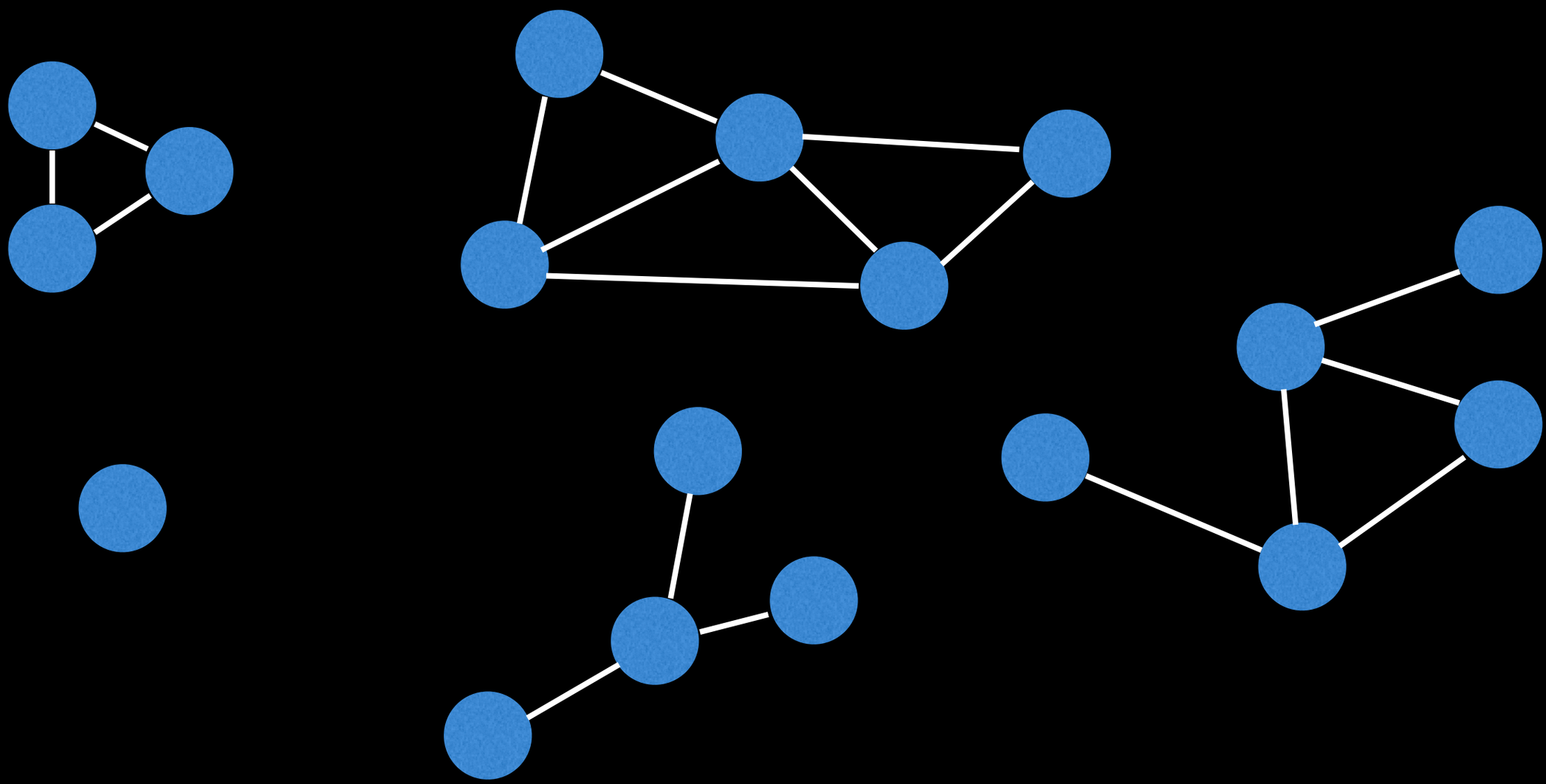
Connected Components

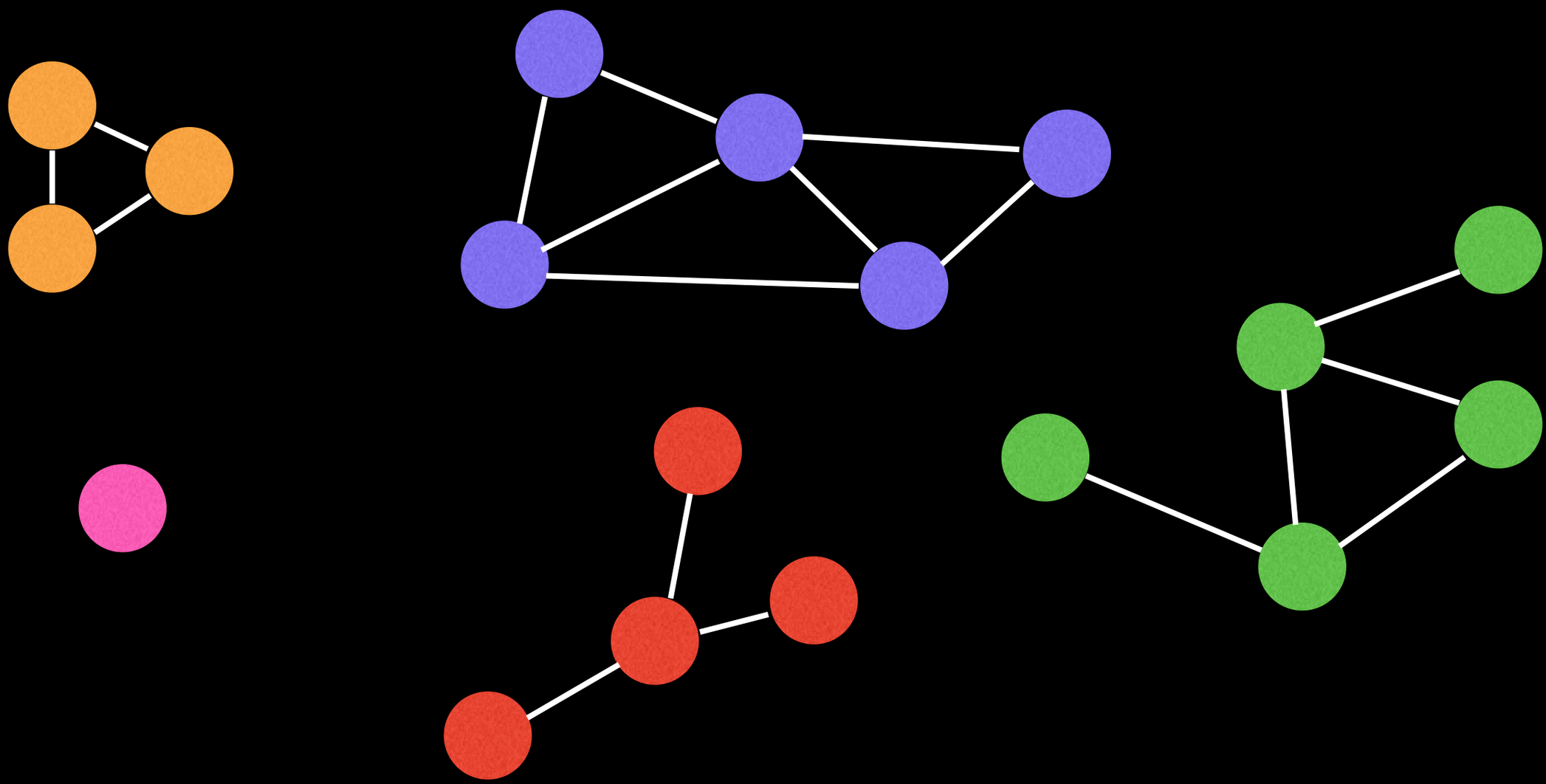
Connected Components

Sometimes a graph is split into multiple components. It’s useful to be able to identify and count these components.



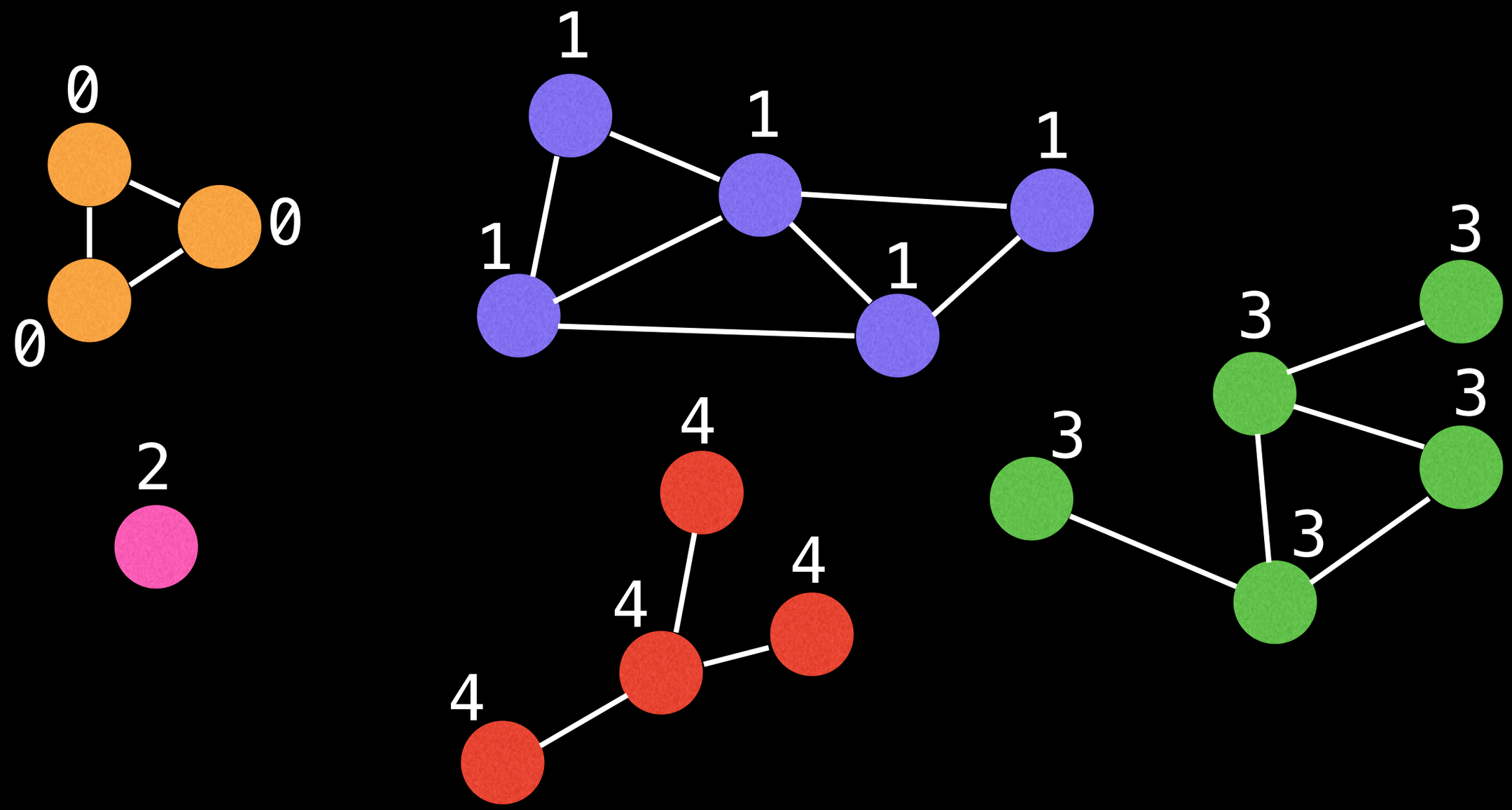
Connected Components

Sometimes a graph is split into multiple components. It’s useful to be able to identify and count these components.

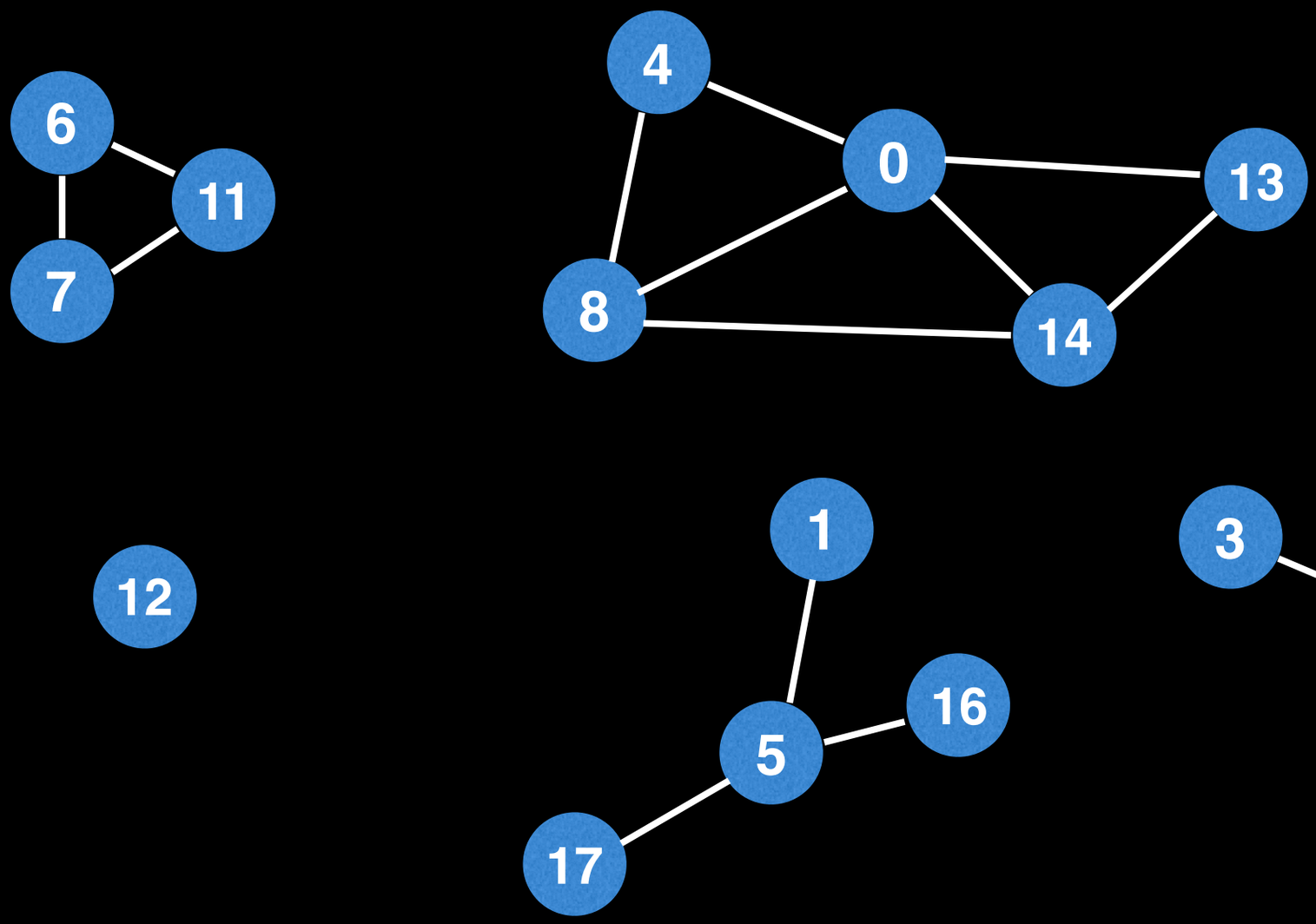


Connected Components

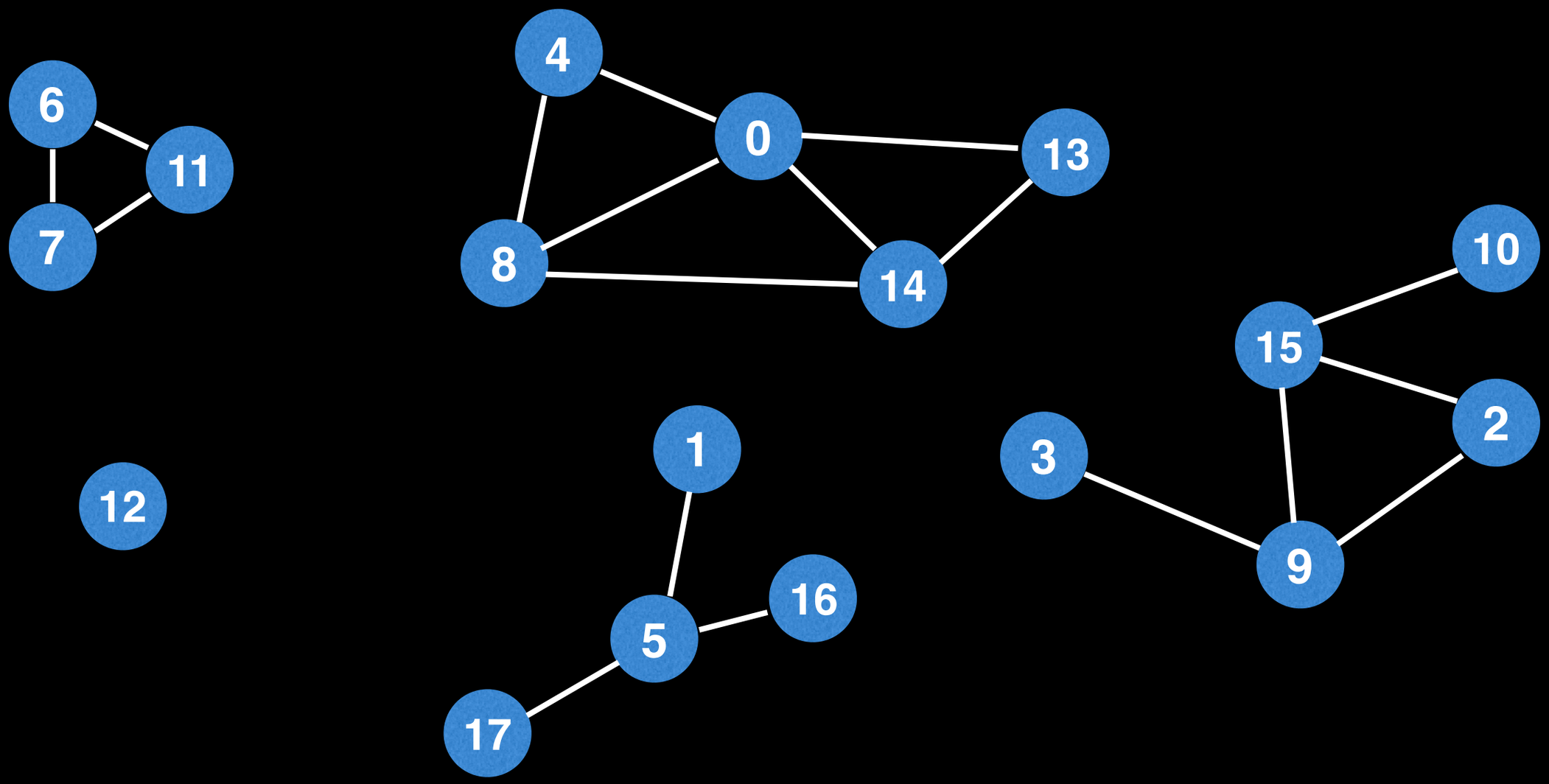
Assign an integer value to each group to be able to tell them apart.



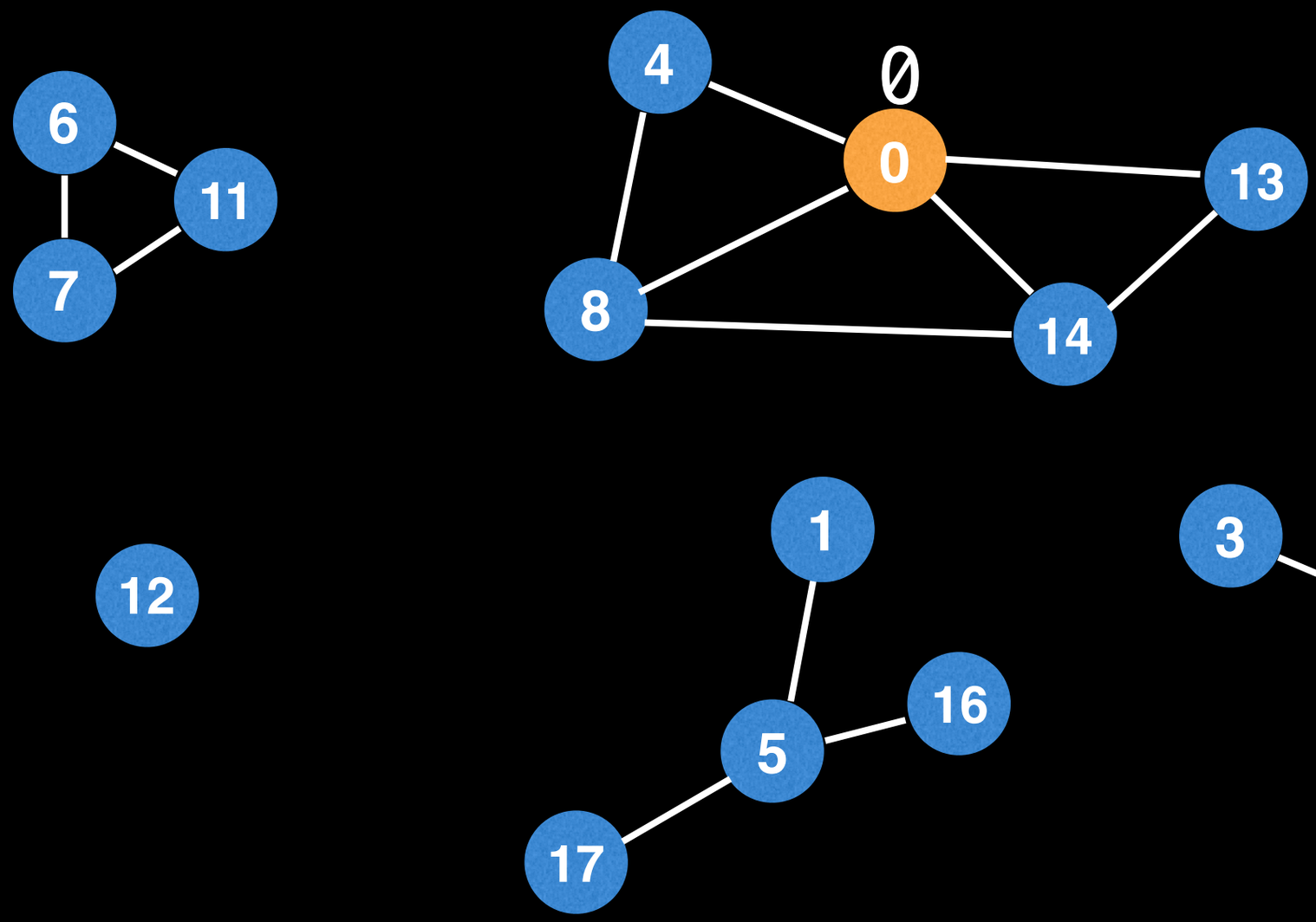
We can use a DFS to identify components. First, make sure all the nodes are labeled from [0, n) where n is the number of nodes.



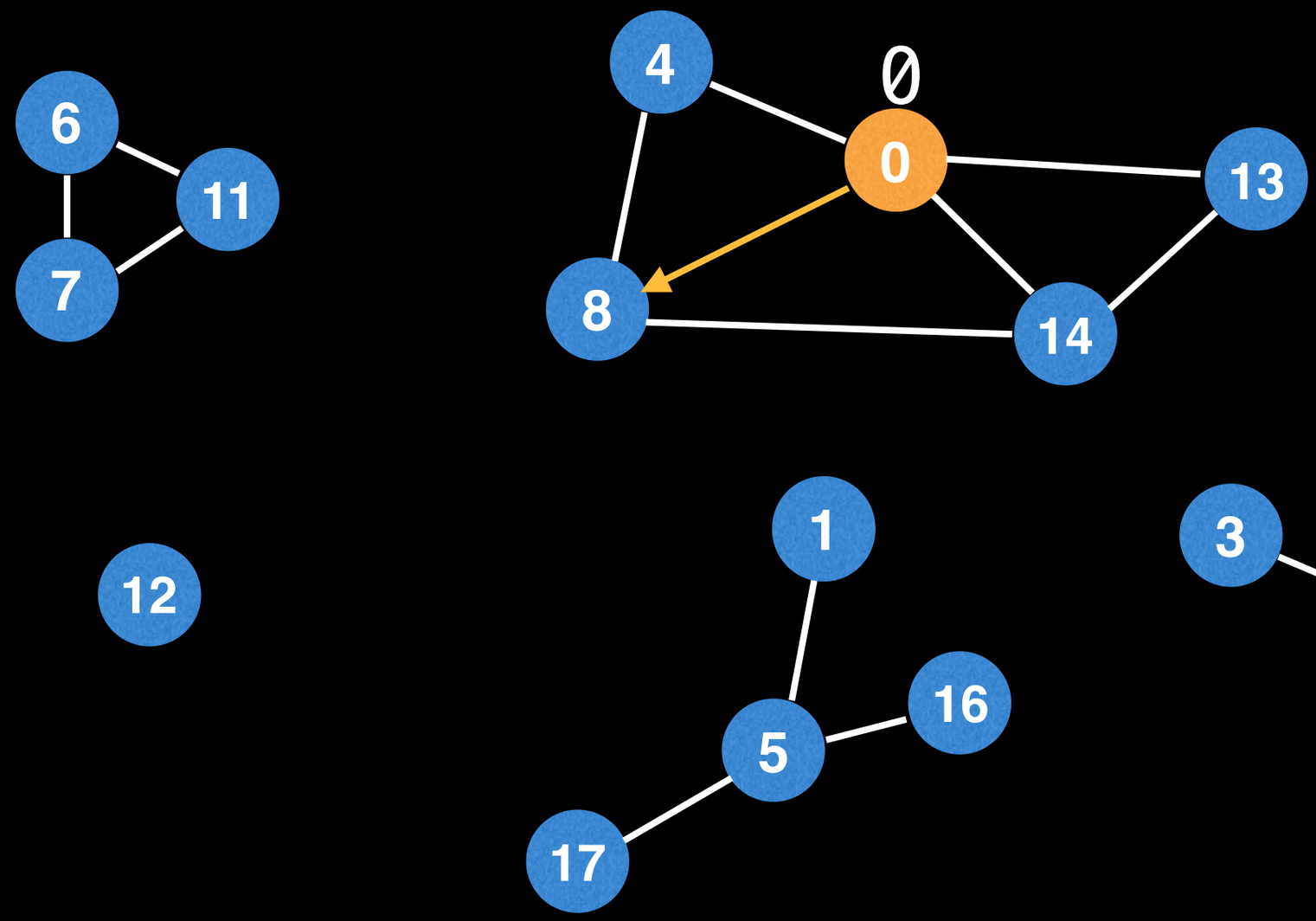
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



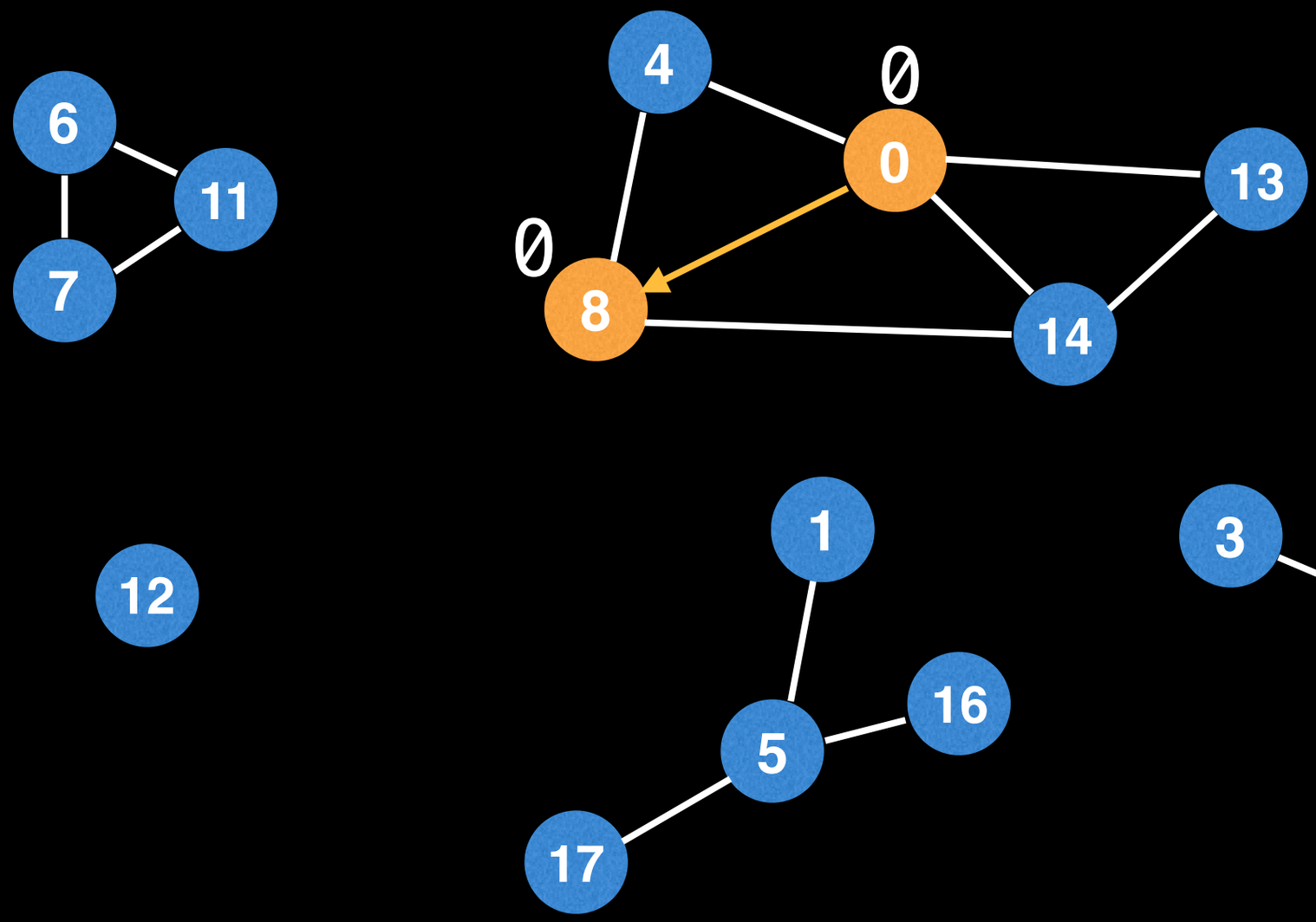
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



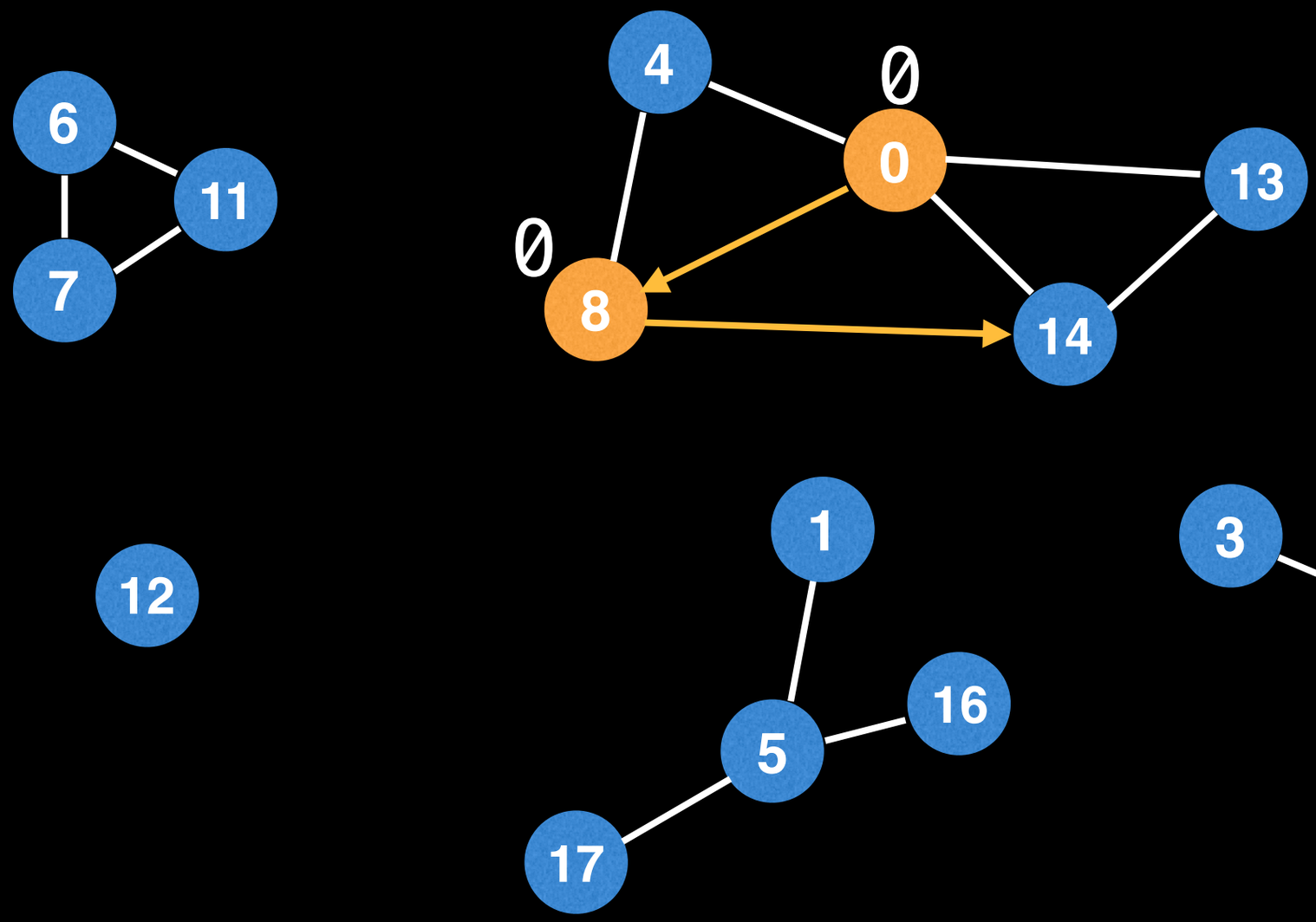
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



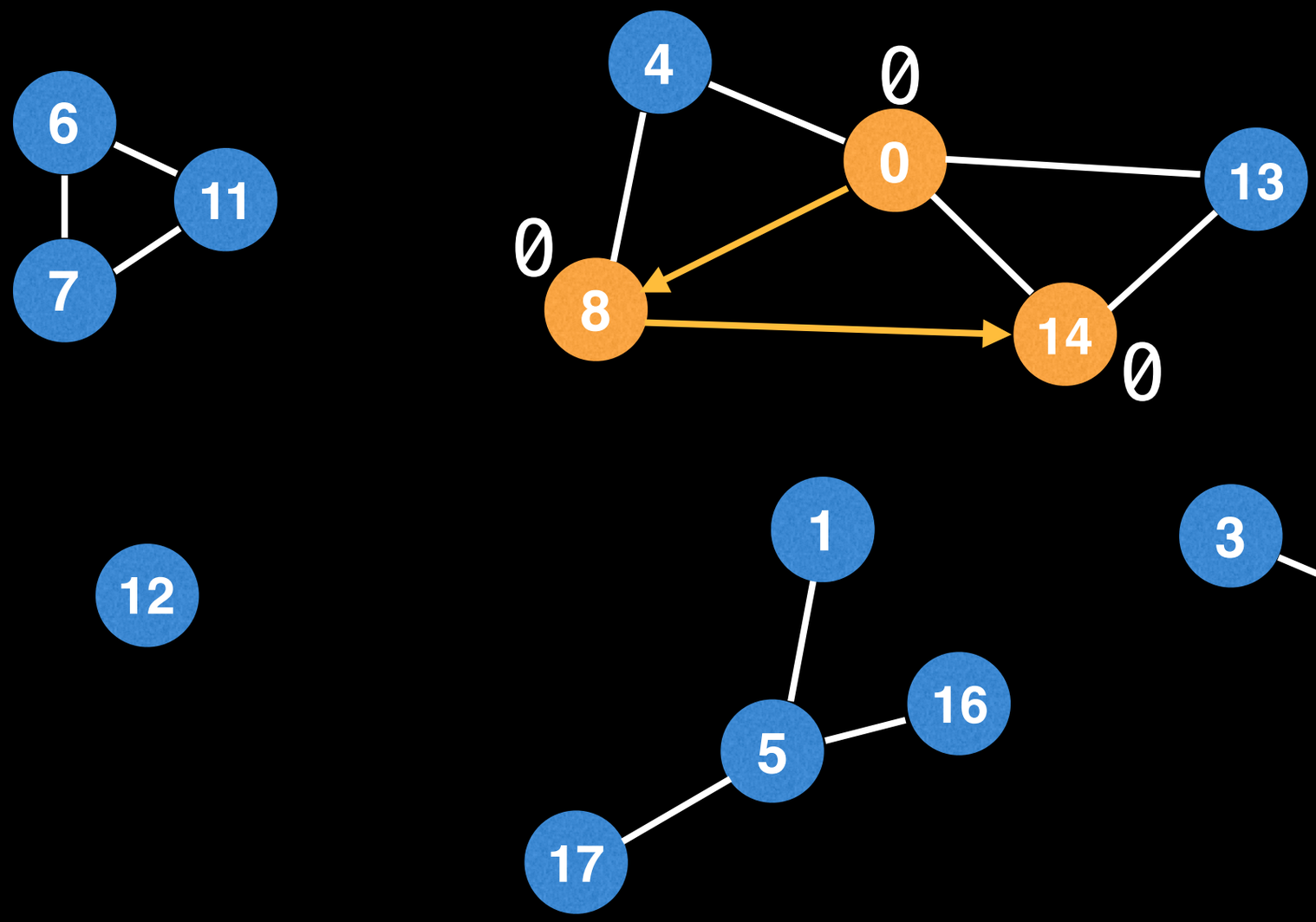
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



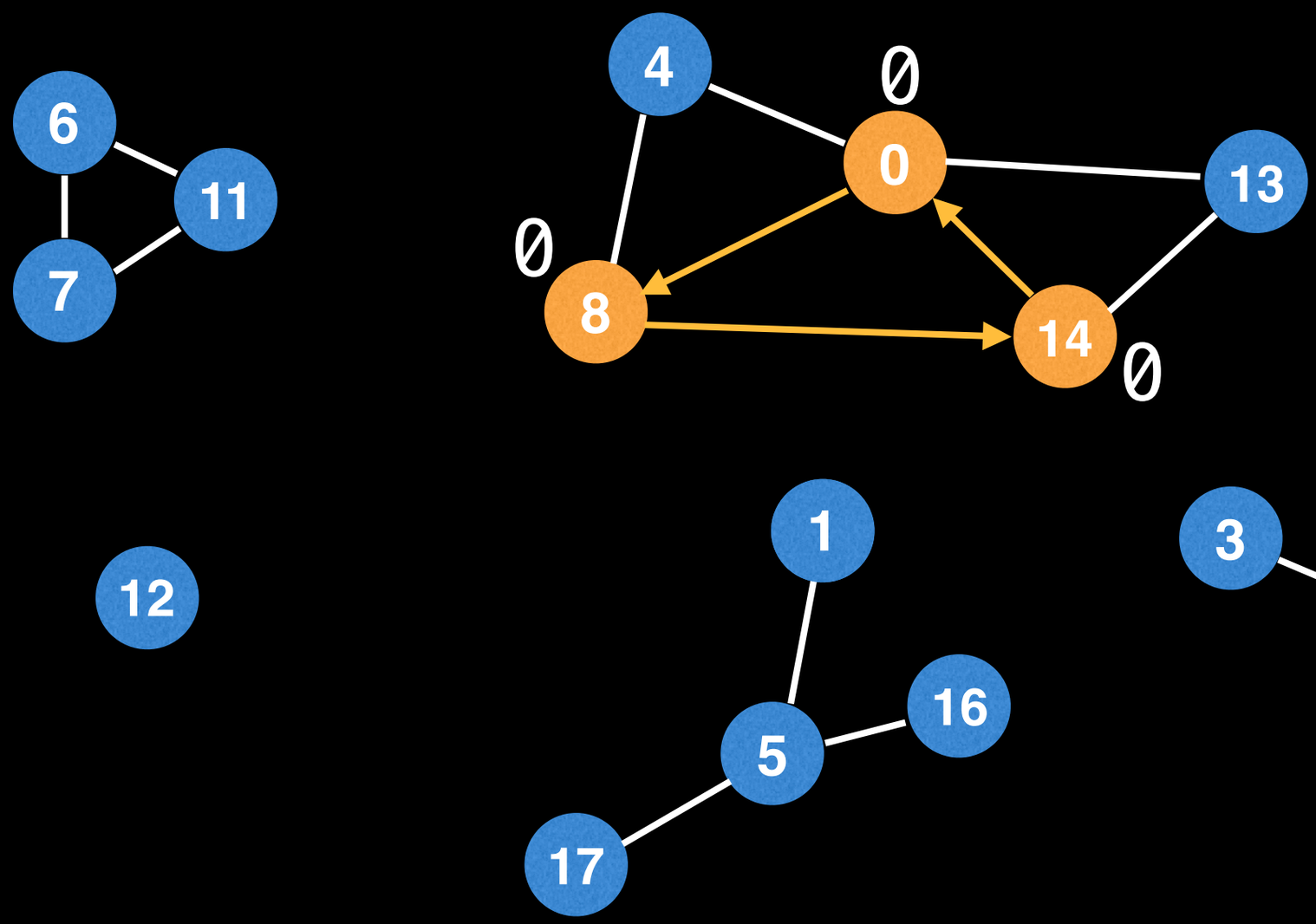
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



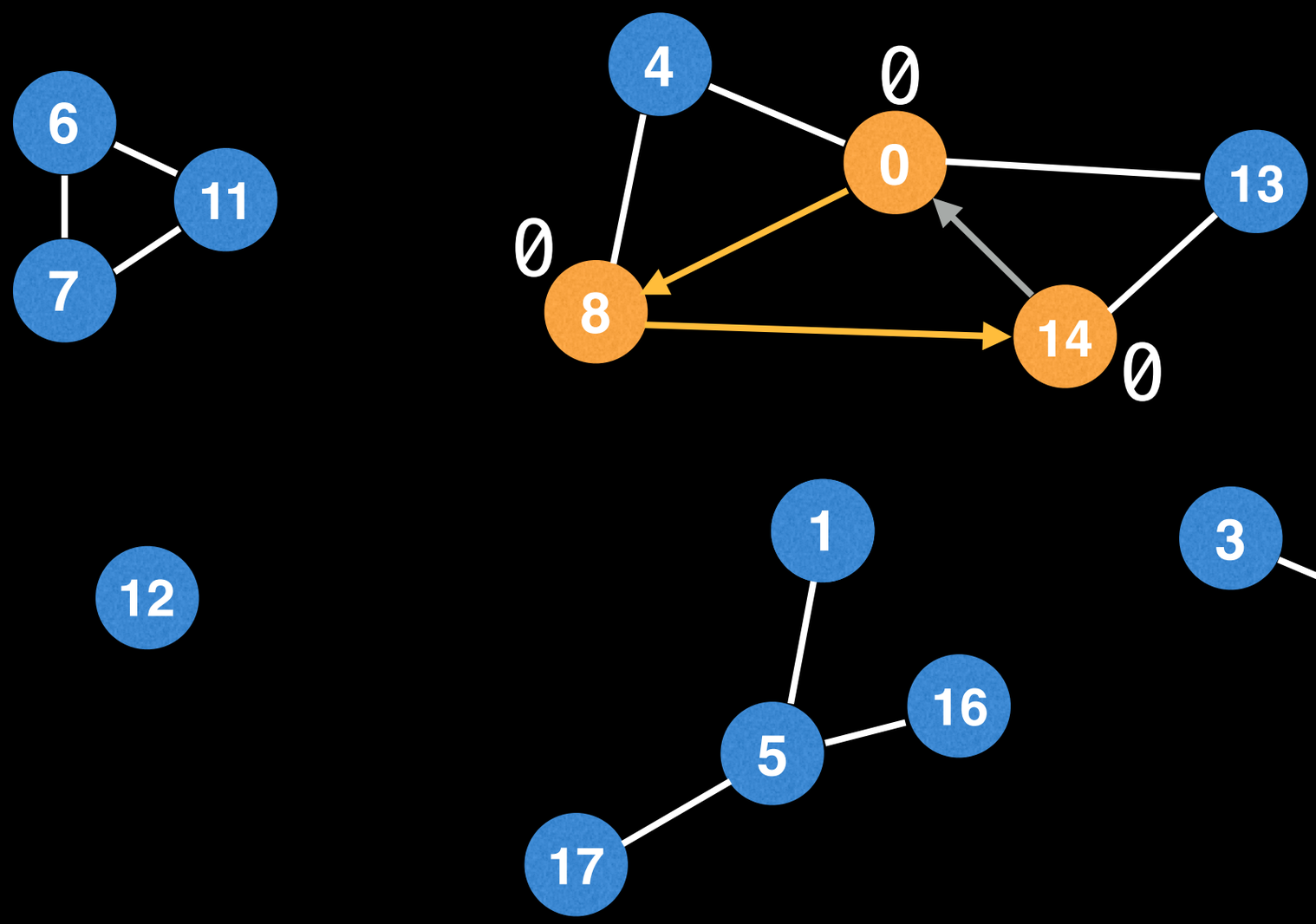
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



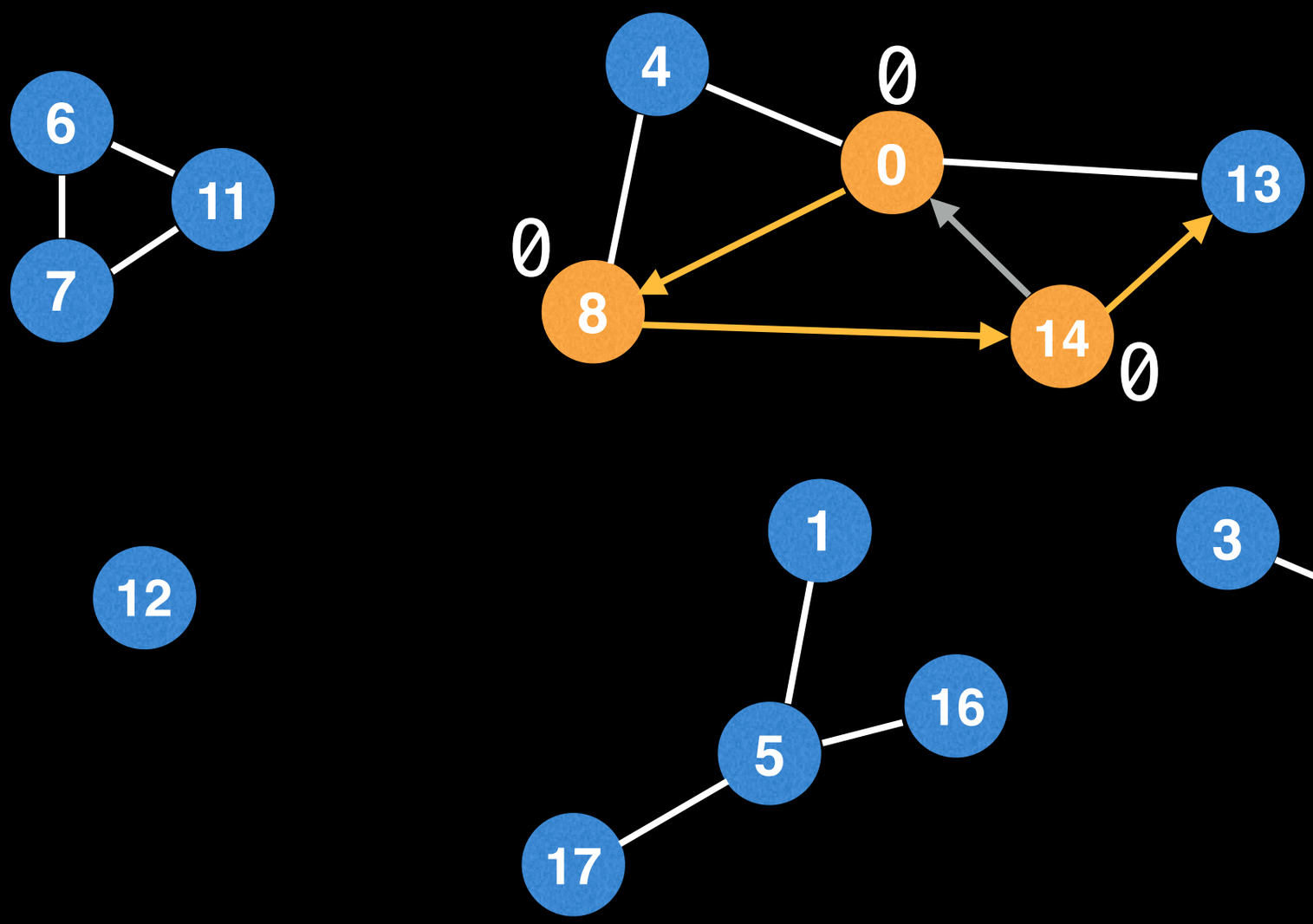
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



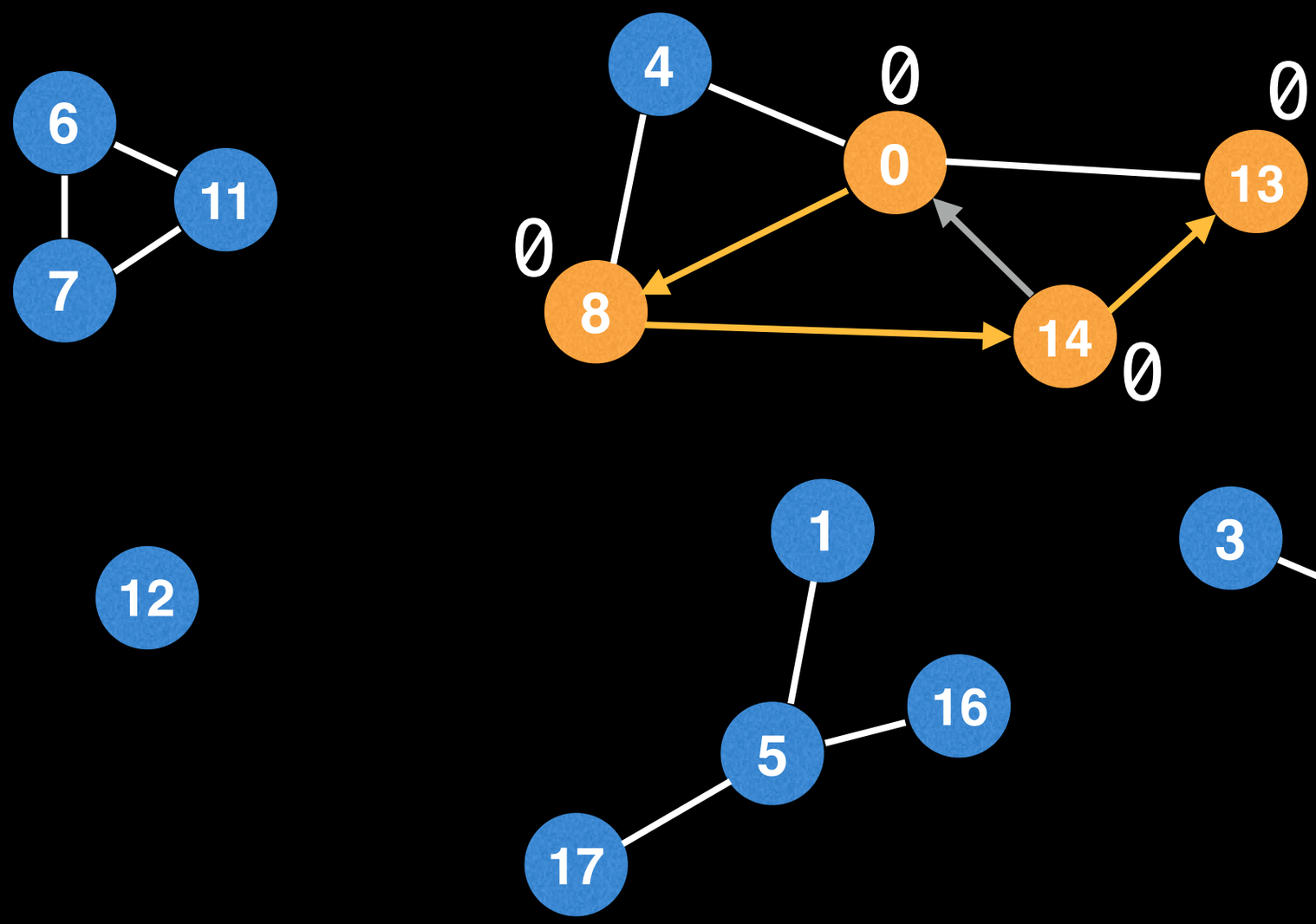
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



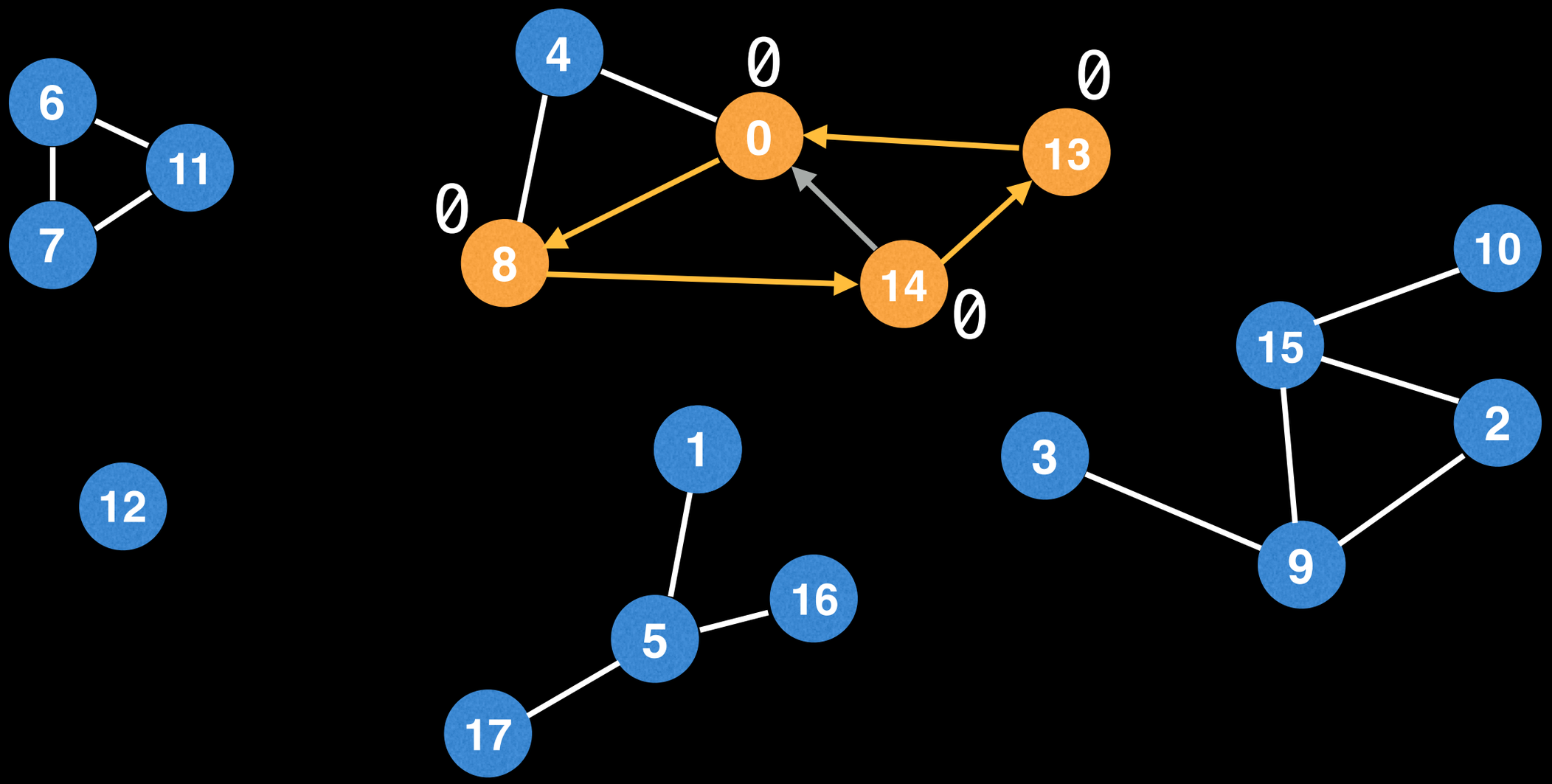
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



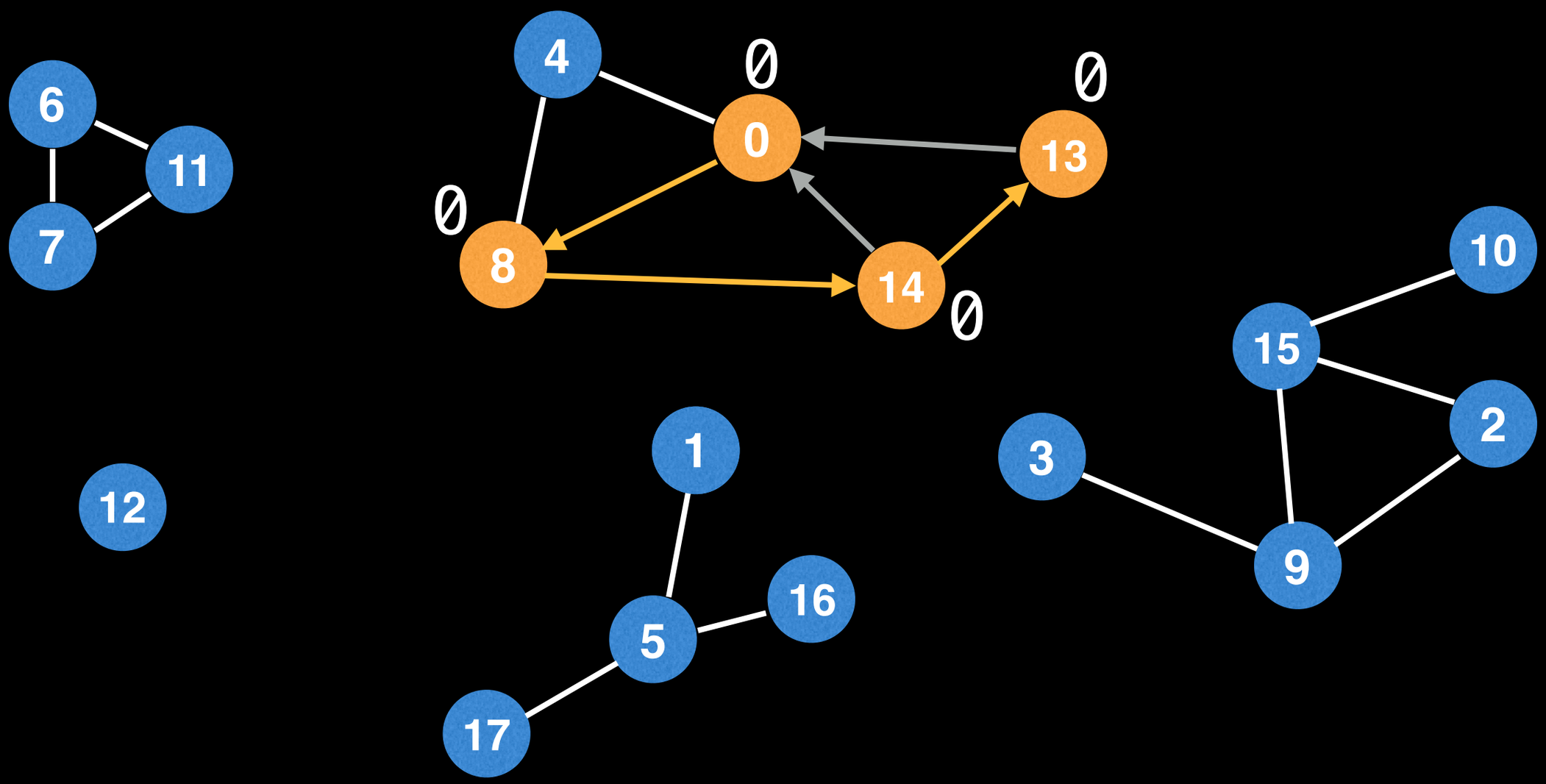
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



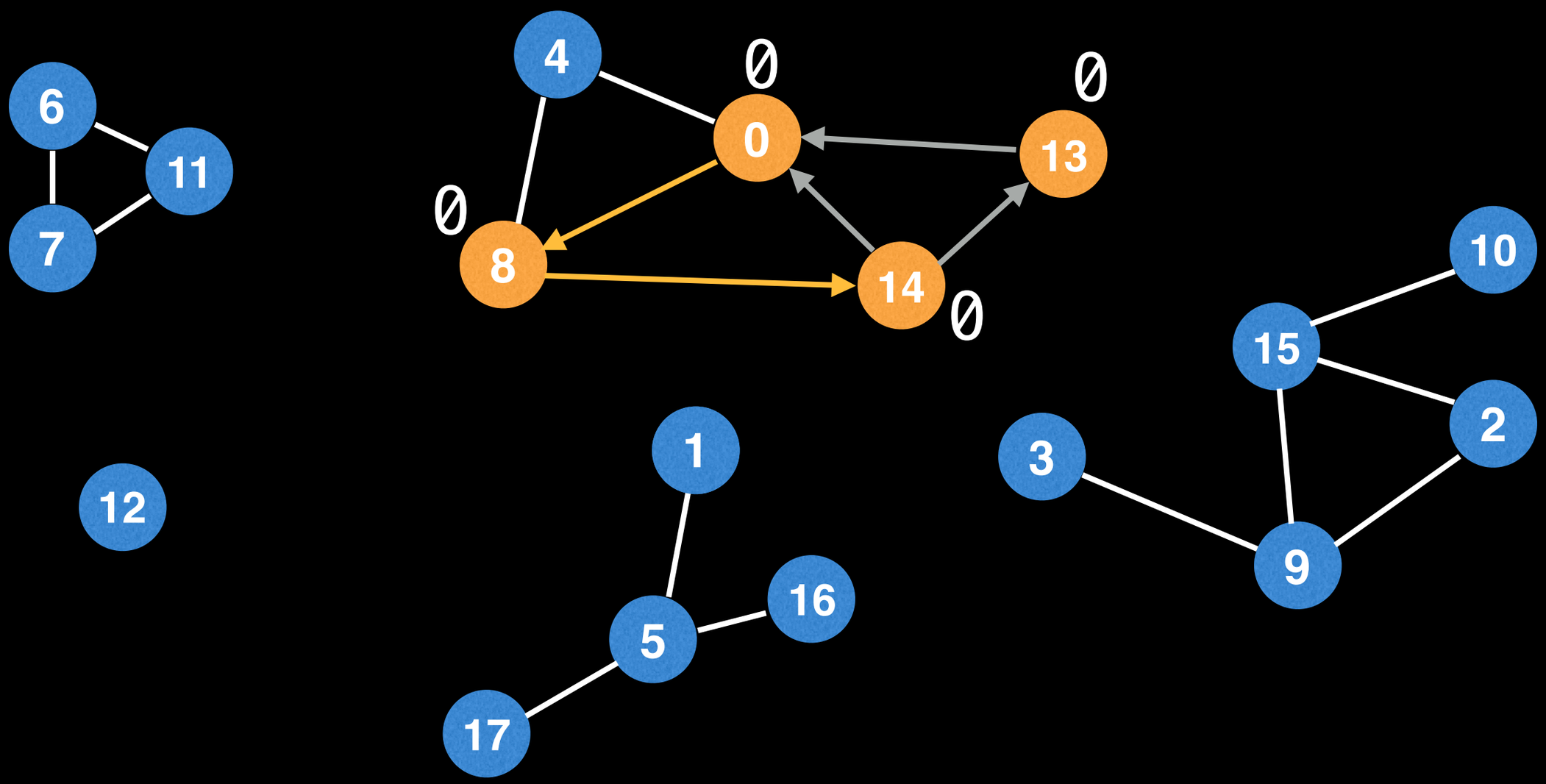
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



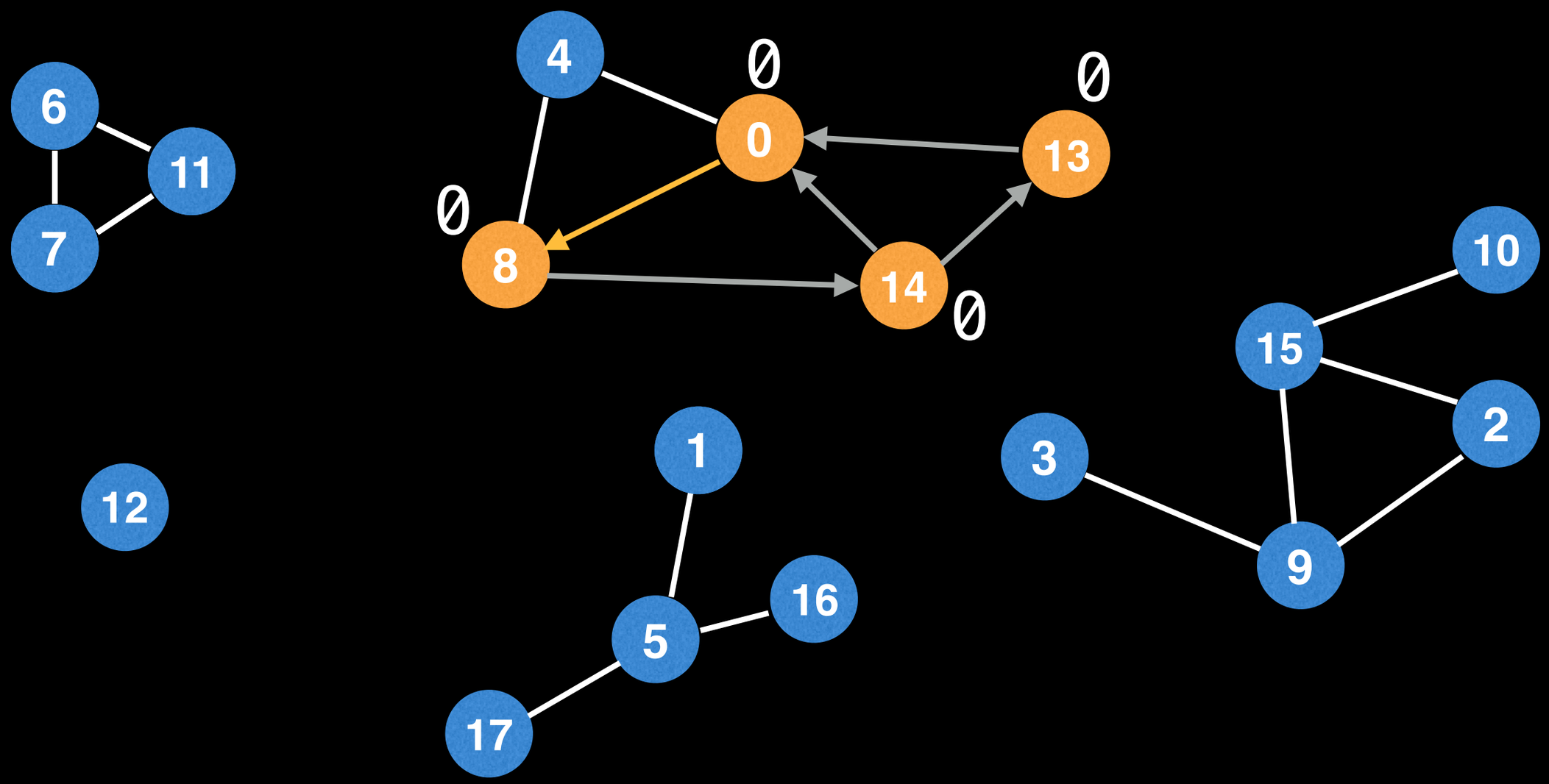
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



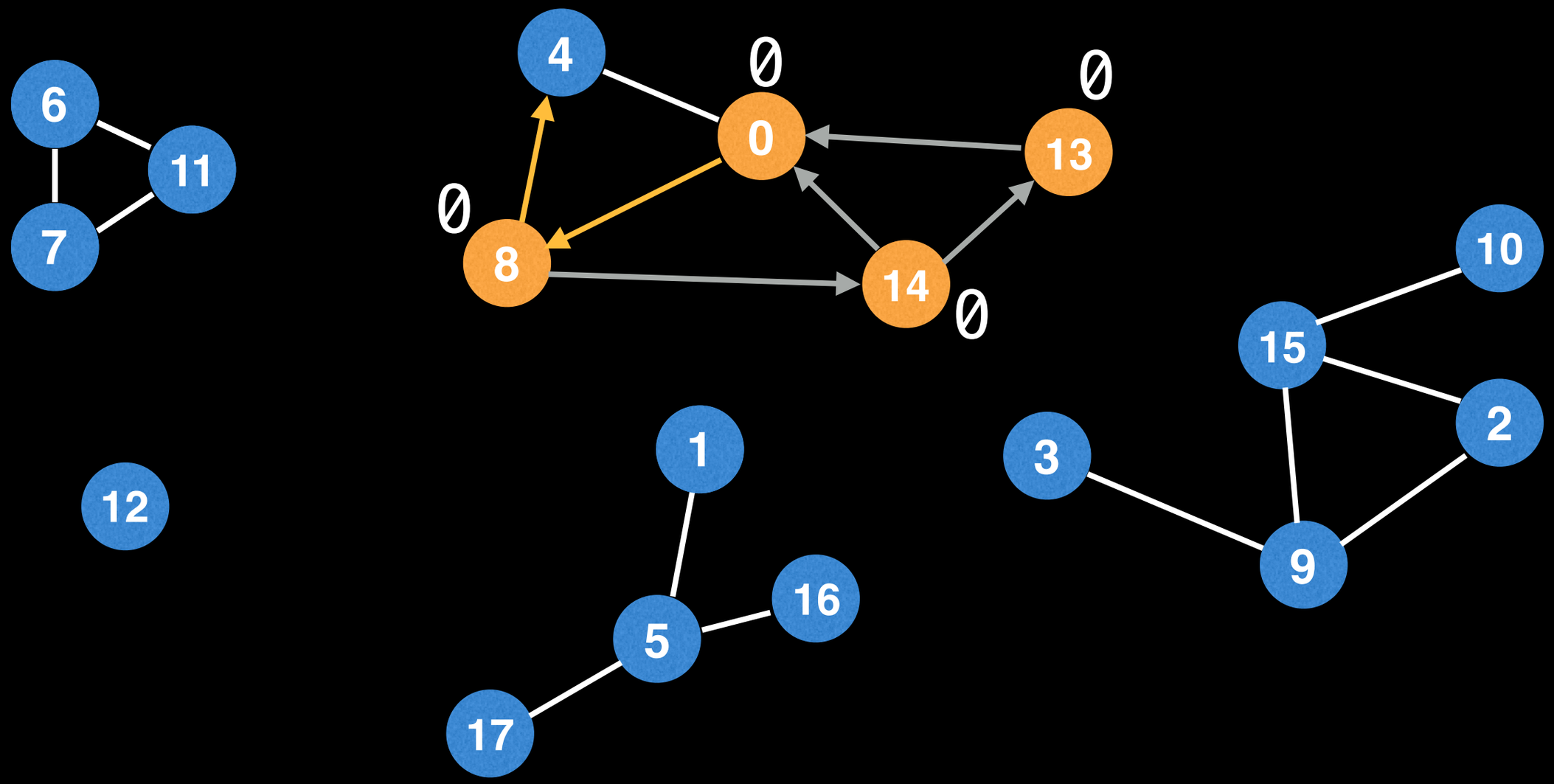
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



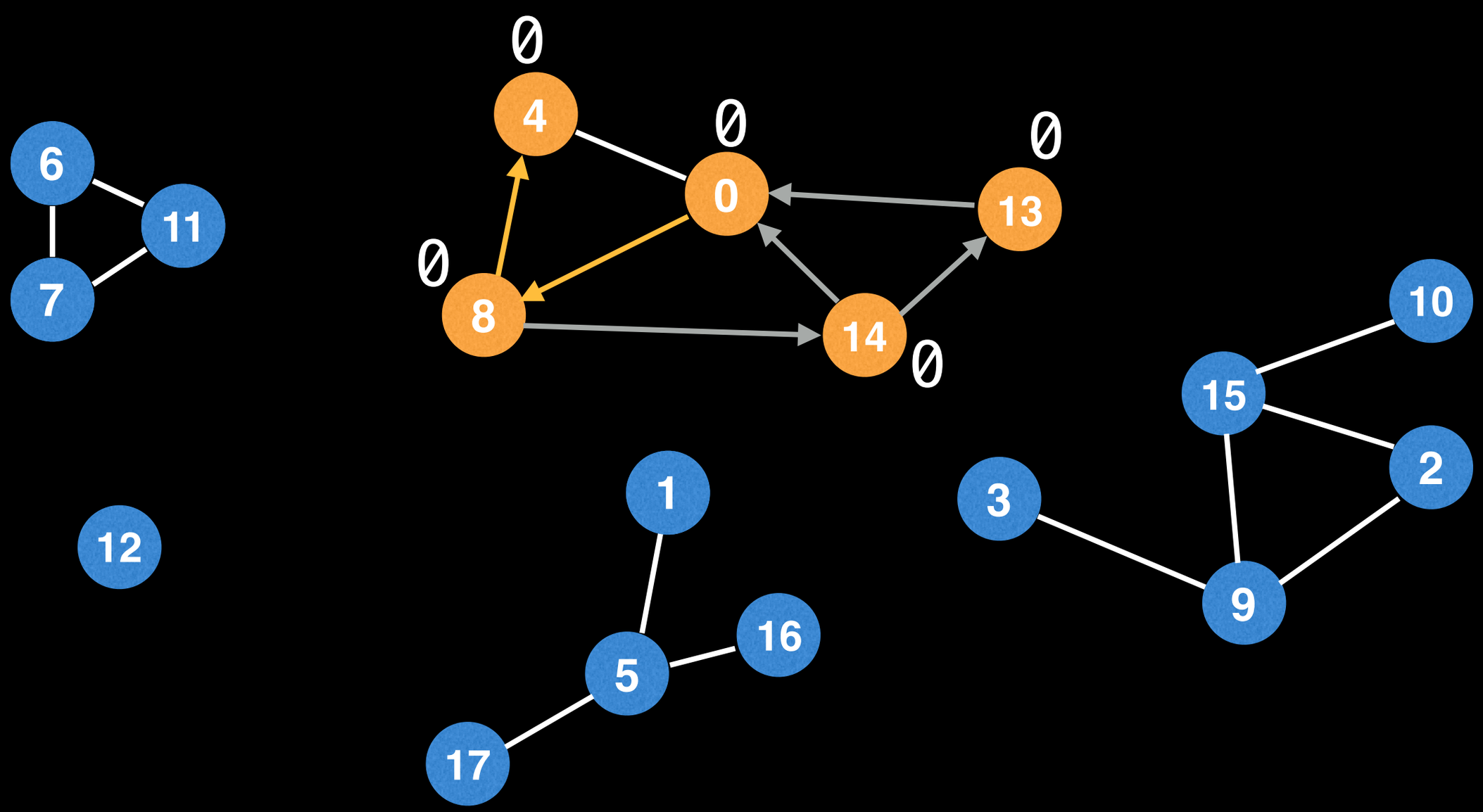
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



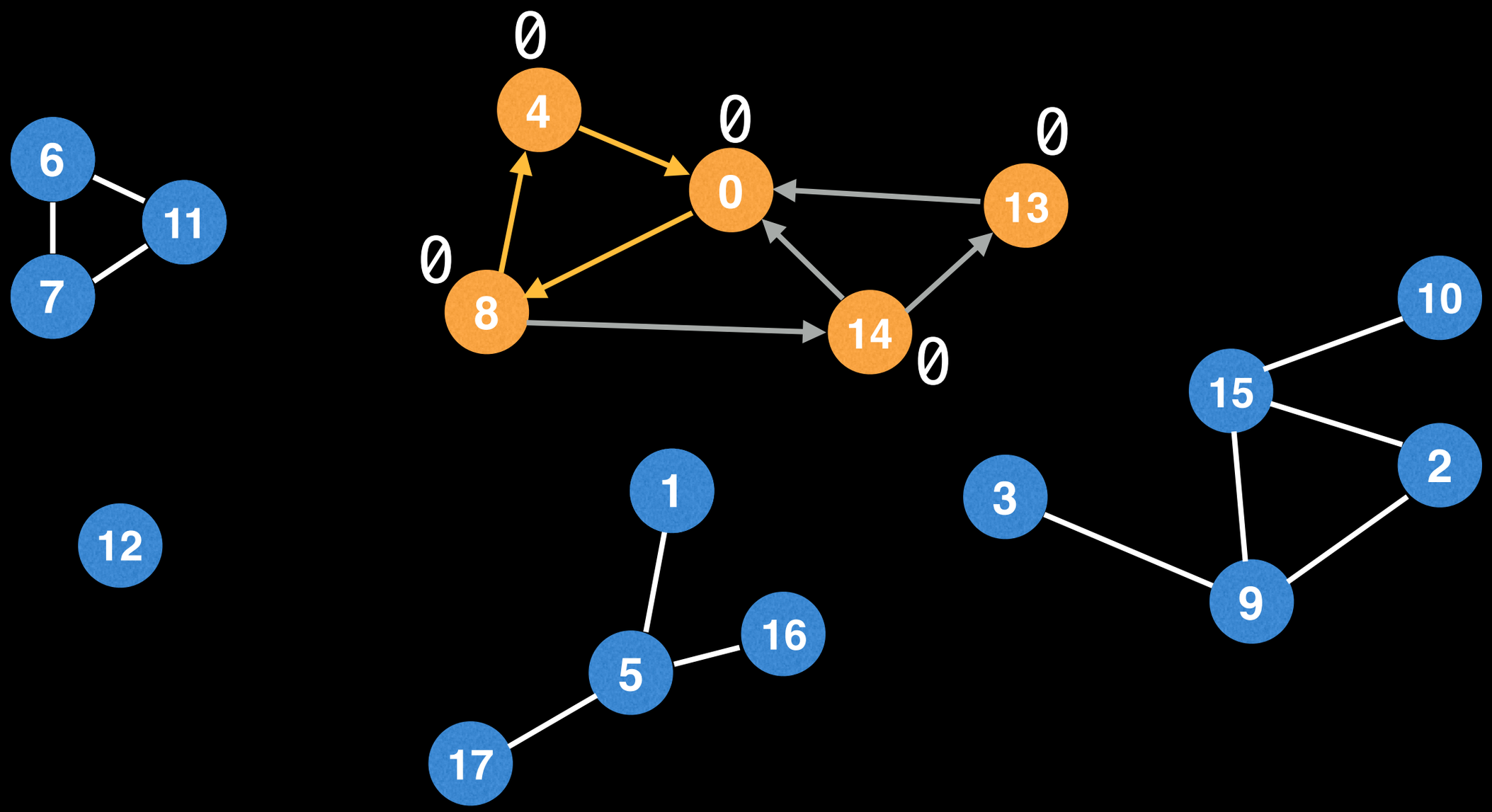
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



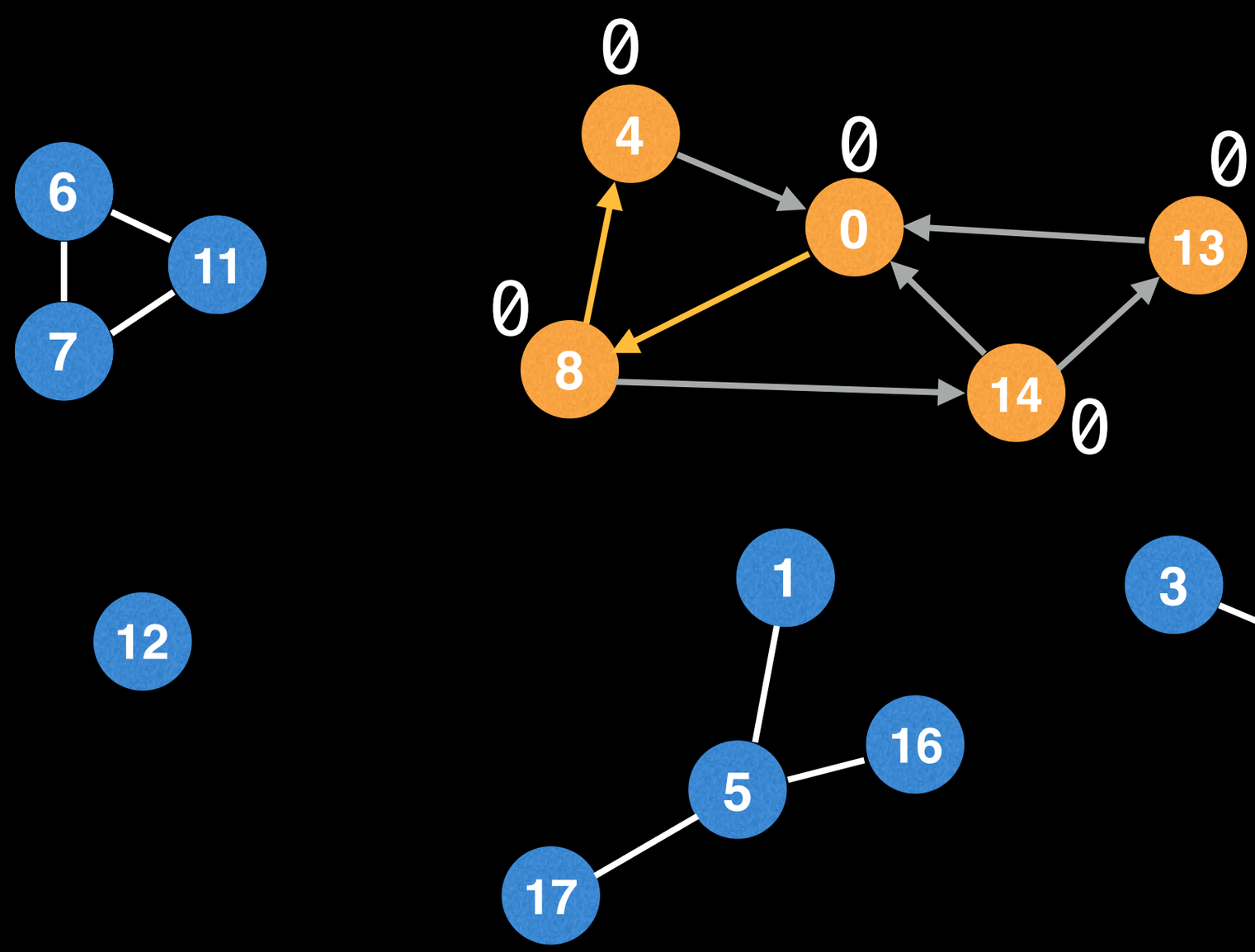
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



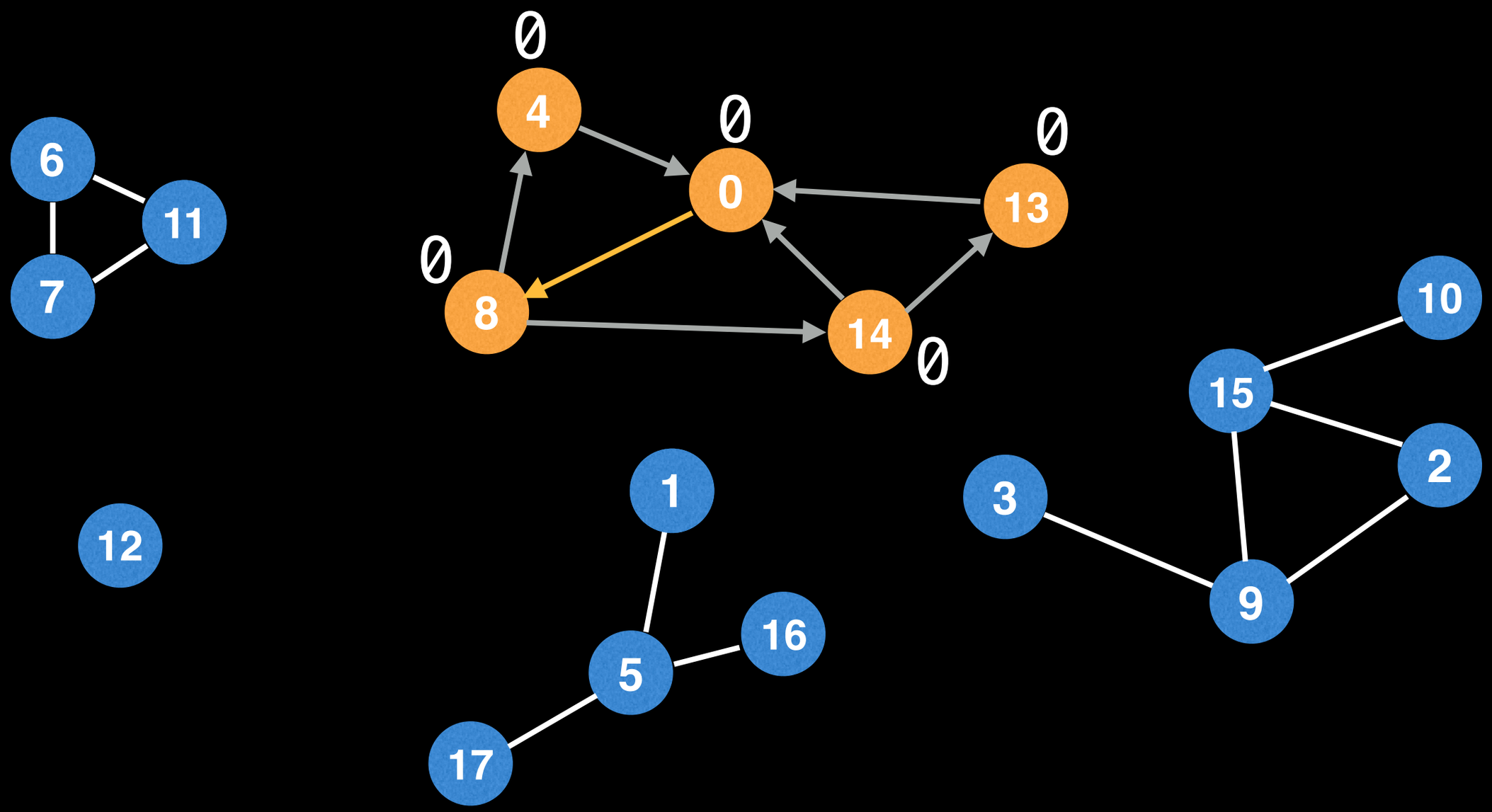
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



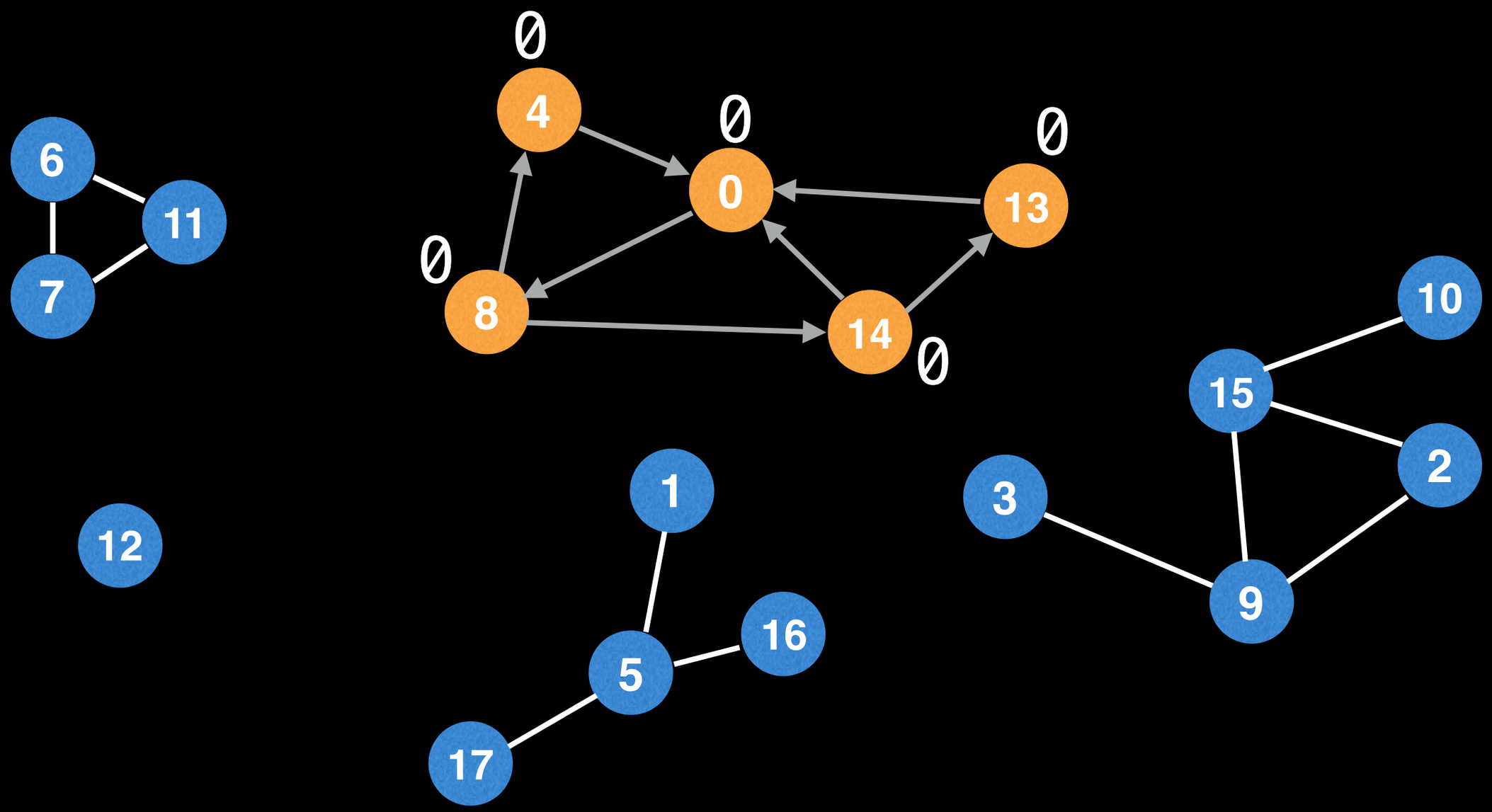
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



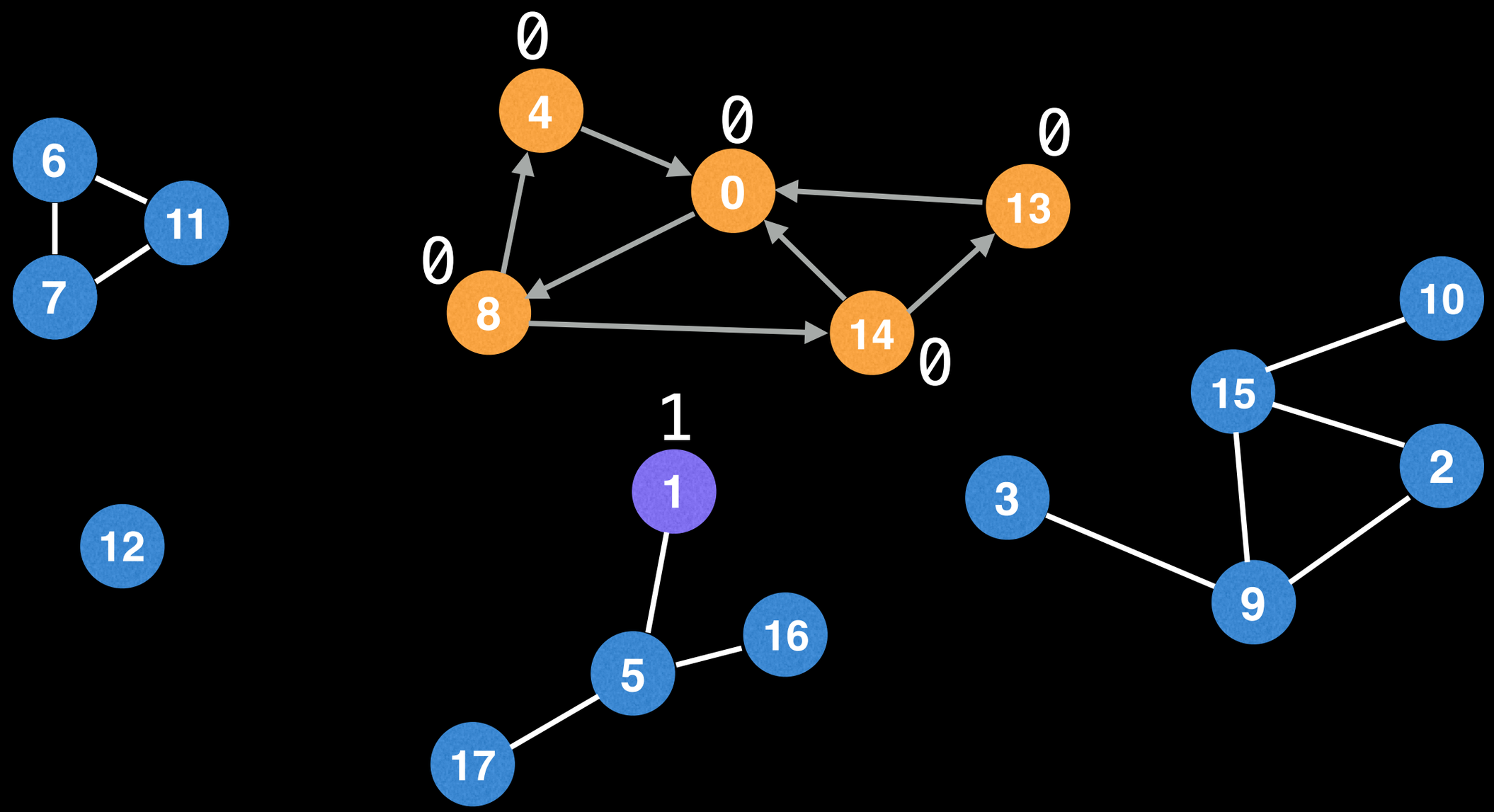
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



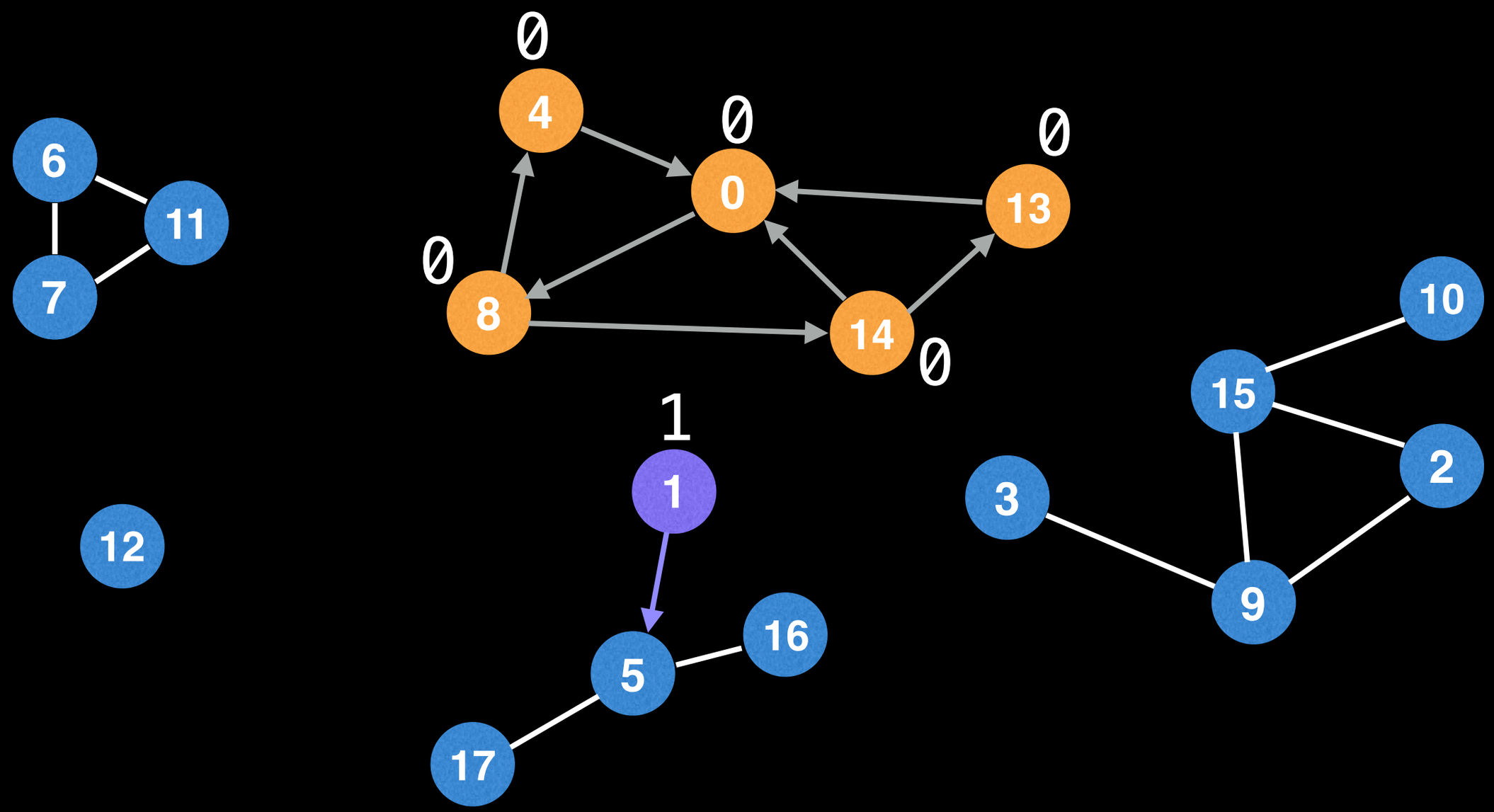
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



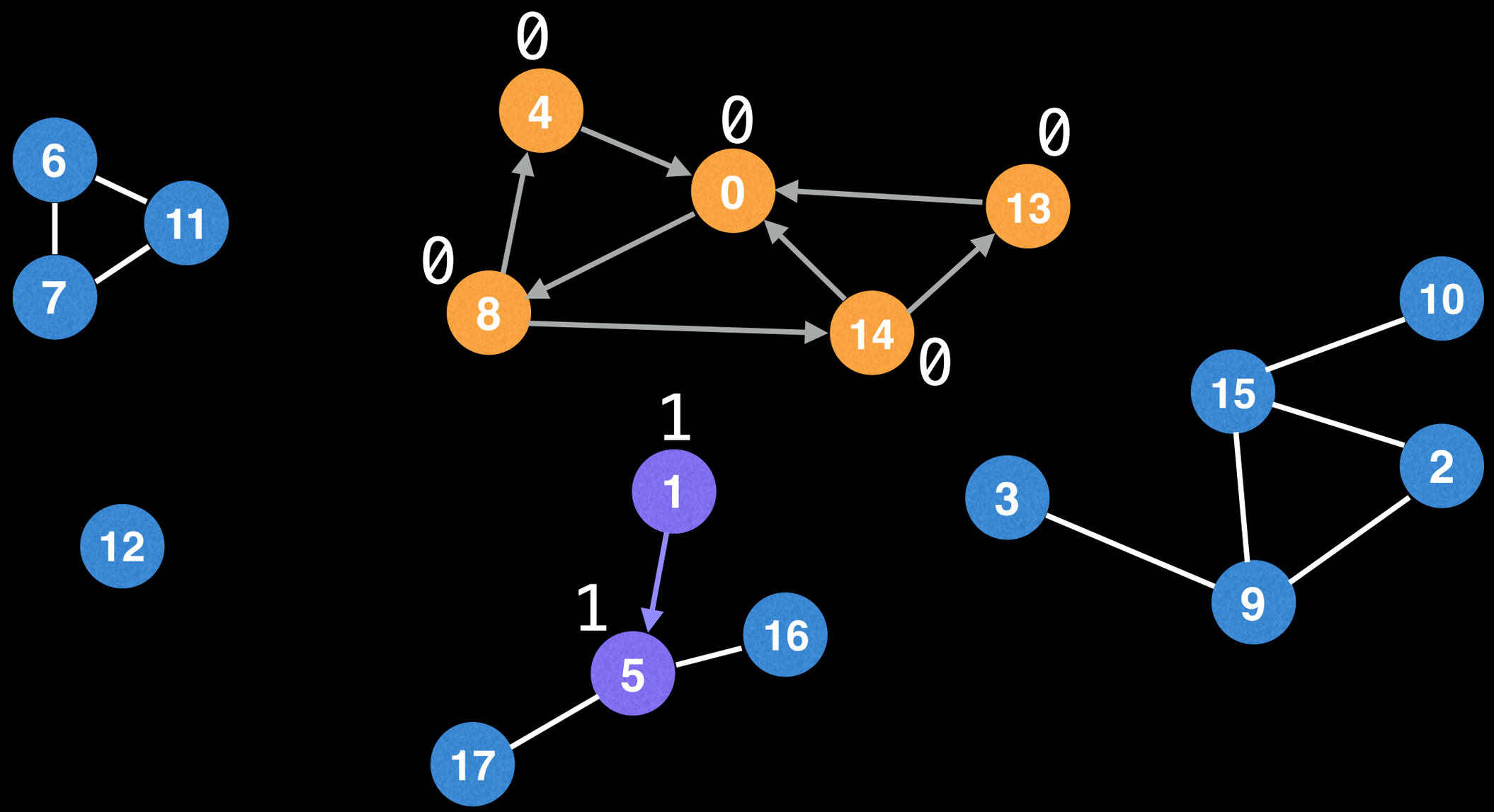
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



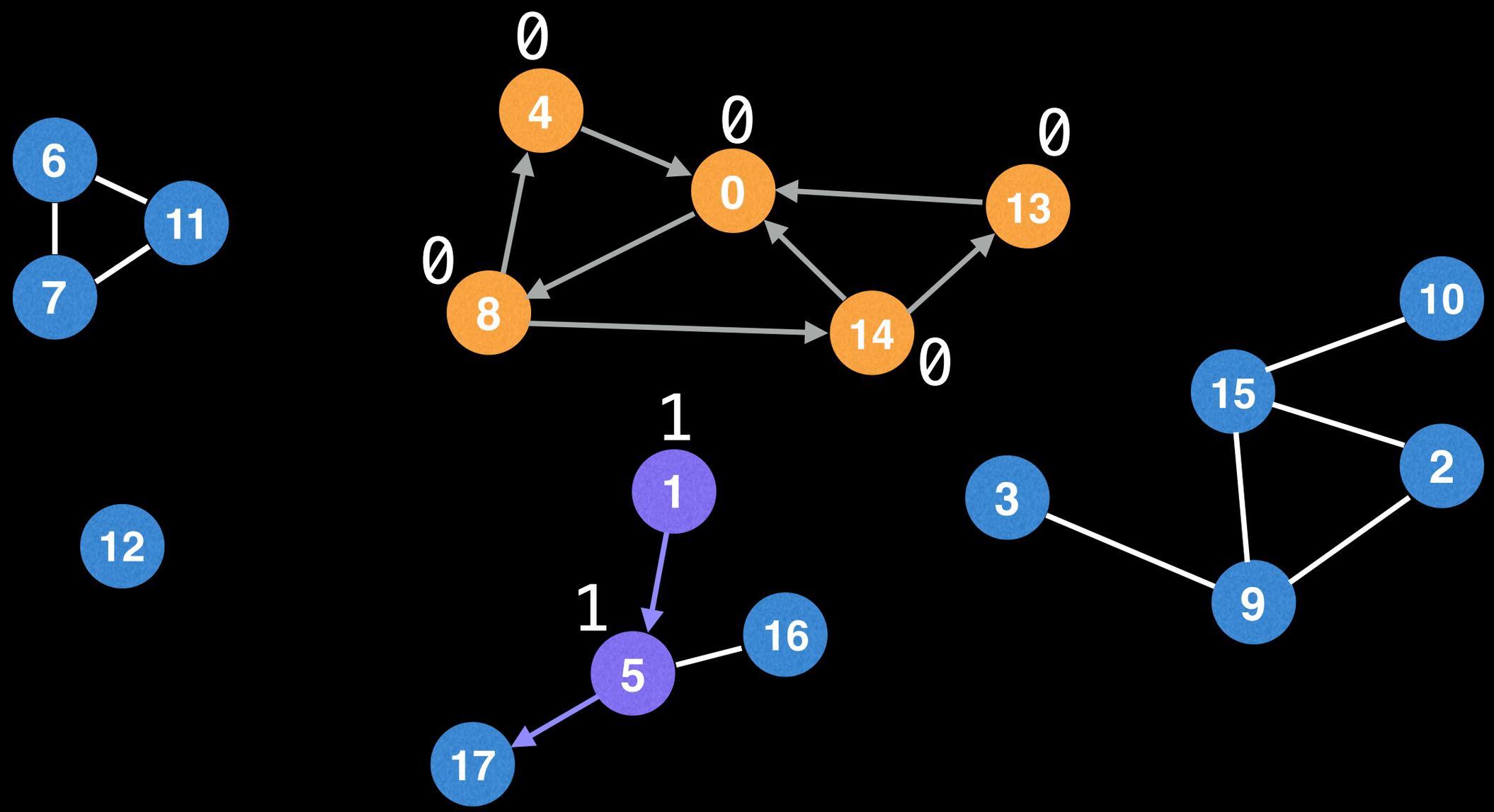
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



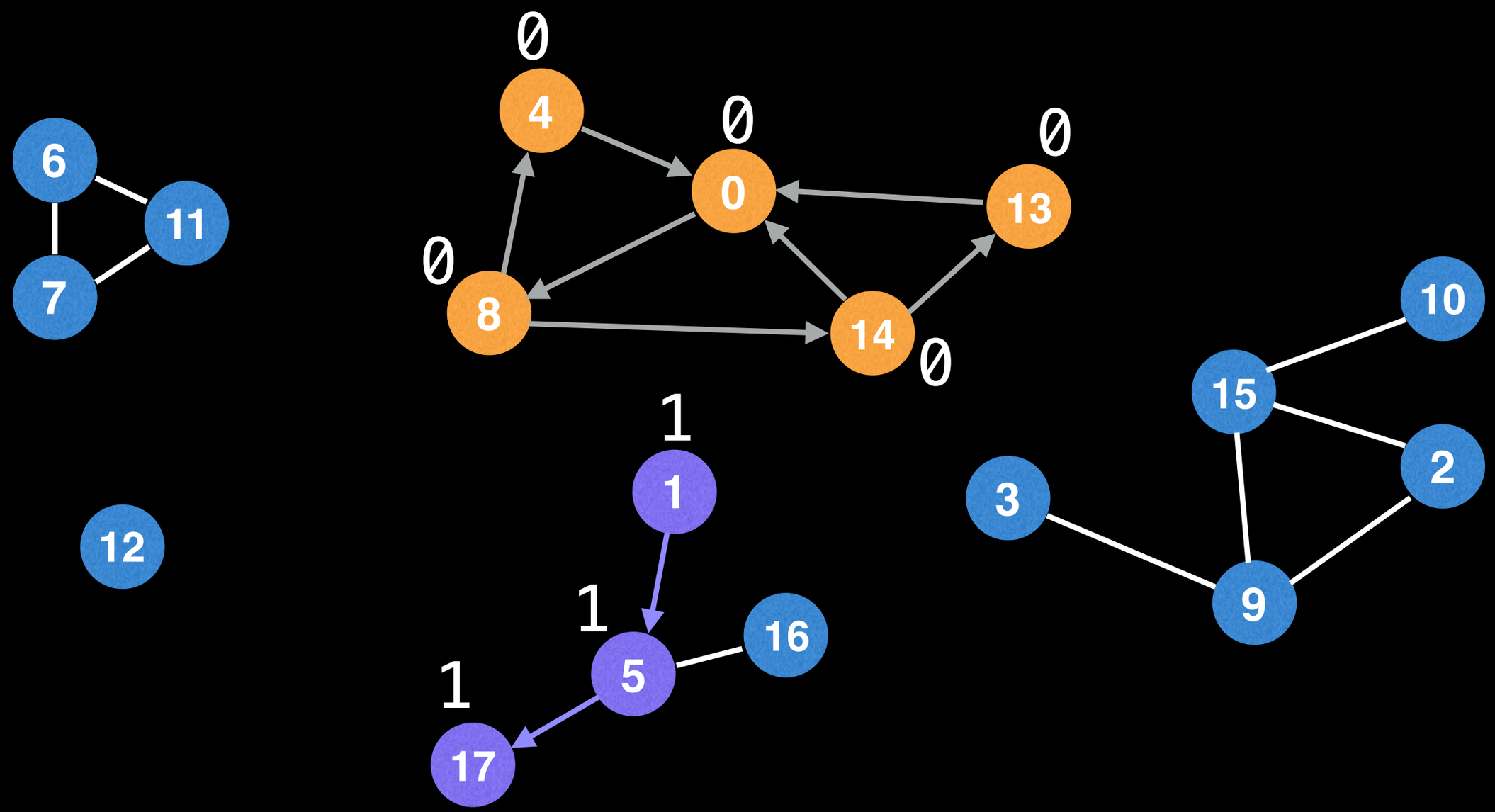
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



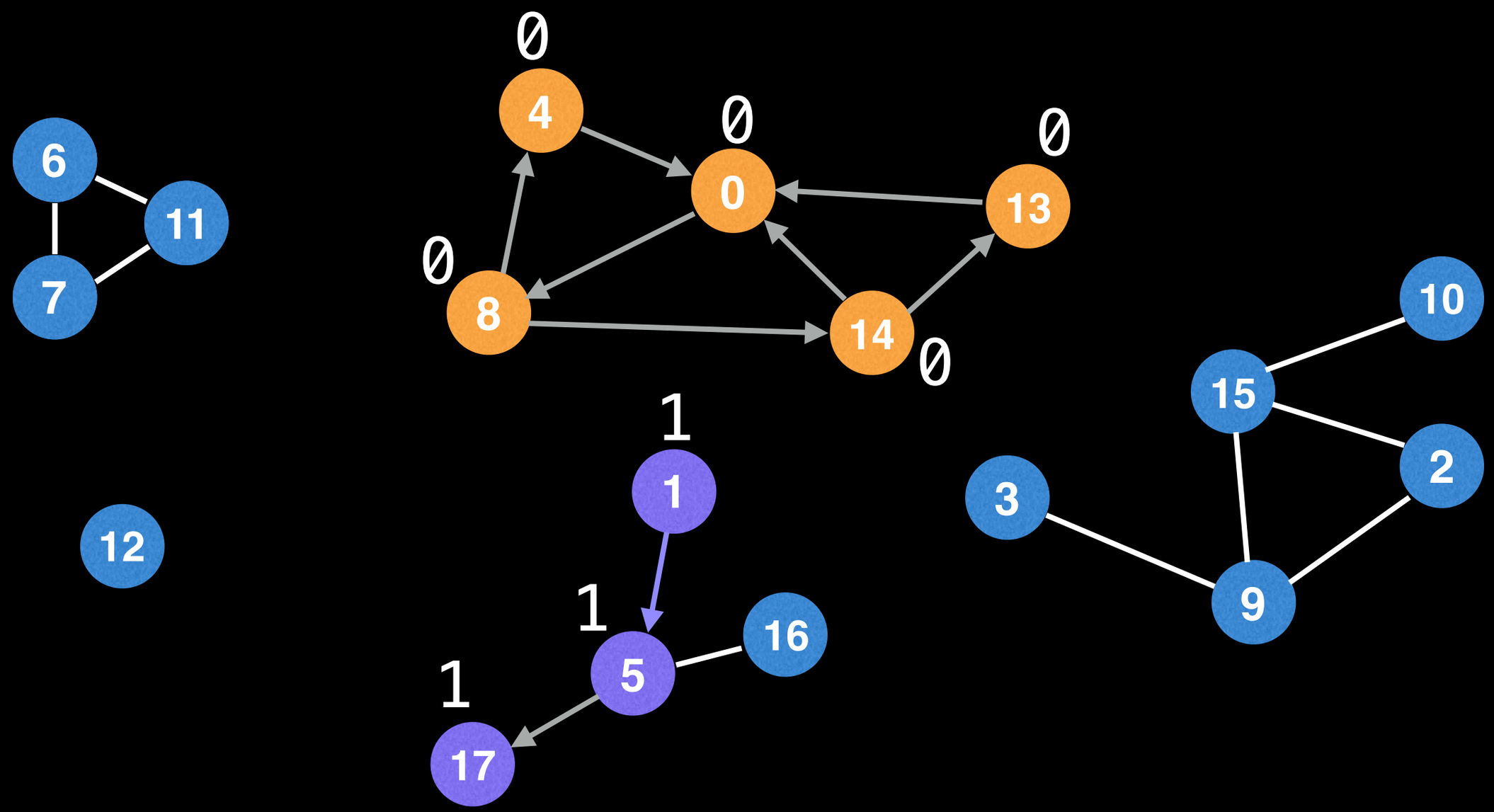
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



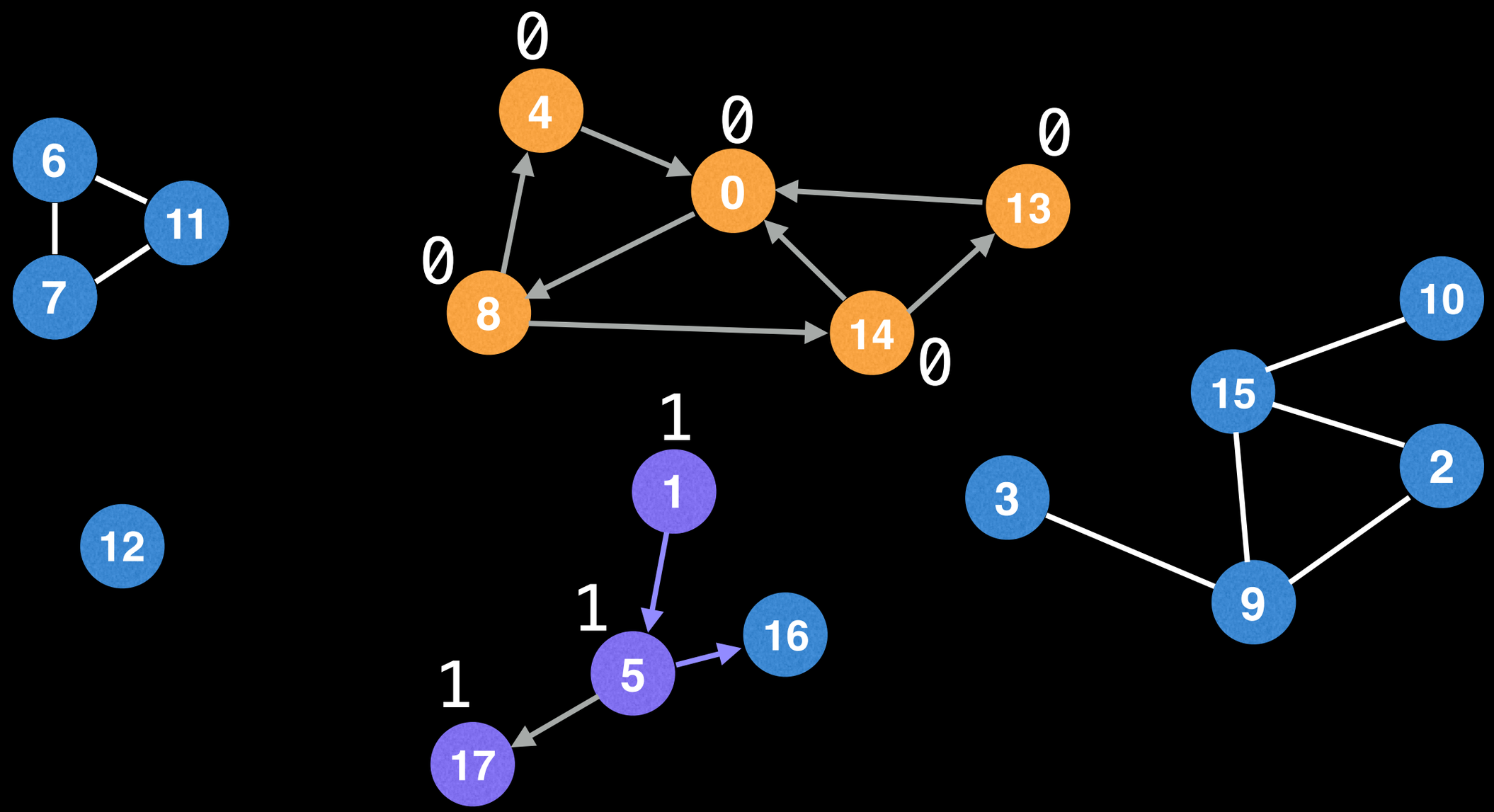
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



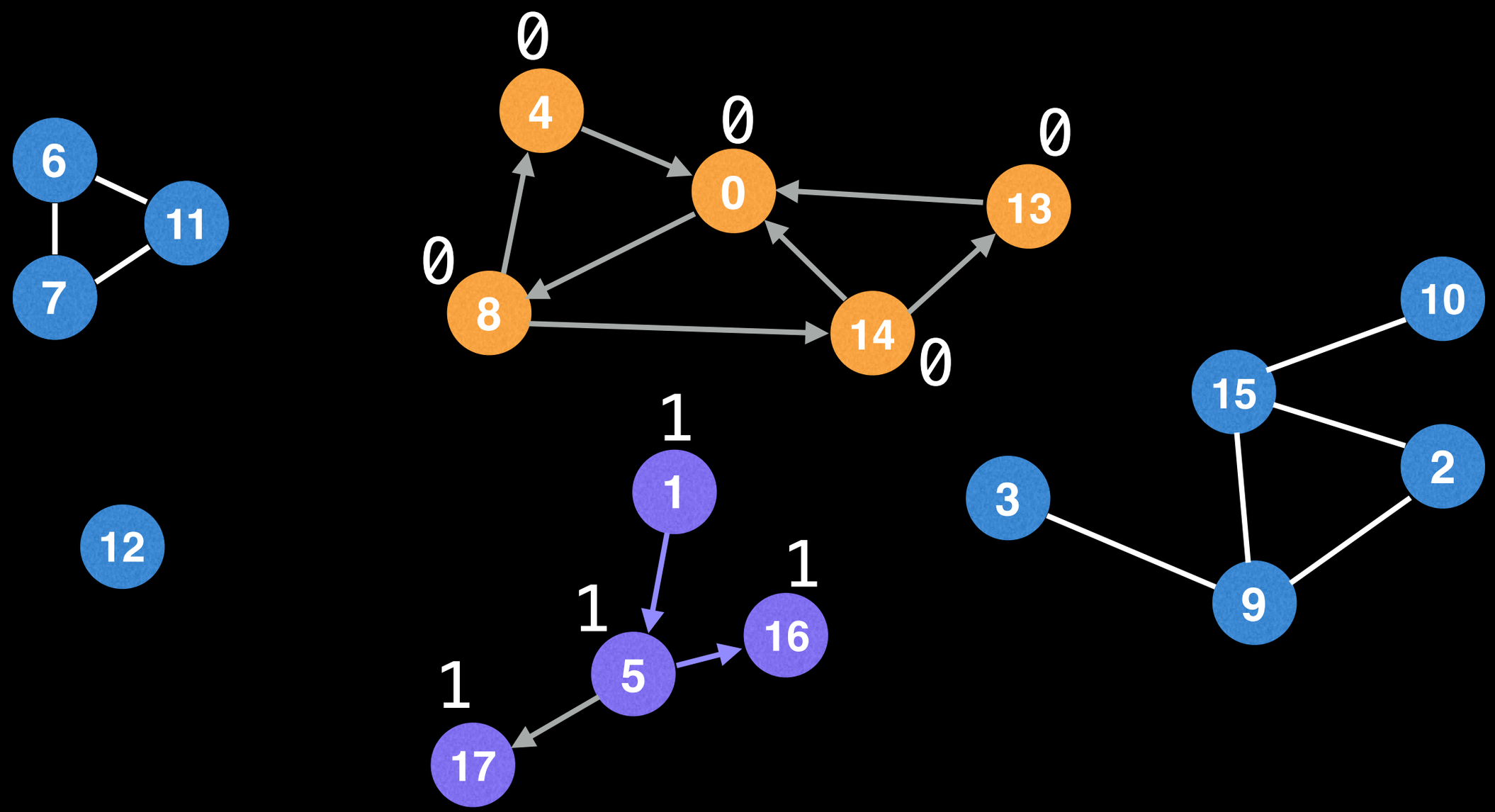
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



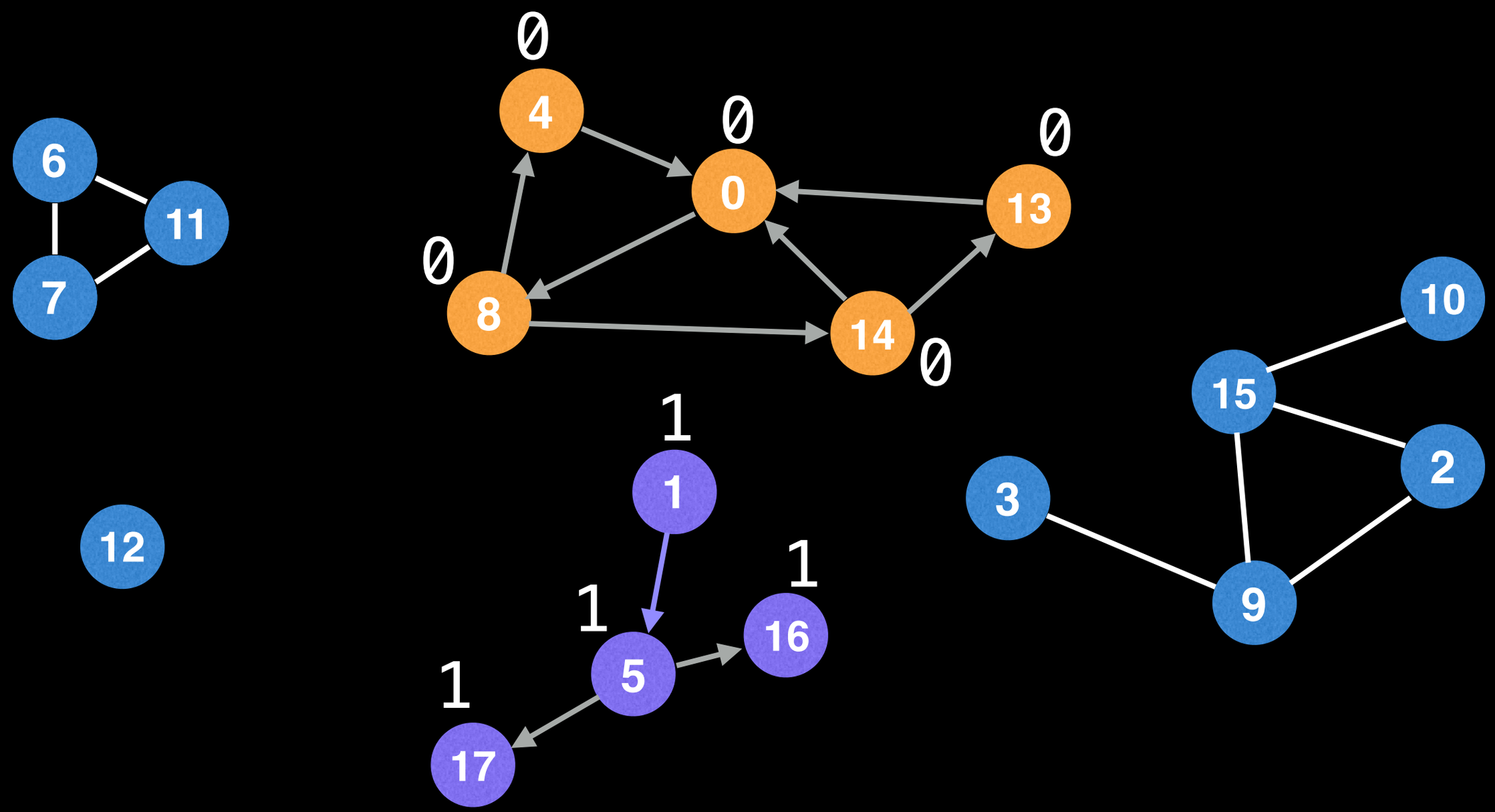
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



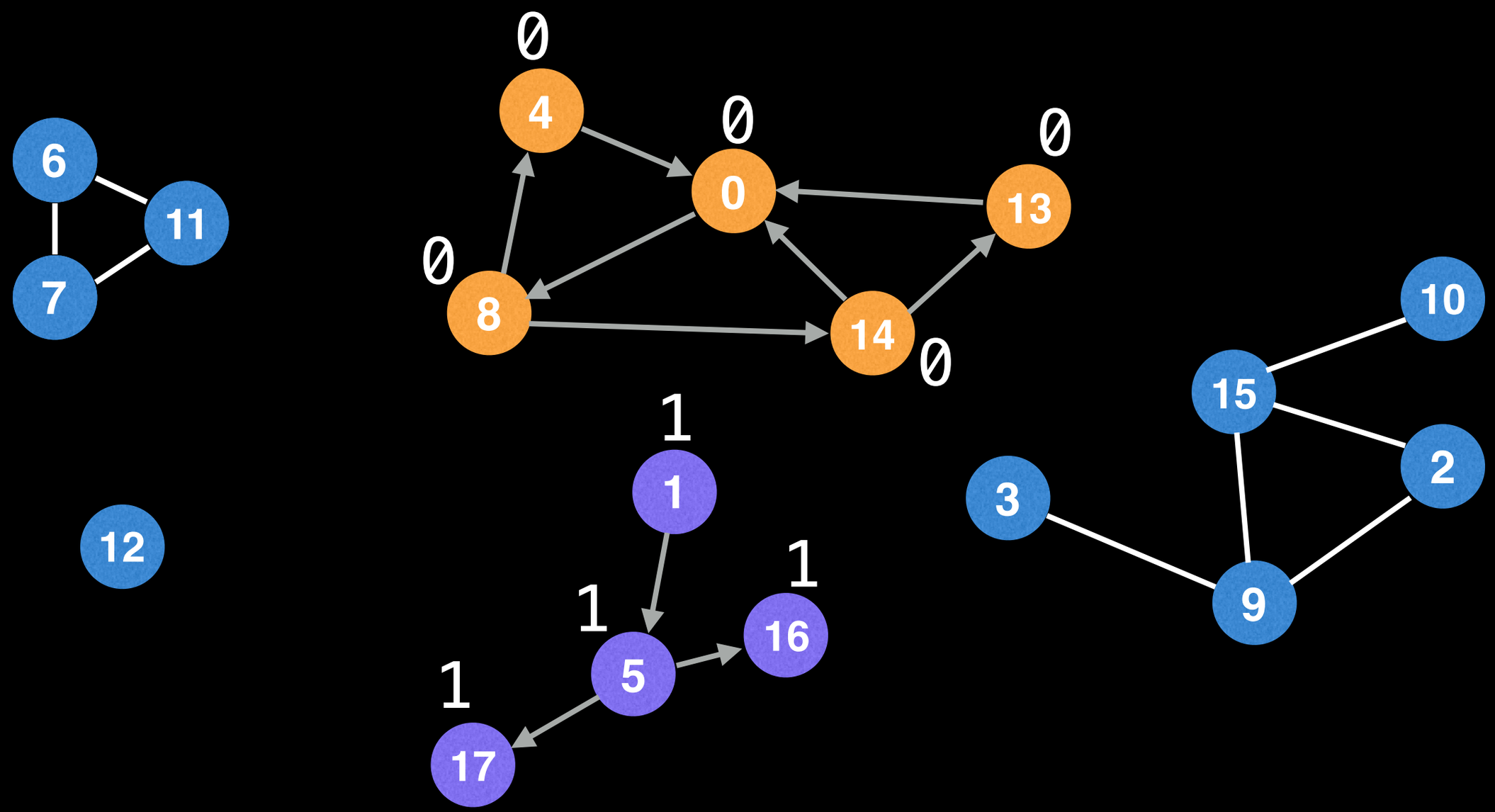
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



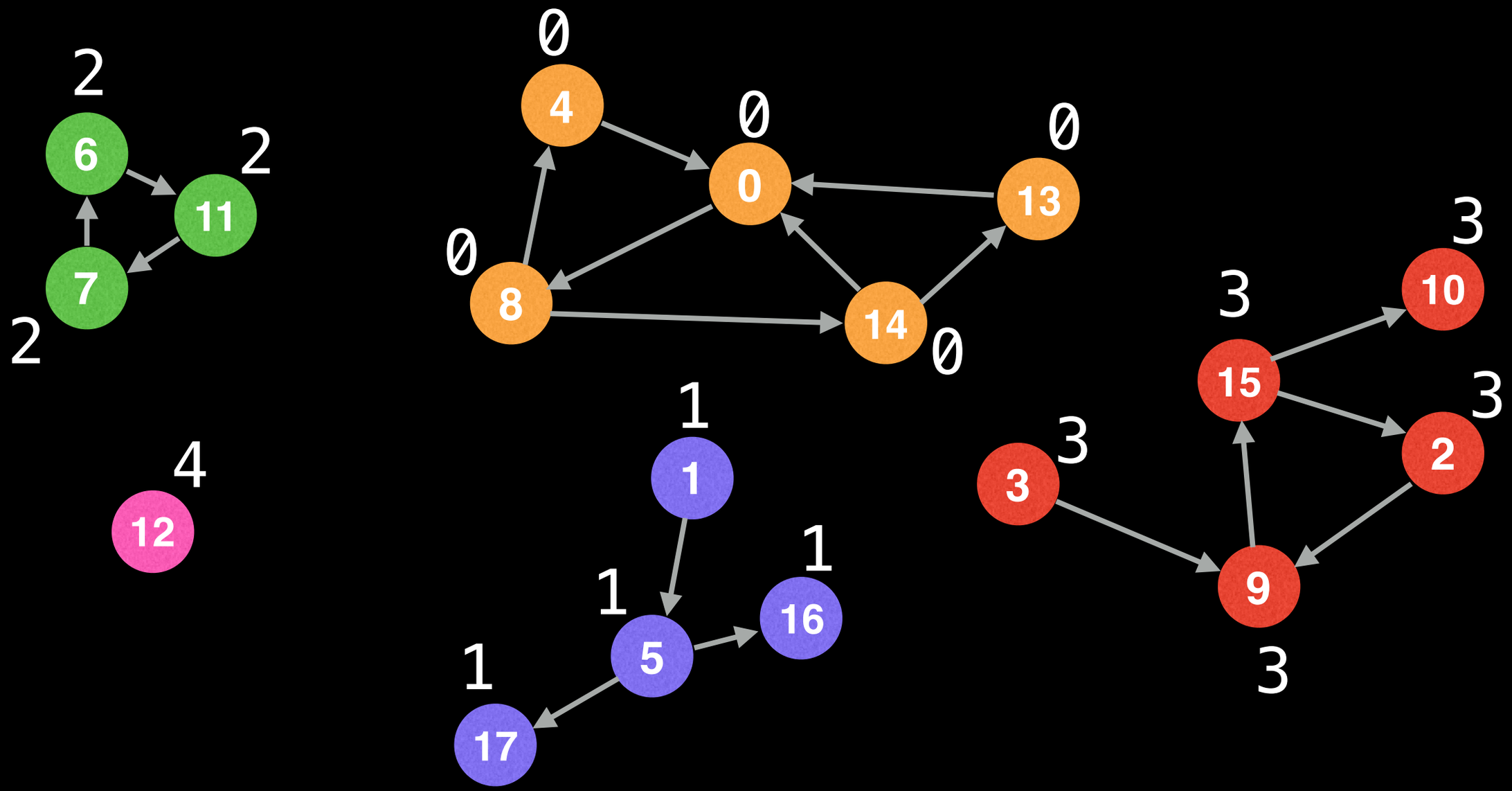
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



… and so on for the other components



```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

What else can DFS do?

We can augment the DFS algorithm to:

- Compute a graph’s minimum spanning tree.
- Detect and find cycles in a graph.
- Check if a graph is bipartite.
- Find strongly connected components.
- Topologically sort the nodes of a graph.
- Find bridges and articulation points.
- Find augmenting paths in a flow network.
- Generate mazes.