DEPTH FIRST SEARCH

DFS

Depth-first search

- Depth-first search is a widely used graph traversal algorithm besides breadth-first search
- It was investigated as strategy for solving mazes by Trémaux in the 19th century
- It explores as far as possible along each branch before backtracking // BFS was a layer-by-layer algorithm
- Time complexity of traversing a graph with DFS: **O(V+E)**
- Memory complexity: a bit better than that of BFS !!!
- By itself the DFS isn't all that useful, but when augmented to perform other tasks such as count connected components, determine connectivity, or find bridges/articulation points then DFS really shines.

Depth-first search

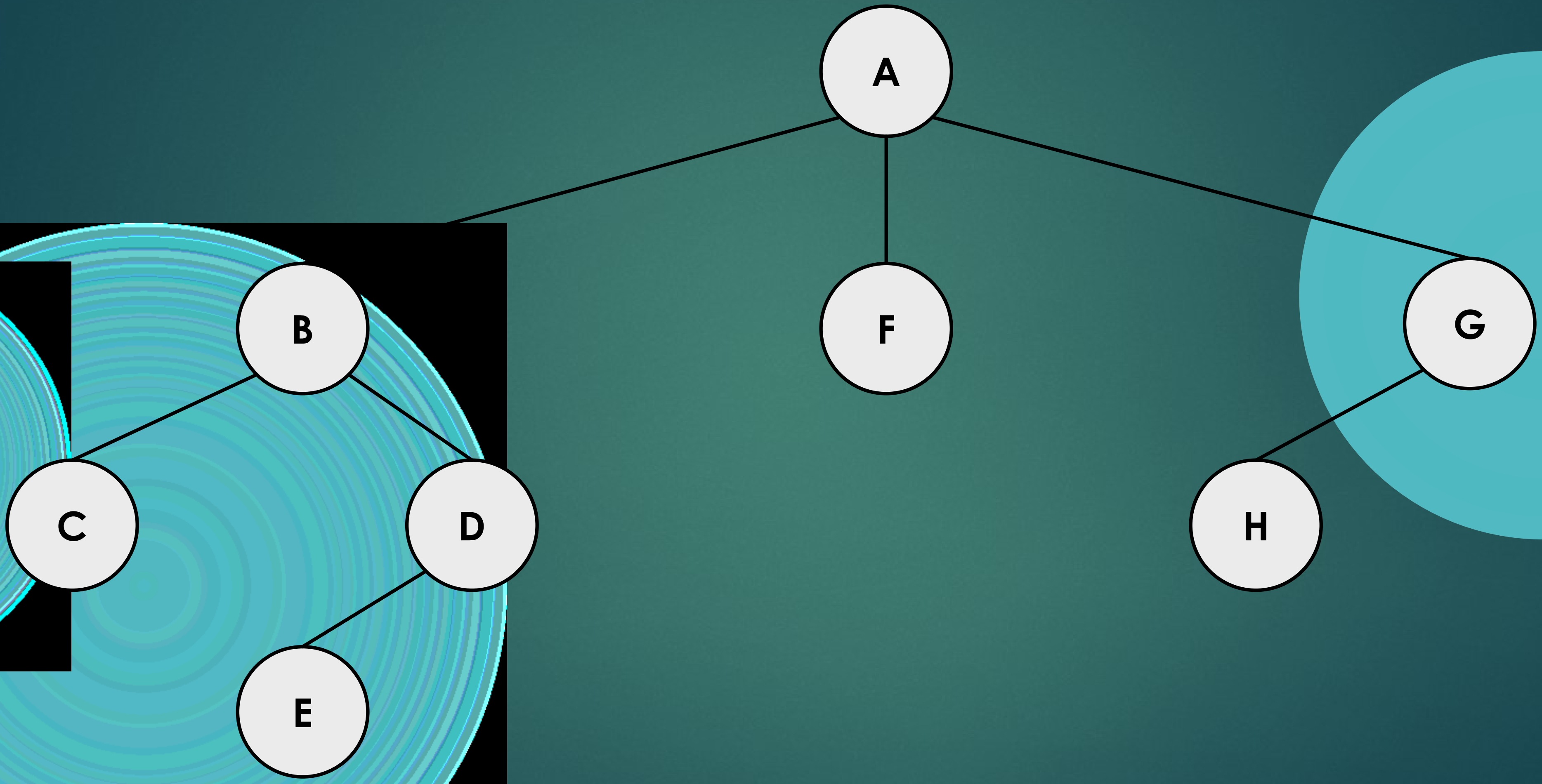
```
dfs(vertex)

        vertex set visited true
        print vertex

        for v in vertex neighbours
                if v is not visited
                        dfs(v)
```

RECURSION

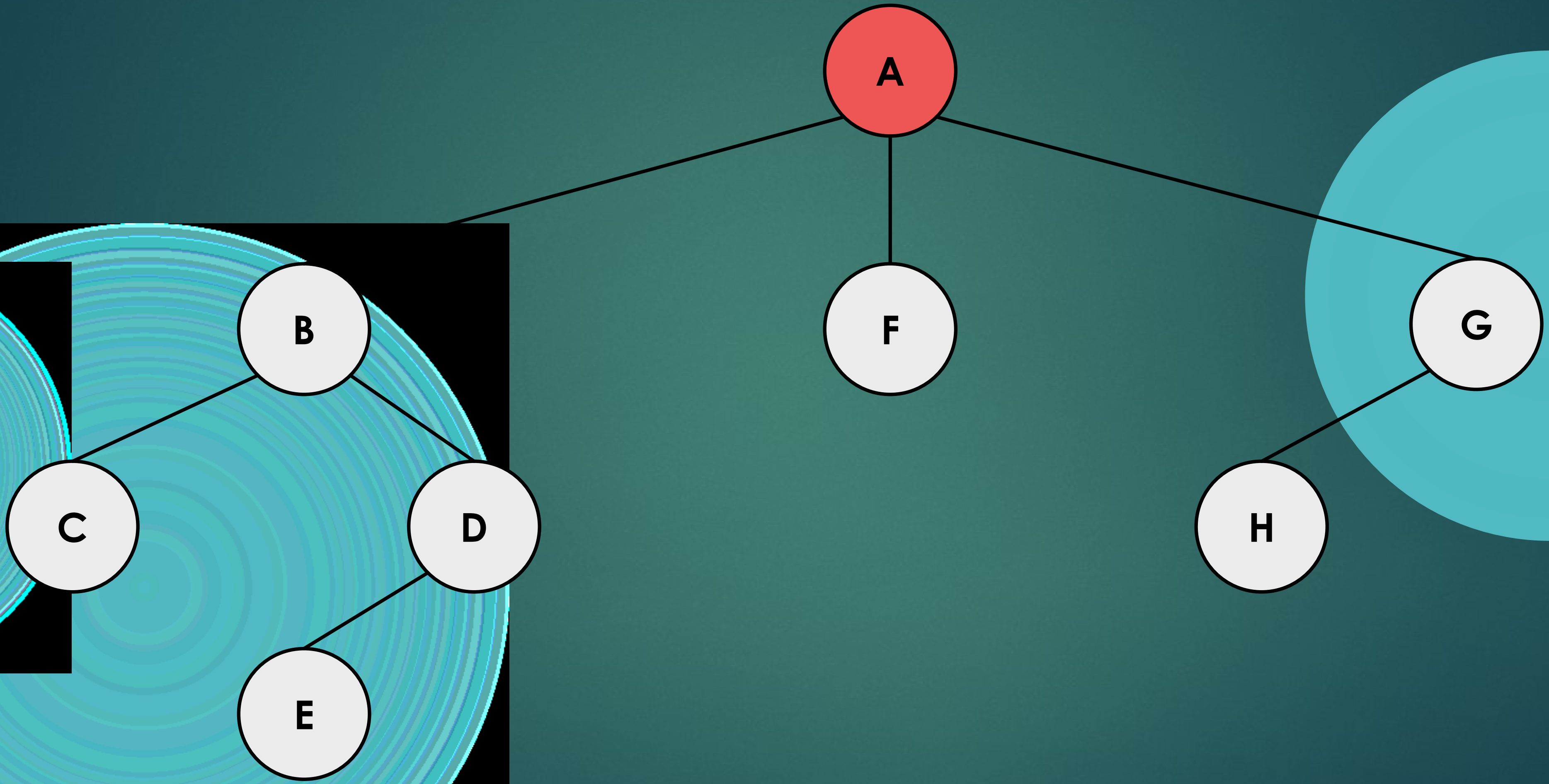
```
dfs(vertex)

        Stack stack
        vertex set visited true
        stack.push(vertex)

        while stack not empty
                actual = stack.pop()

                for v in actual neighbours
                        if v is not visited
                                v set visited true
                                stack.push(v)
```

ITERATION

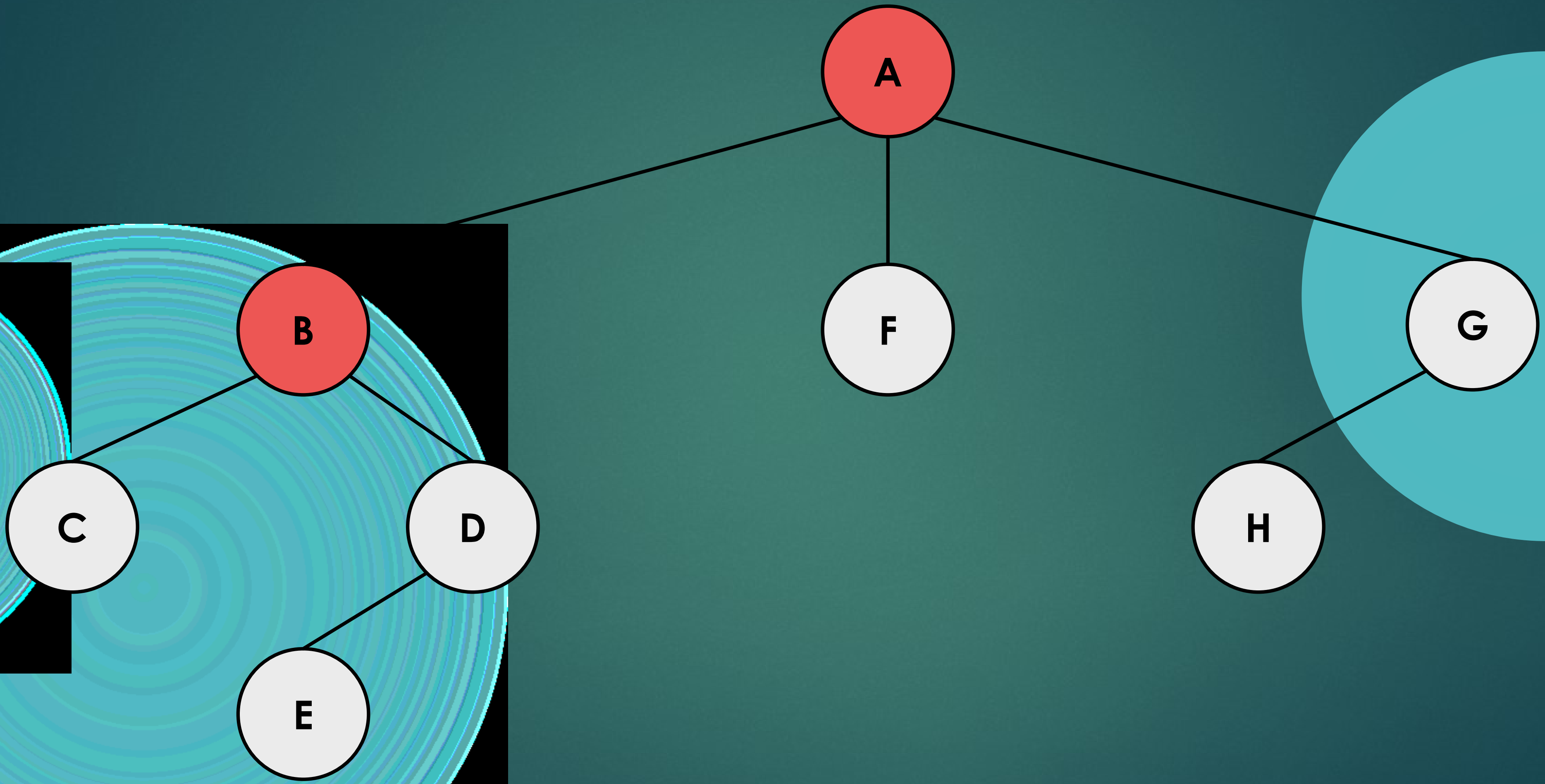
Recursively check every unmarked vertices!



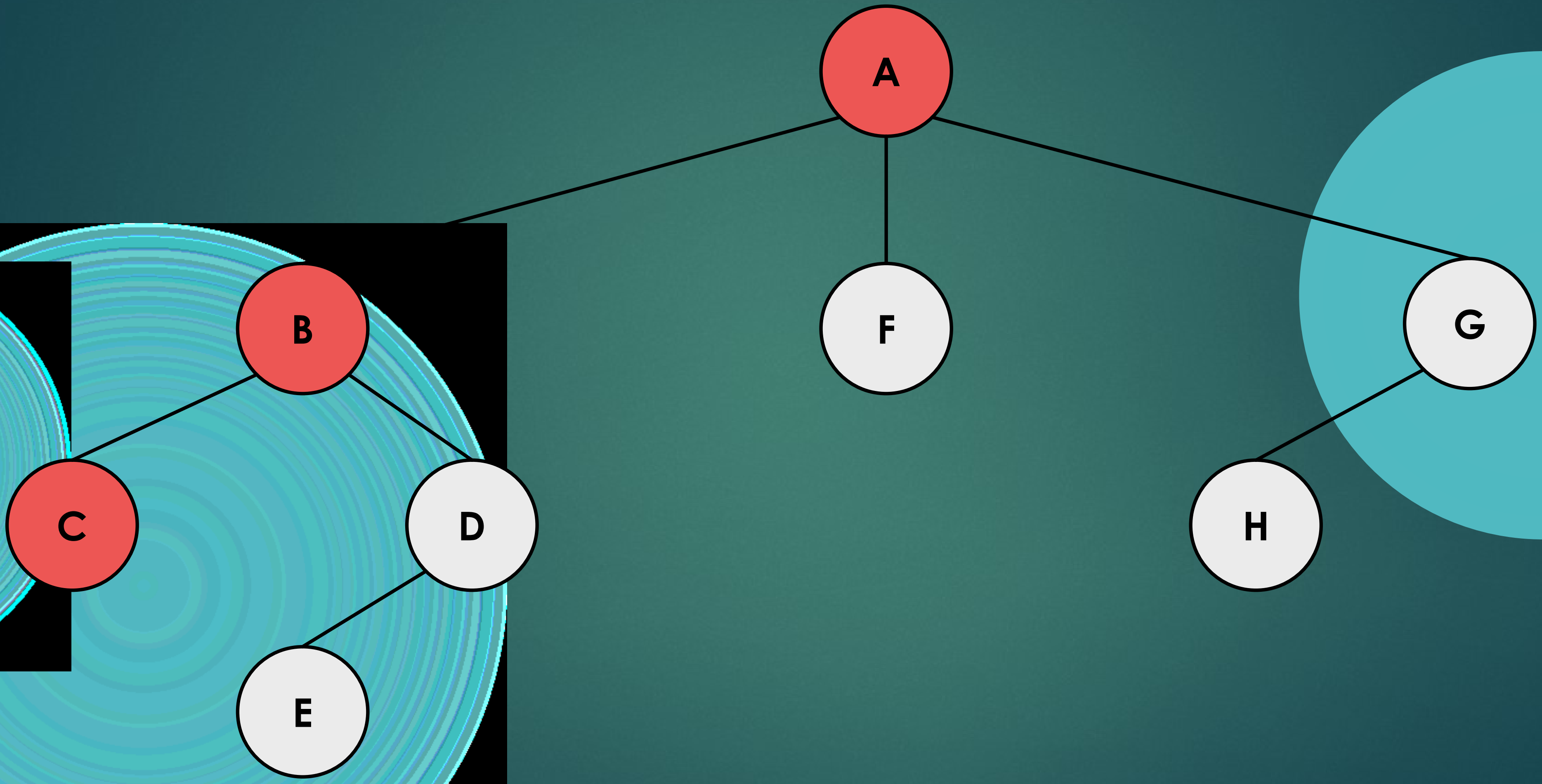
Recursively check every unmarked vertices!



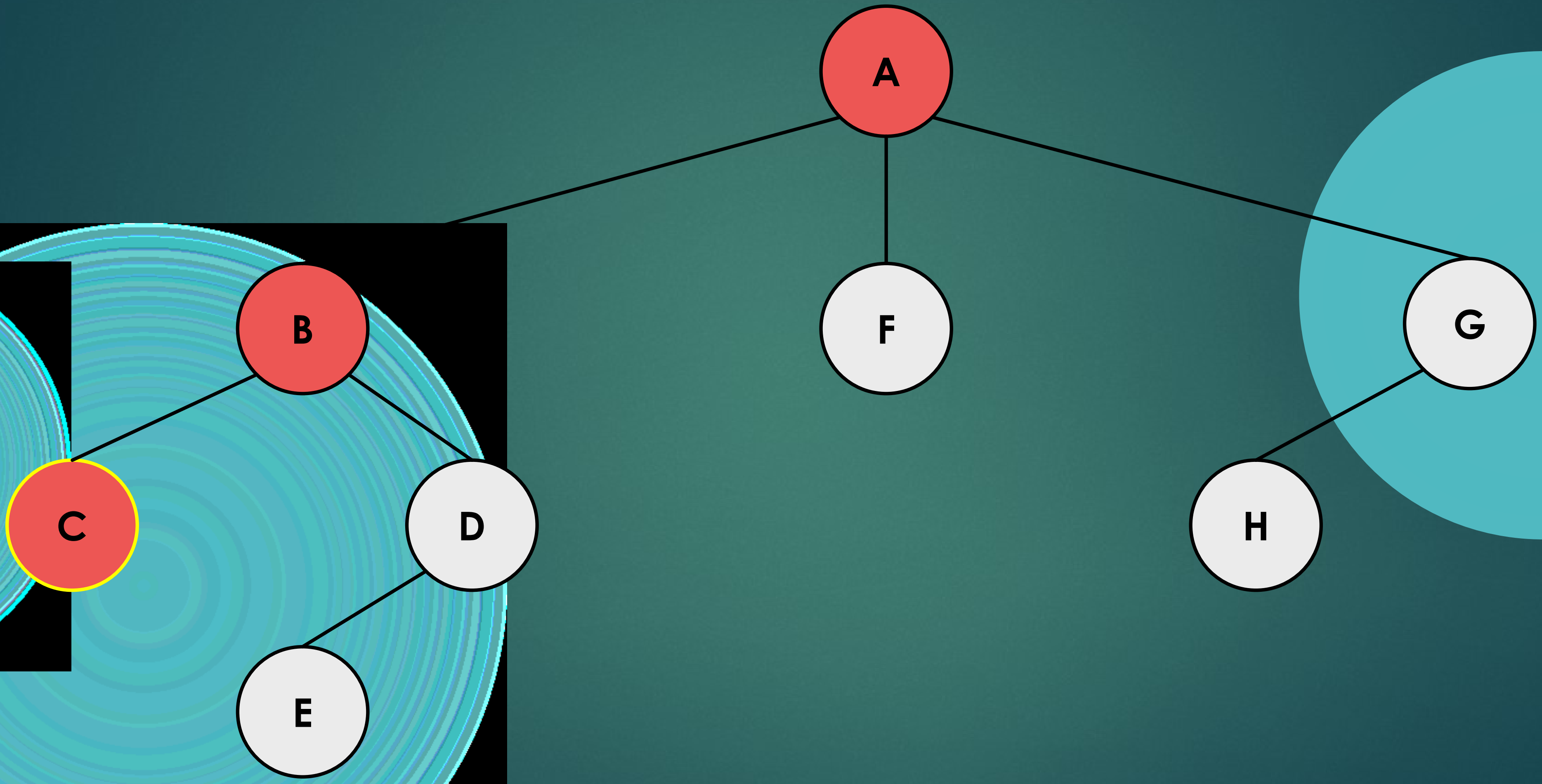
Recursively check every unmarked vertices!



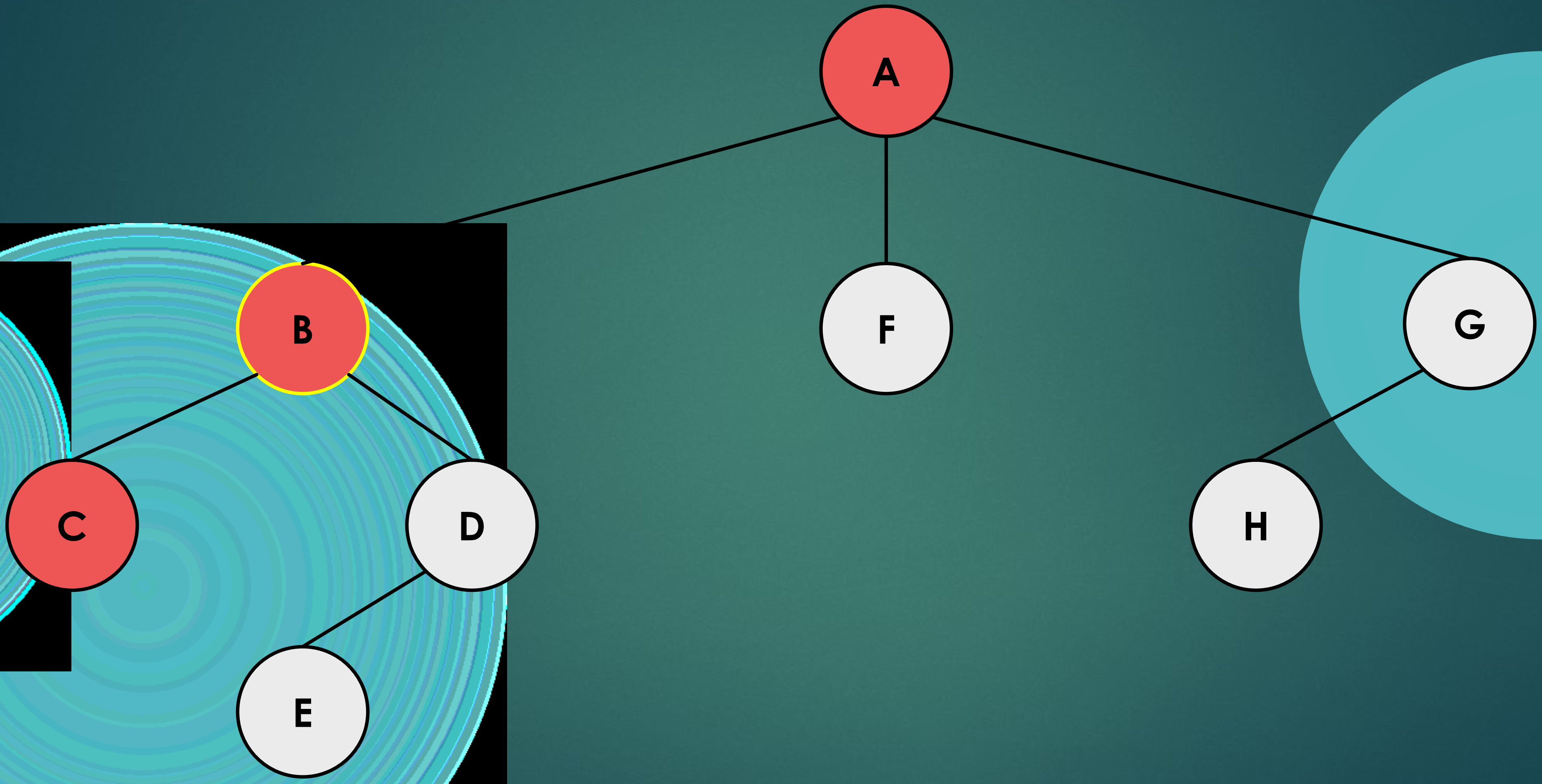
Recursively check every unmarked vertices!



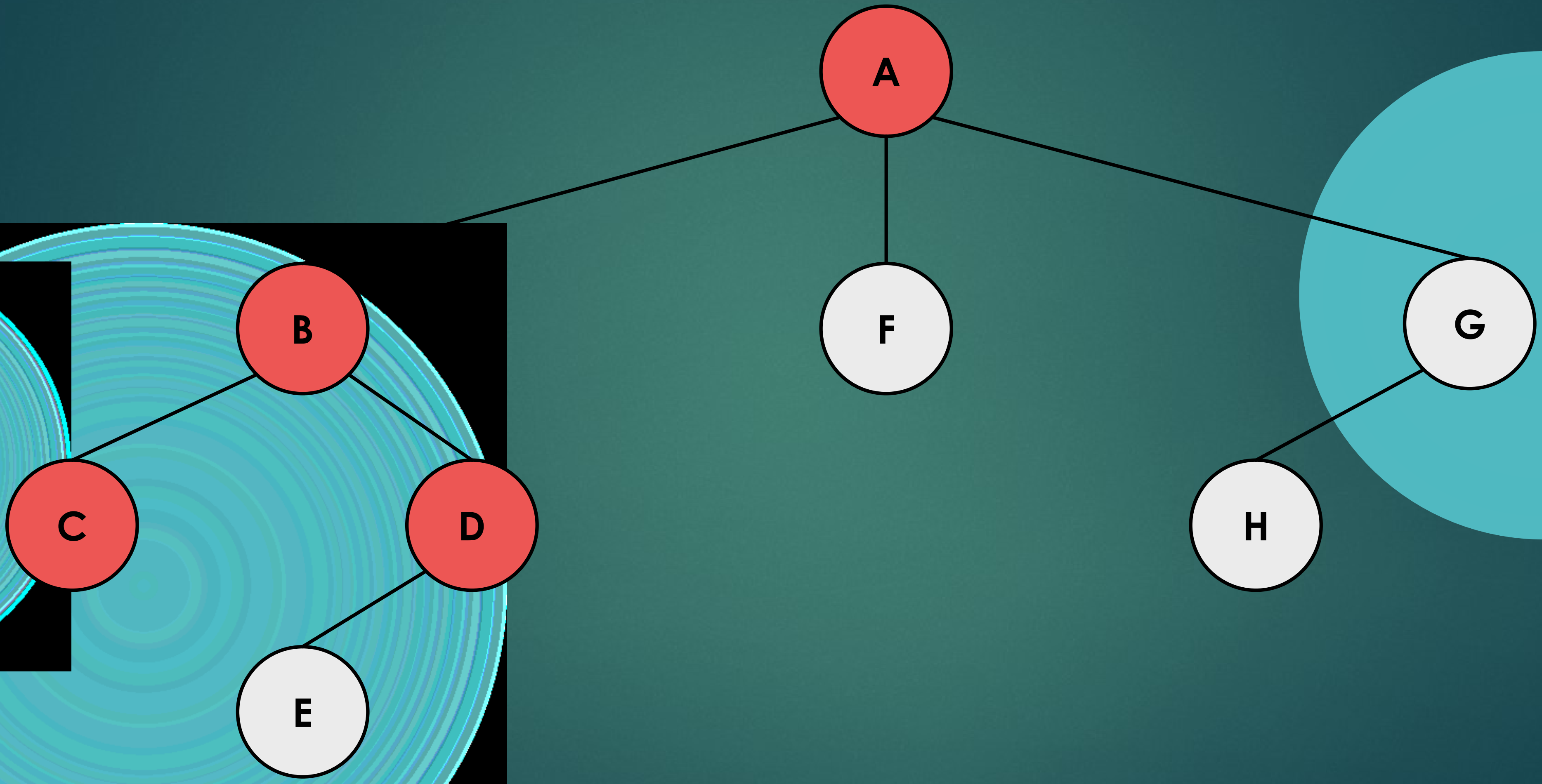
Recursively check every unmarked vertices!



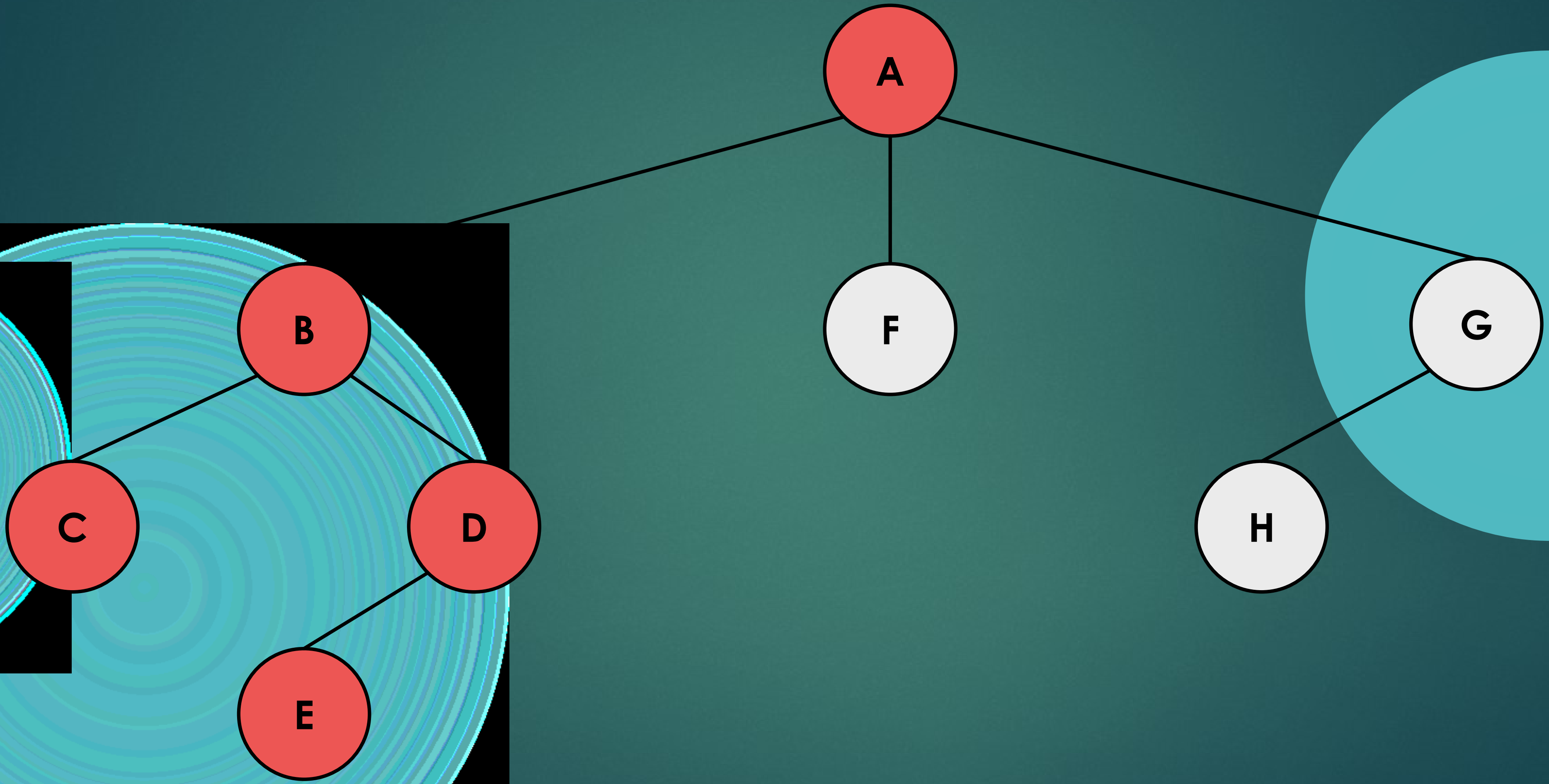
Recursively check every
unmarked vertices!



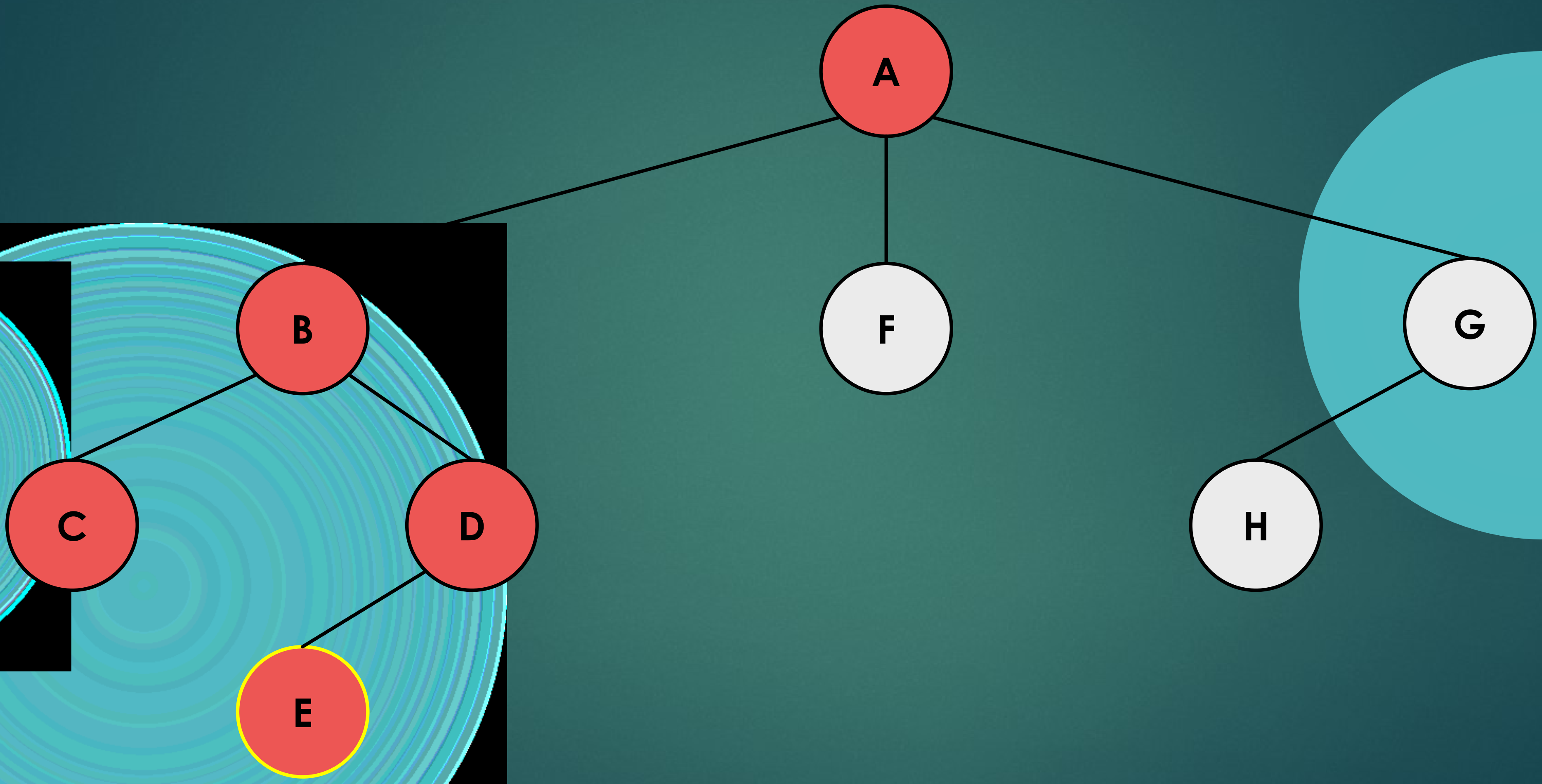
Recursively check every
unmarked vertices!



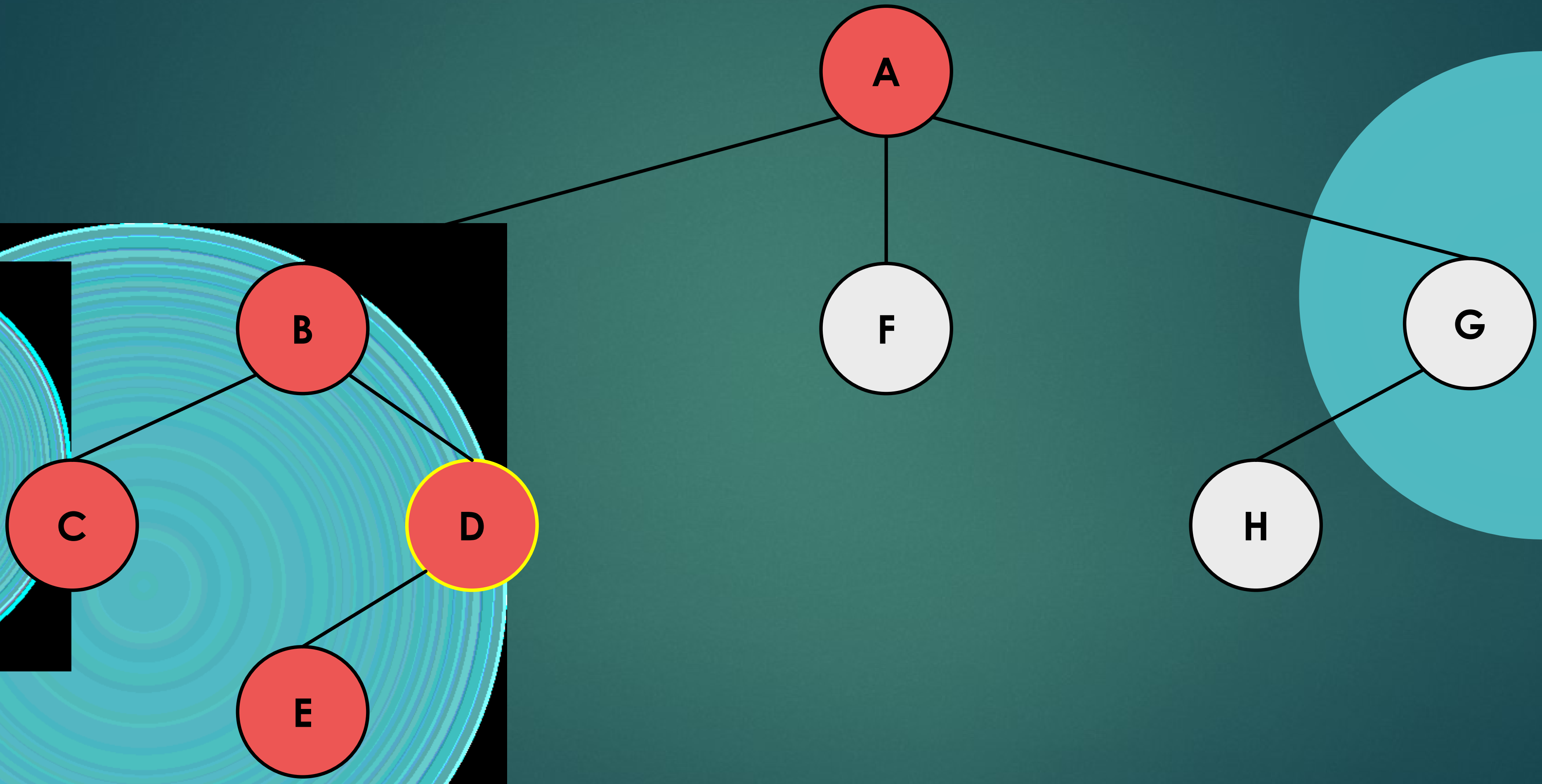
Recursively check every unmarked vertices!



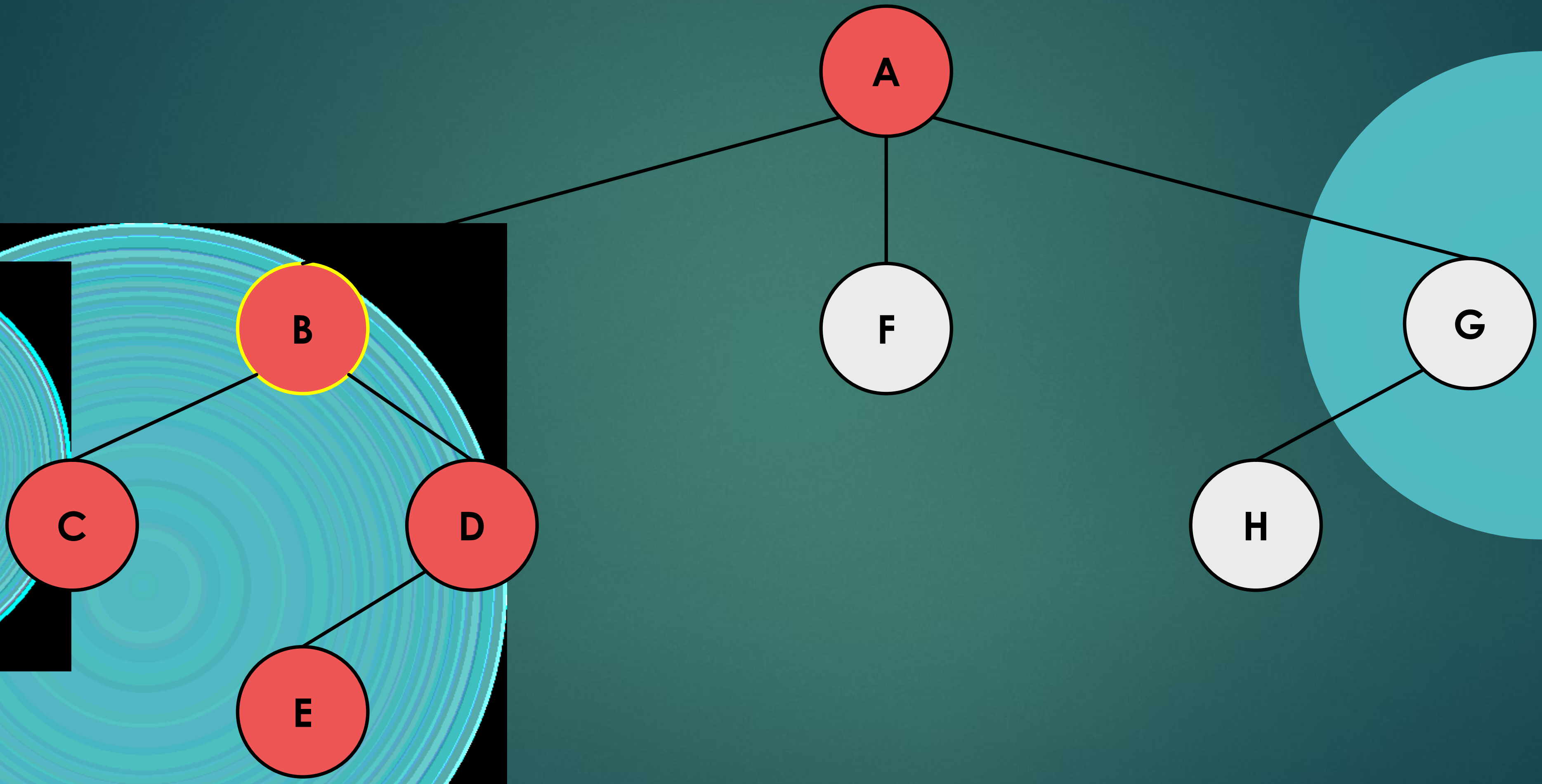
Recursively check every unmarked vertices!



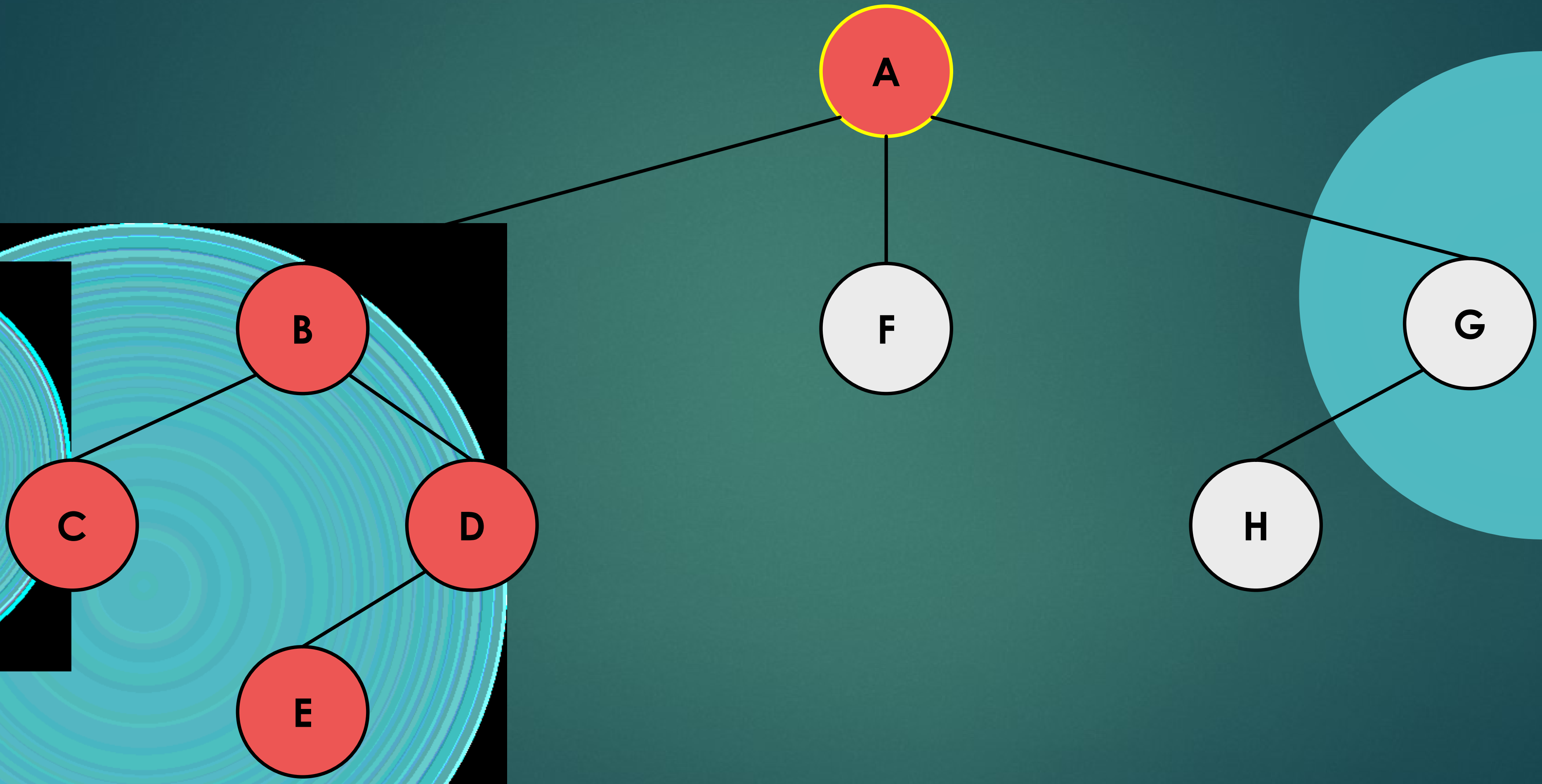
Recursively check every unmarked vertices!



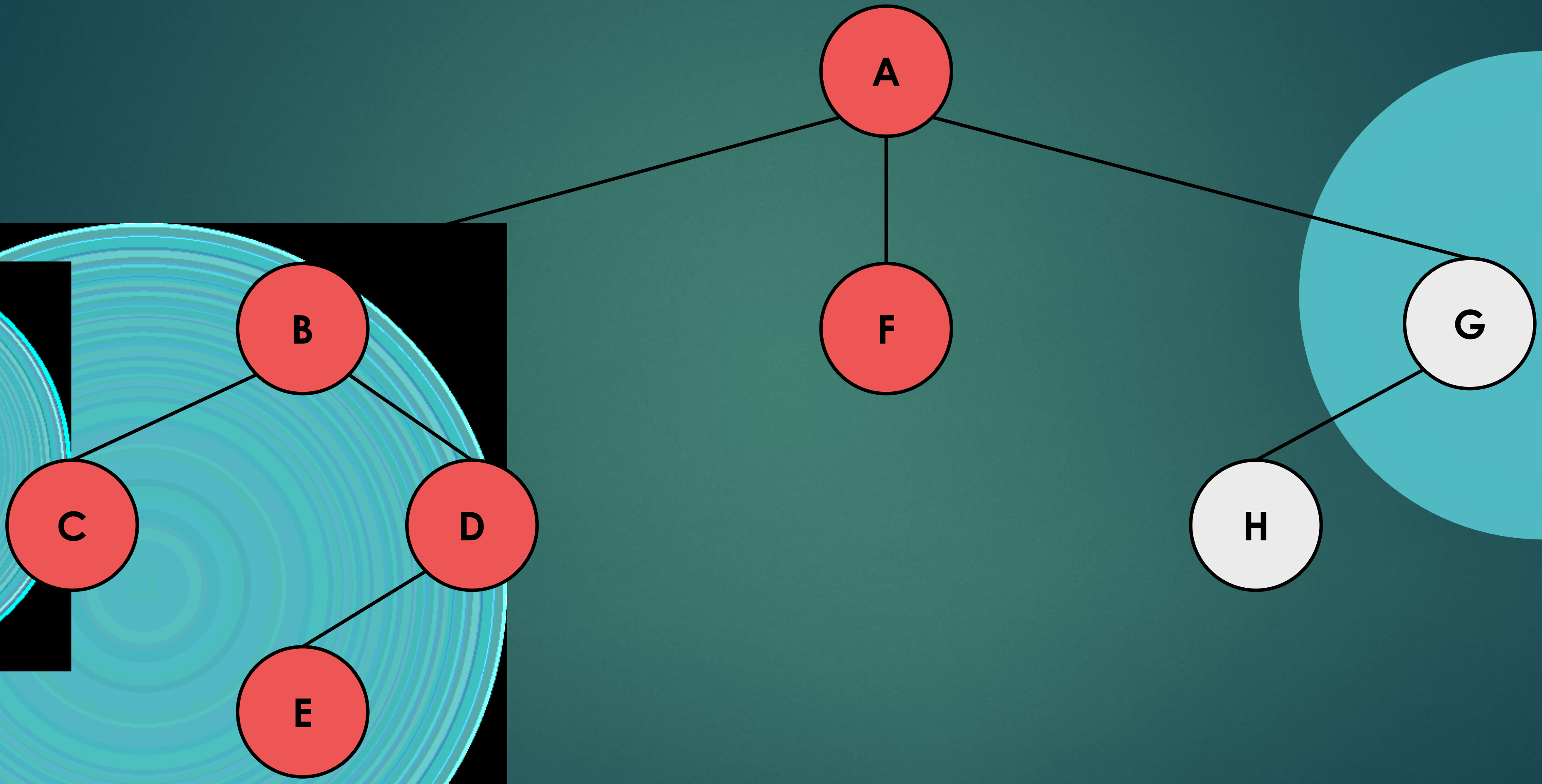
Recursively check every unmarked vertices!



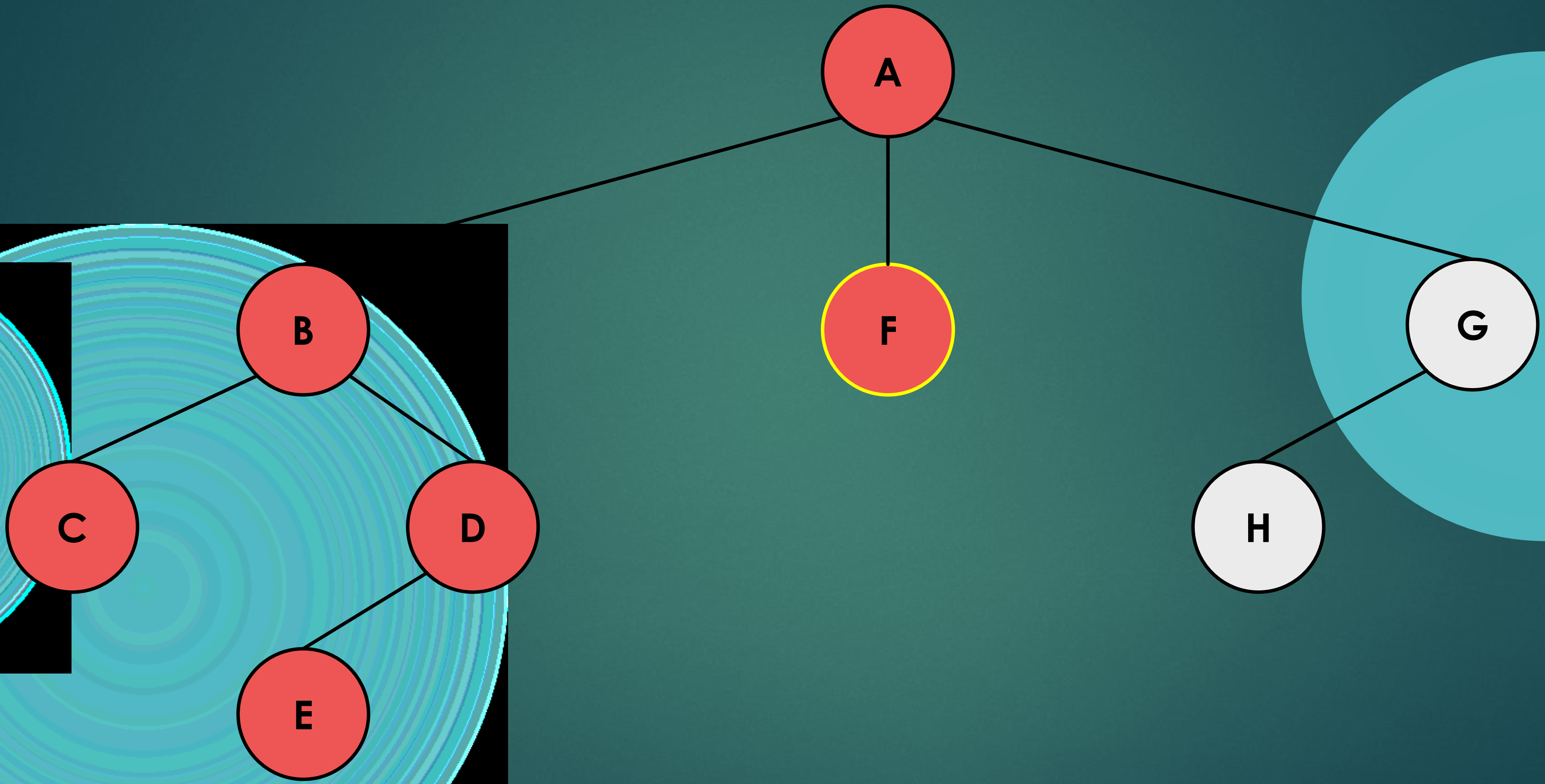
Recursively check every unmarked vertices!



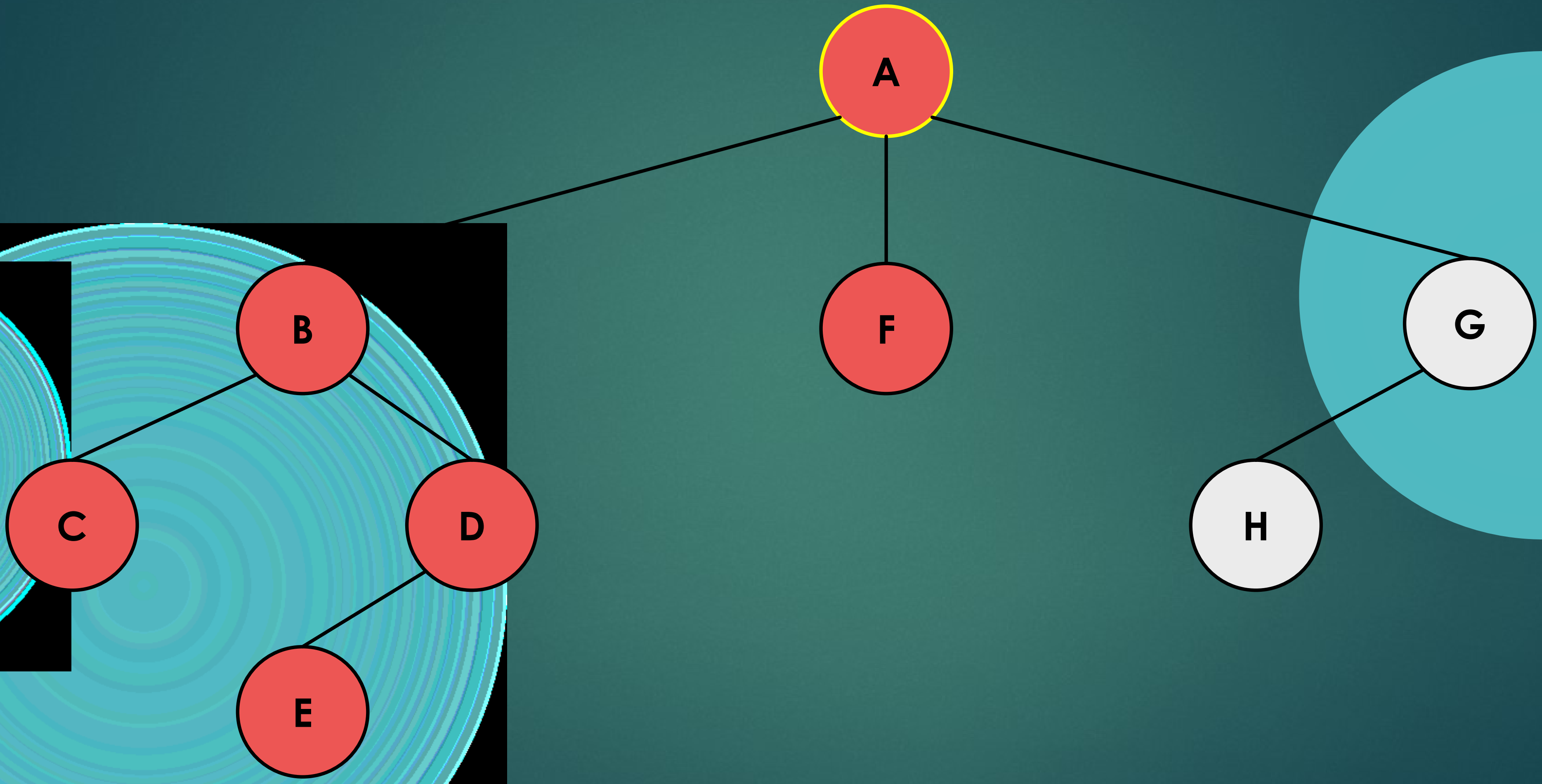
Recursively check every unmarked vertices!



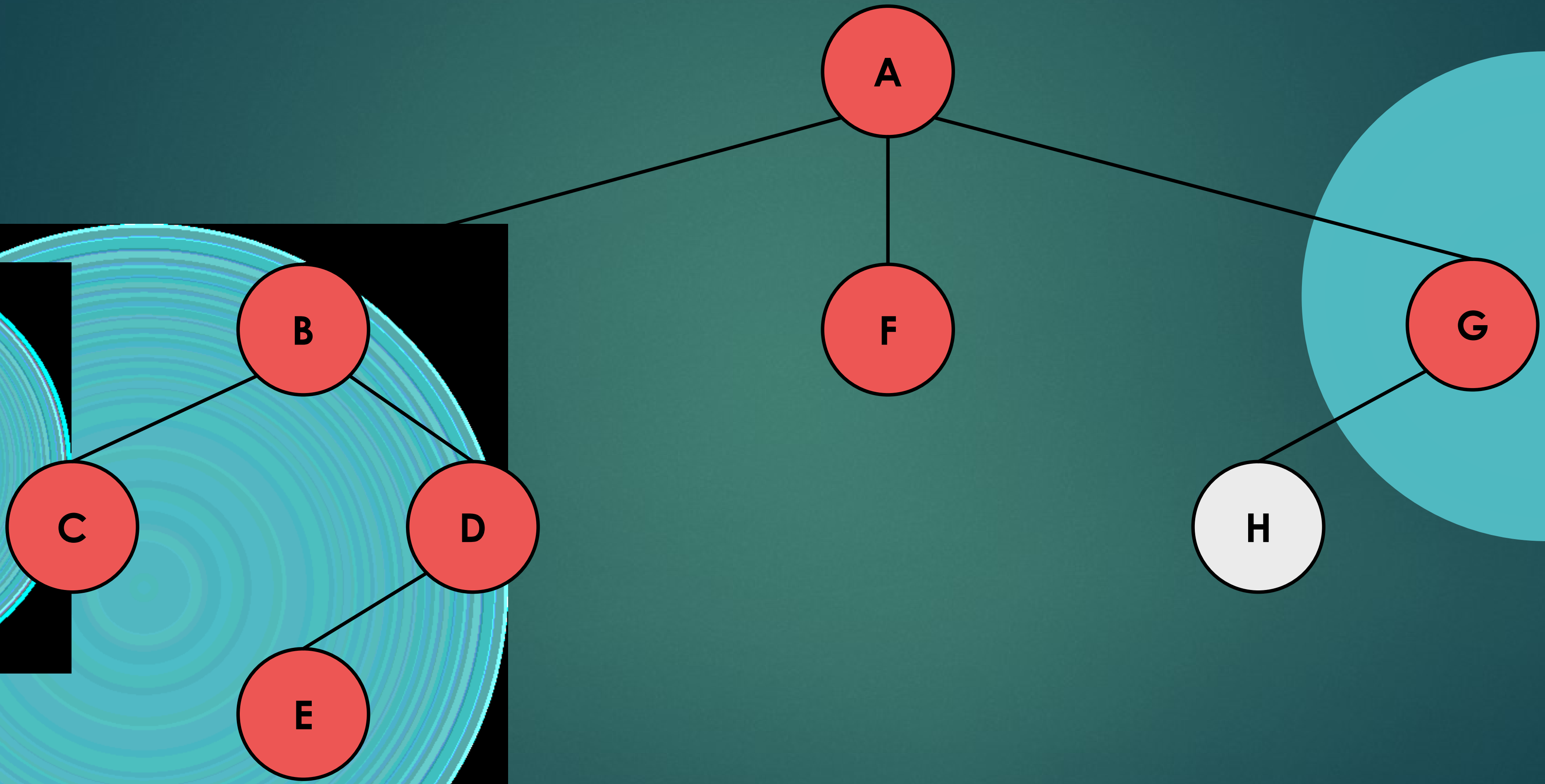
Recursively check every unmarked vertices!



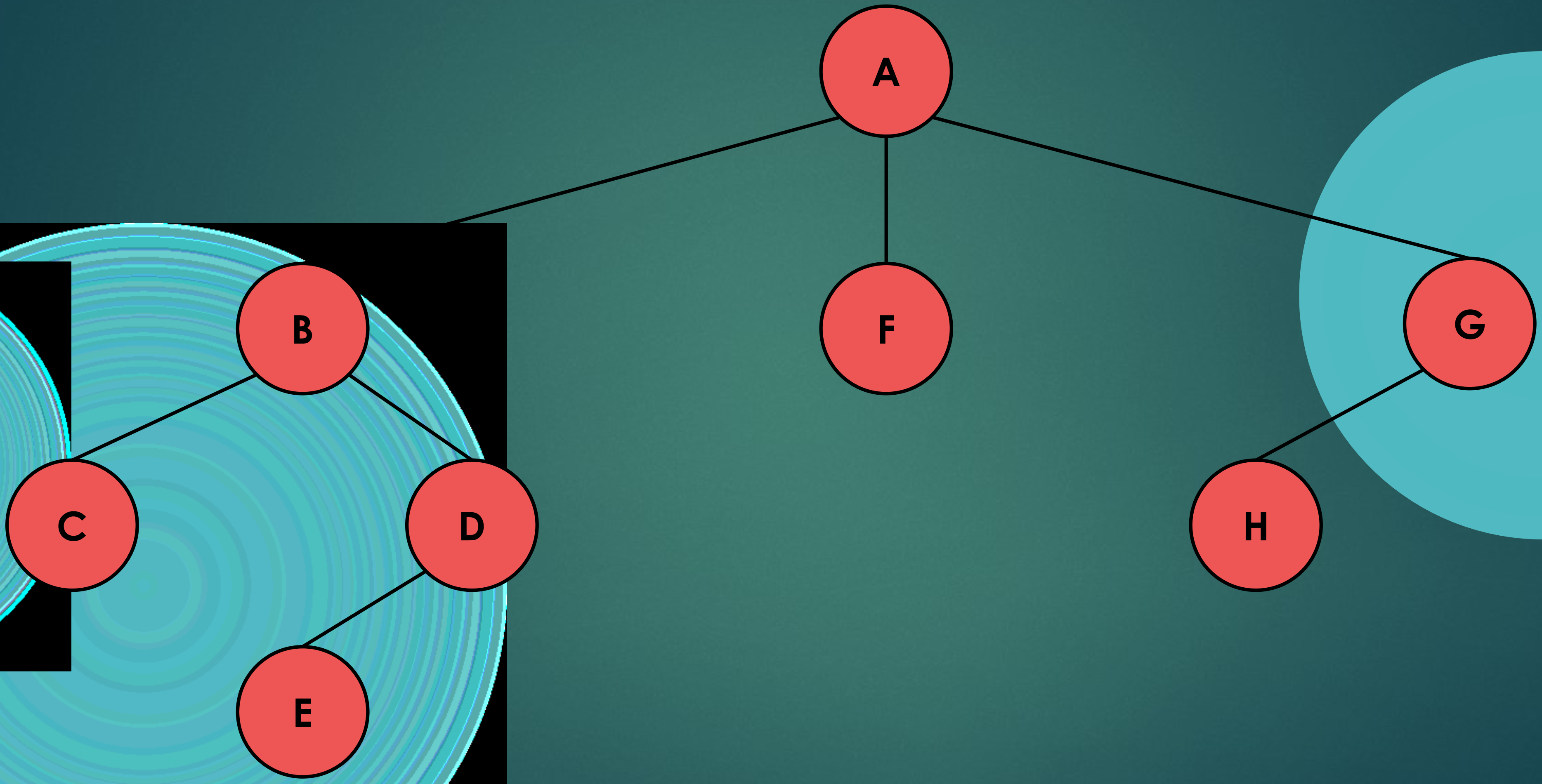
Recursively check every unmarked vertices!



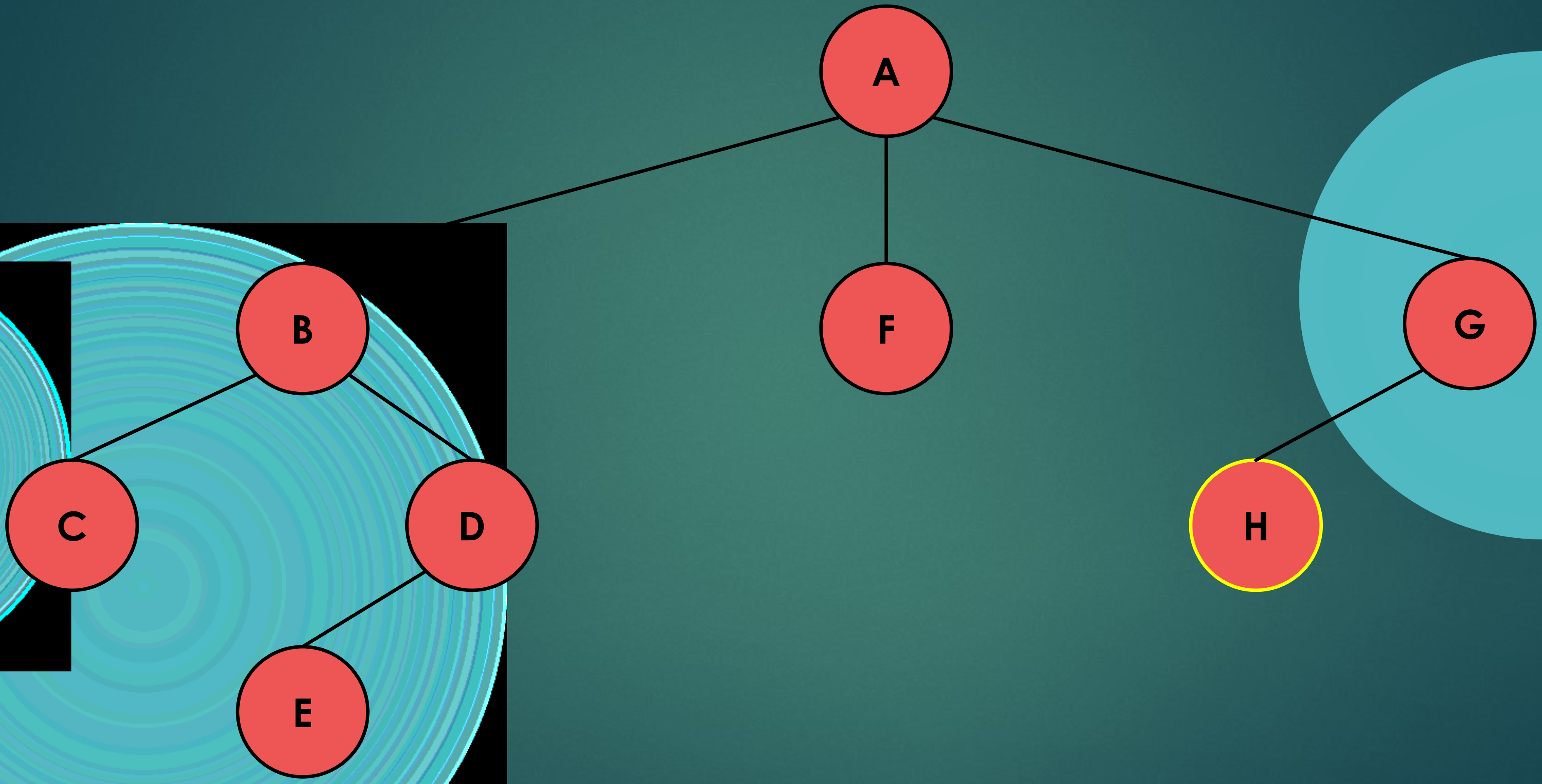
Recursively check every unmarked vertices!



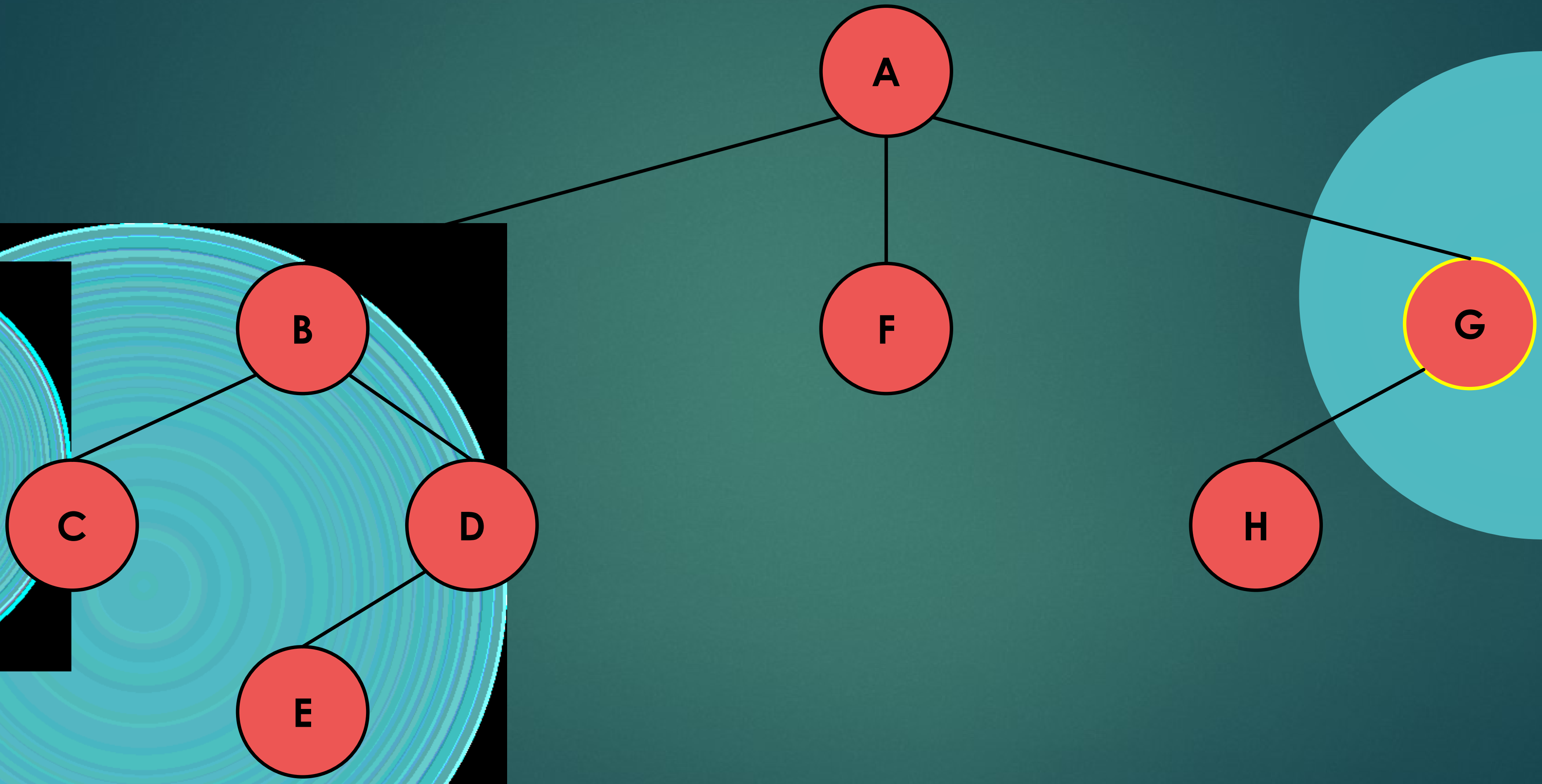
Recursively check every
unmarked vertices!



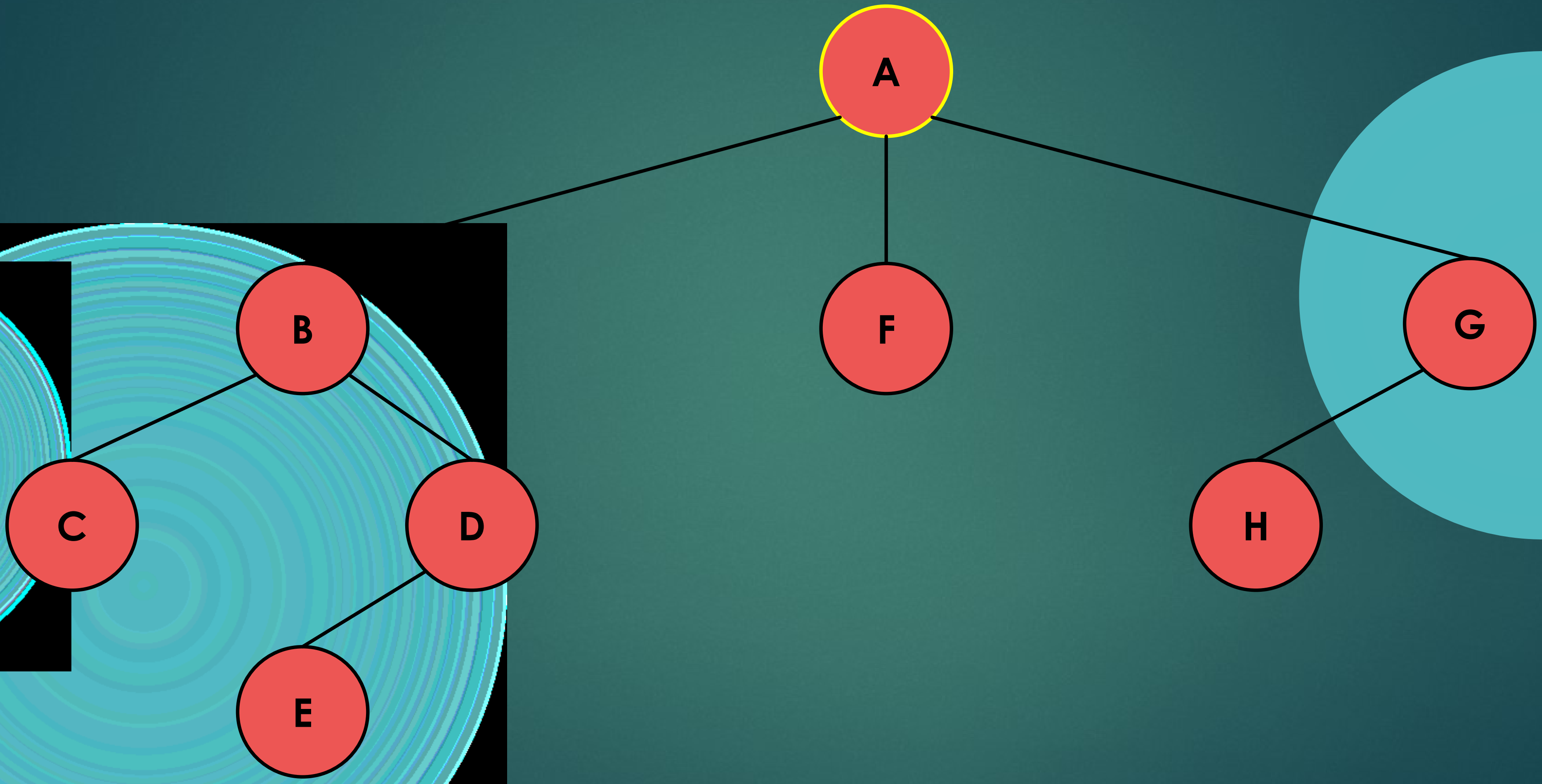
Recursively check every
unmarked vertices!



Recursively check every unmarked vertices!

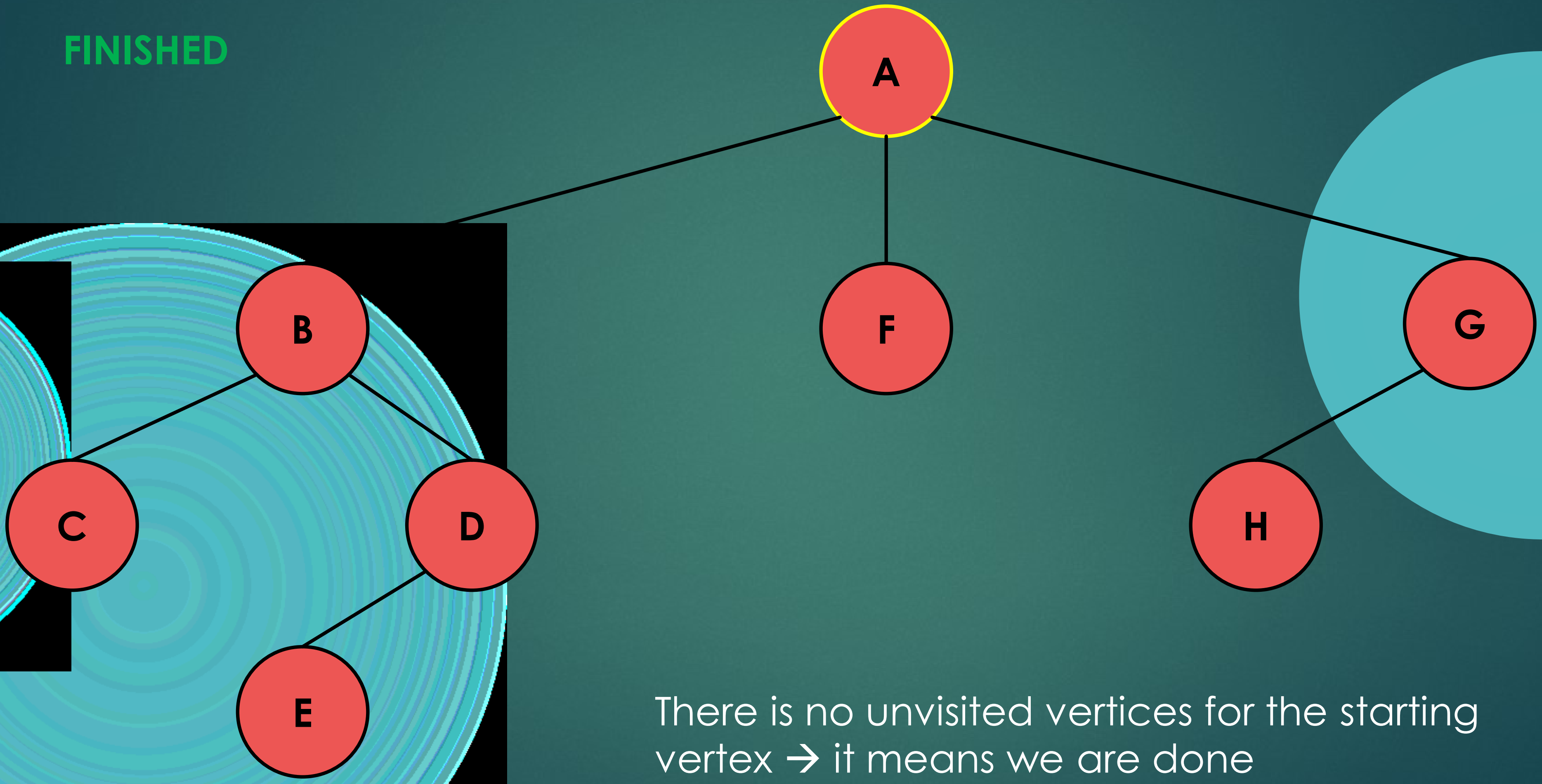


Recursively check every unmarked vertices!



Recursively check every
unmarked vertices!

FINISHED

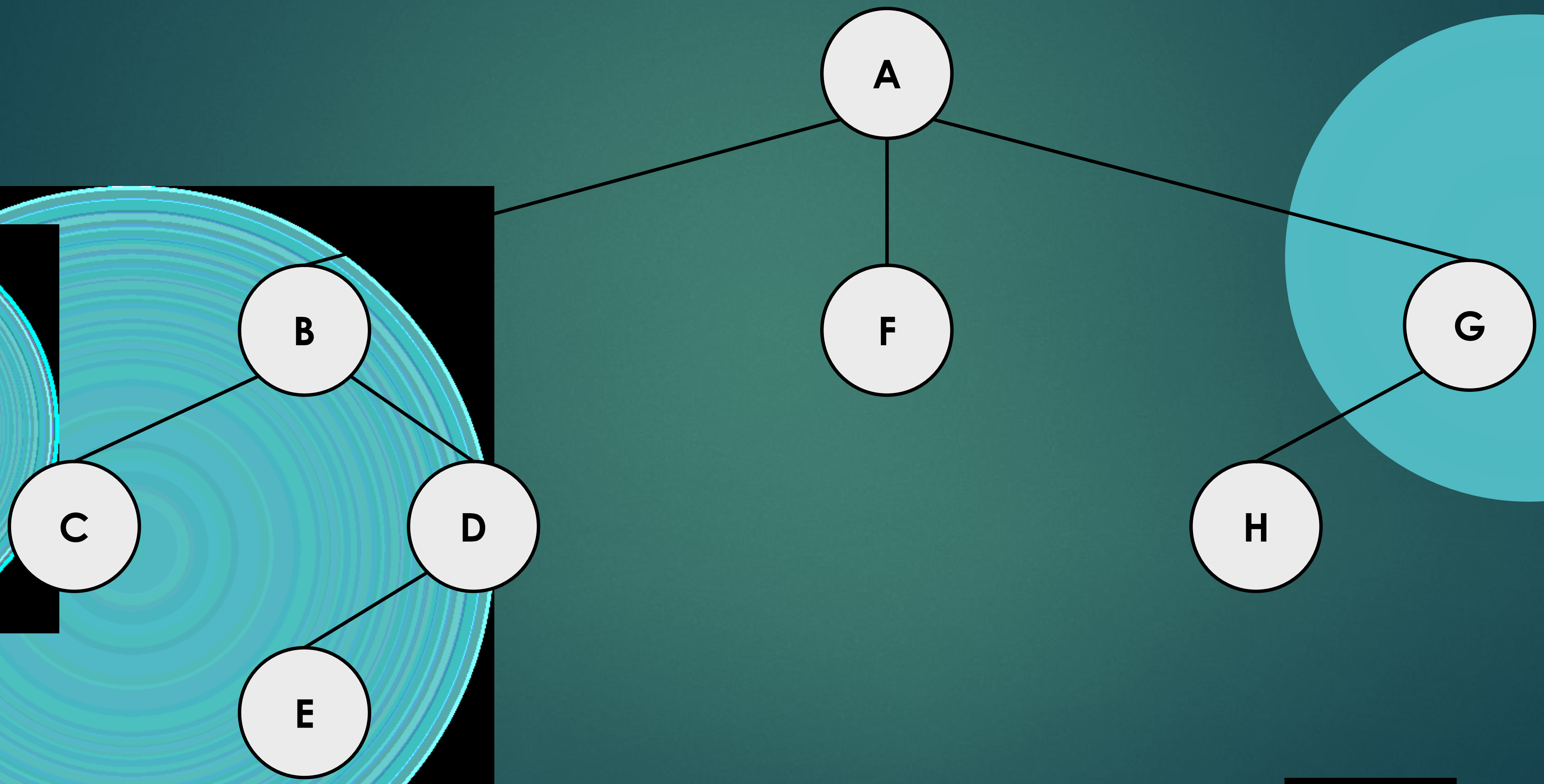


There is no unvisited vertices for the starting
vertex → it means we are done

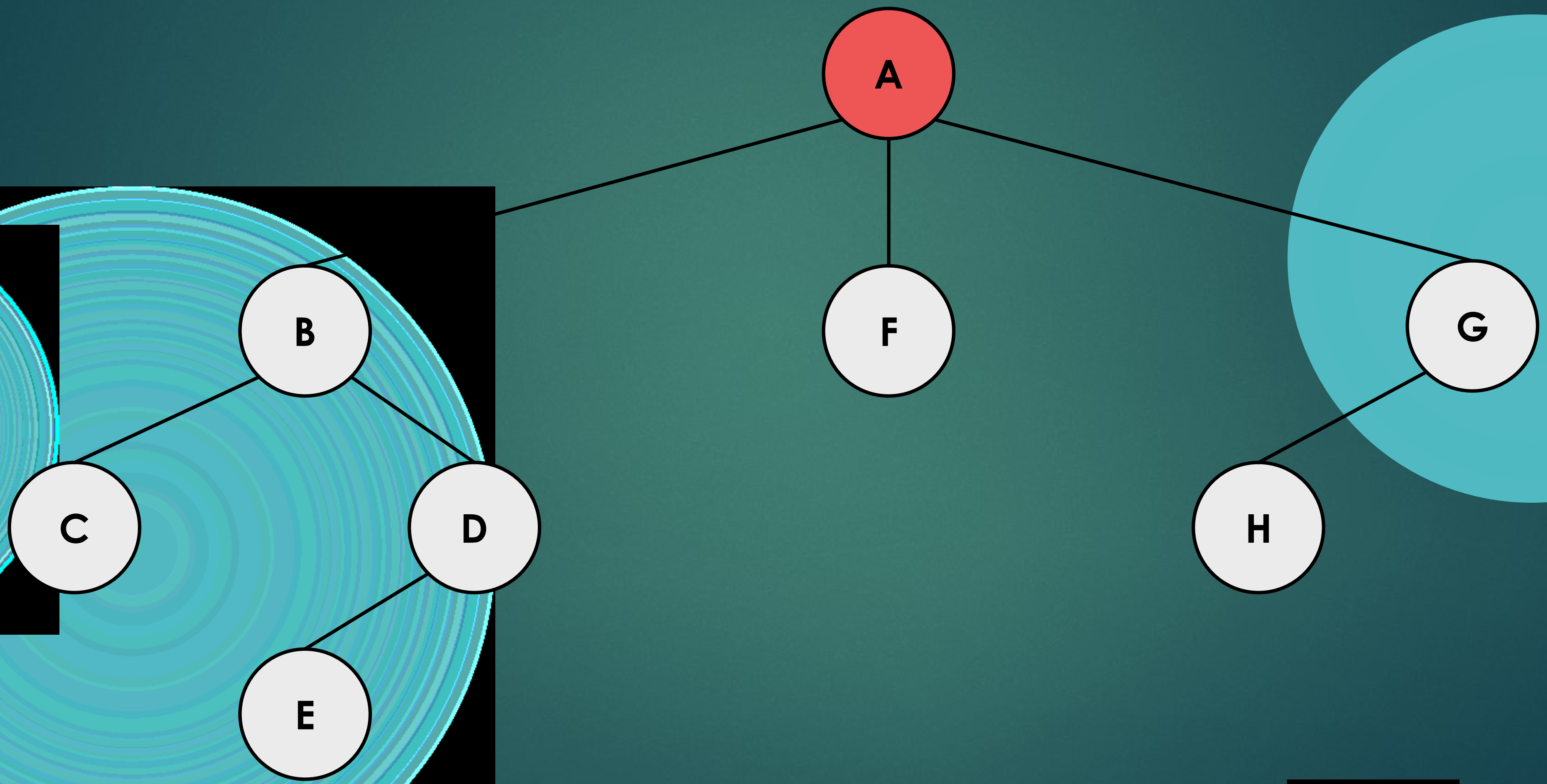
Applications

- Topological ordering
- Kosaraju algorithm for finding strongly connected components in a graph which can be proved to be very important in recommendation systems (youtube)
- Detecting cycles (checking whether a graph is a DAG or not)
  - Processes waiting for each other ➔ this is a cycle
- Generating mazes OR finding way out of a maze

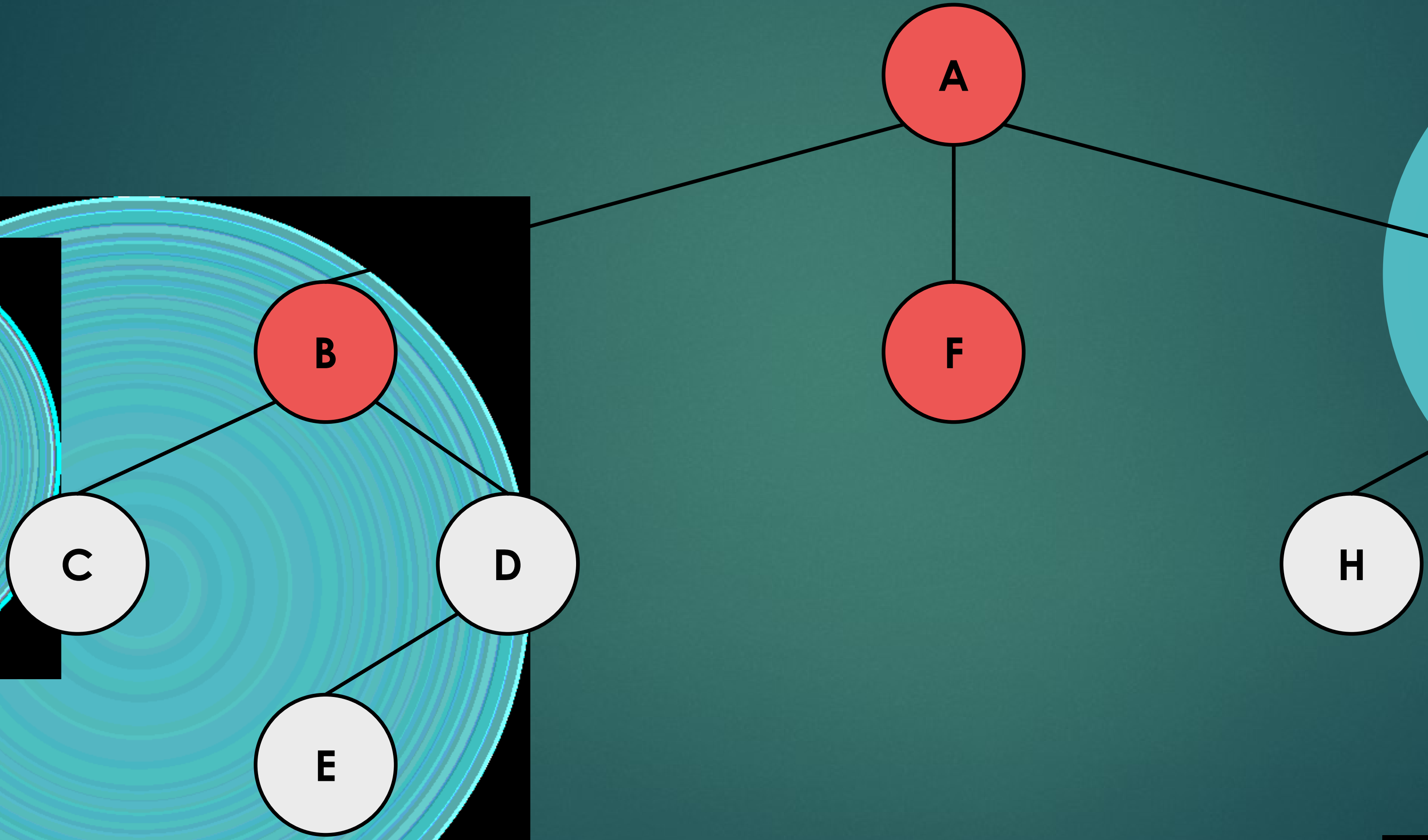
Revisiting breadth-first search



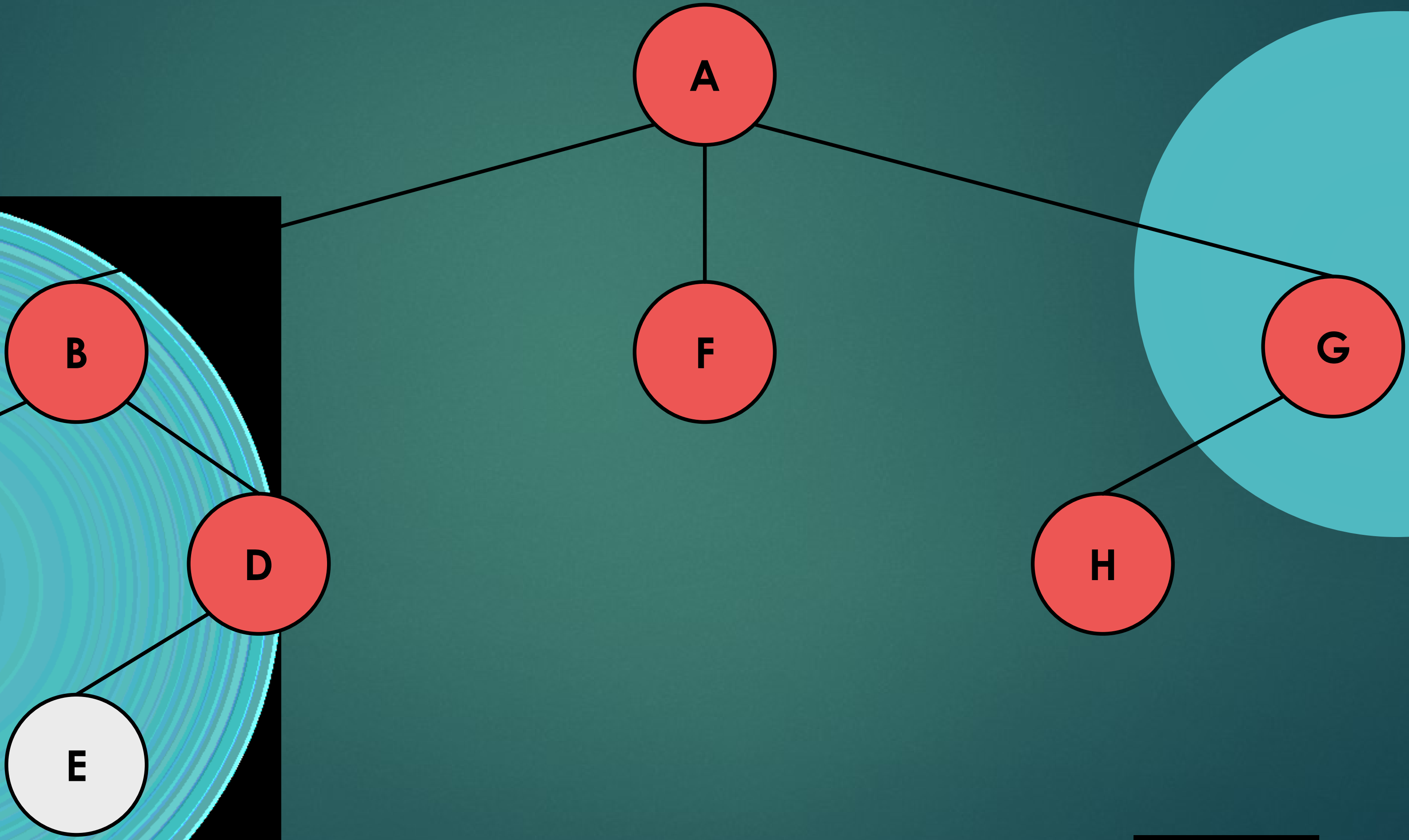
Revisiting breadth-first search



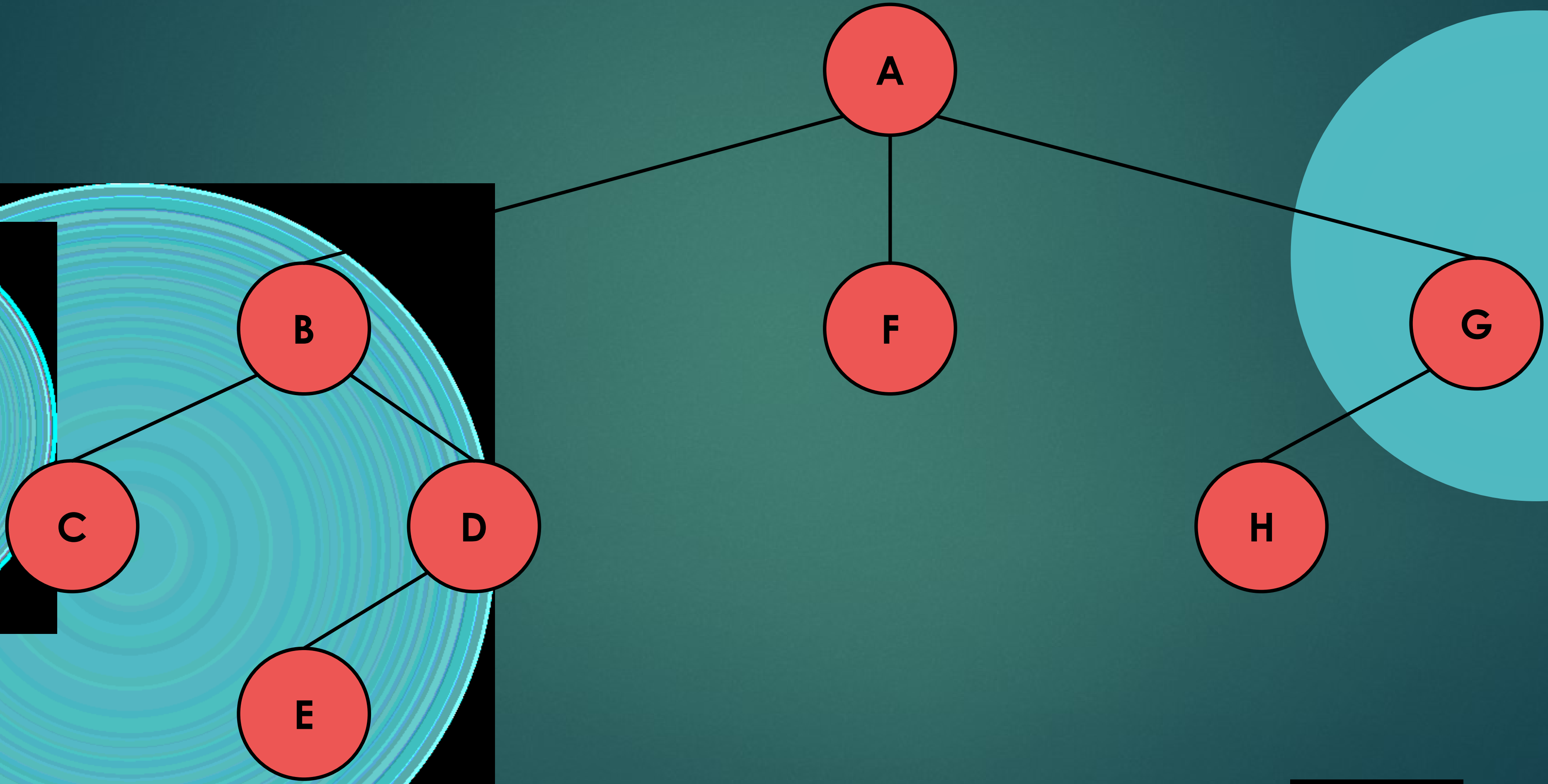
Revisiting breadth-first search



Revisiting breadth-first search

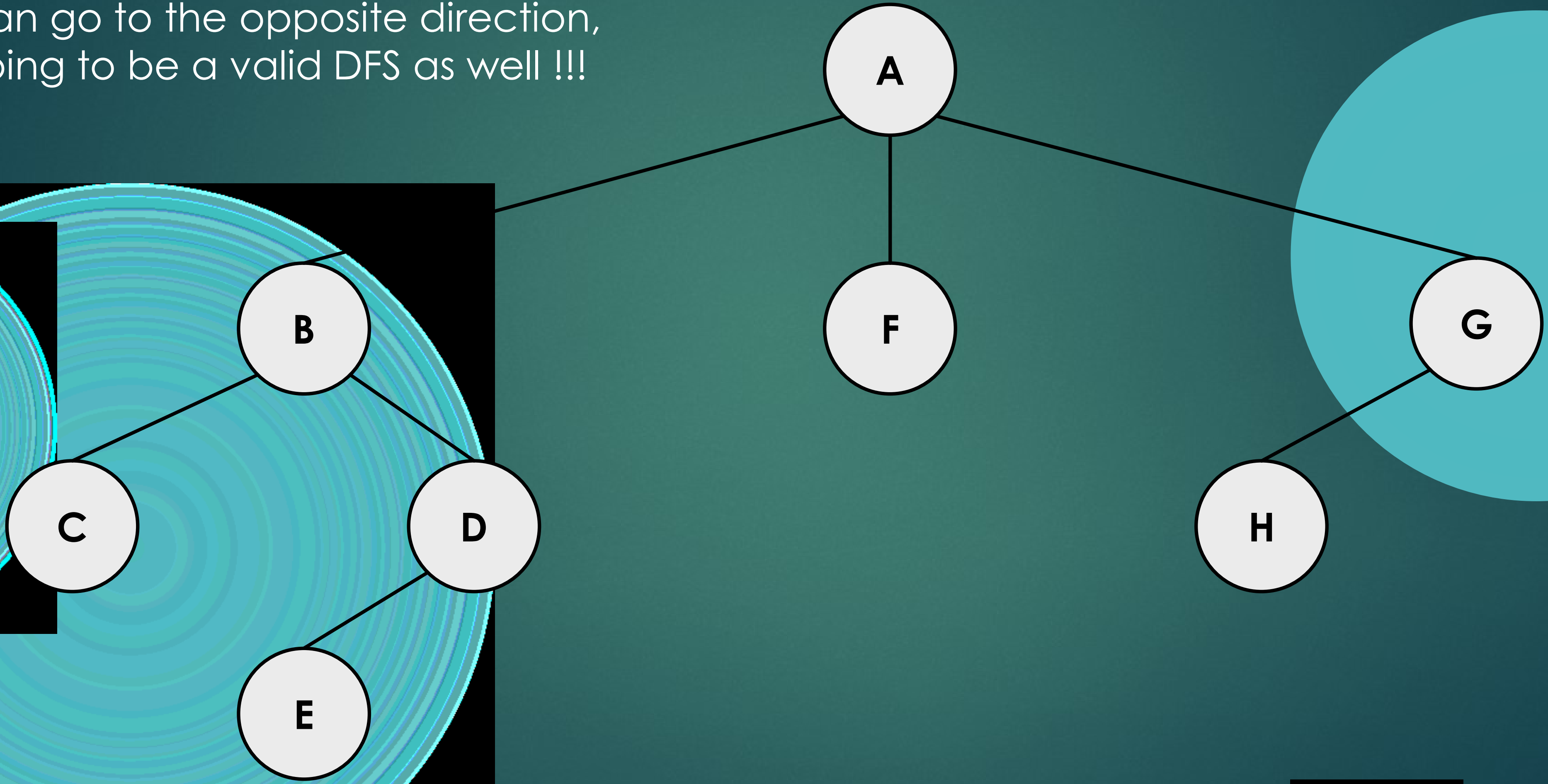


Revisiting breadth-first search



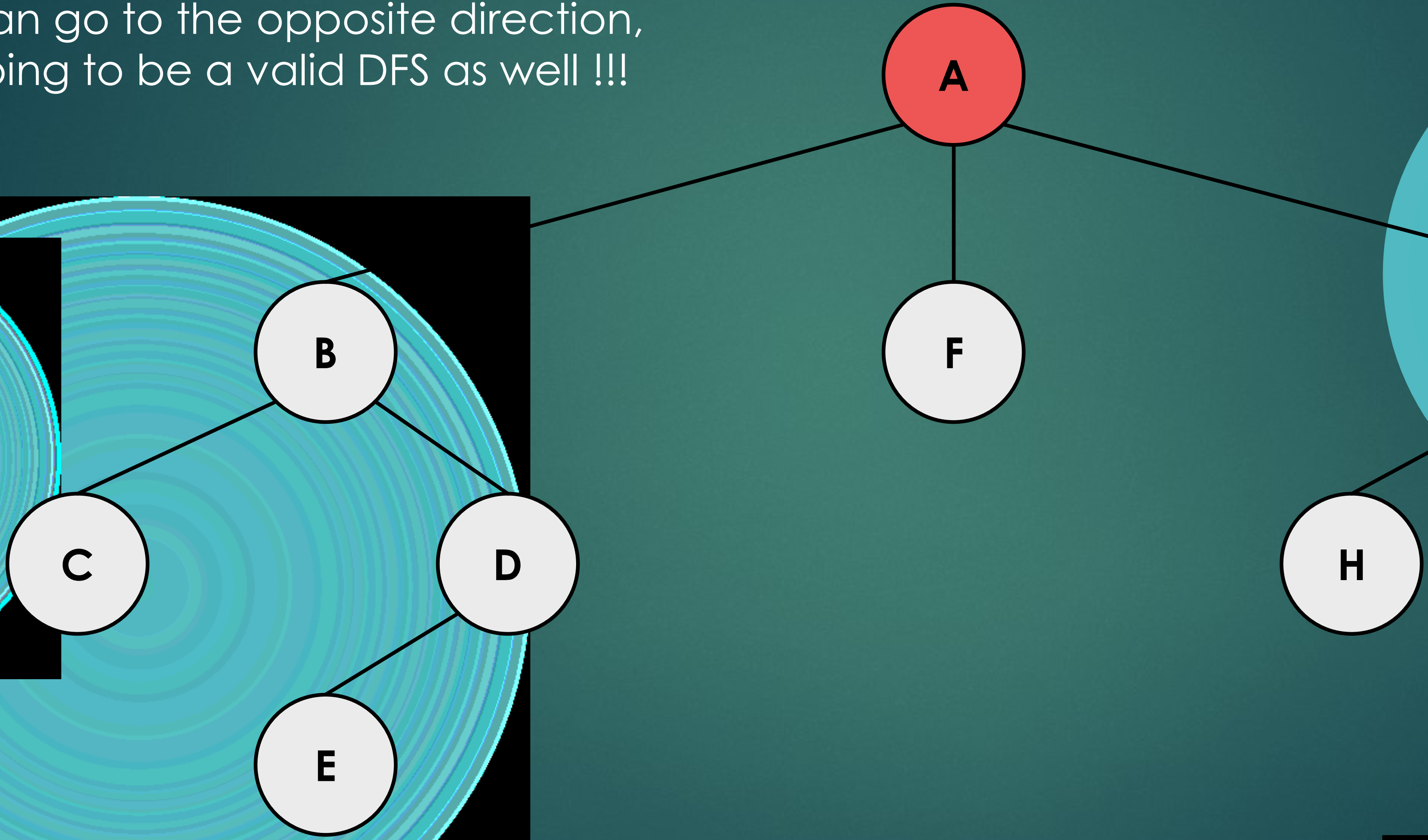
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



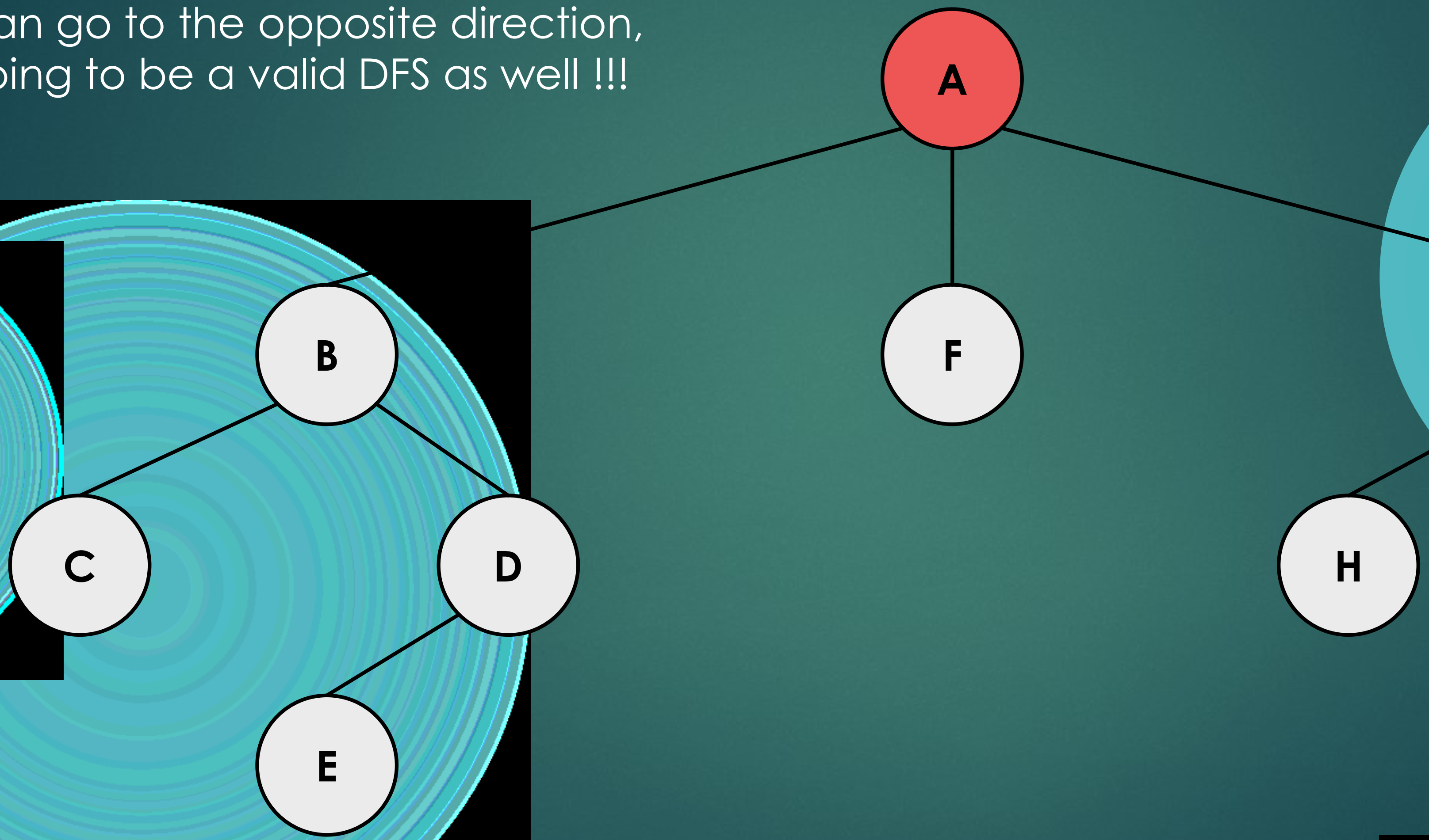
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



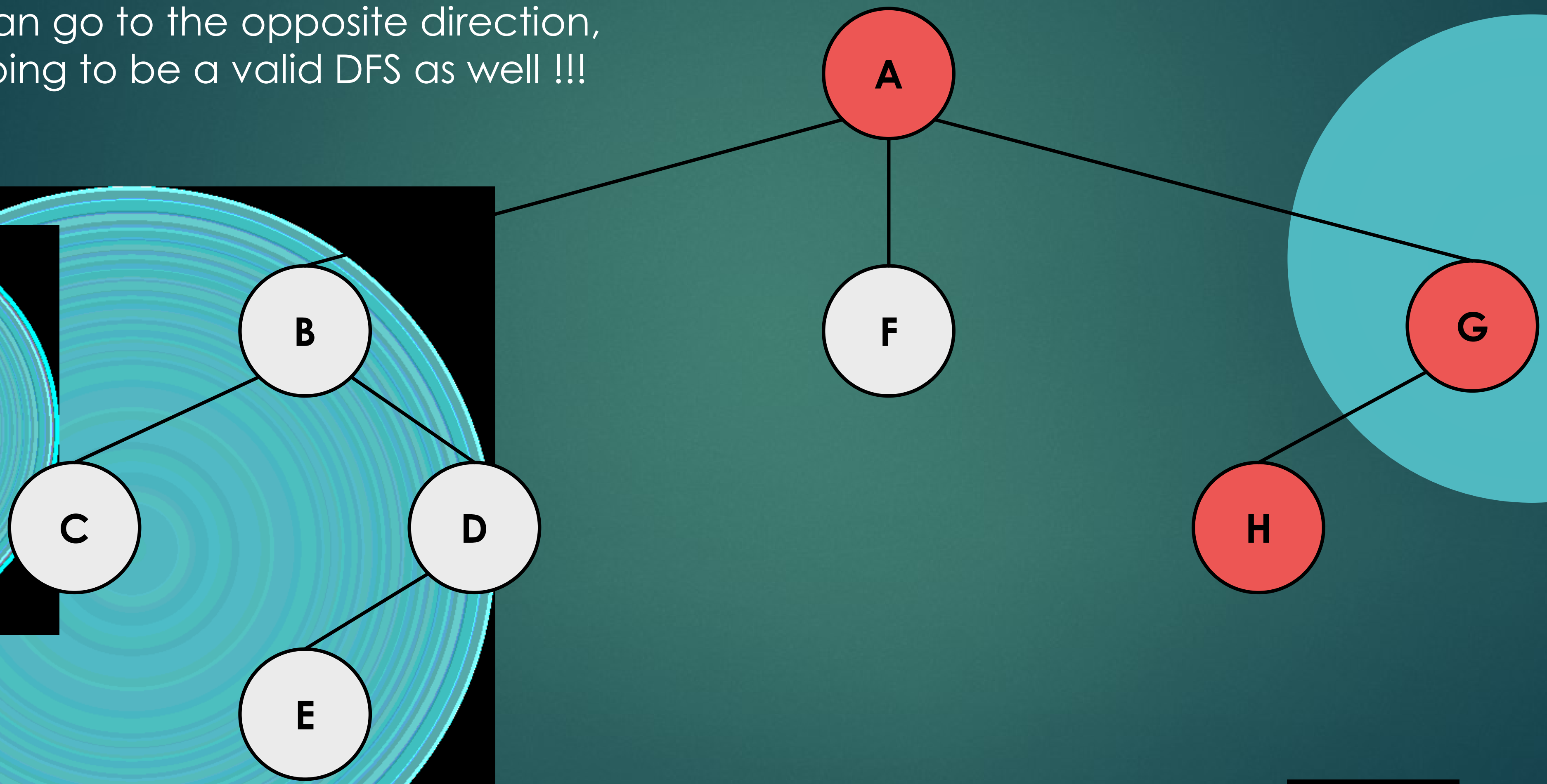
Symmetry in DFS

We can go to the opposite direction, it is going to be a valid DFS as well !!!



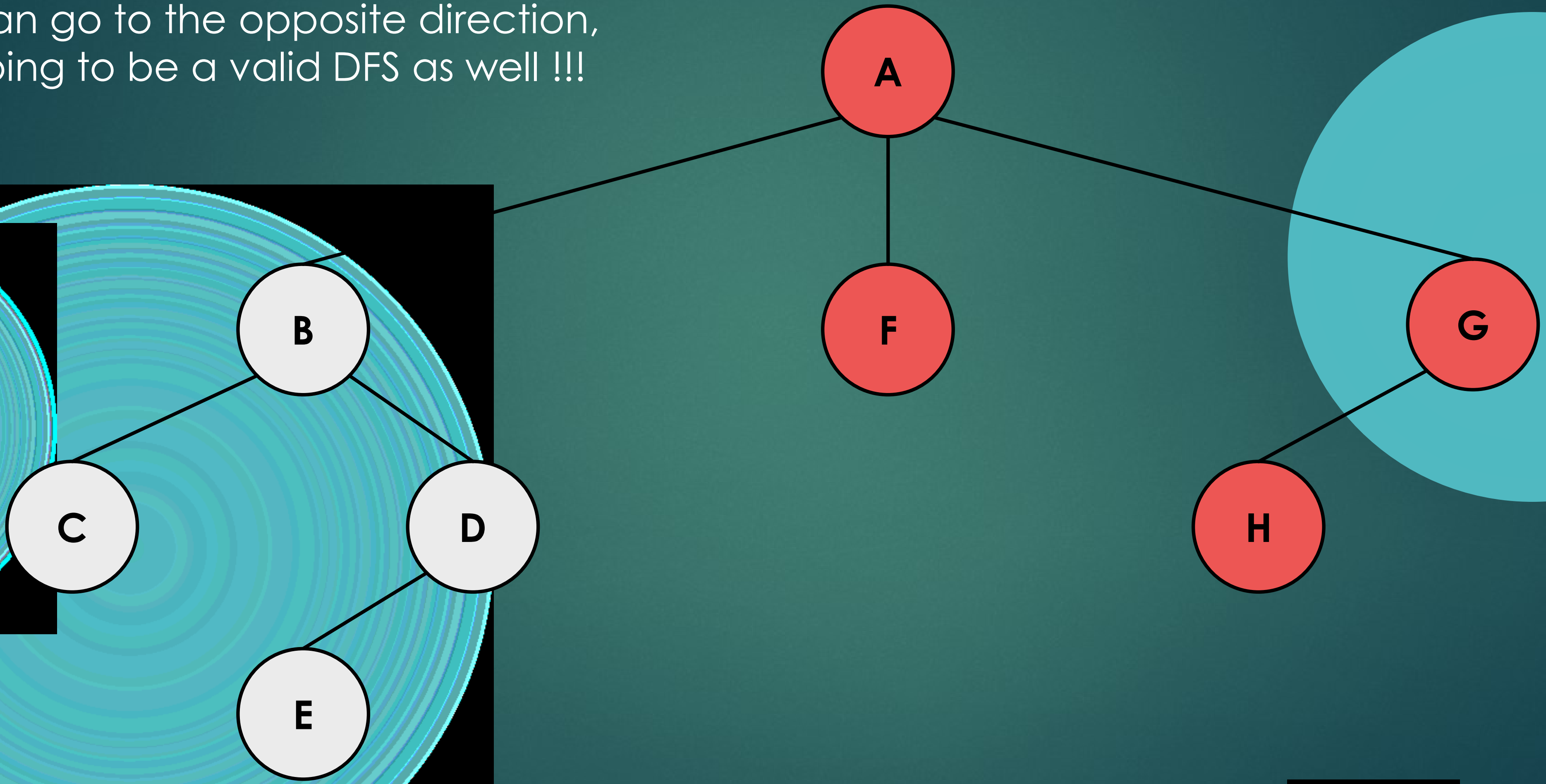
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



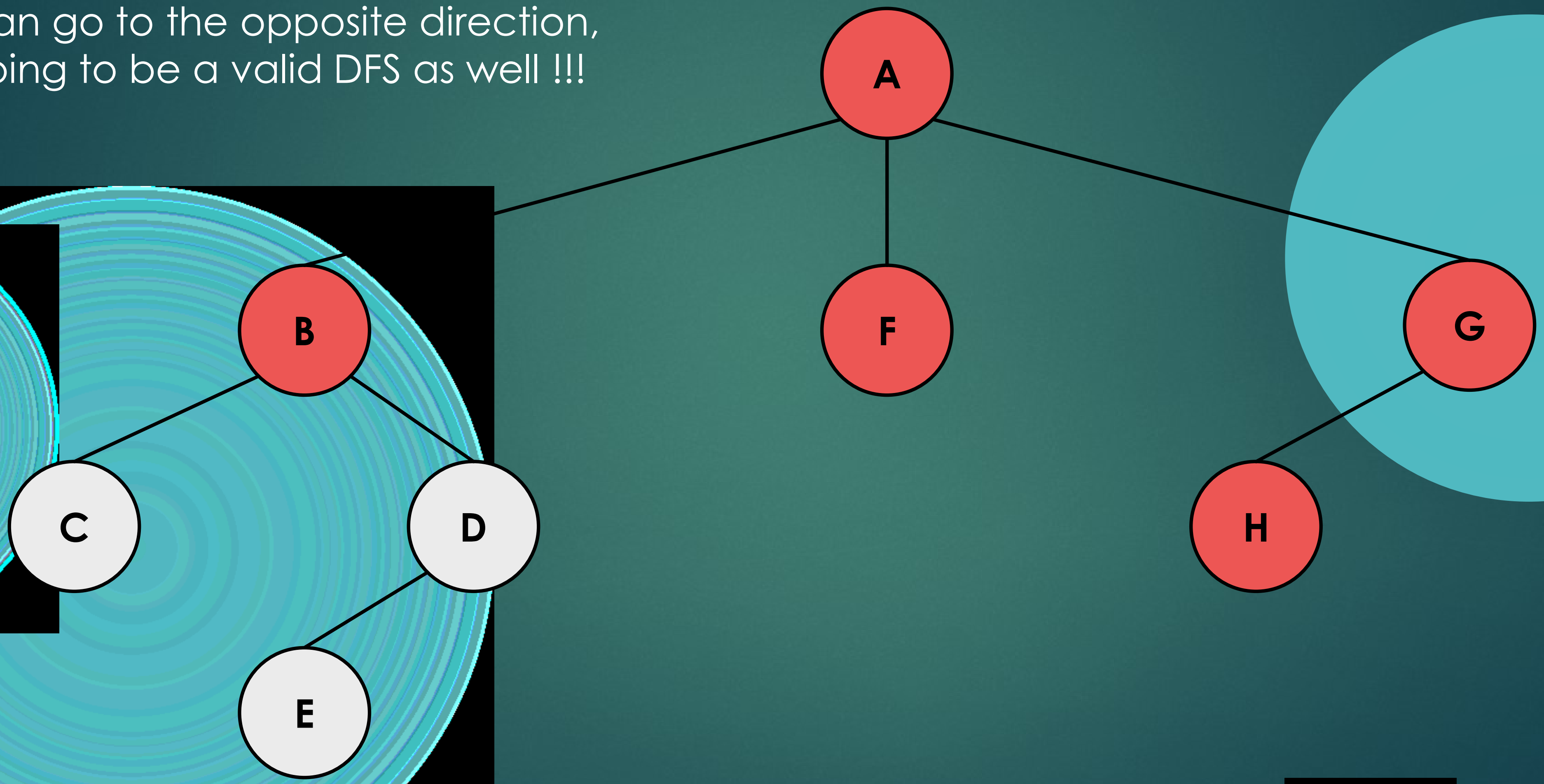
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



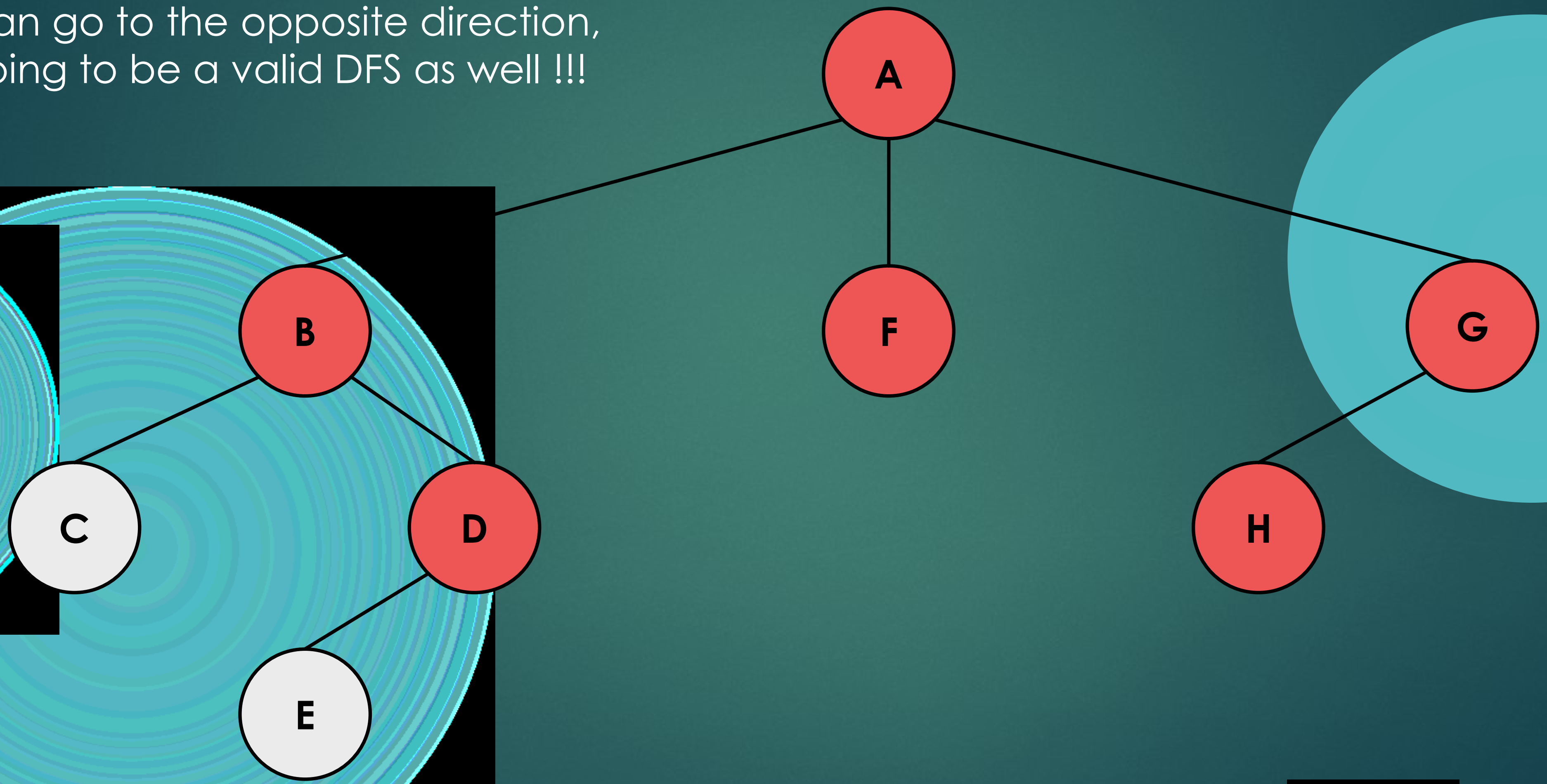
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



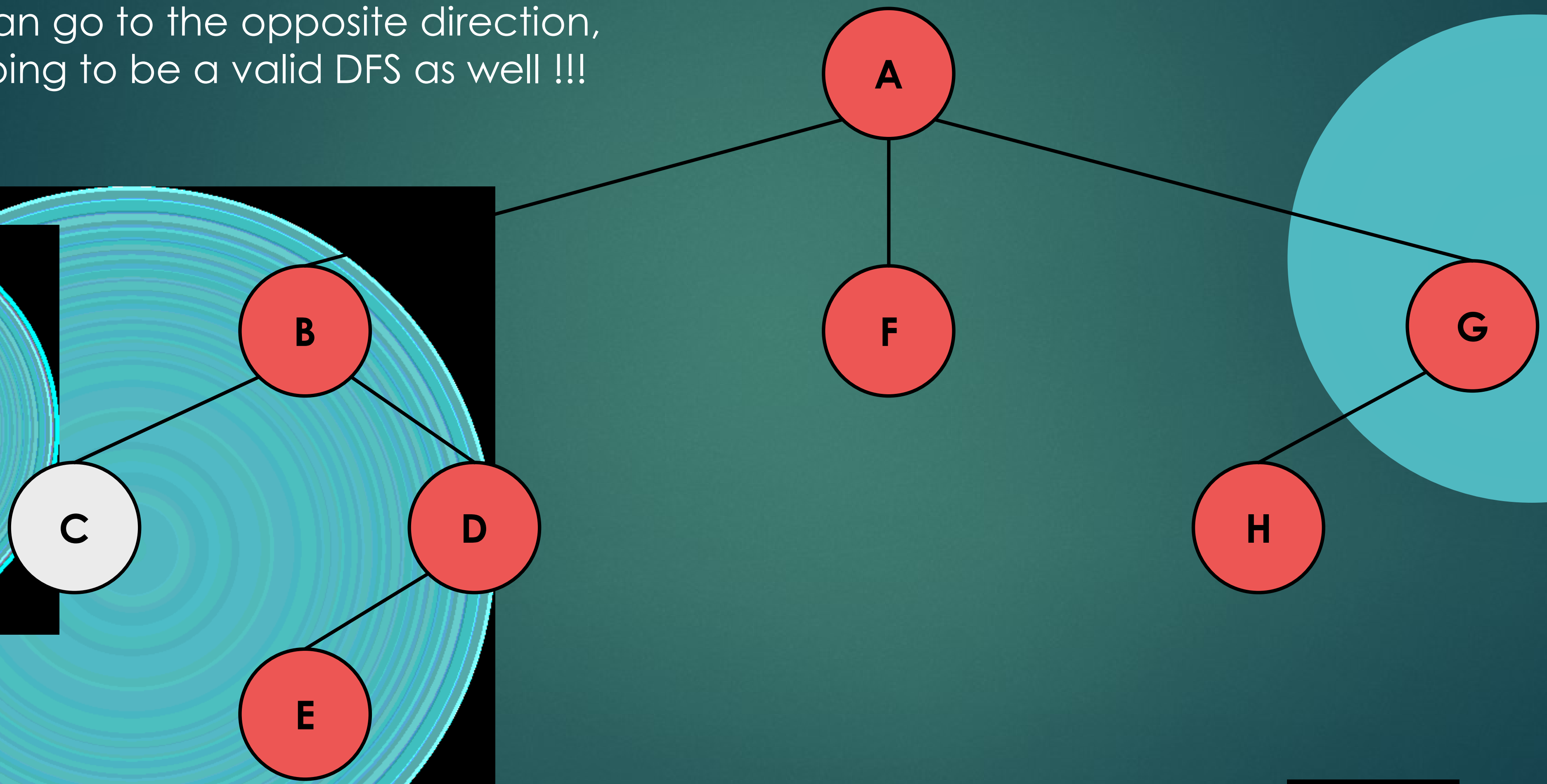
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



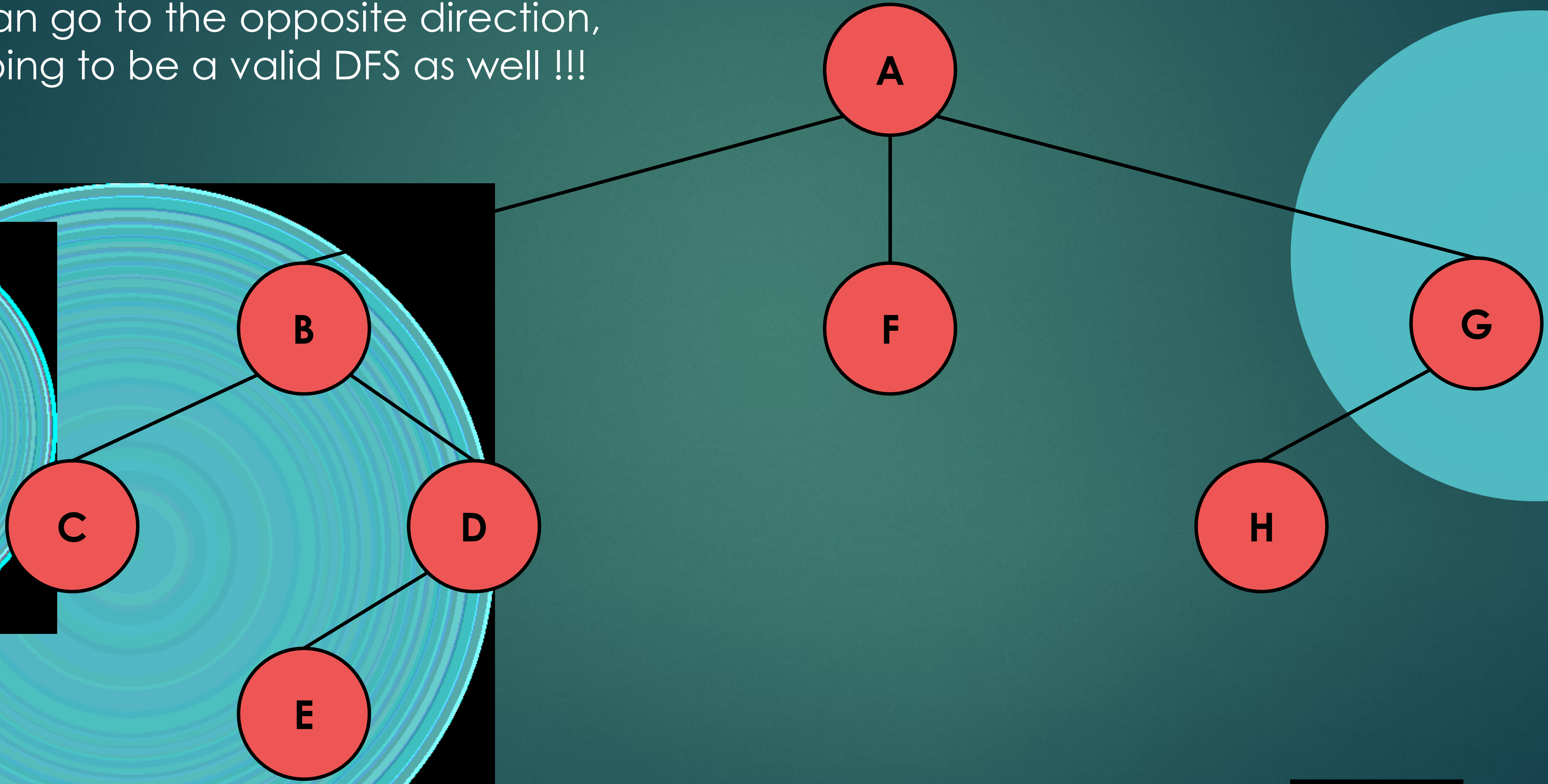
Symmetry in DFS

We can go to the opposite direction, it is going to be a valid DFS as well !!!



Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!

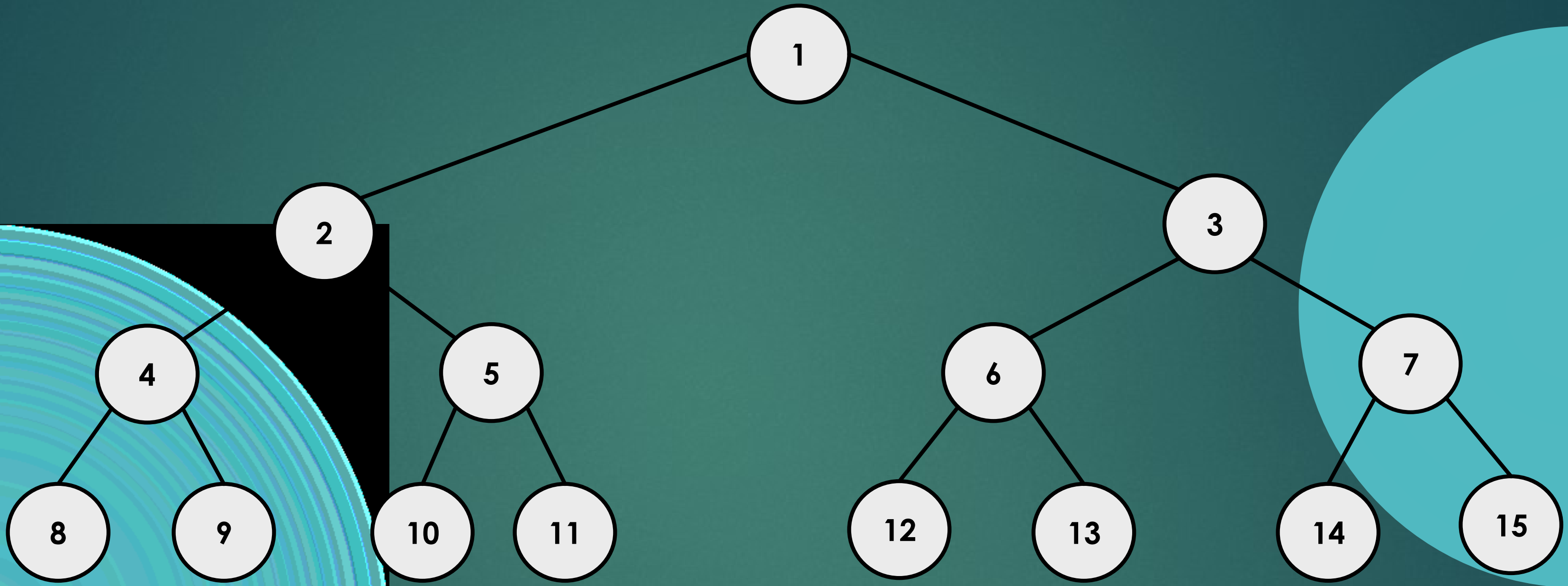


DFS IMPLEMENTATION

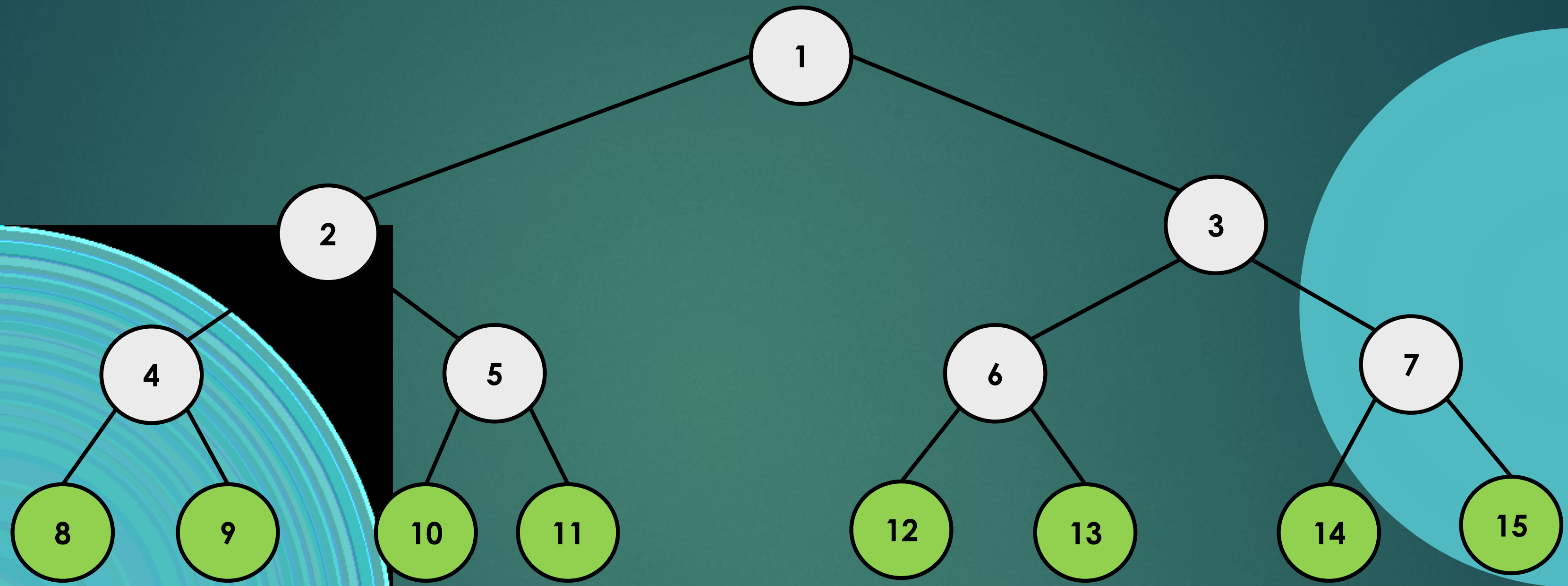
Memory Complexity

BFS VS DFS

Memory complexity: BFS vs DFS



Memory complexity: BFS vs DFS

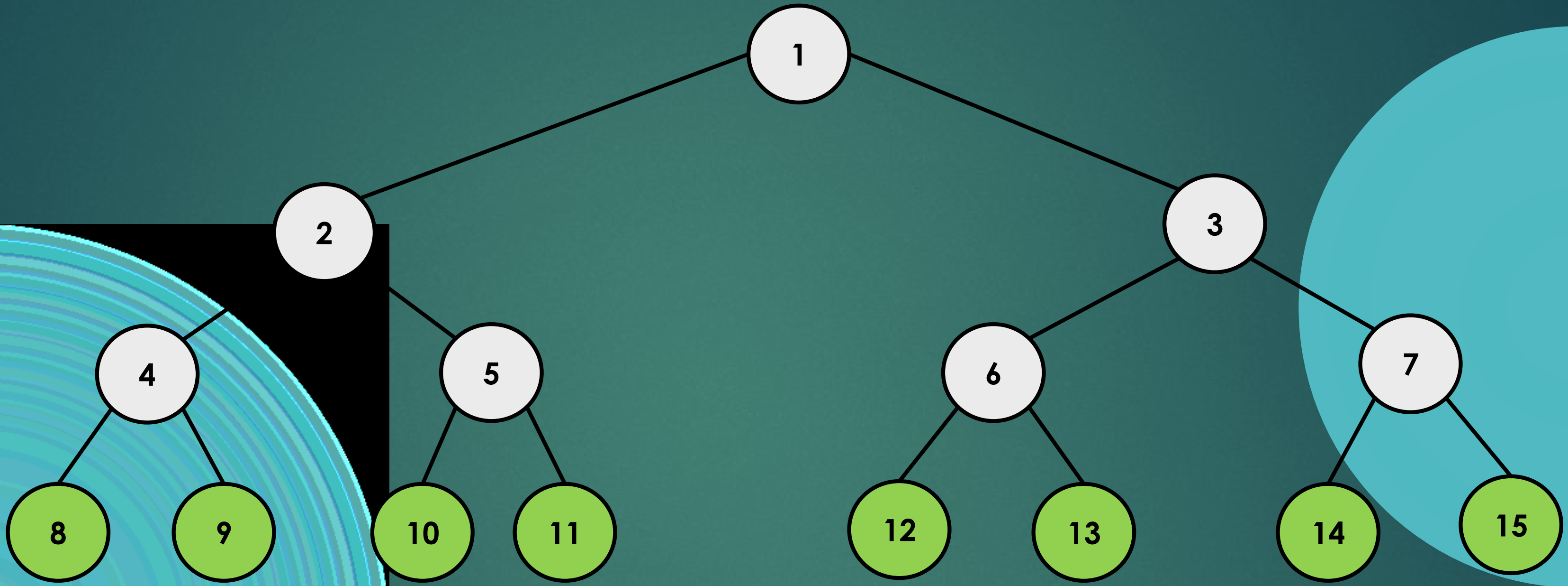


What does it mean?

At the leaves → if we have N items stored in the balanced tree

→ then there will be N/2 leave nodes

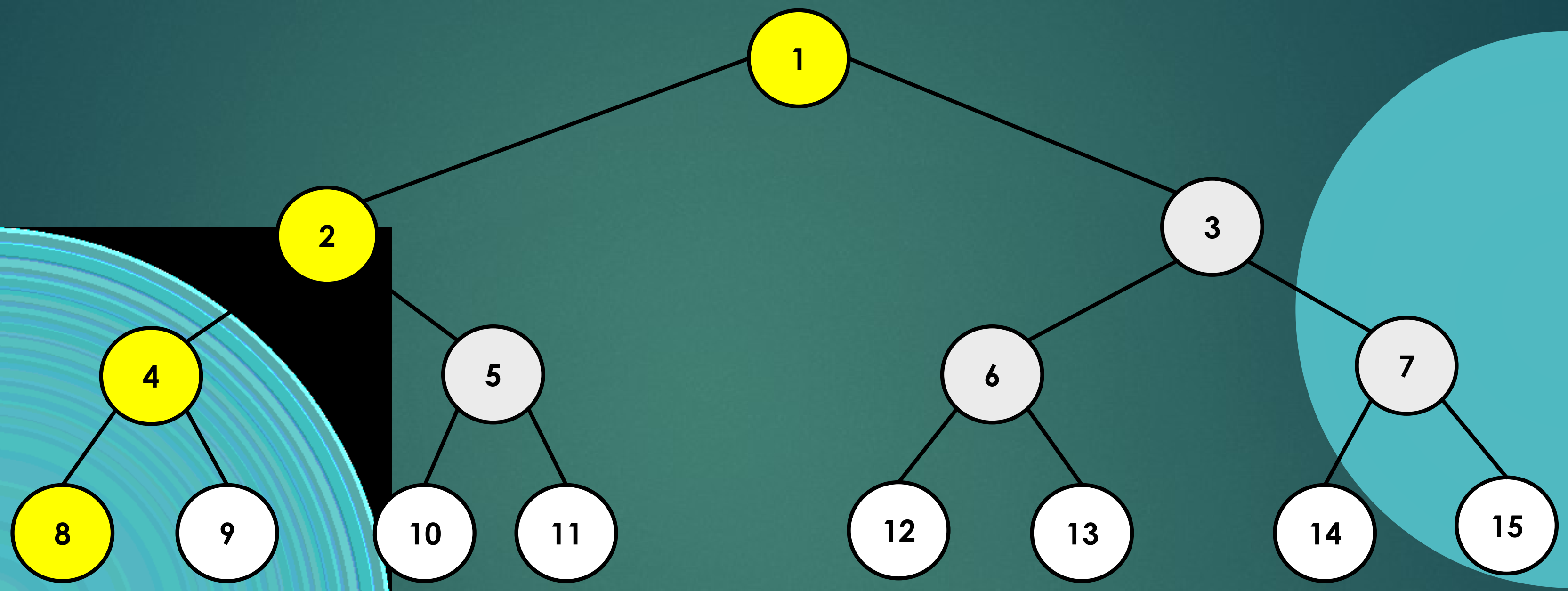
Memory complexity: BFS vs DFS



What does it mean?
At the leaves → if we have N items stored in the balanced tree
→ then there will be N/2 leave nodes

So we have to store **O(N)** times if we want to traverse a tree that contains N items!!!

Memory complexity: BFS vs DFS



Here we have to backtrack (pop item from stack)

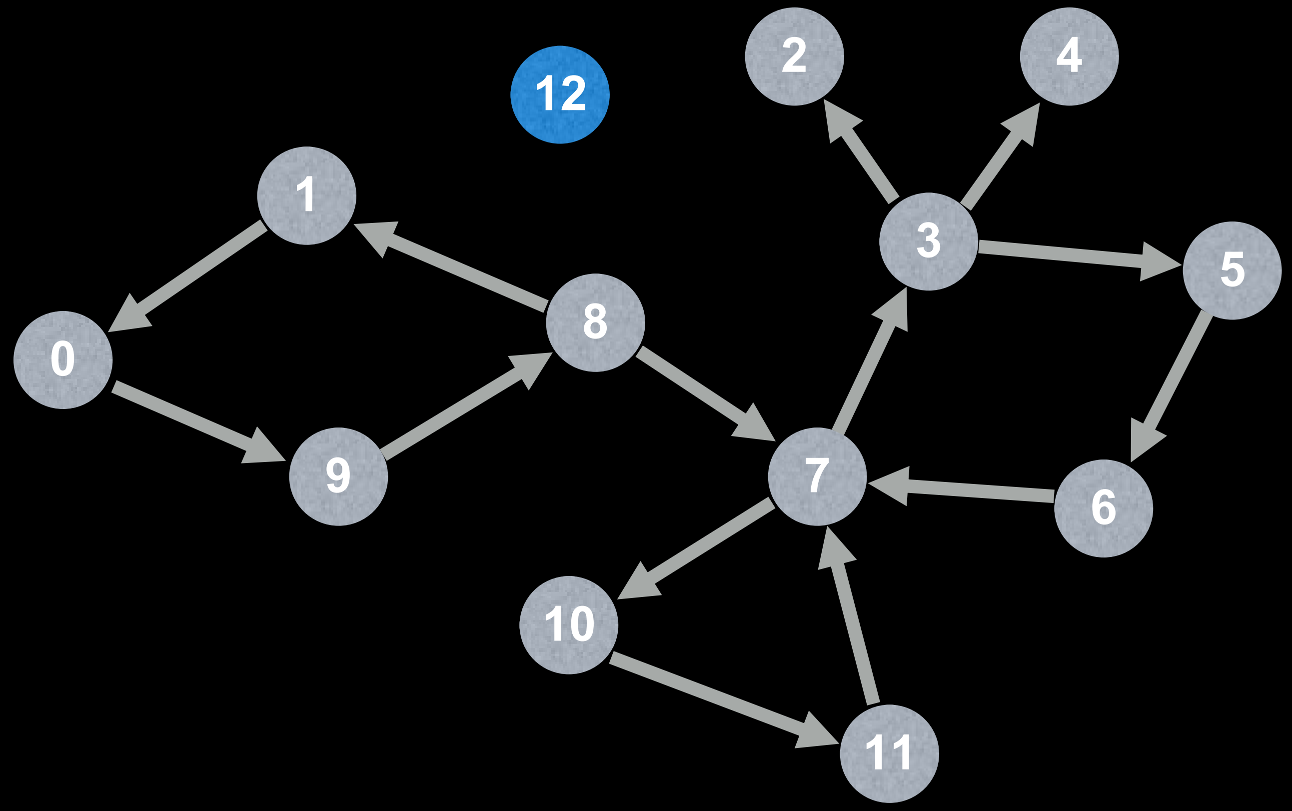
- We just have to store as many items in the stack as the height of the tree
- Which is log N !!!
- The memory complexity will be **O(logN)**

Memory complexity: BFS vs DFS

- Breadth-First Search: **O(N)**
- Depth-First Search: **O(logN)**

- Depth-First Search is **preferred** most of the time
- There may be some situations where BFS is better
  - Artificial intelligence
  - Robot movements

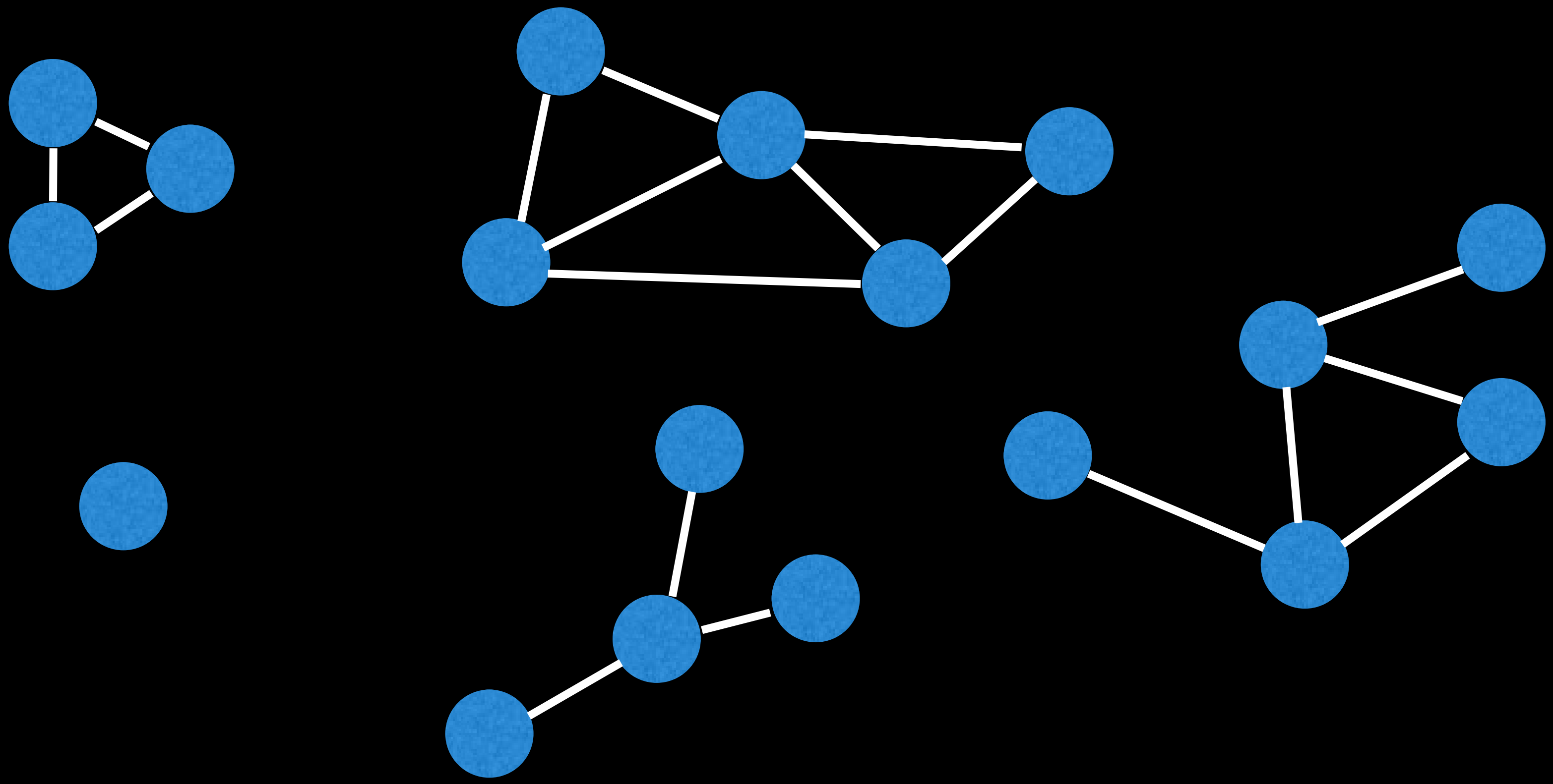
Basic DFS



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

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
    dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

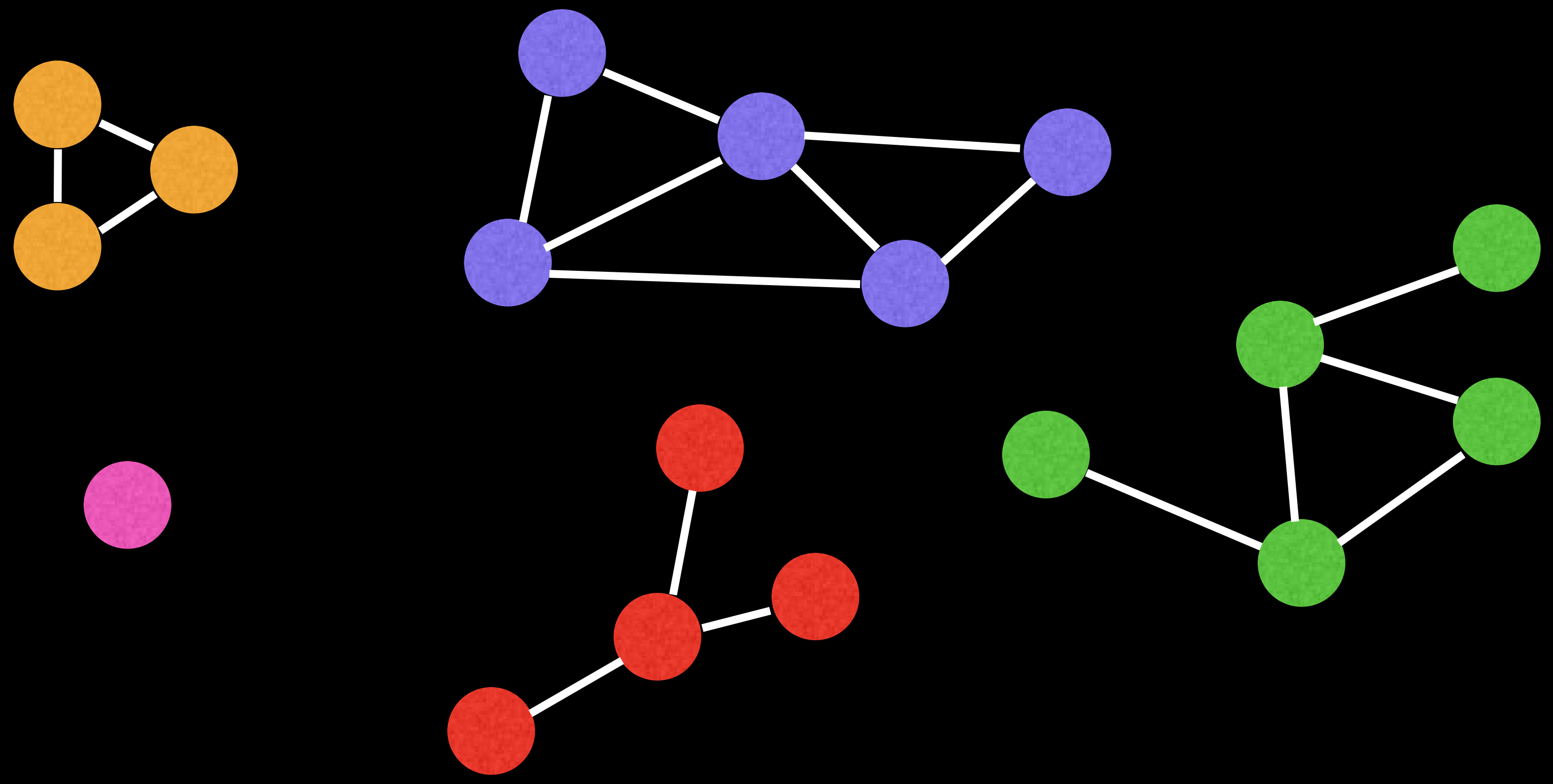
Connected Components

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



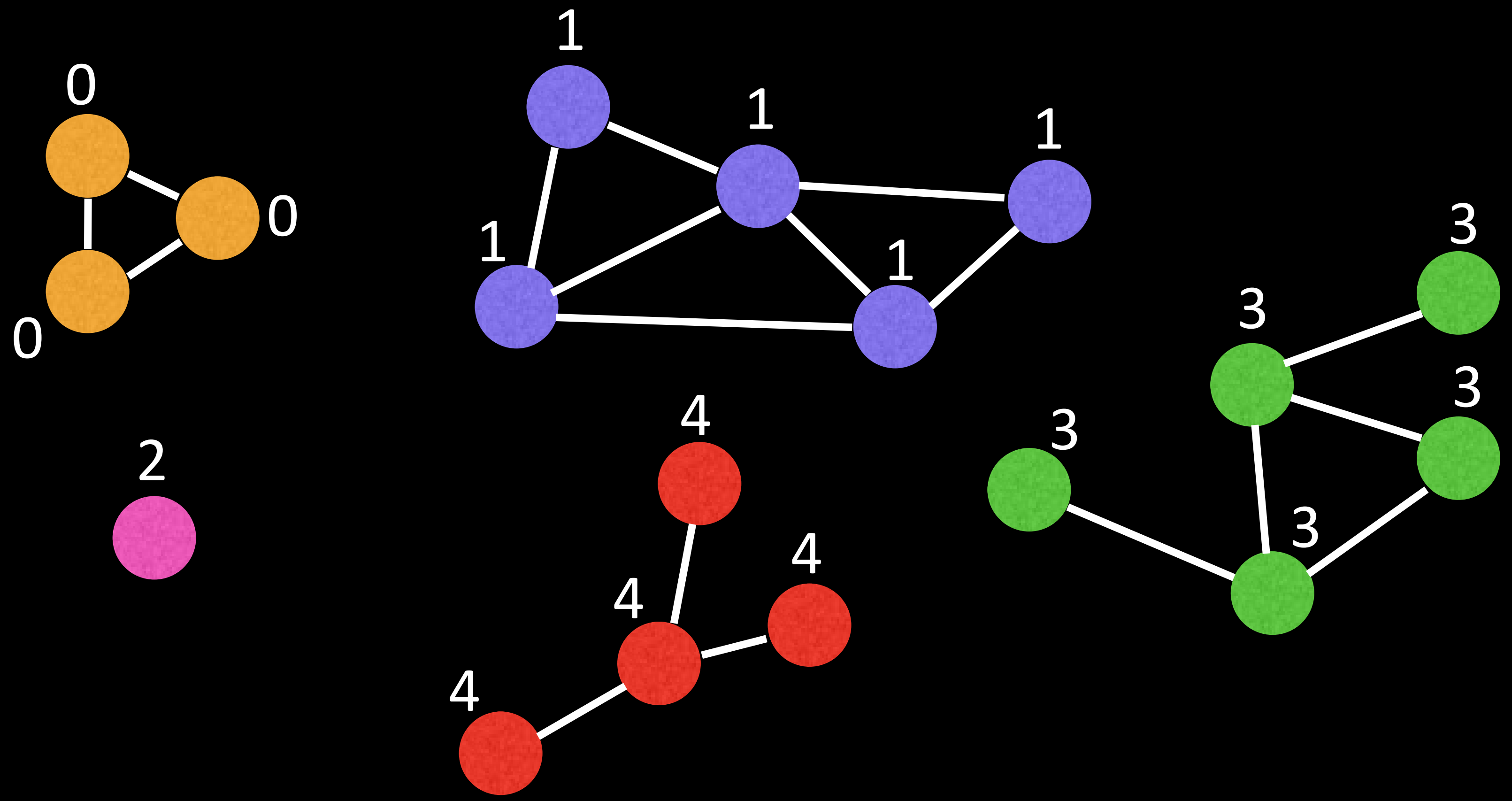
Connected Components

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

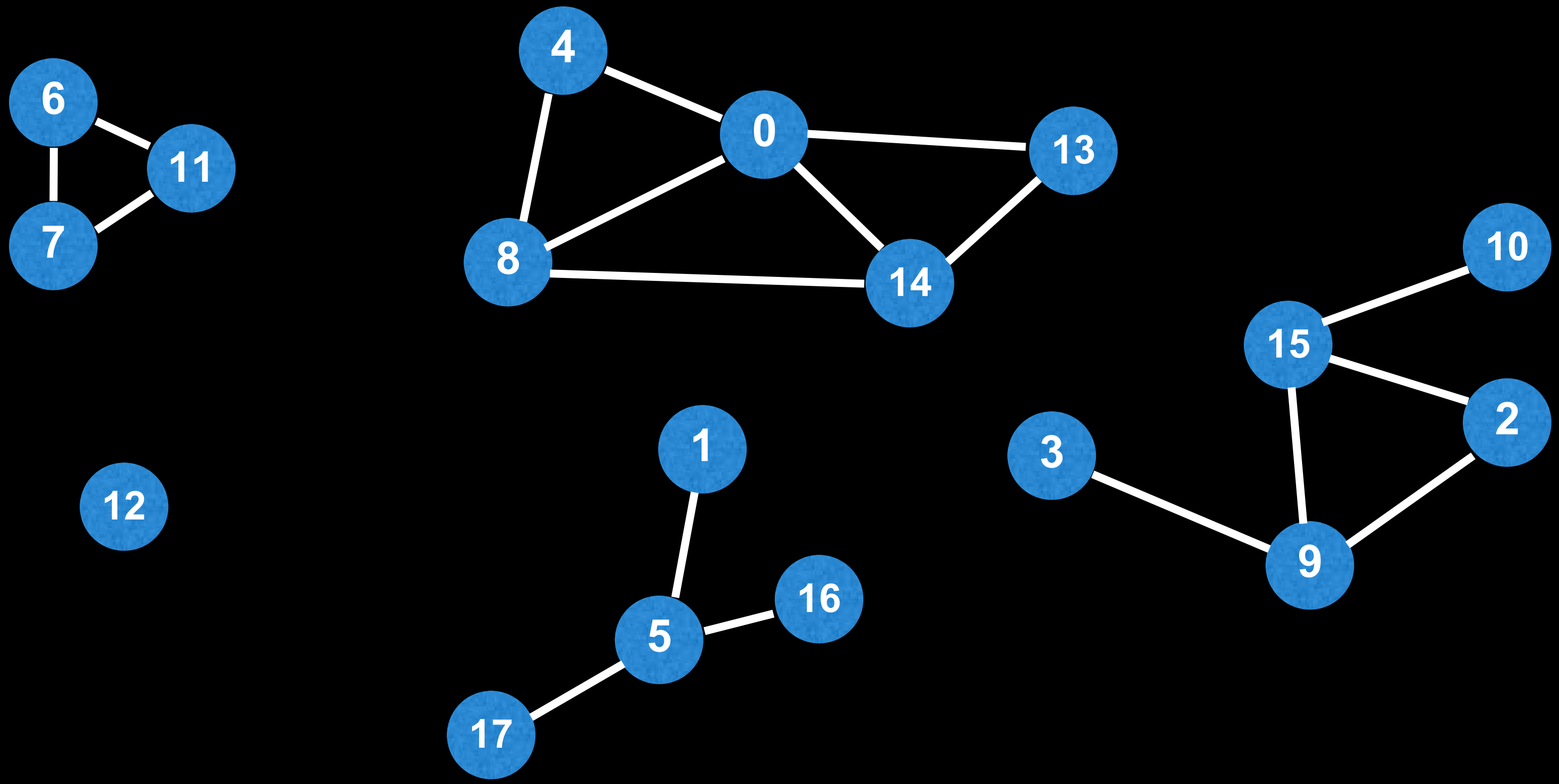


Connected Components

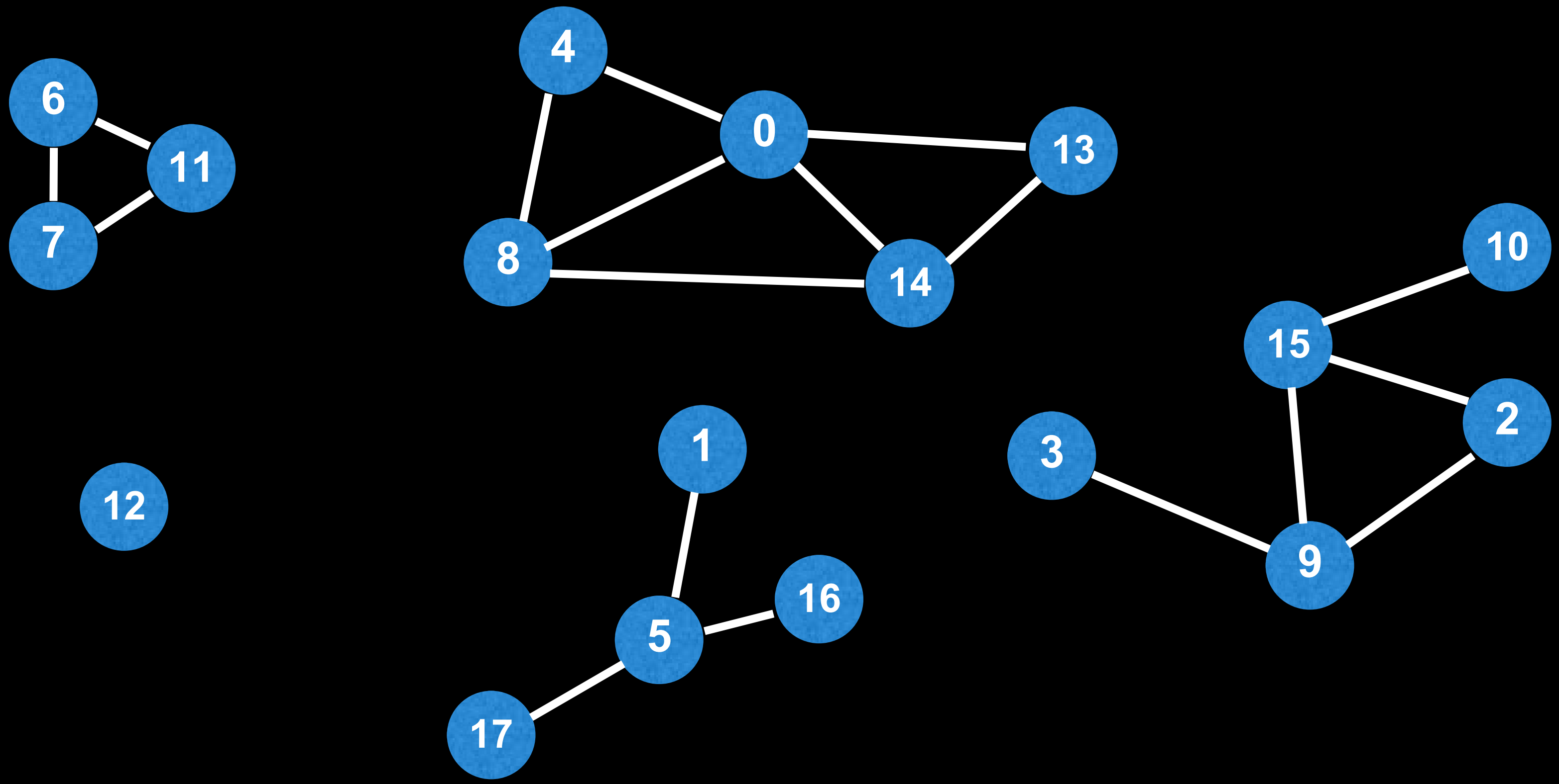
Assign an integer value to each group to be able to tell them apart.



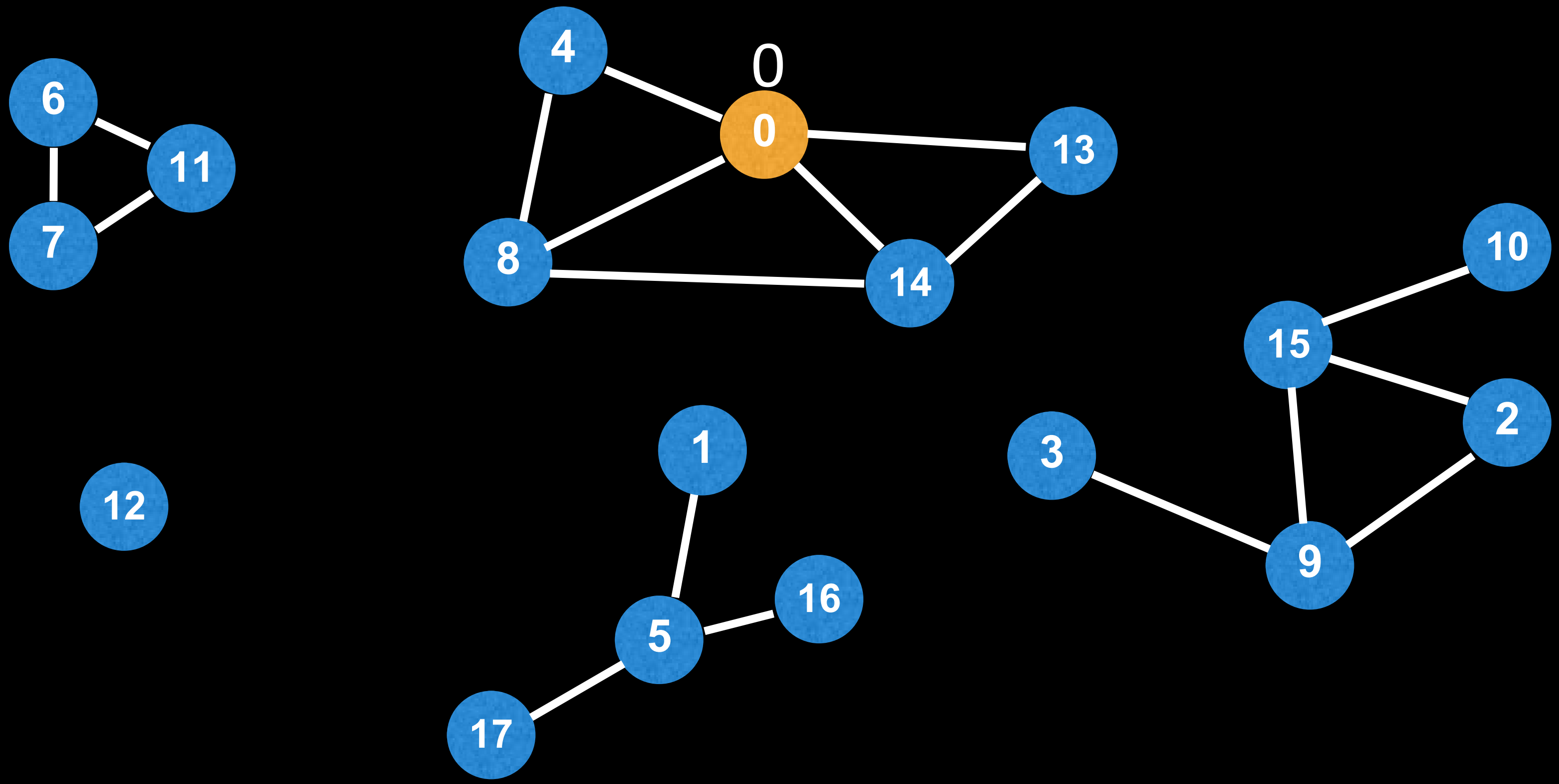
We can use a DFS to identify components. First, make sure all the nodes are labeled from [0, n) where n is the number of nodes.



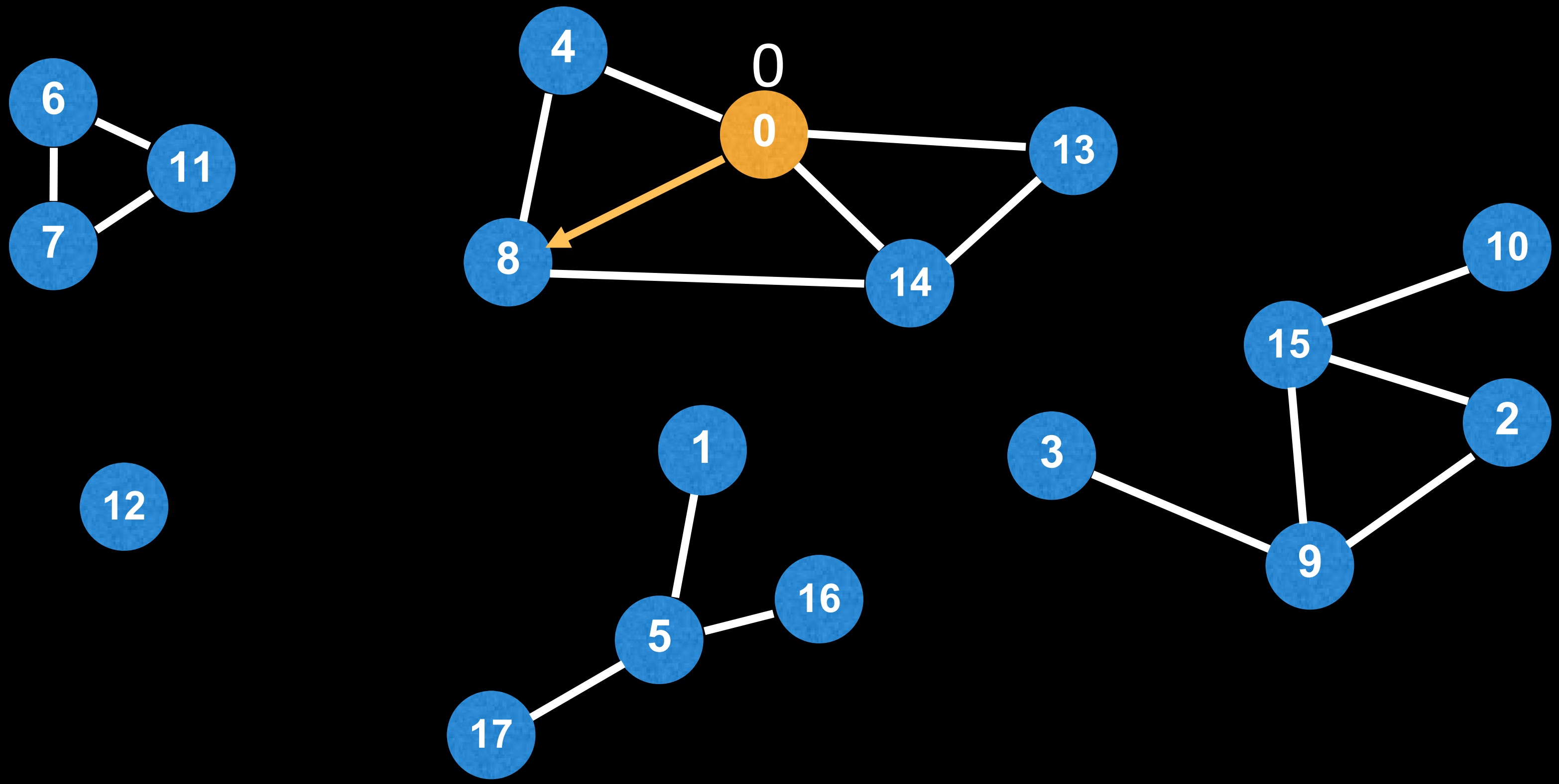
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



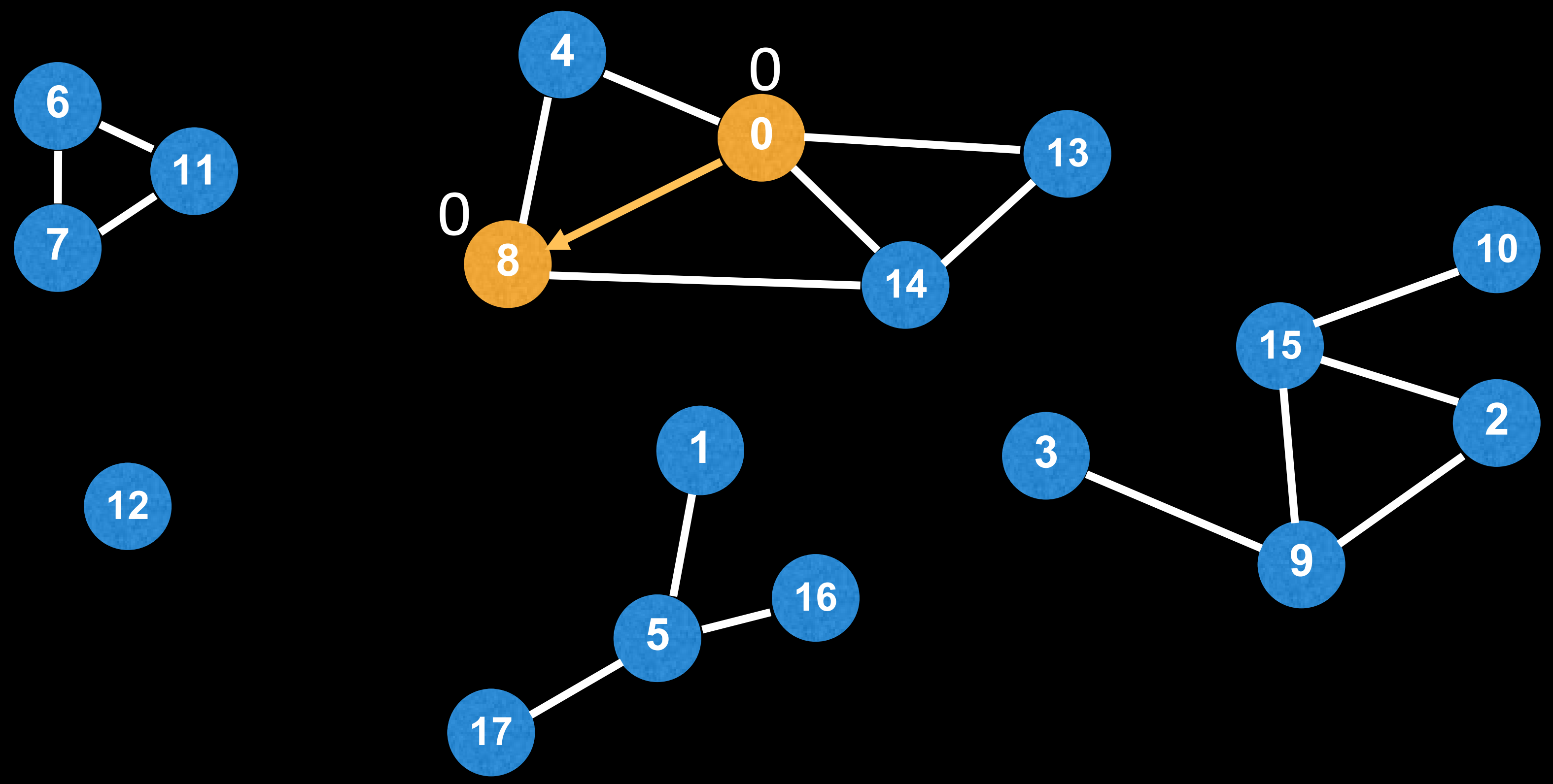
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



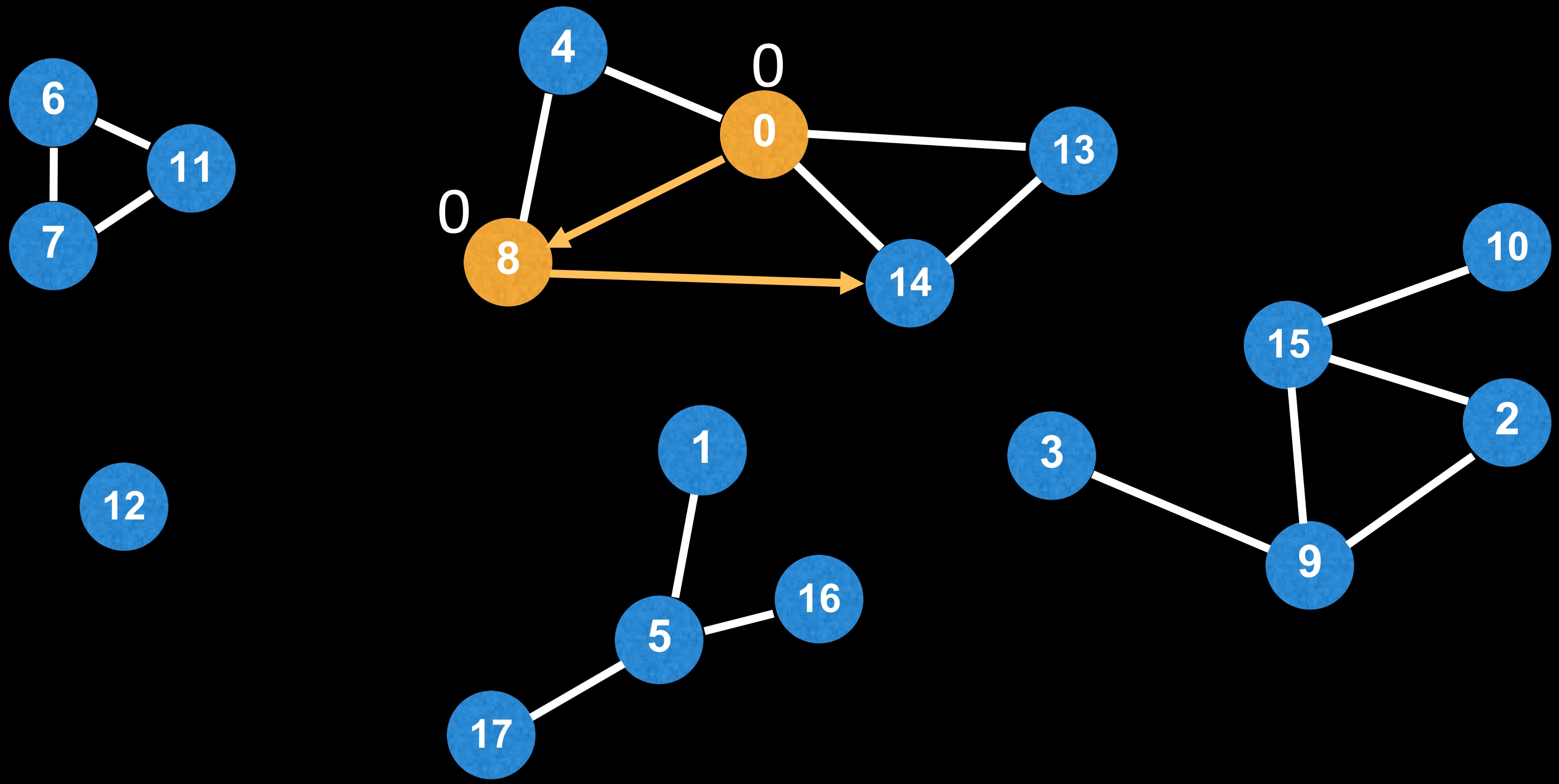
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



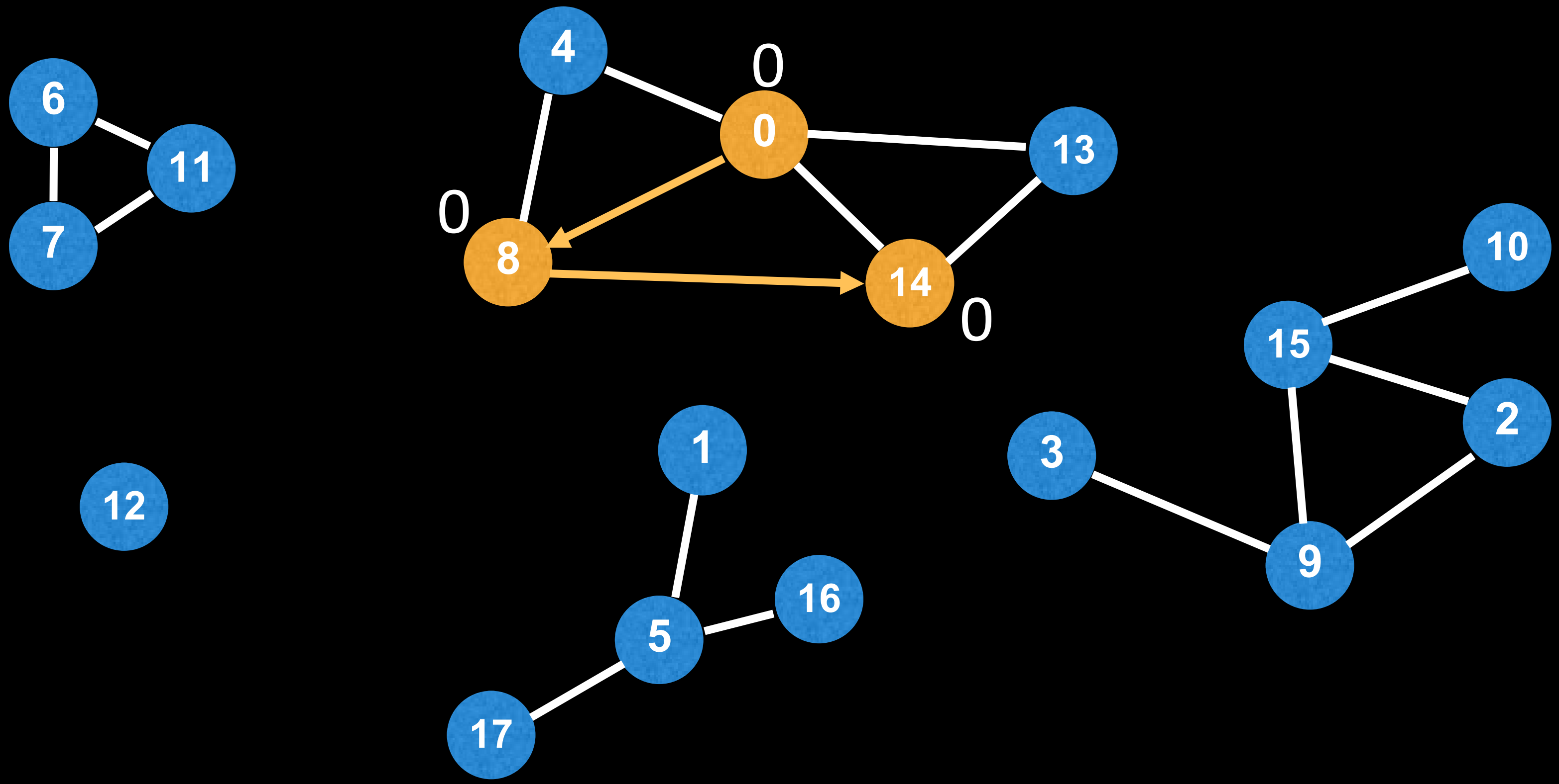
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



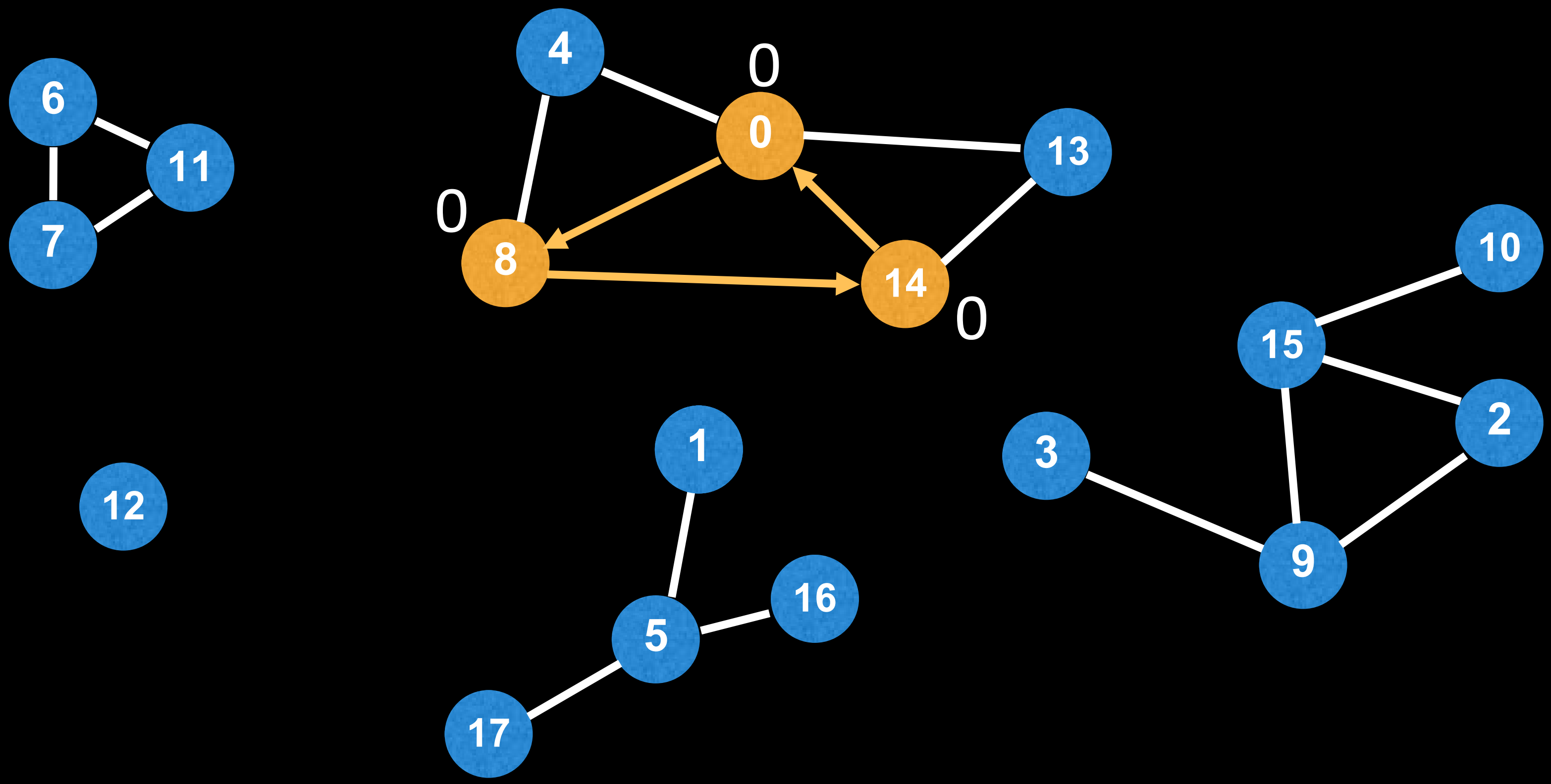
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



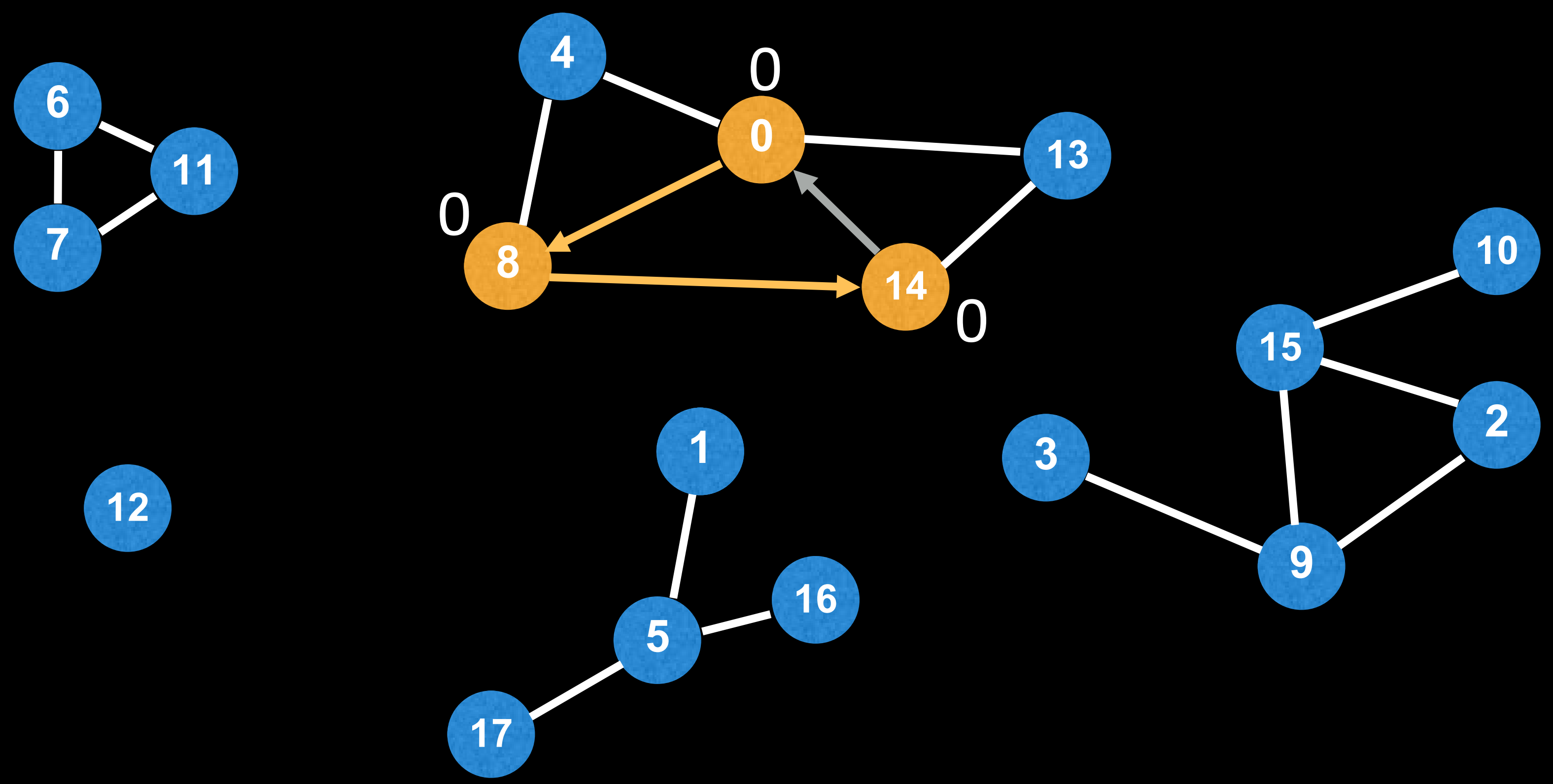
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



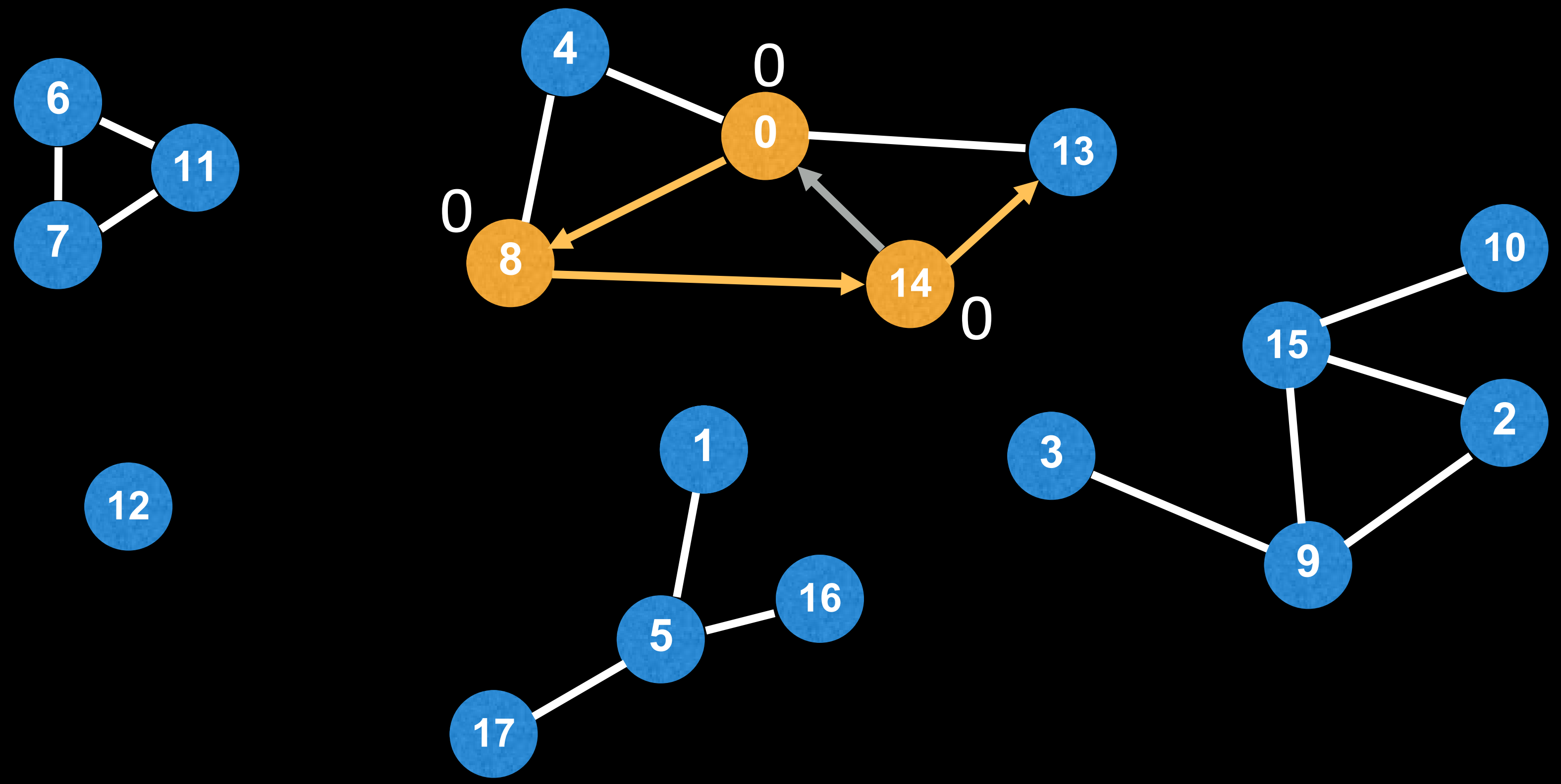
Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.



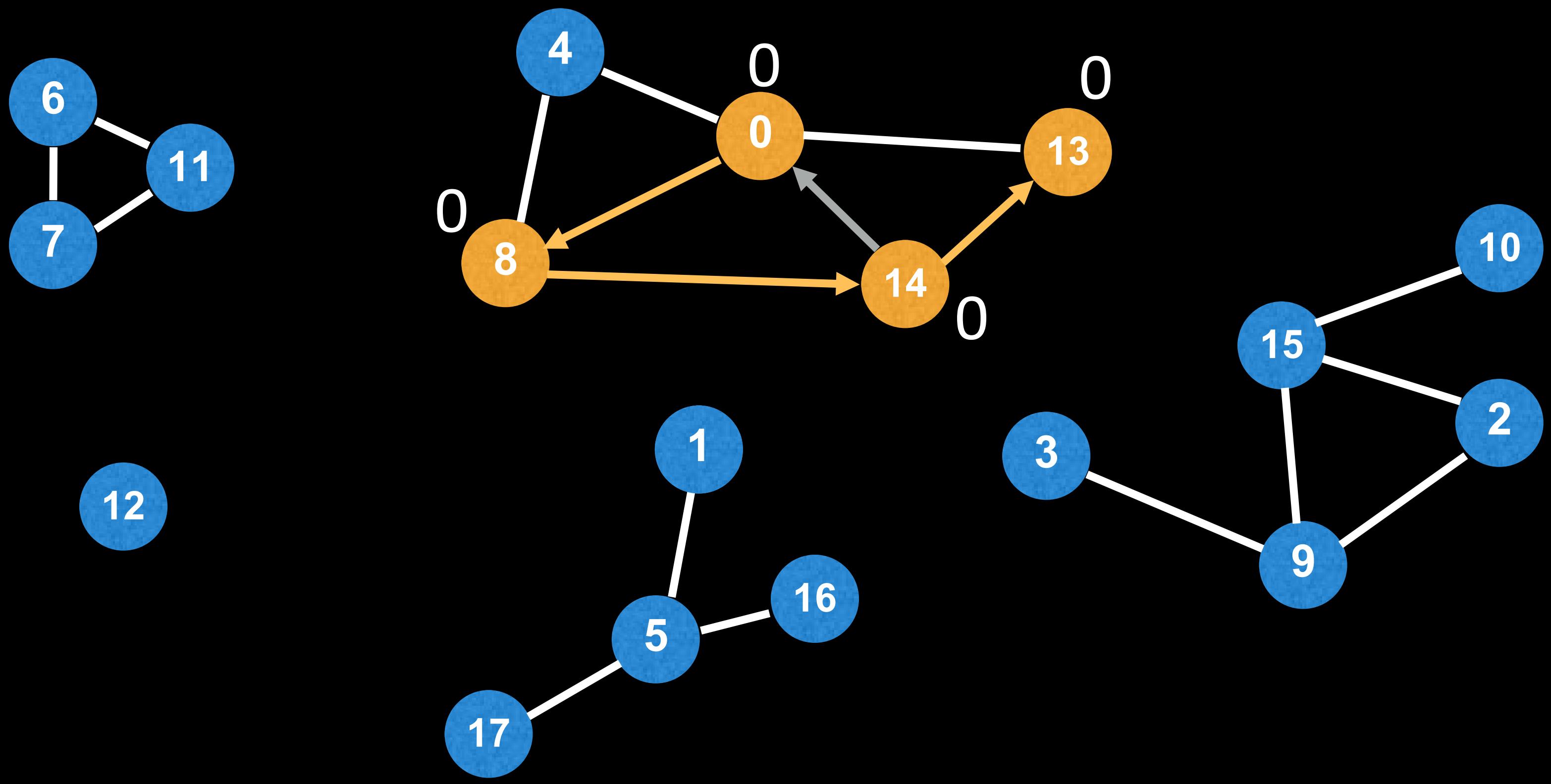
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



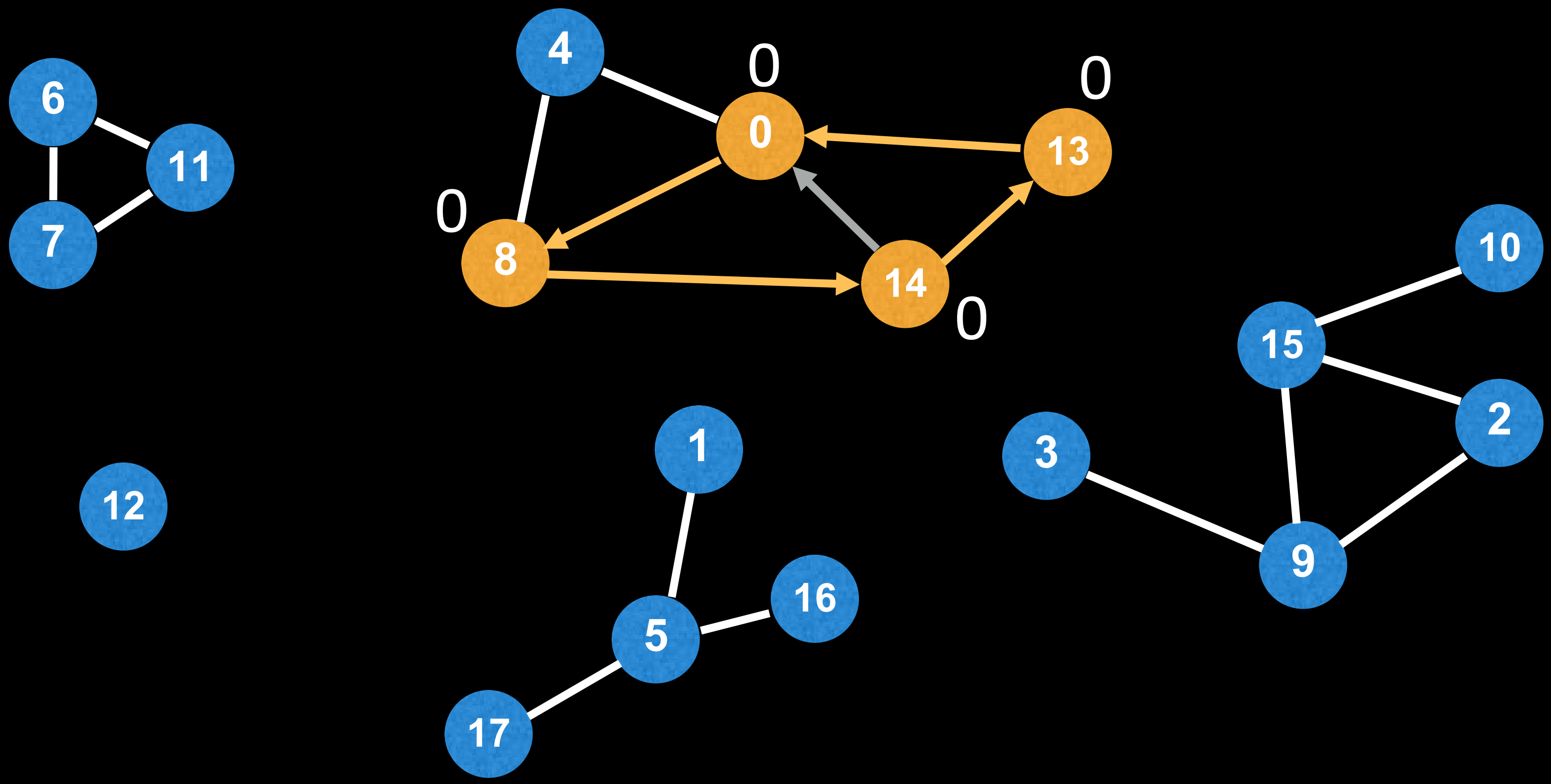
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



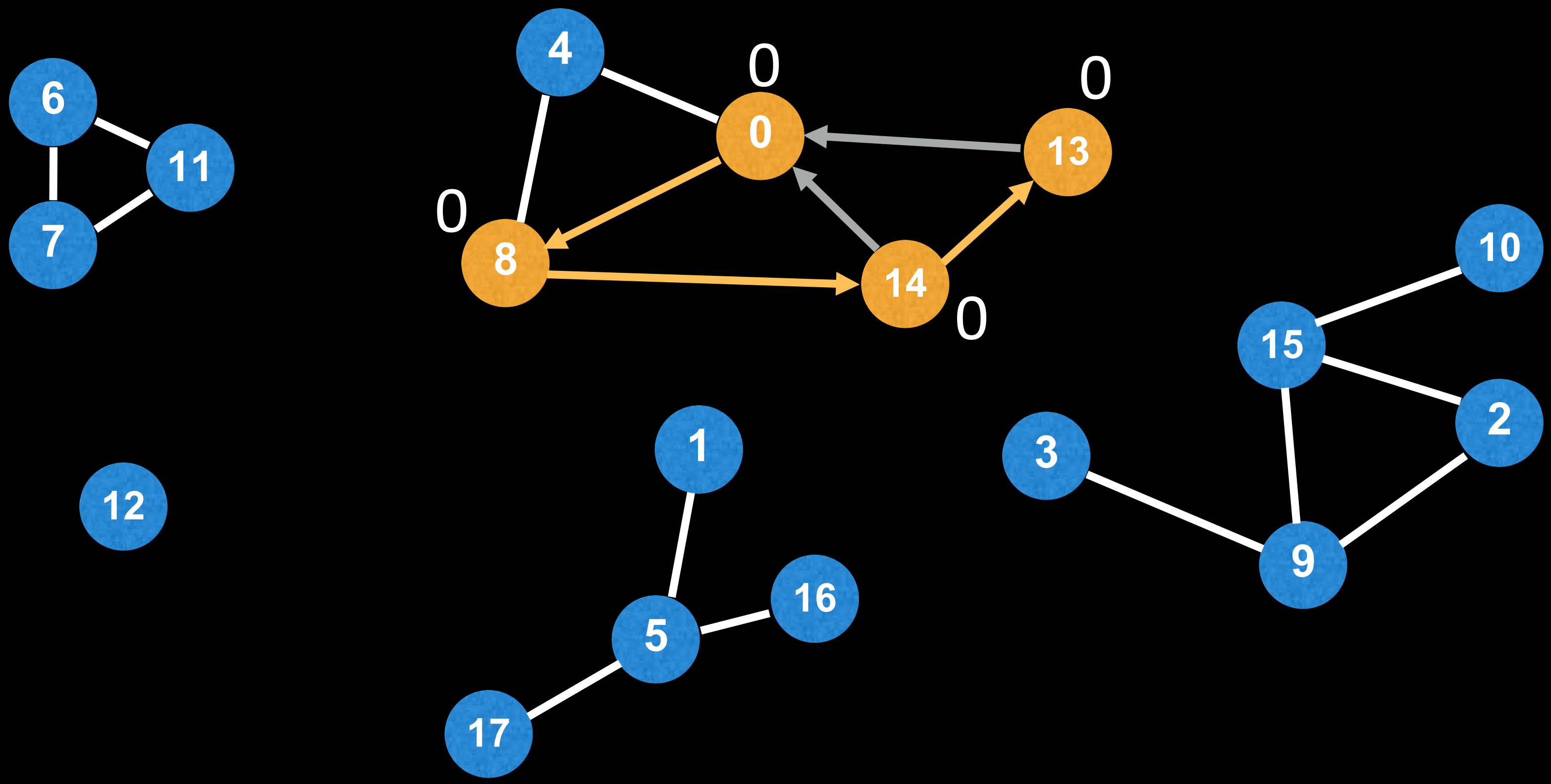
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



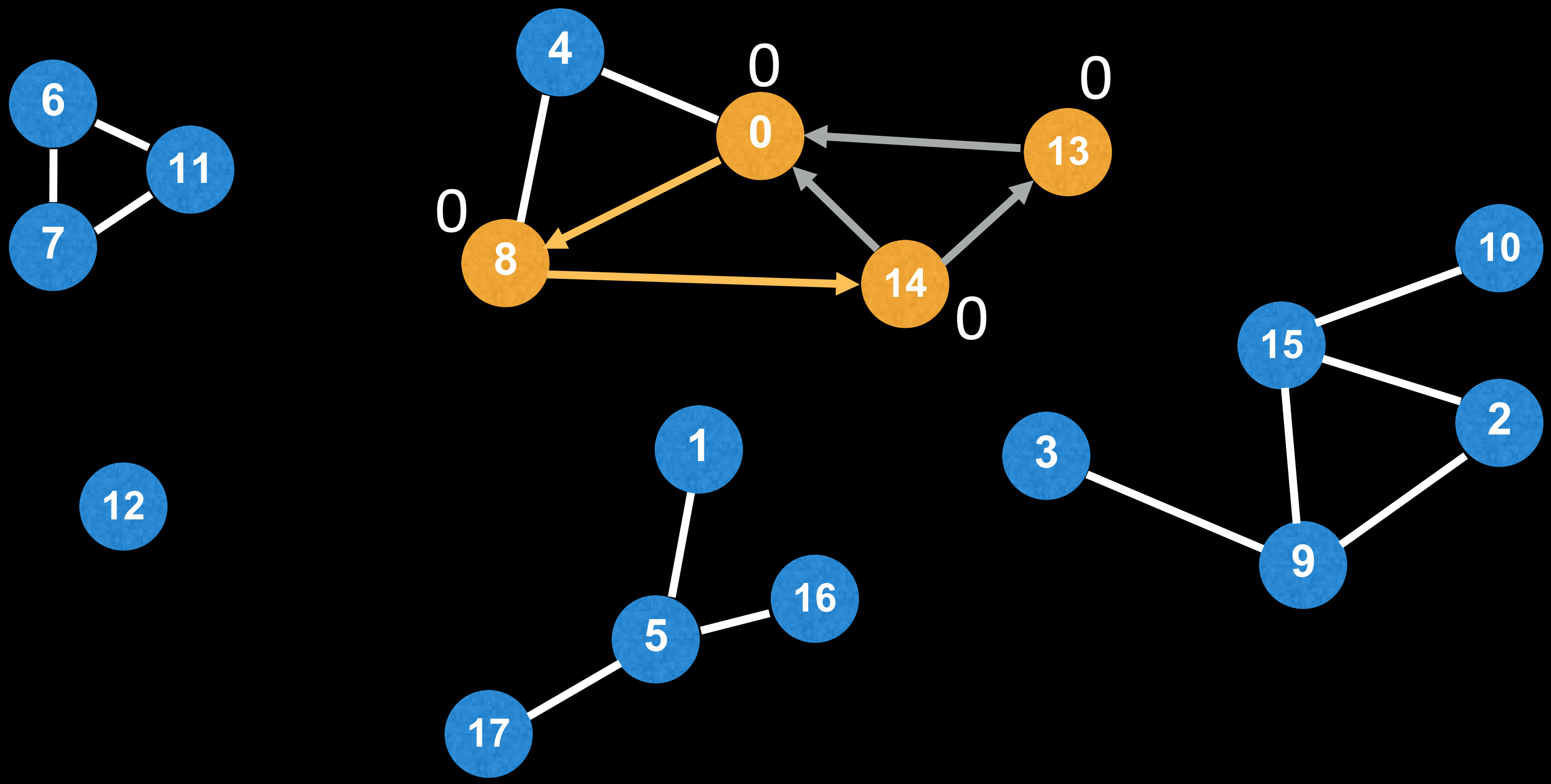
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



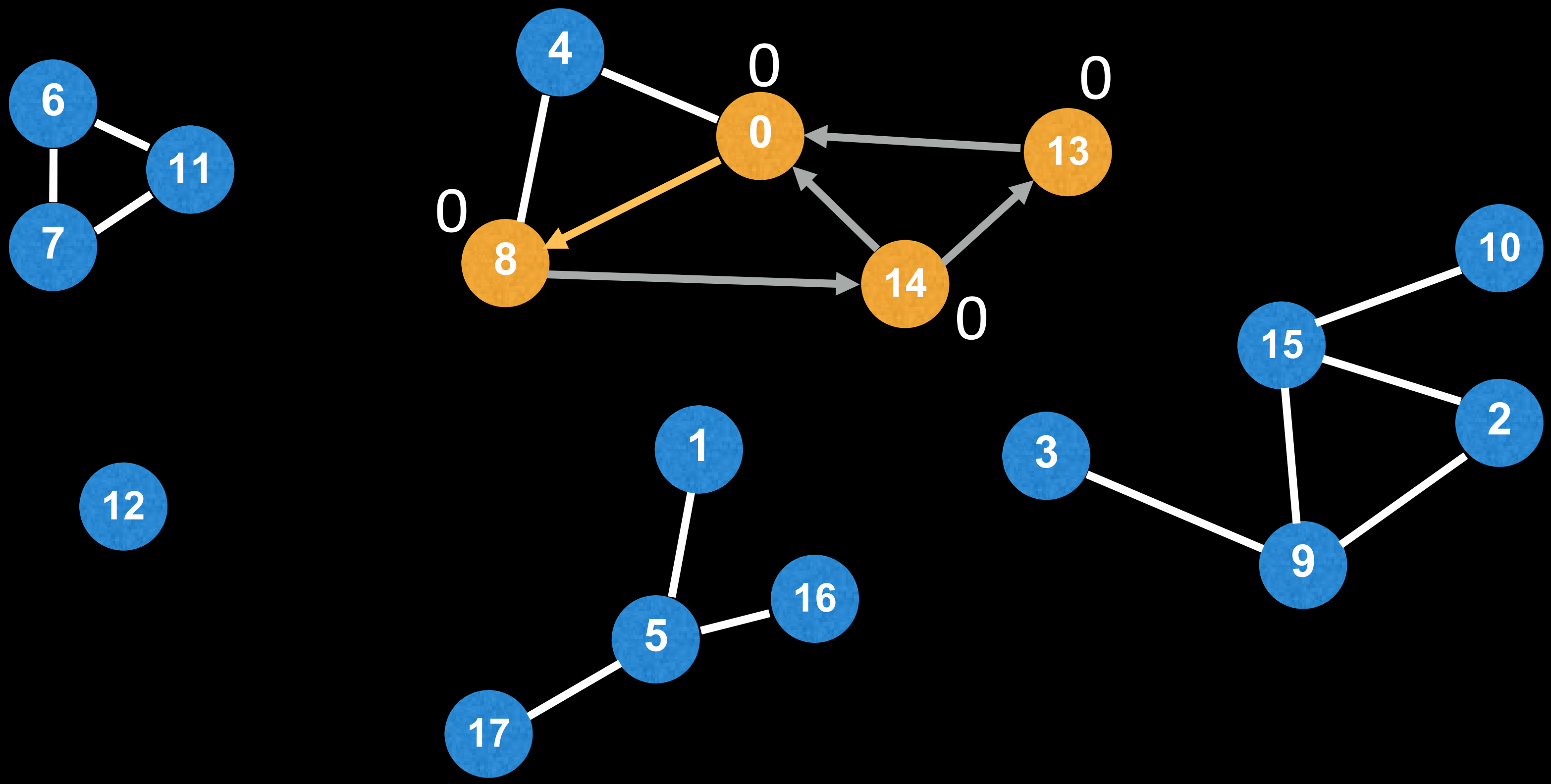
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



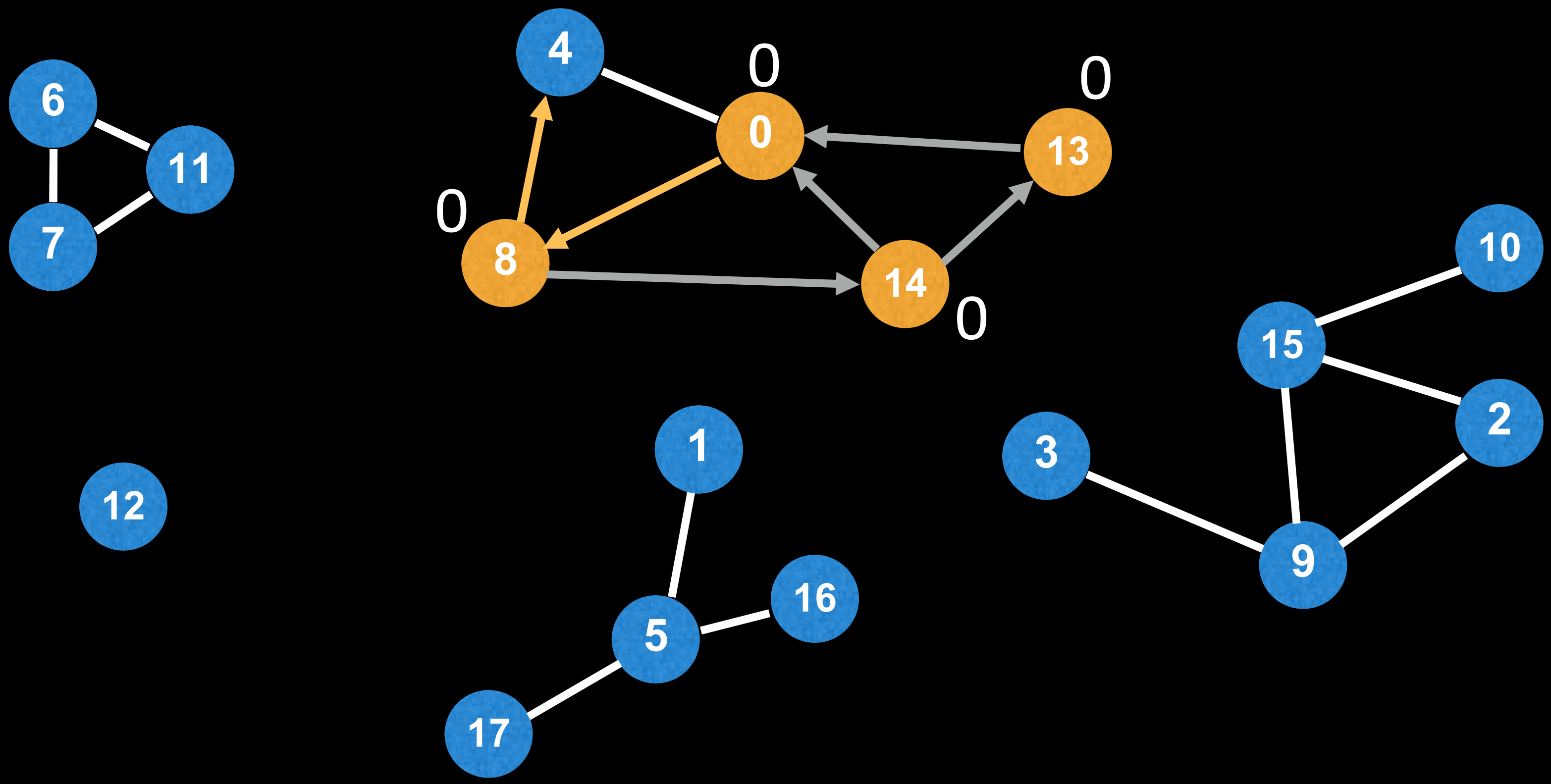
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



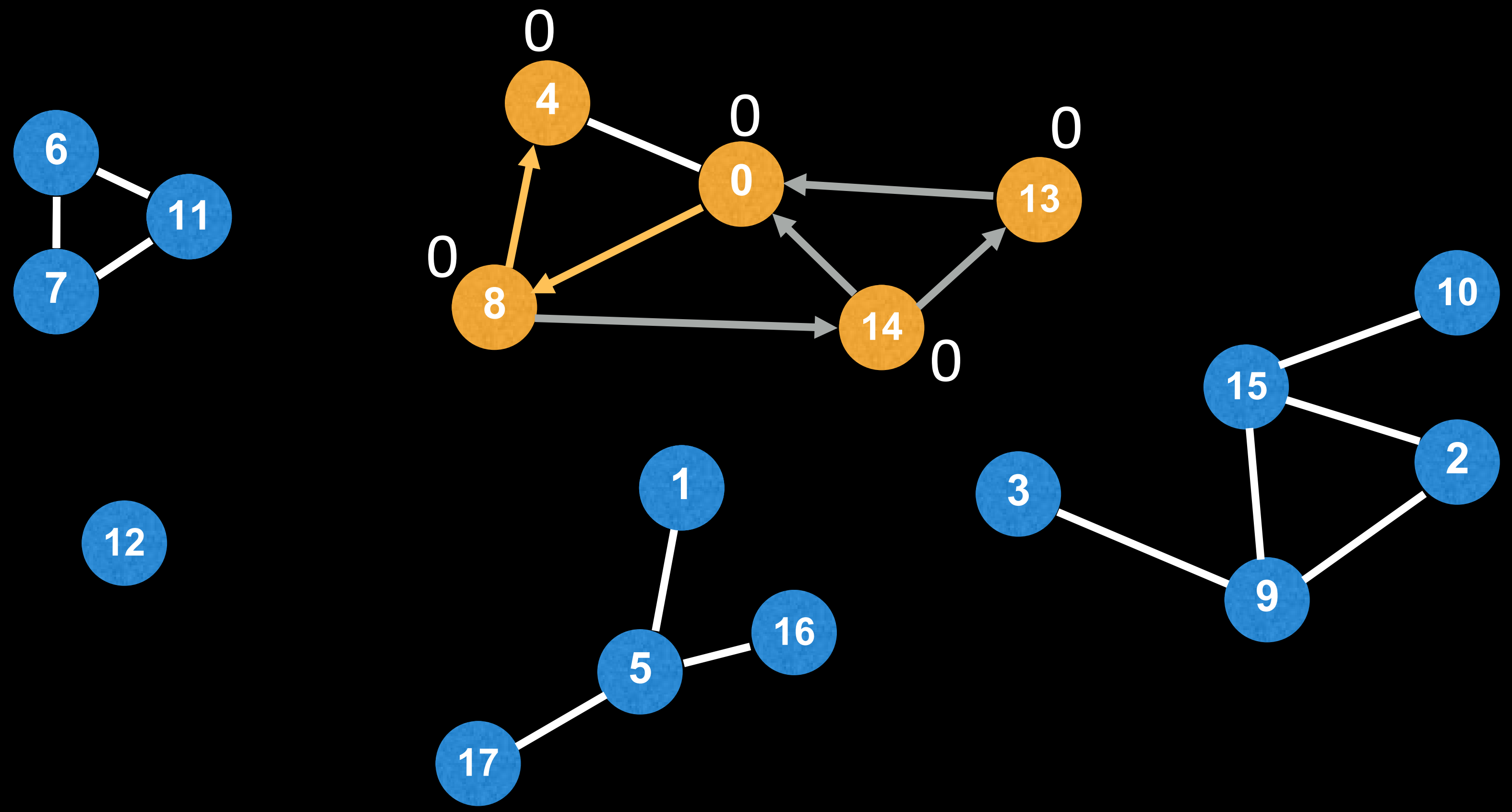
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



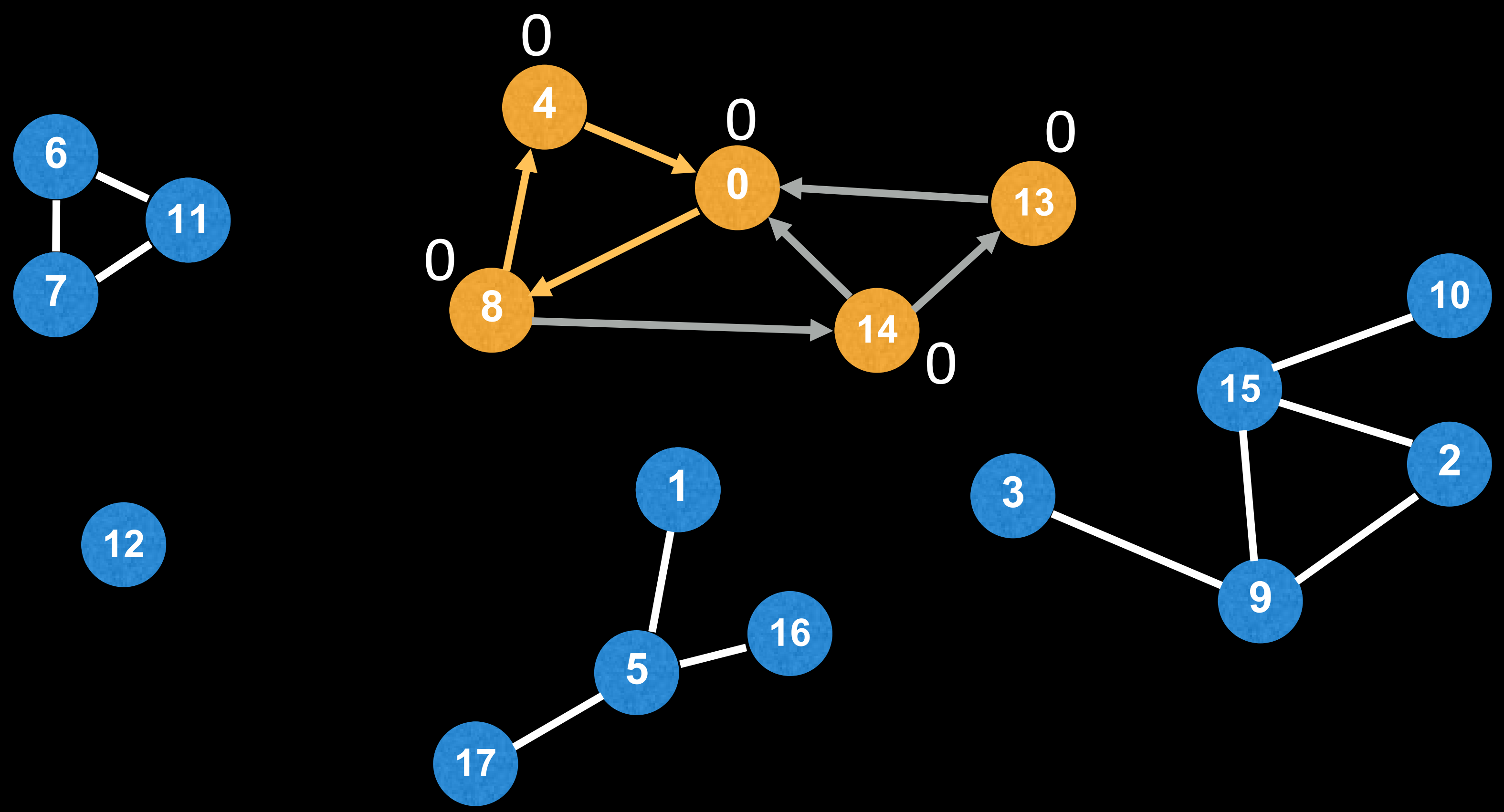
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



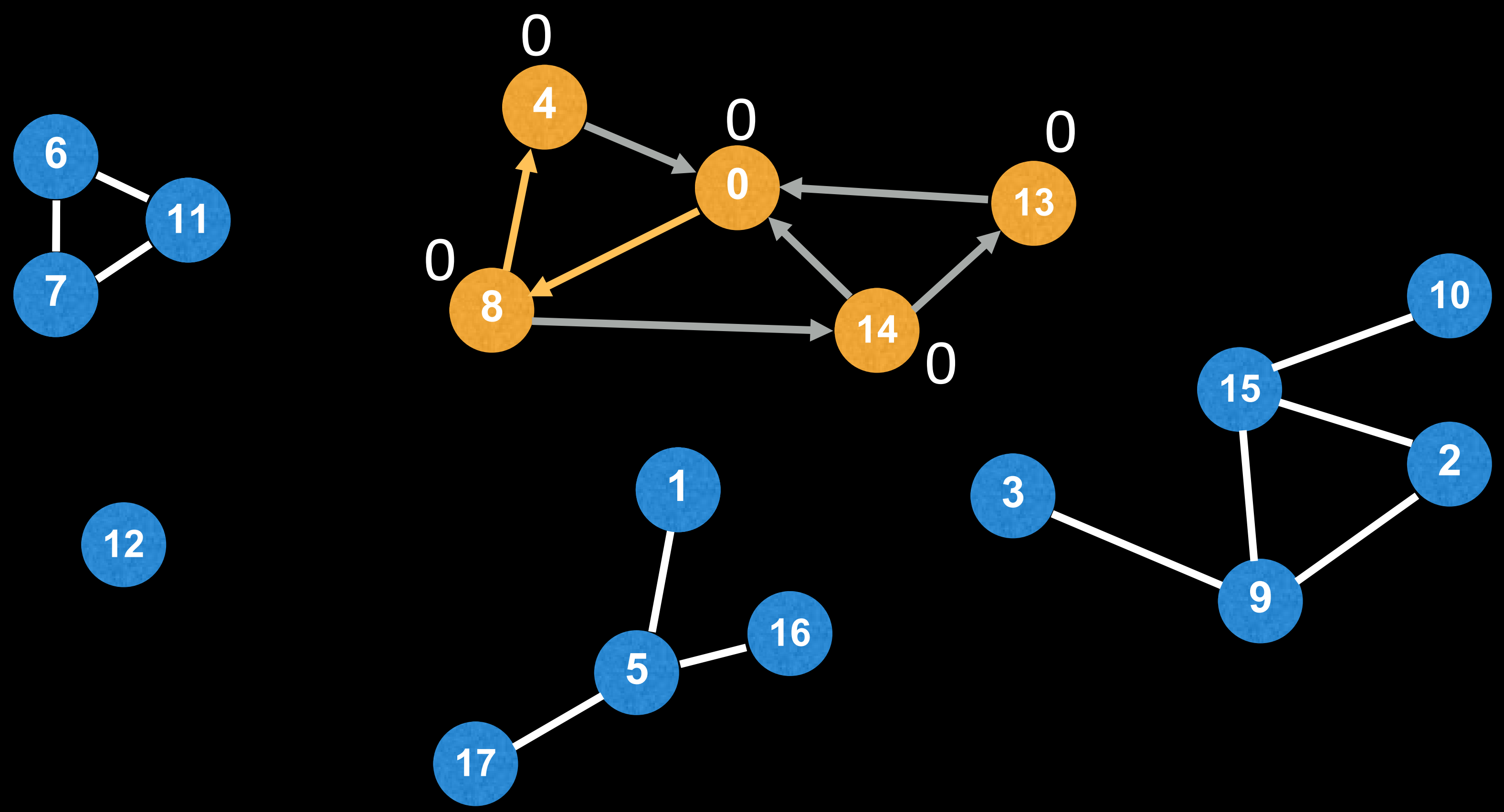
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



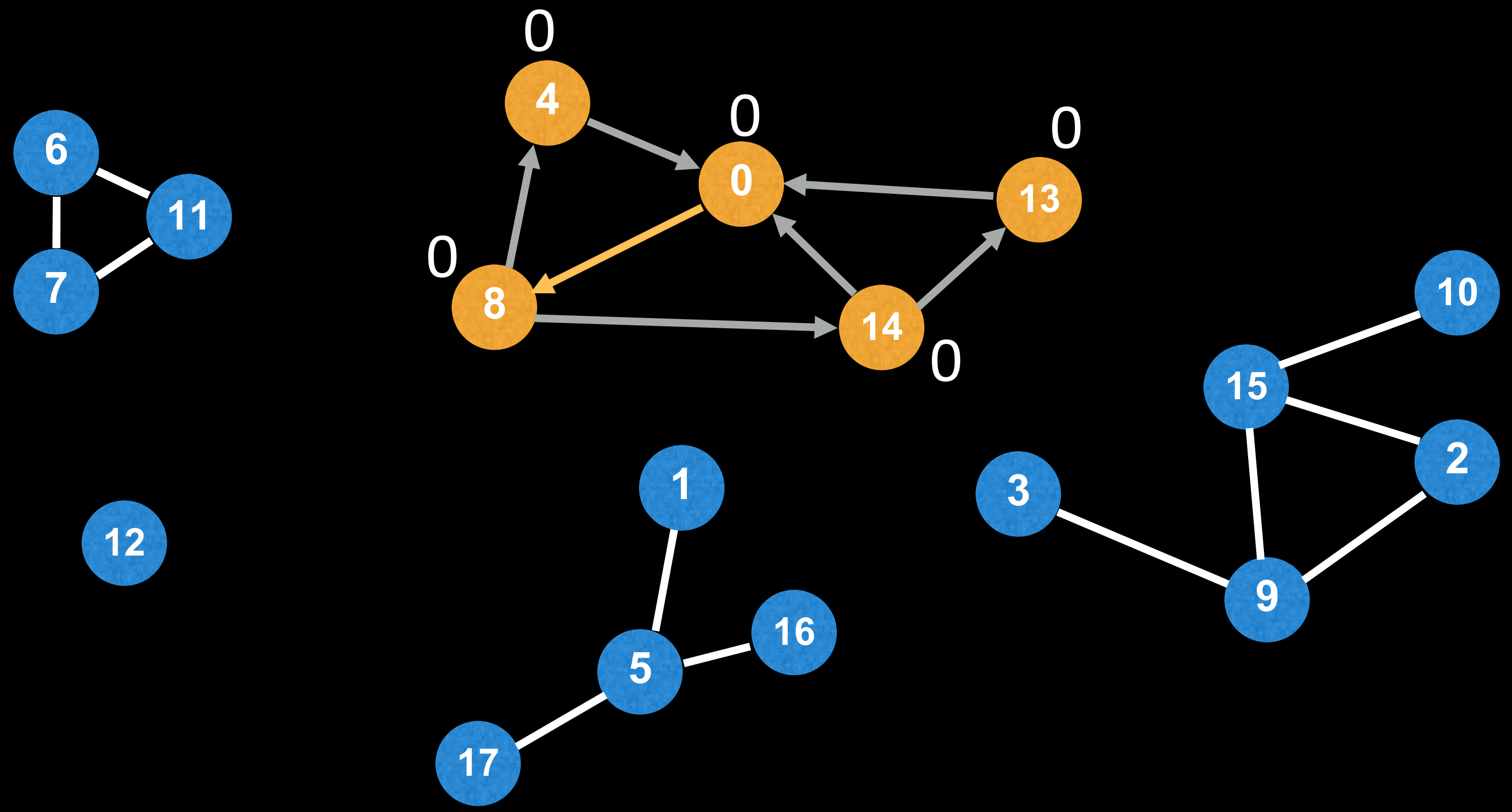
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



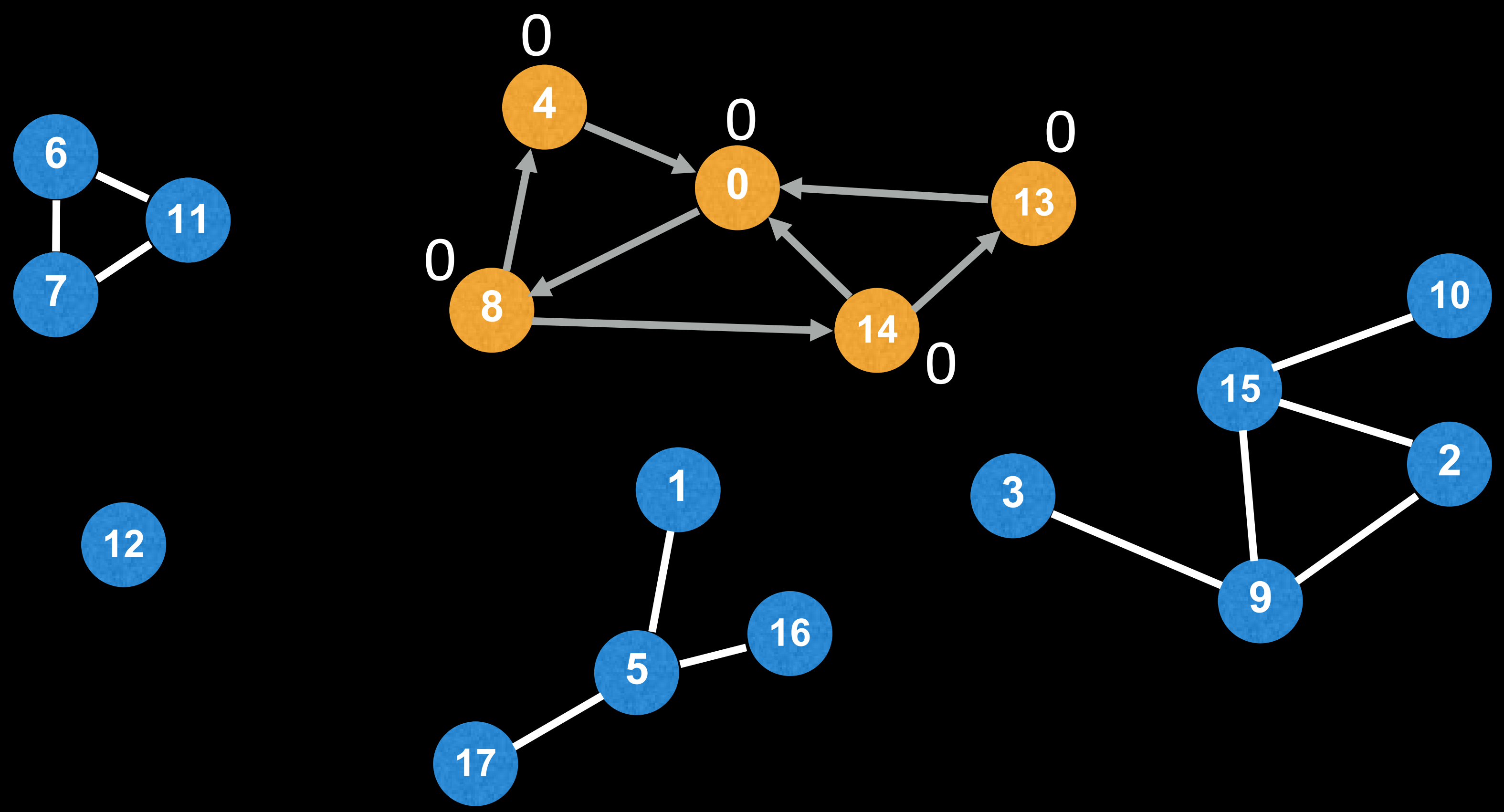
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



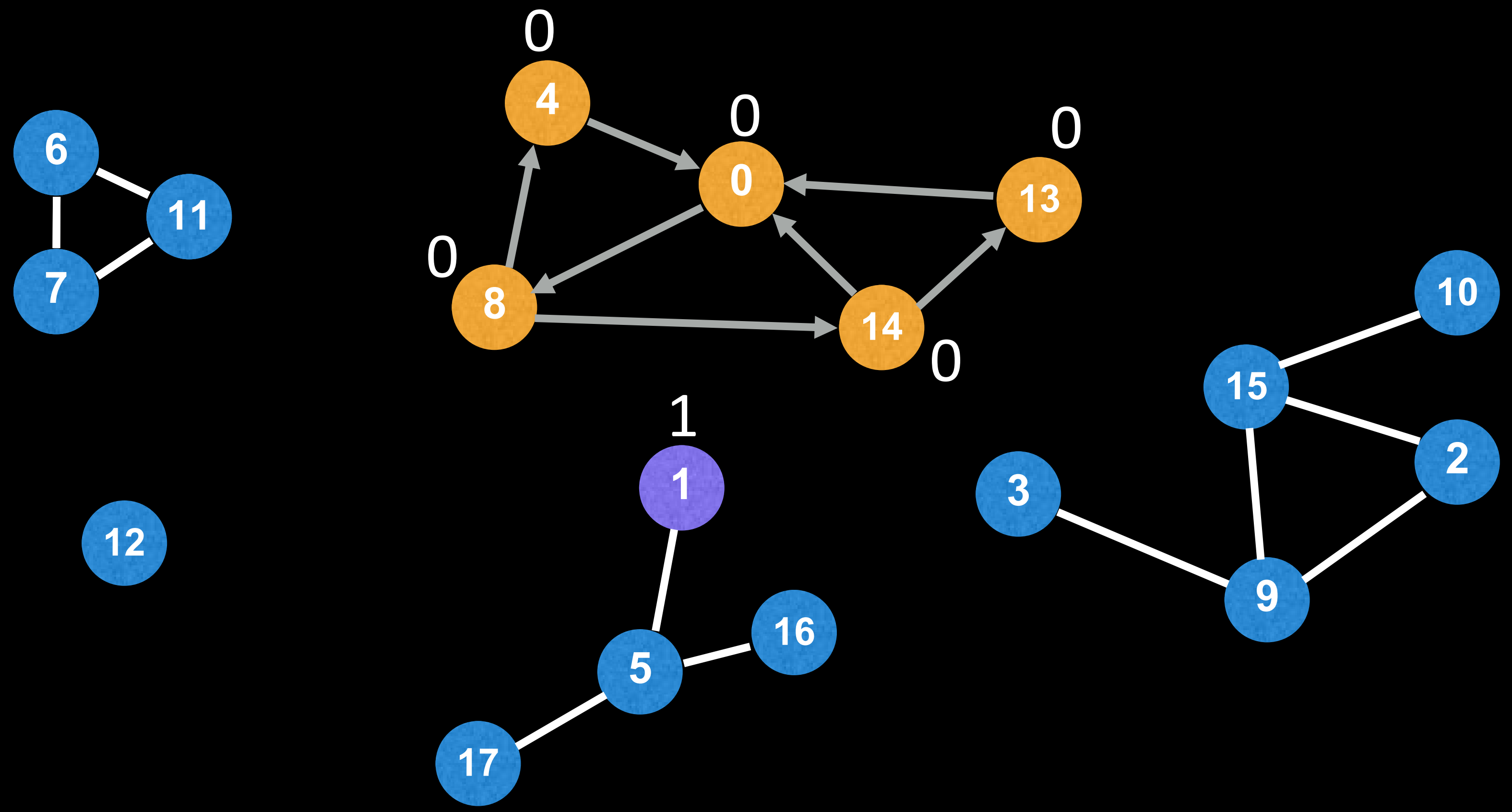
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



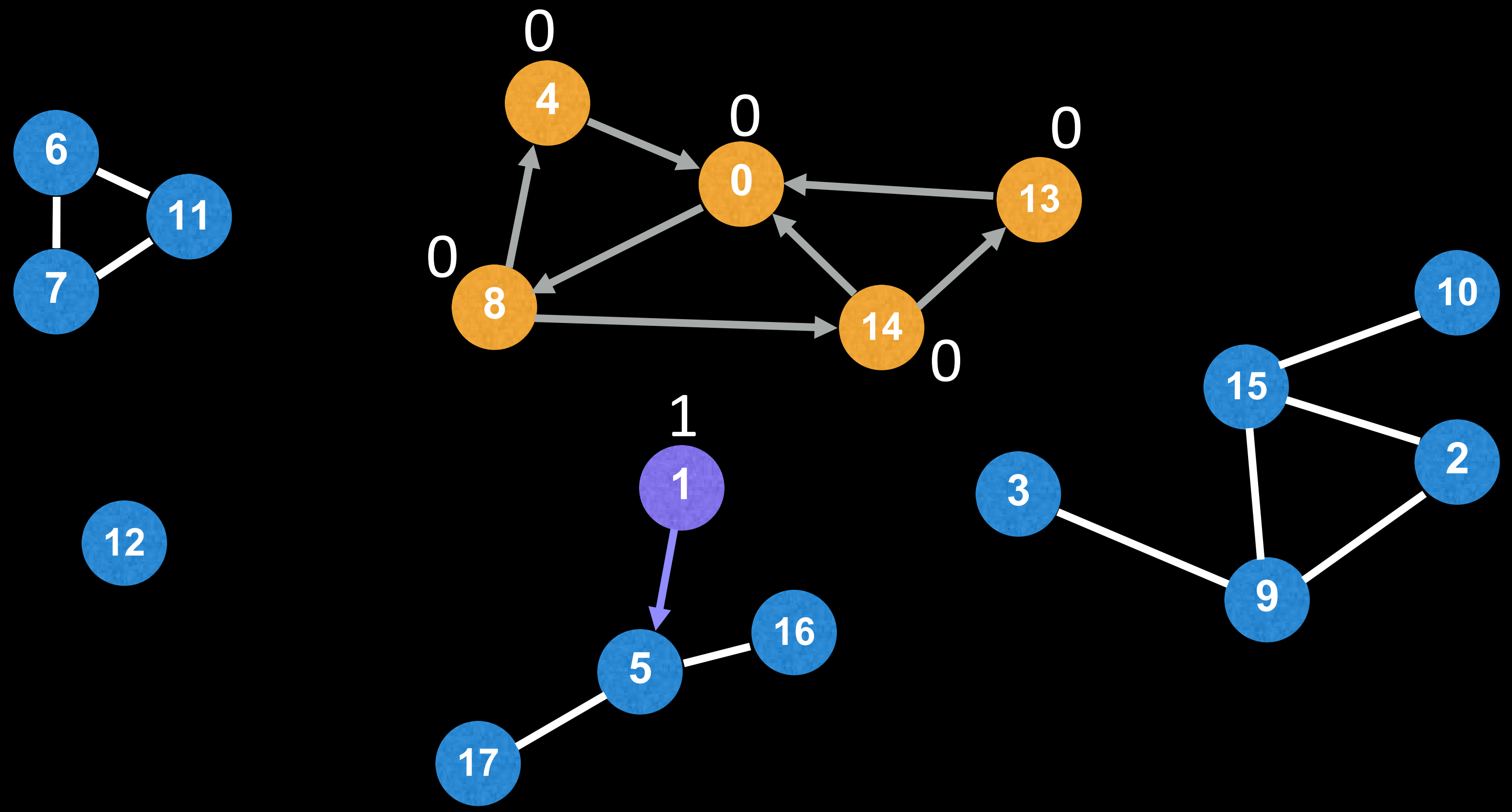
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



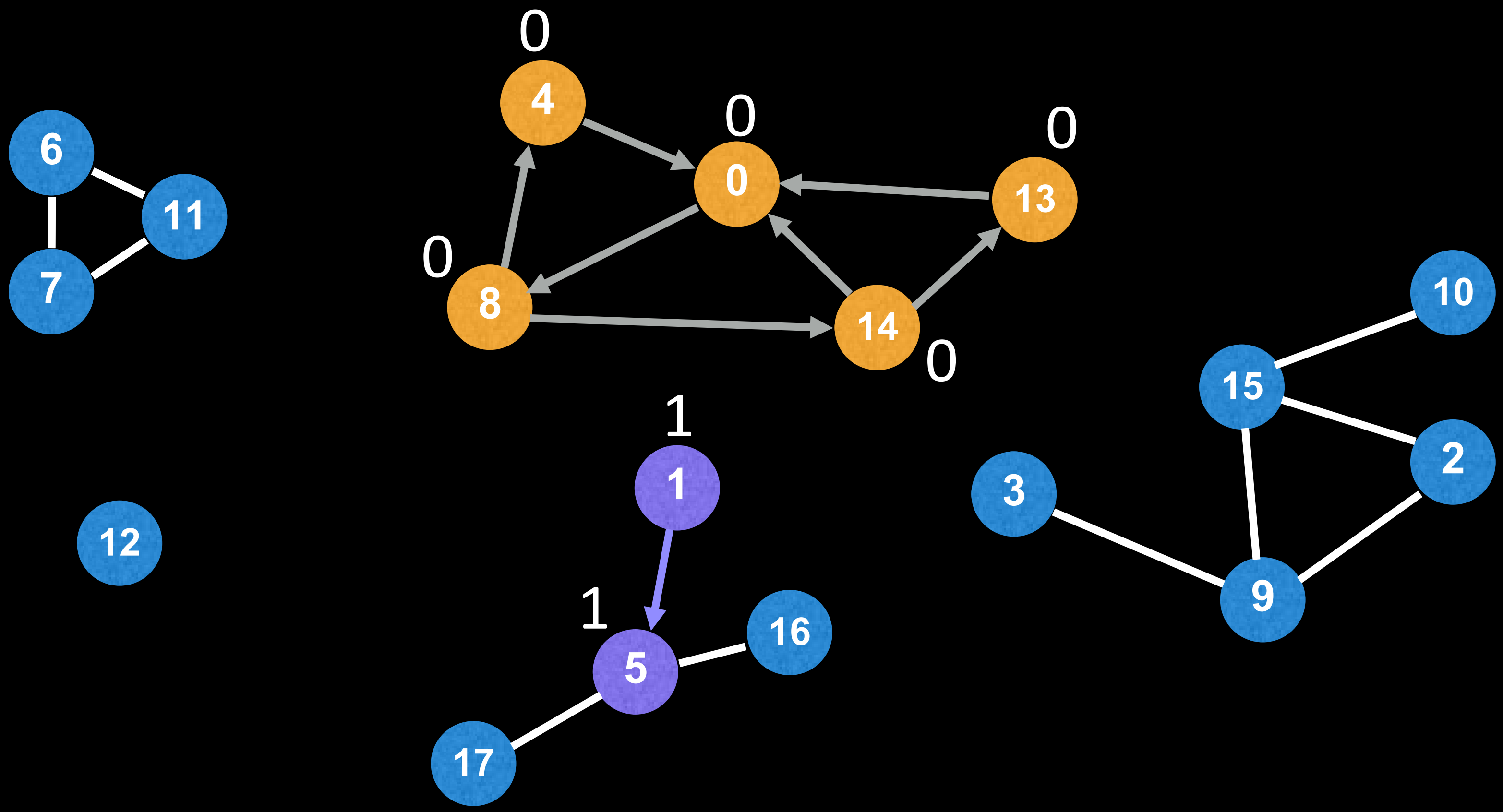
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



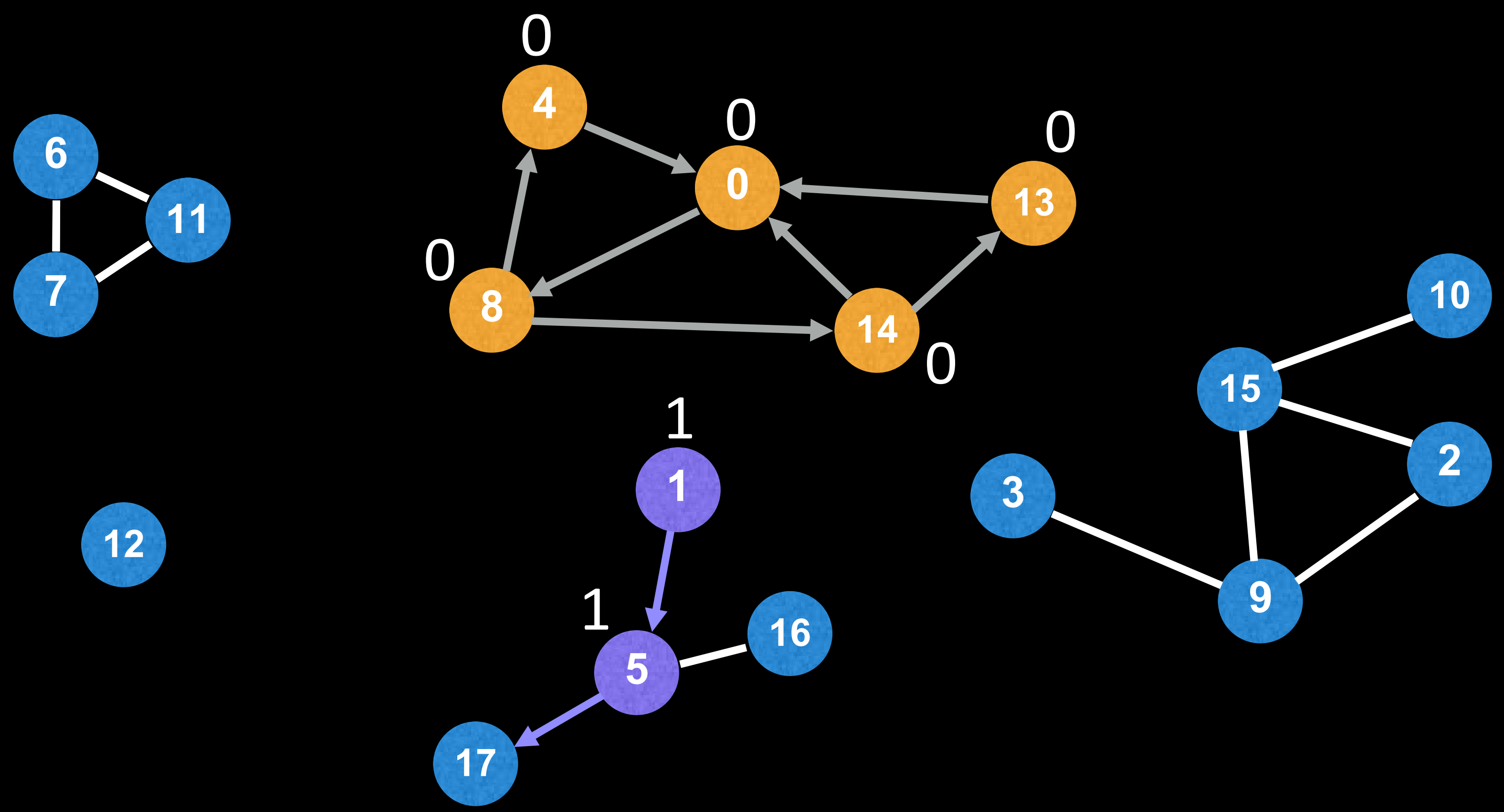
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



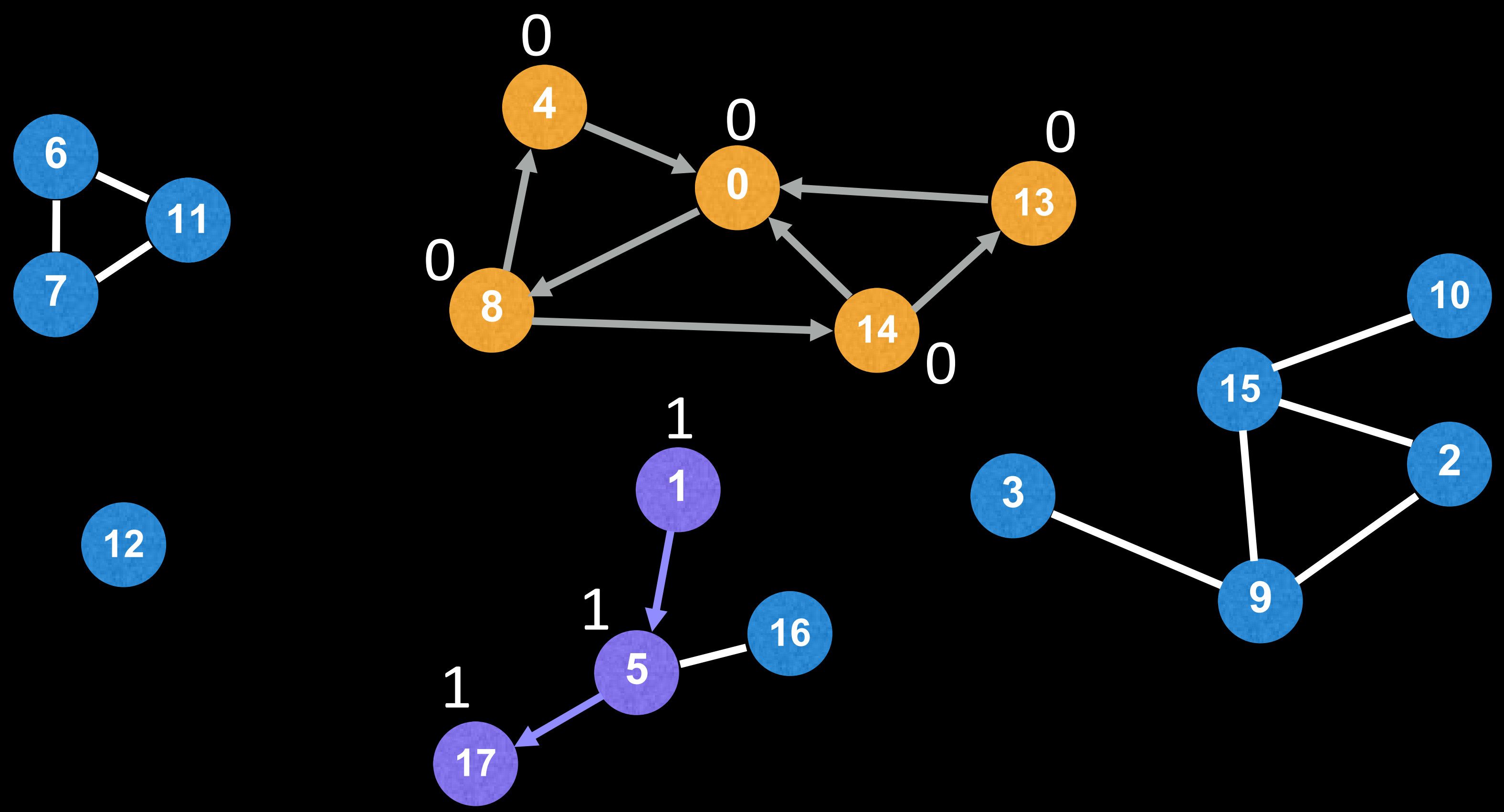
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



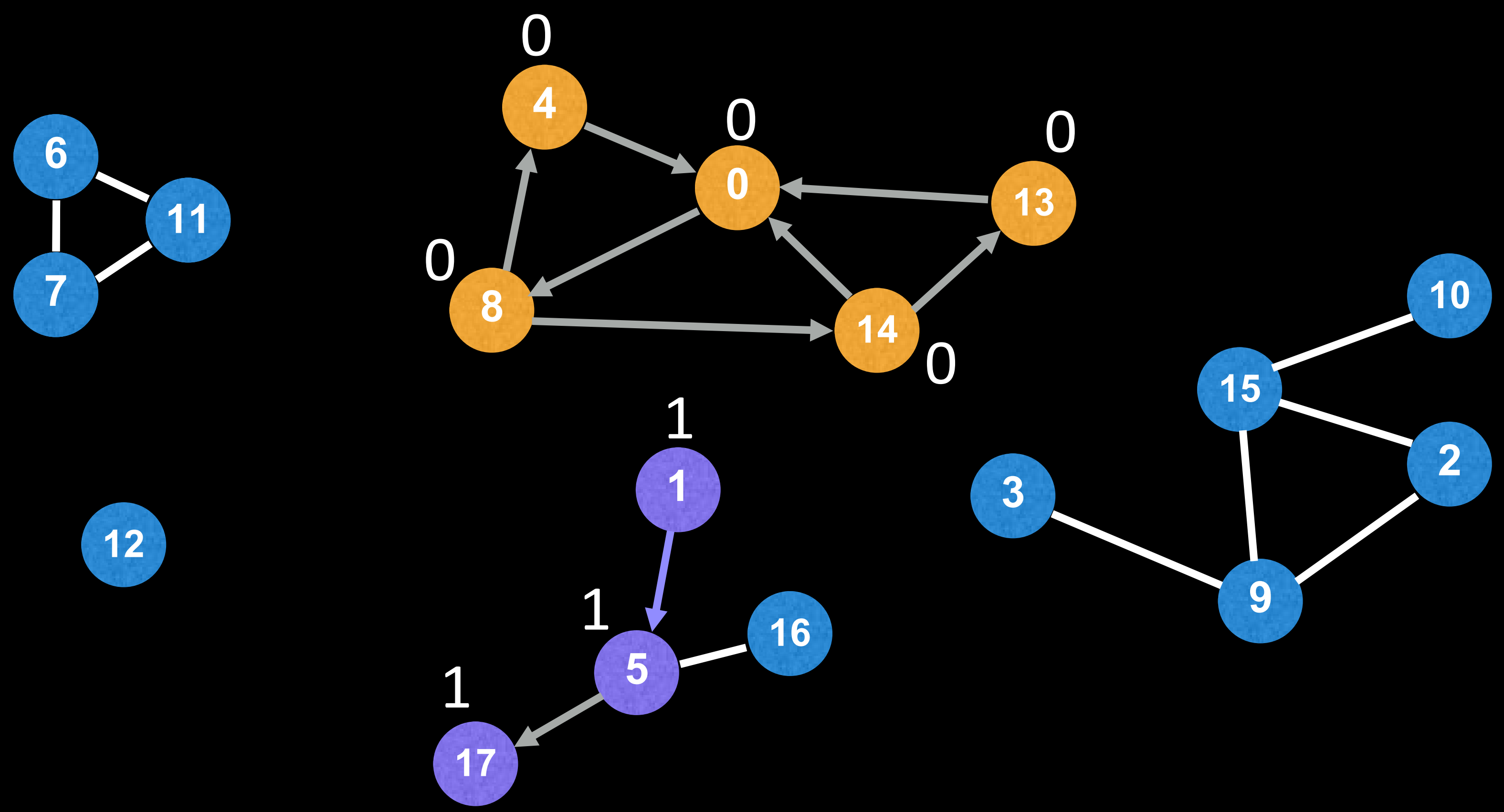
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



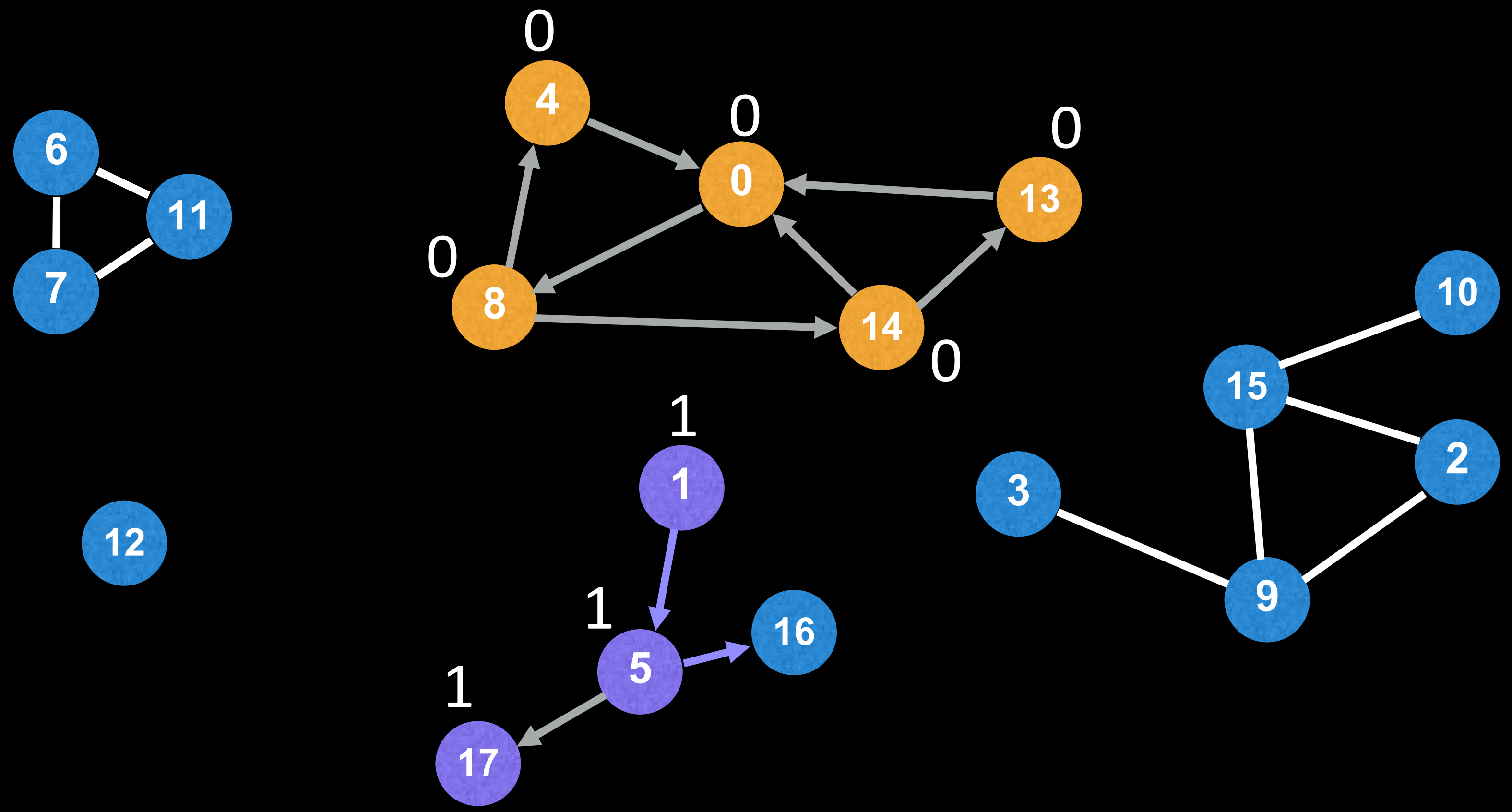
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



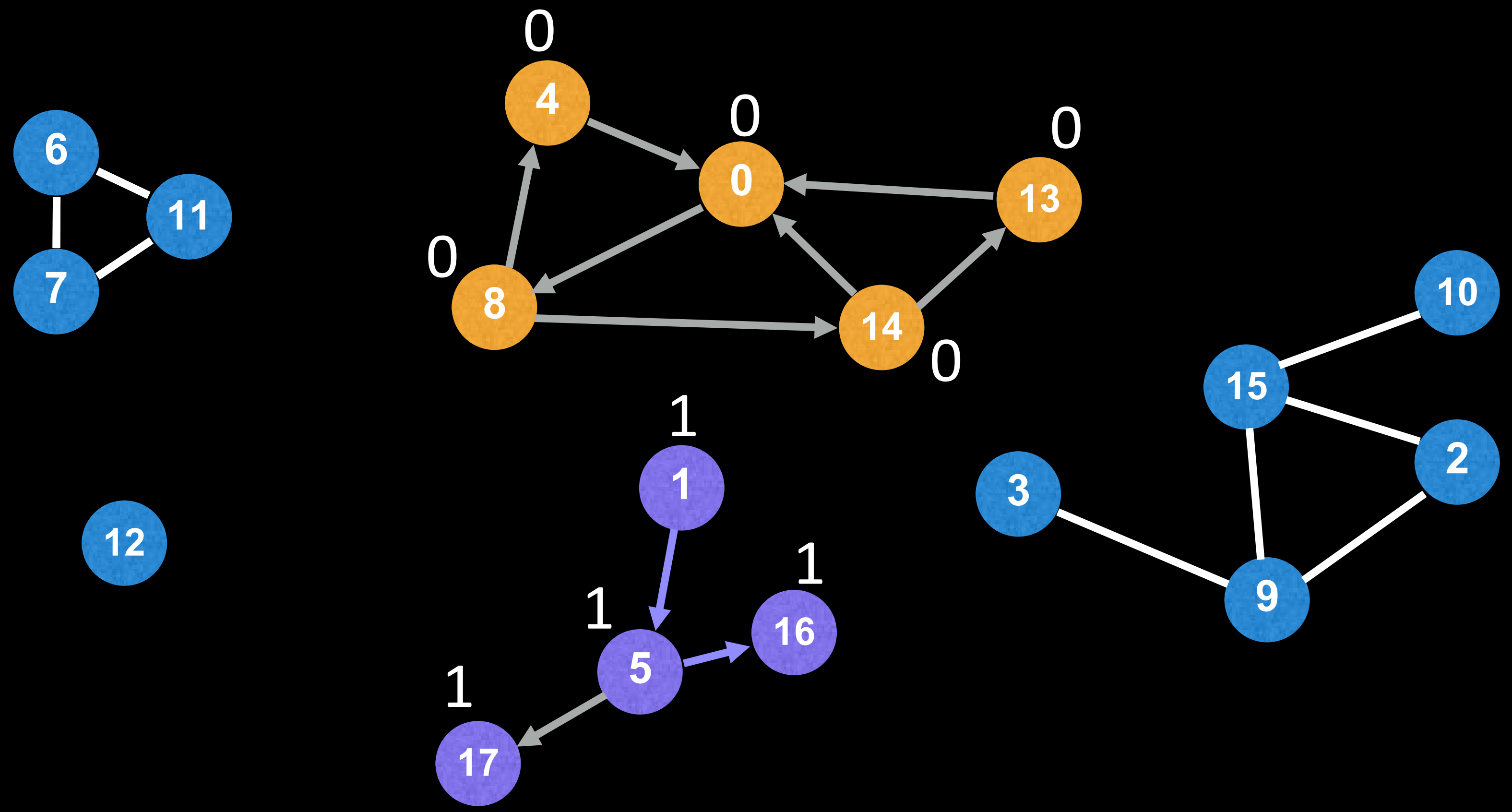
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



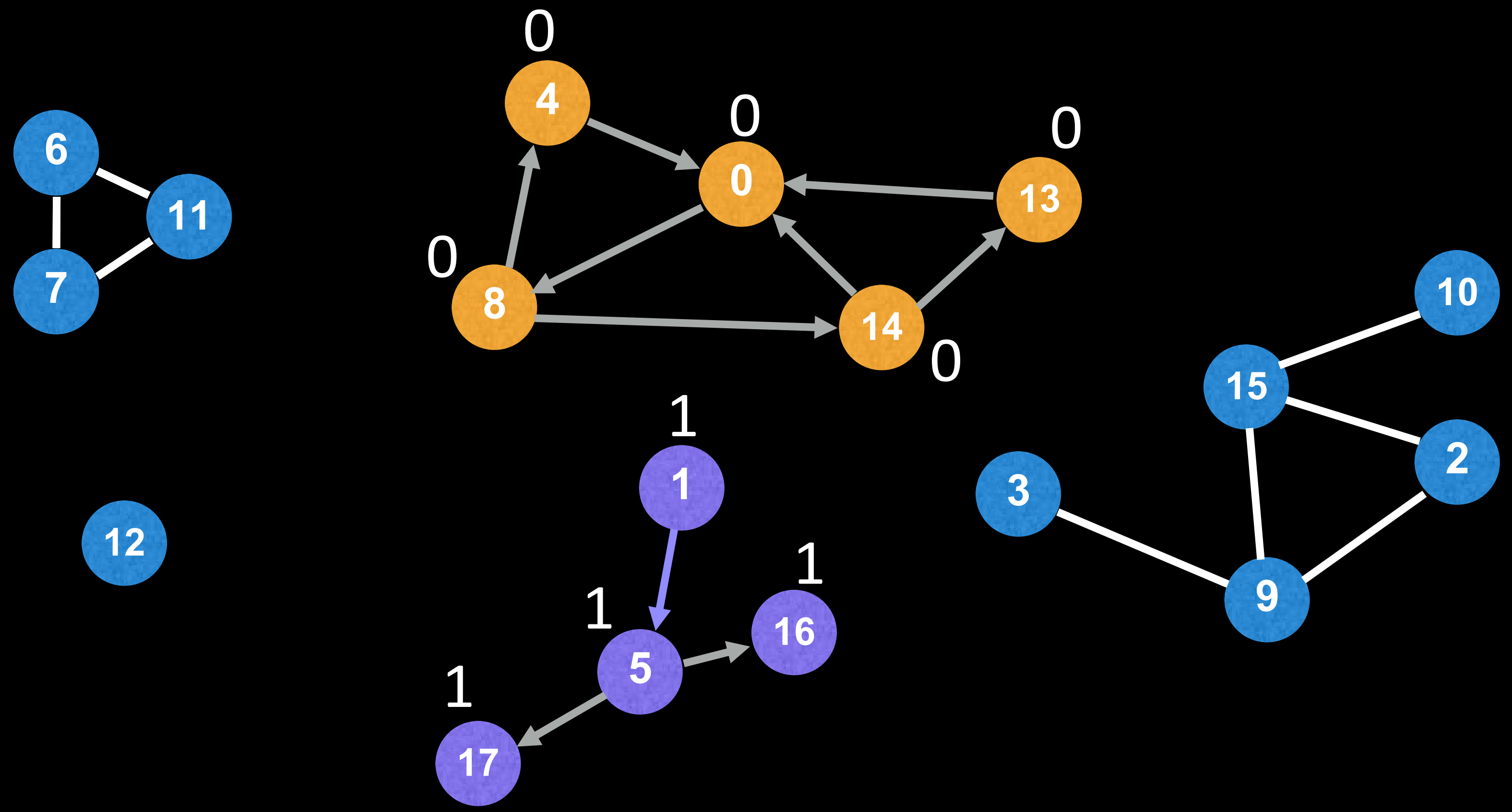
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



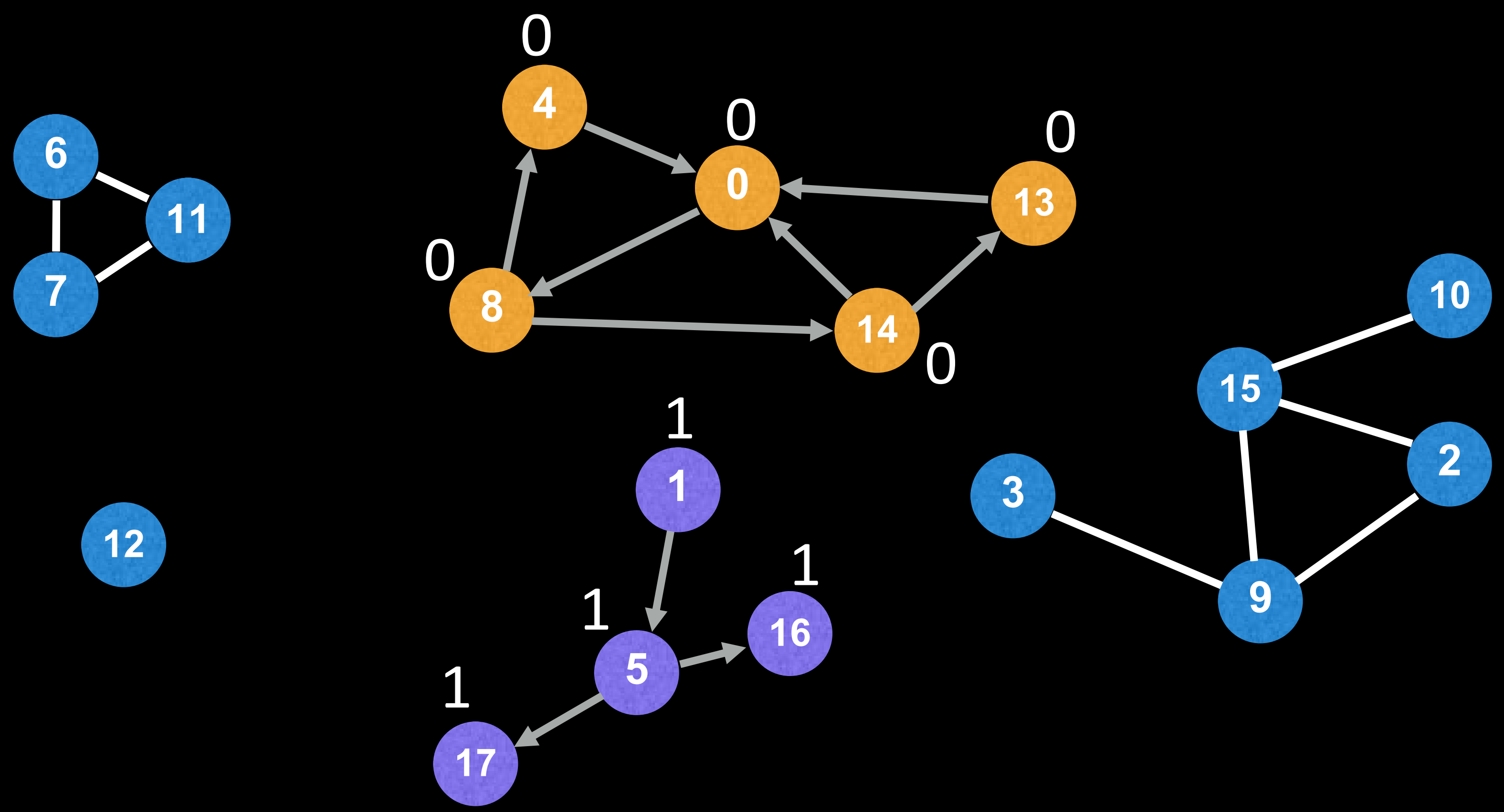
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



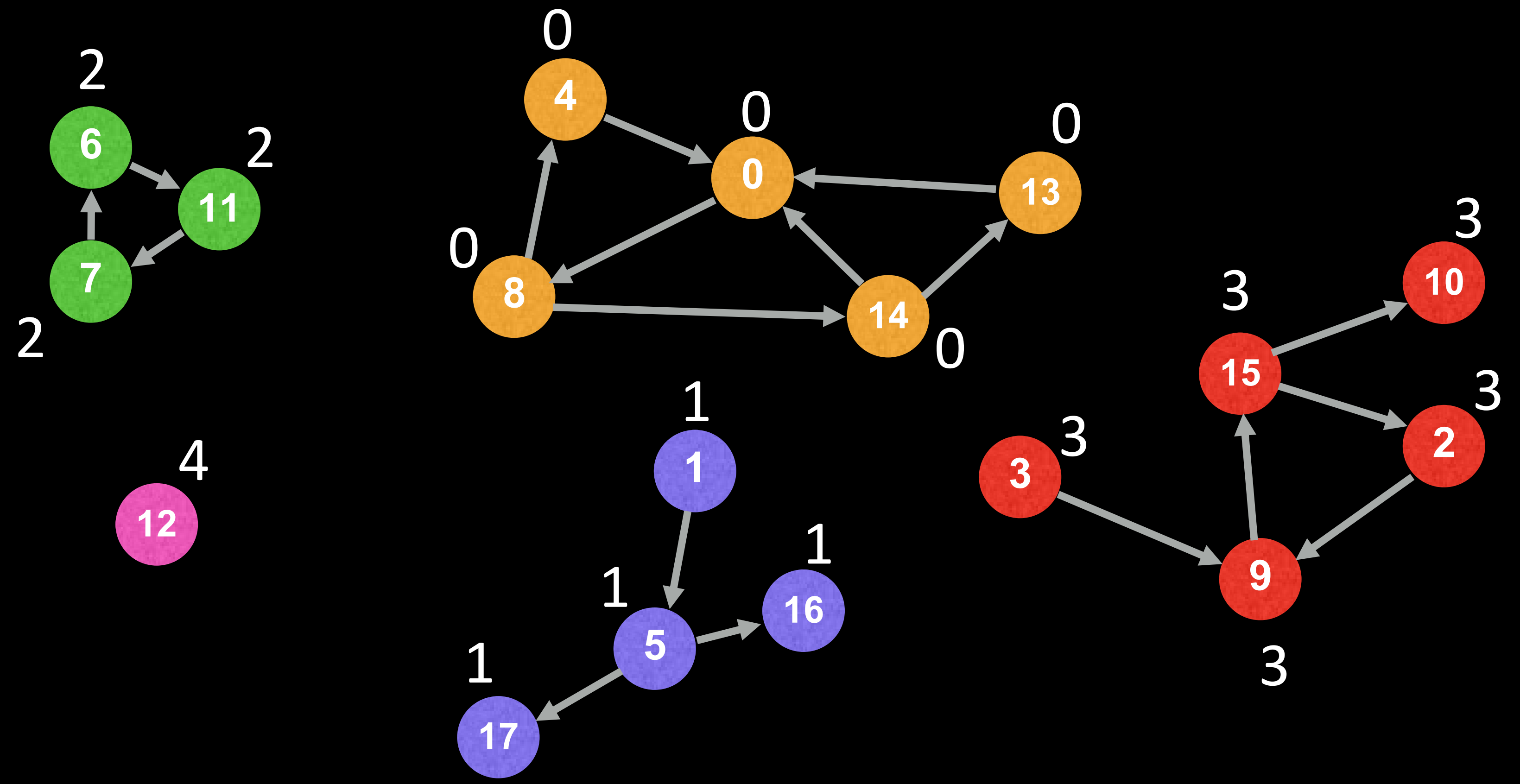
Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.



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)
```

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.