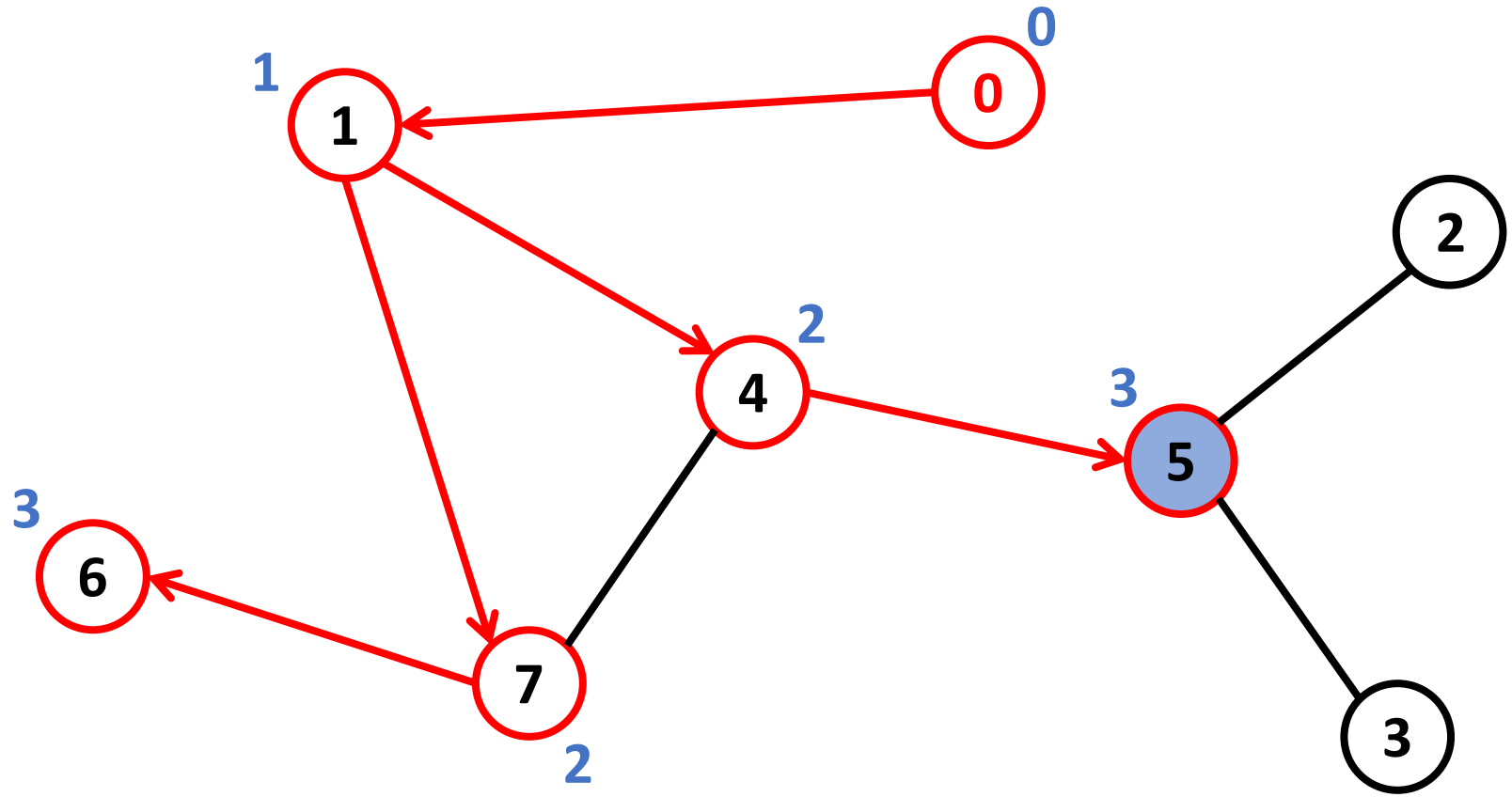
BFS



Cola: {~~1, 7, 4, 6,~~ 5}

Nodo: 6

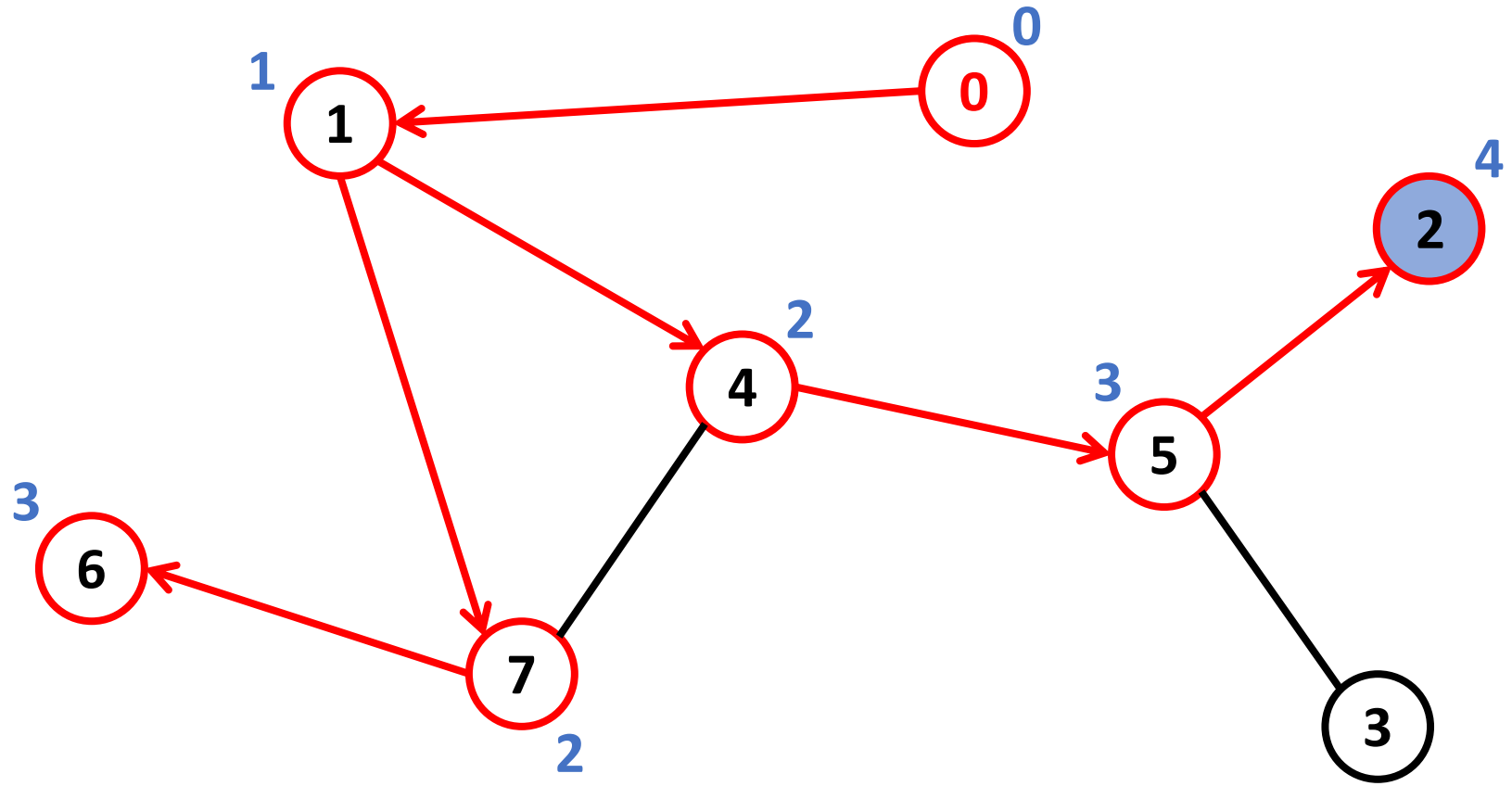
BFS



Cola: {~~1, 7, 4, 6, 5,~~ 2, 3}

Nodo: 5

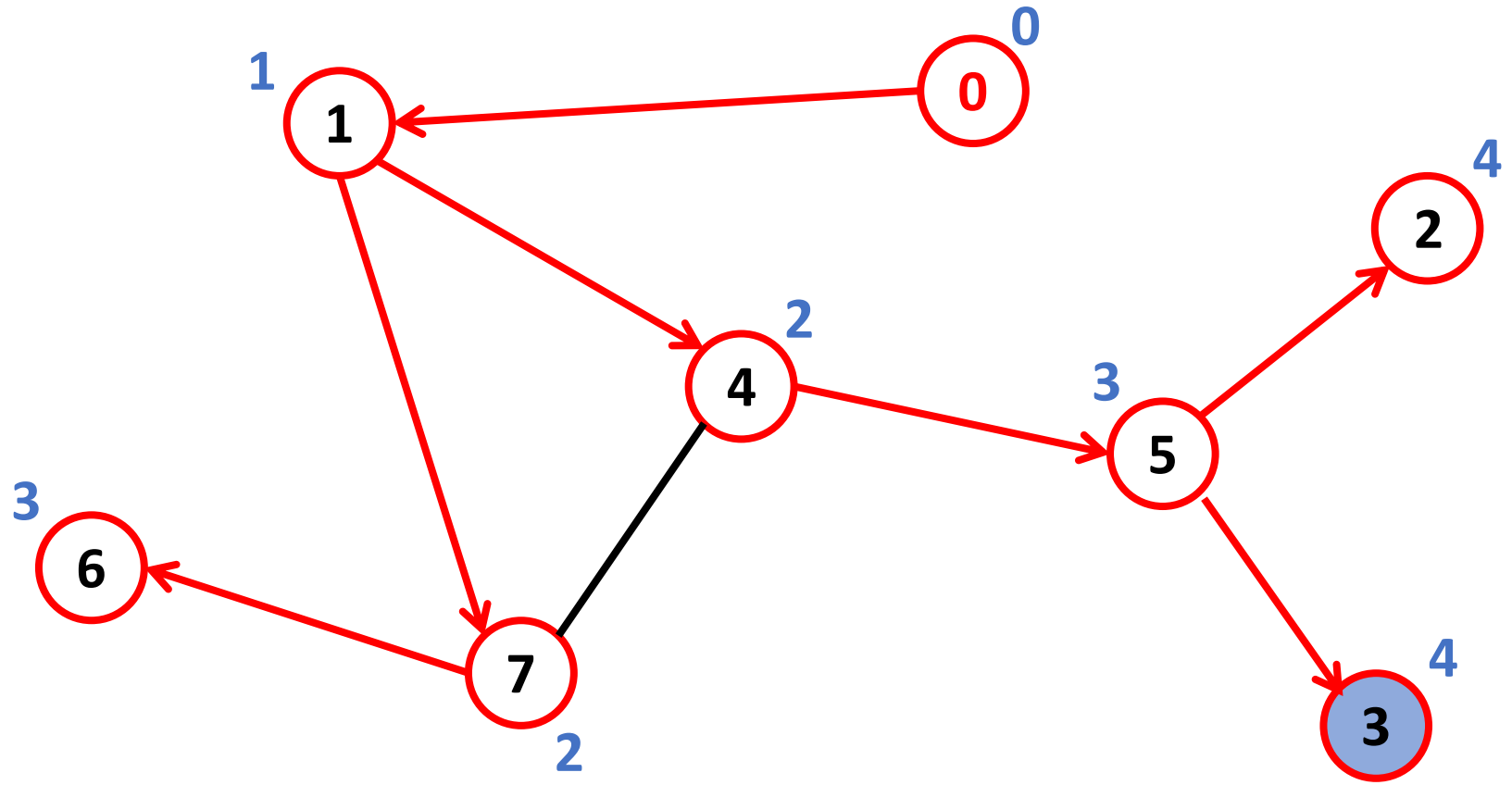
BFS



Cola: {~~1, 7, 4, 6, 5, 2,~~ 3}

Nodo: 2

BFS



Cola: {~~1, 7, 4, 6, 5, 2, 3~~}

Nodo: 3

Código

Necesitamos:

```
vector<int> g[n];

vector<int> nivel(n, -1);

queue<int> q;
```

```
int inicio = 0;
nivel[inicio] = 0;
q.push(inicio);

while (!q.empty()) {
  int u = q.front();
  q.pop();
  for (int v : g[u]) {
    if (nivel[v] == -1) {
      nivel[v] = nivel[u] + 1;
      q.push(v);
    }
  }
}
```