Advanced Algorithmic Techniques (COMP523)

Graph Algorithms

Recap and plan

Recap and plan

- **First five lectures:**
  - Basic Algorithms
  - Divide and Conquer algorithms
    - Searching, Sorting, Majority, Distance between points, Integer Multiplication, Median

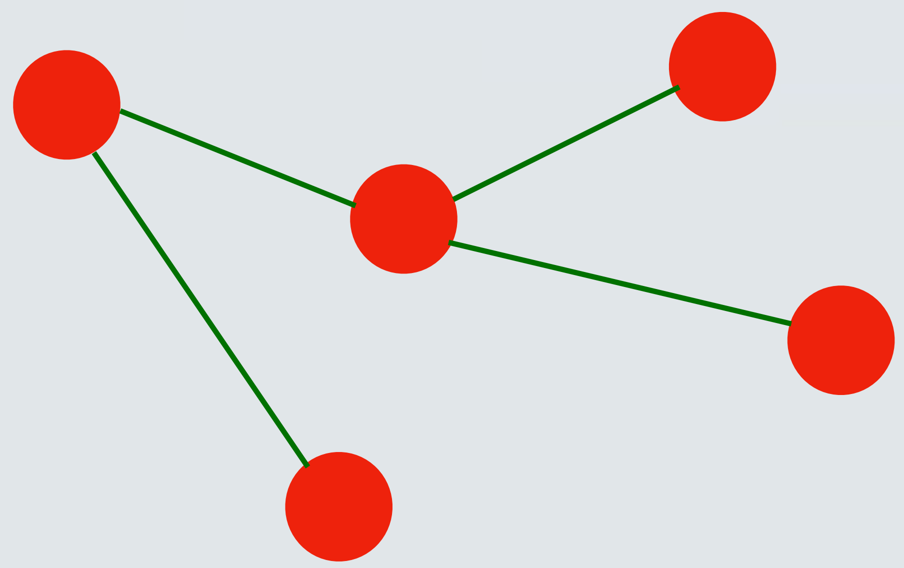
Recap and plan

- **First five lectures:**
  - Basic Algorithms
  - Divide and Conquer algorithms
    - Searching, Sorting, Majority, Distance between points, Integer Multiplication, Median
- **This lecture:**
  - Graph Algorithms
    - Graph Definitions
    - Graph Representations
    - Depth-First Search, Breadth-First Search

Graph Definitions

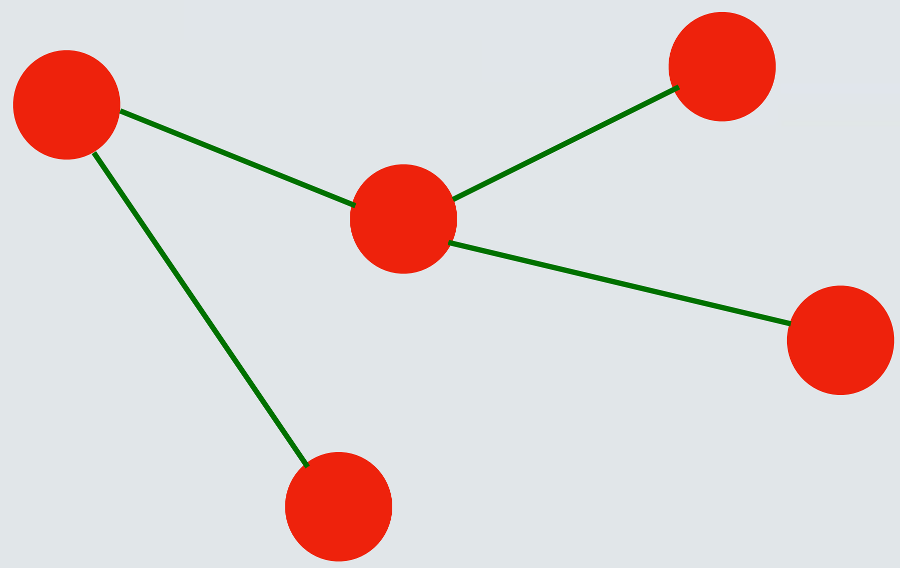
Graph **G**=(**V**,**E**)

- Set of nodes (or vertices) **V**, with |**V**| = n
- Set of edges **E**, with |**E**| = m
    - Undirected: edge e = {v,w}
    - Directed: edge e = (v,w)



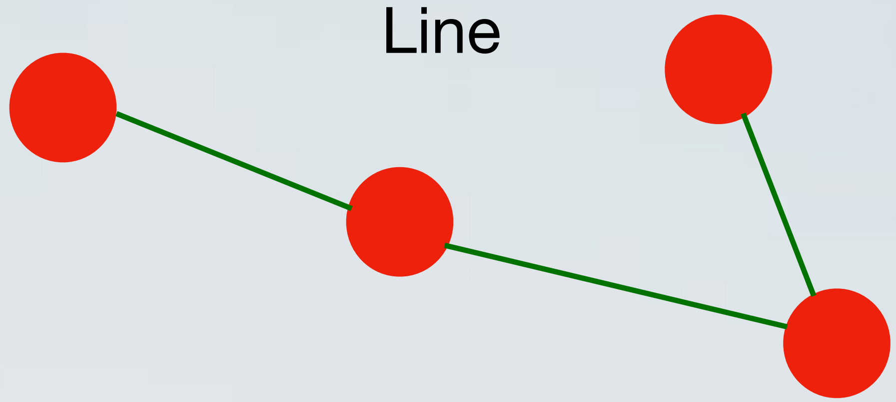
Graph Definitions

Neighbours of v : Set of nodes connected by an edge with v
Degree of a node: number of neighbours
Directed graphs: *in-degree* and *out-degree*
Path: A sequence of (non-repeating) nodes with consecutive nodes being connected by an edge.
Length: # nodes - 1
Distance between u and v : length of the shortest path u and v,
Graph diameter: The longest distance in the graph

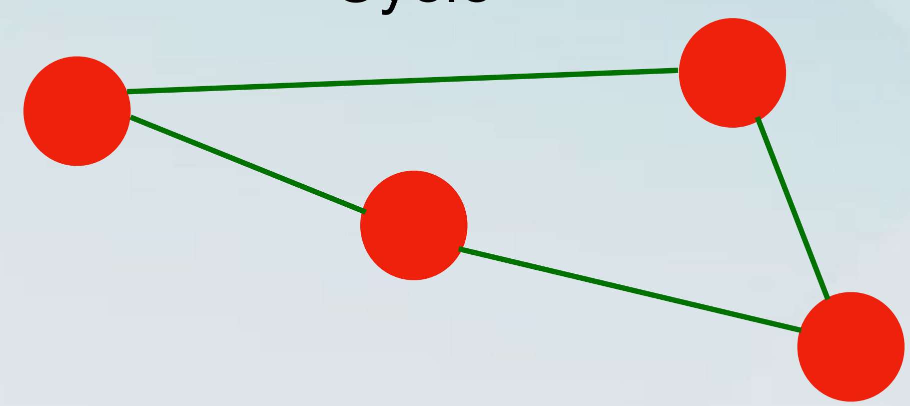


Lines, cycles, trees and cliques

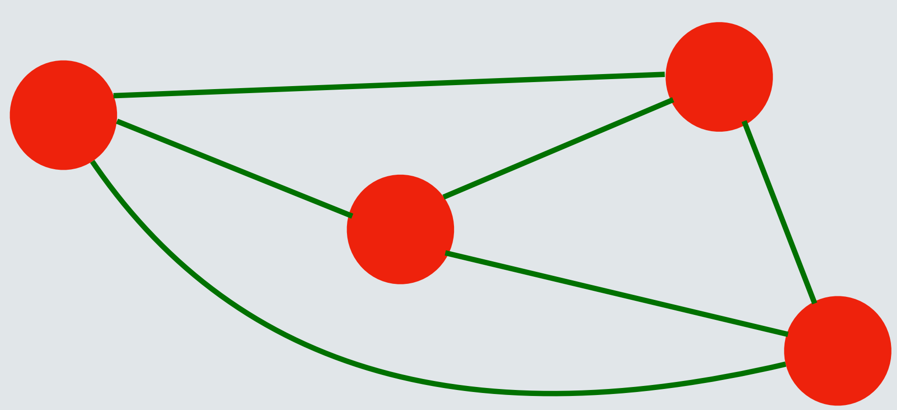
Line



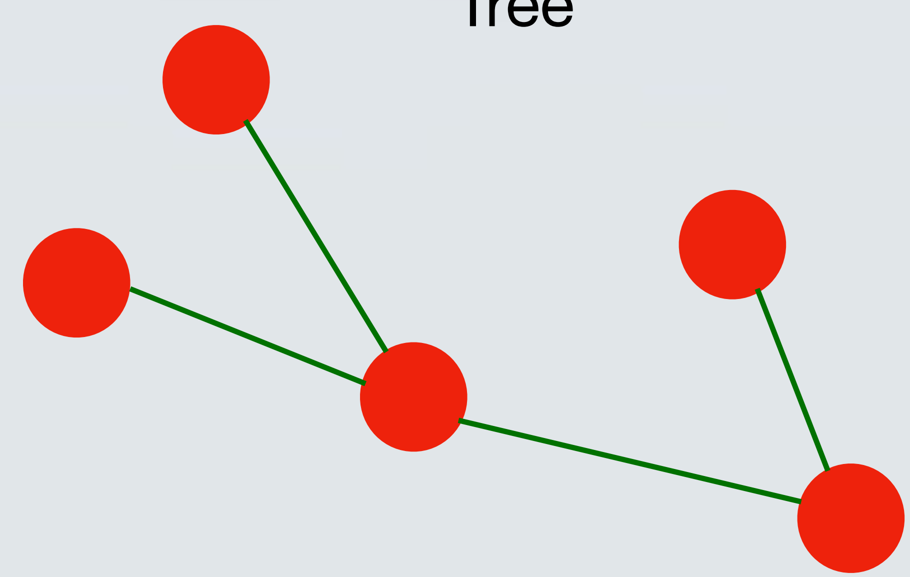
Cycle



Clique



Tree

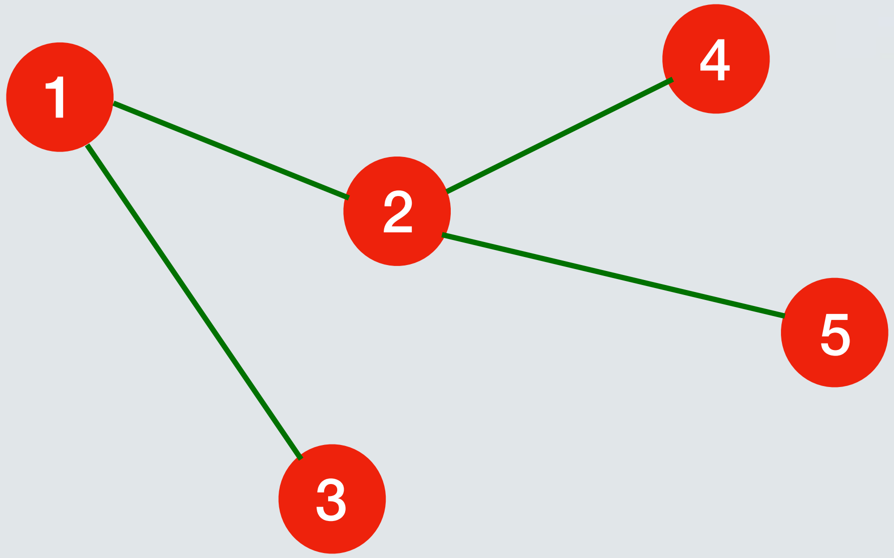


Graph Representations

- How do we represent a graph **G**=(**V**,**E**)?
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix A

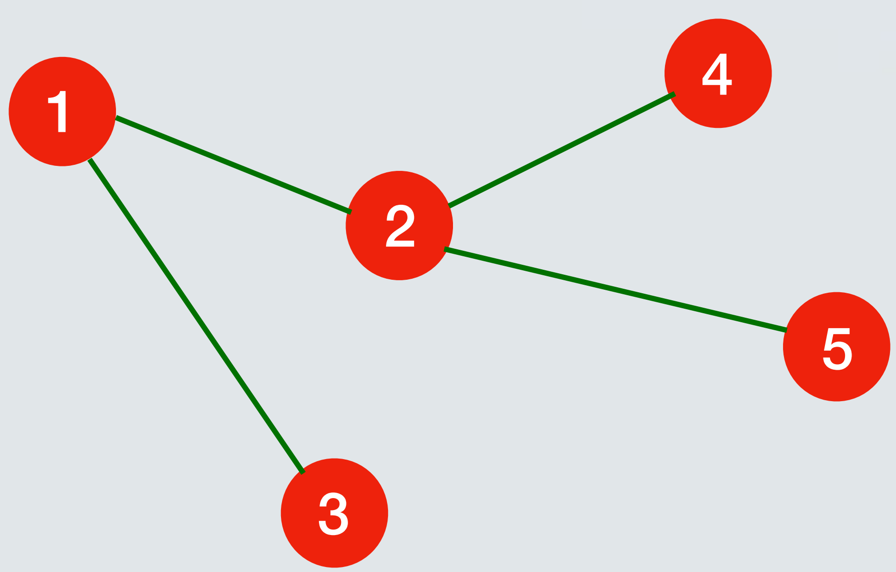
- The i^{th} node corresponds to the i^{th} row and the i^{th} column.
- If there is an edge between i and j in the graph, then we have **A**[i,j] = 1, otherwise **A**[i,j] = 0.
- For undirected graphs, necessarily **A**[i,j] = **A**[j,i]. For directed graphs, it could be that **A**[i,j] ≠ **A**[j,i].

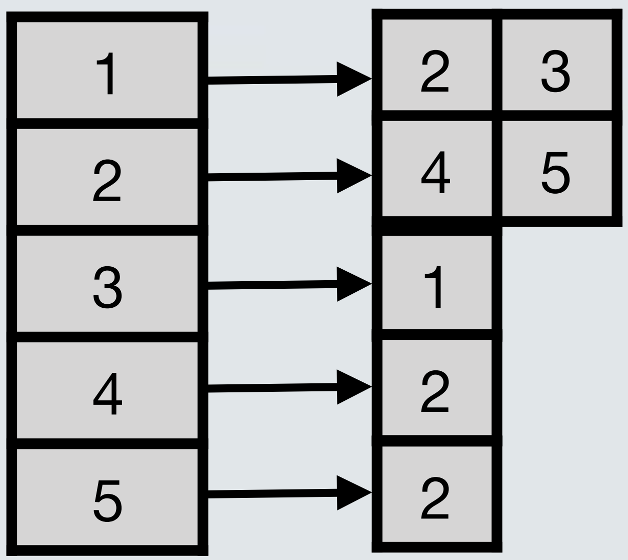


0	1	1	0	0
1	0	0	1	1
1	0	0	0	0
0	1	0	0	0
0	1	0	0	0

Adjacency List L

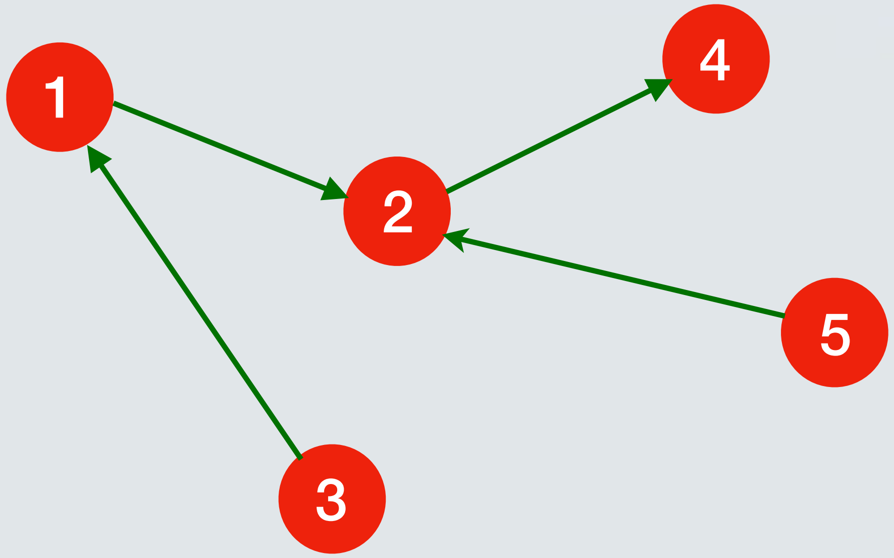
- Nodes are arranged as a list, each node points to the neighbours.
- For undirected graphs, the node points only in one direction.
- For directed graphs, the node points in two directions, for in-degree and for out-degree

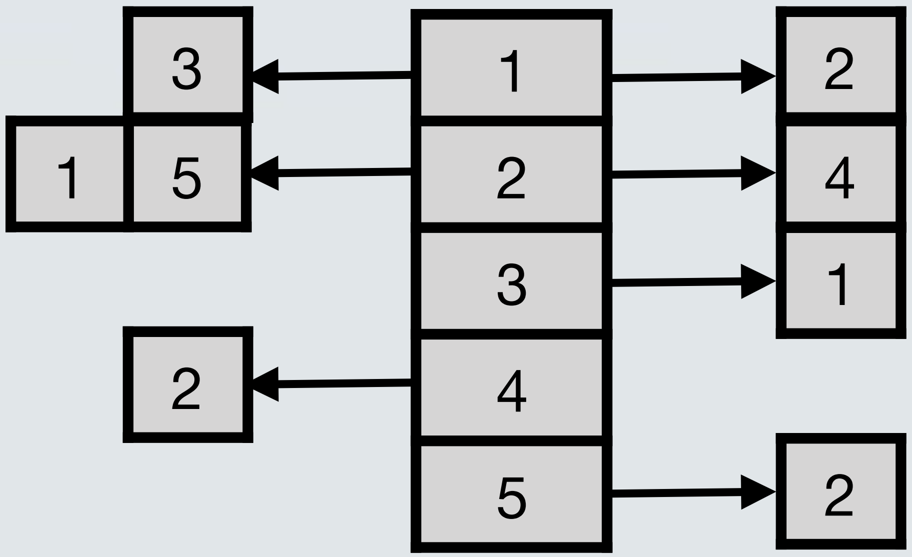




Adjacency List L

- Nodes are arranged as a list, each node points to the neighbours.
- For undirected graphs, the node points only in one direction.
- For directed graphs, the node points in two directions, for in-degree and for out-degree.





Adjacency Matrix vs Adjacency List

Adjacency Matrix

Memory: $O(n^2)$

Checking *adjacency* of u and v
Time: O(1)

Finding *all adjacent nodes* of u
Time: O(n)

Adjacency List

Memory: O(m+n)

Checking *adjacency* of u and v
Time: O(min(deg(u),deg(b))

Finding *all adjacent nodes* of u
Time: O(deg(u))

Adjacency Matrix vs Adjacency List

Adjacency Matrix

Memory: $O(n^2)$

Checking *adjacency* of u and v
Time: $O(1)$

Finding *all adjacent nodes* of u
Time: $O(n)$

Adjacency List

Memory: $O(m+n)$

Checking *adjacency* of u and v
Time: $O(min(deg(u),deg(b))$

Finding *all adjacent nodes* of u
Time: $O(deg(u))$

Question: What kind of graphs are the ones for which Adjacency List is more appropriate?

Adjacency Matrix vs Adjacency List

Adjacency Matrix

Memory: $O(n^2)$

Checking *adjacency* of u and v
Time: O(1)

Finding *all adjacent nodes* of u
Time: O(n)

Adjacency List

Memory: O(m+n)

Checking *adjacency* of u and v
Time: O(min(deg(u),deg(b))

Finding *all adjacent nodes* of u
Time: O(deg(u))

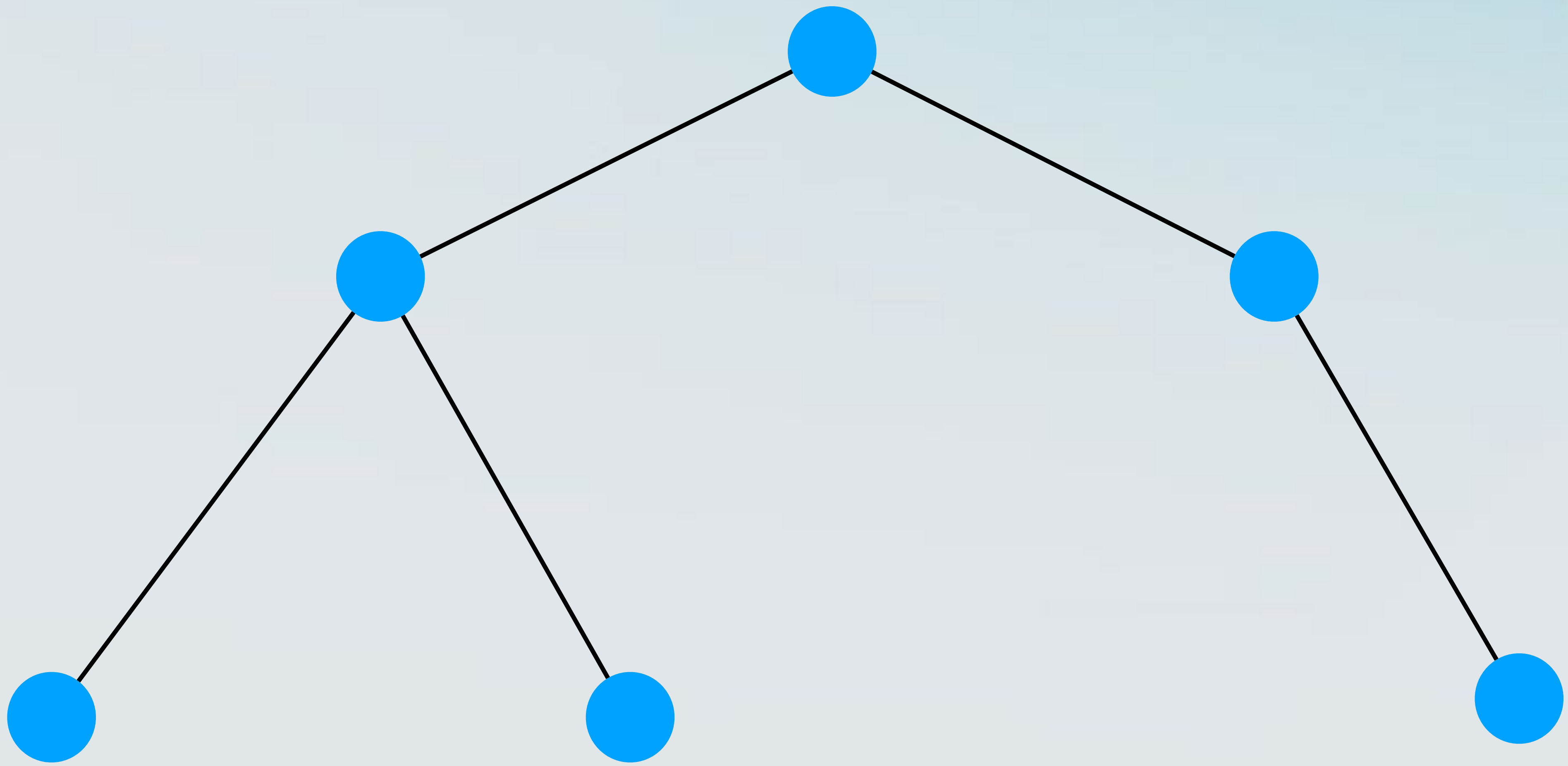
Question: What kind of graphs are the ones for which Adjacency List is more appropriate?
Answer: Sparse graphs (i.e., graphs were n >> m)

Searching a graph

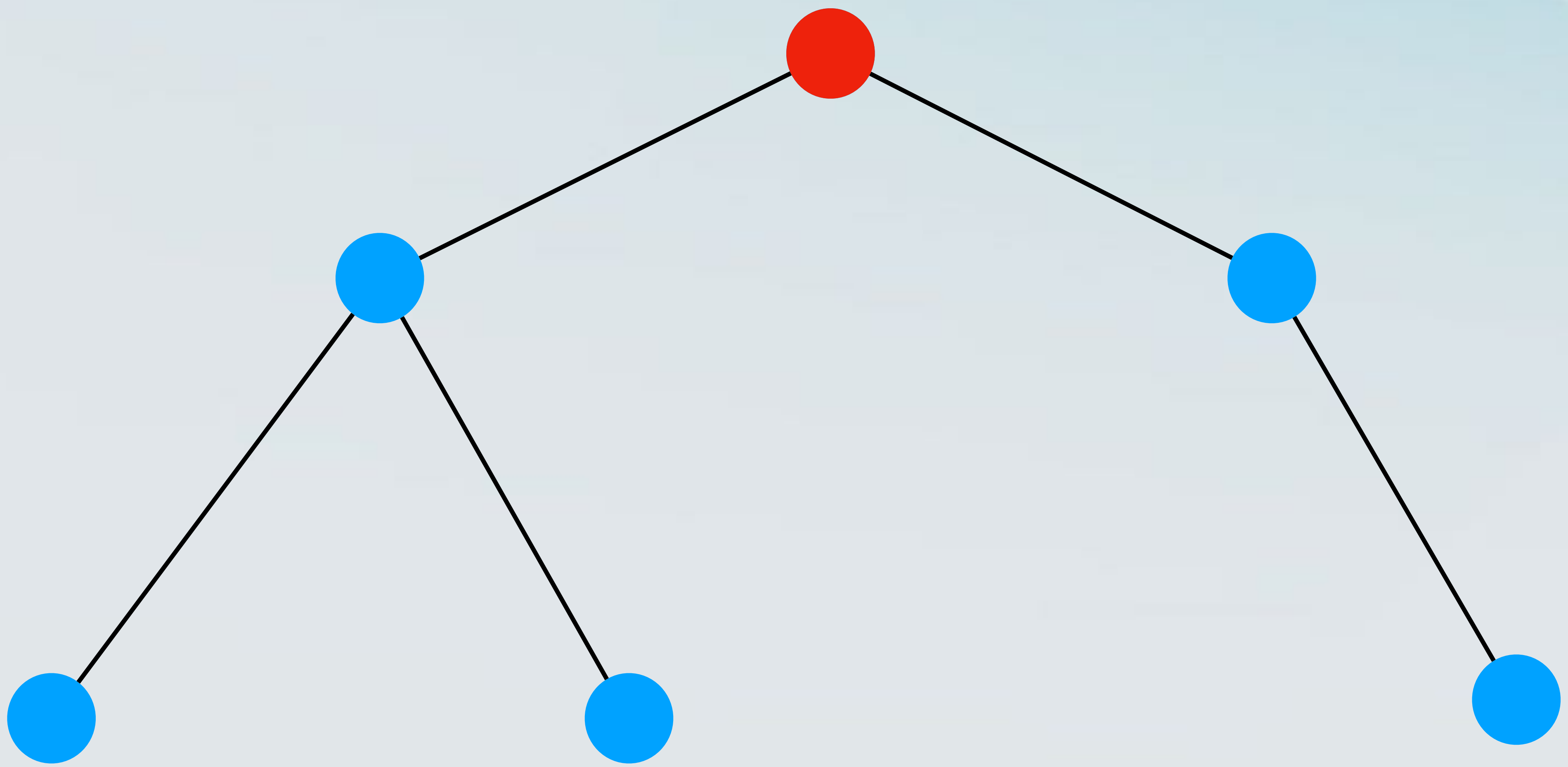
Searching a graph

- Consider the problem of finding a specific node of a graph.
- Imagine that nodes have numbers (but you don't know them), and you want to find the node with the number **x.**
  - Or answer that there is no such node.
- You need to search all the nodes to be sure.

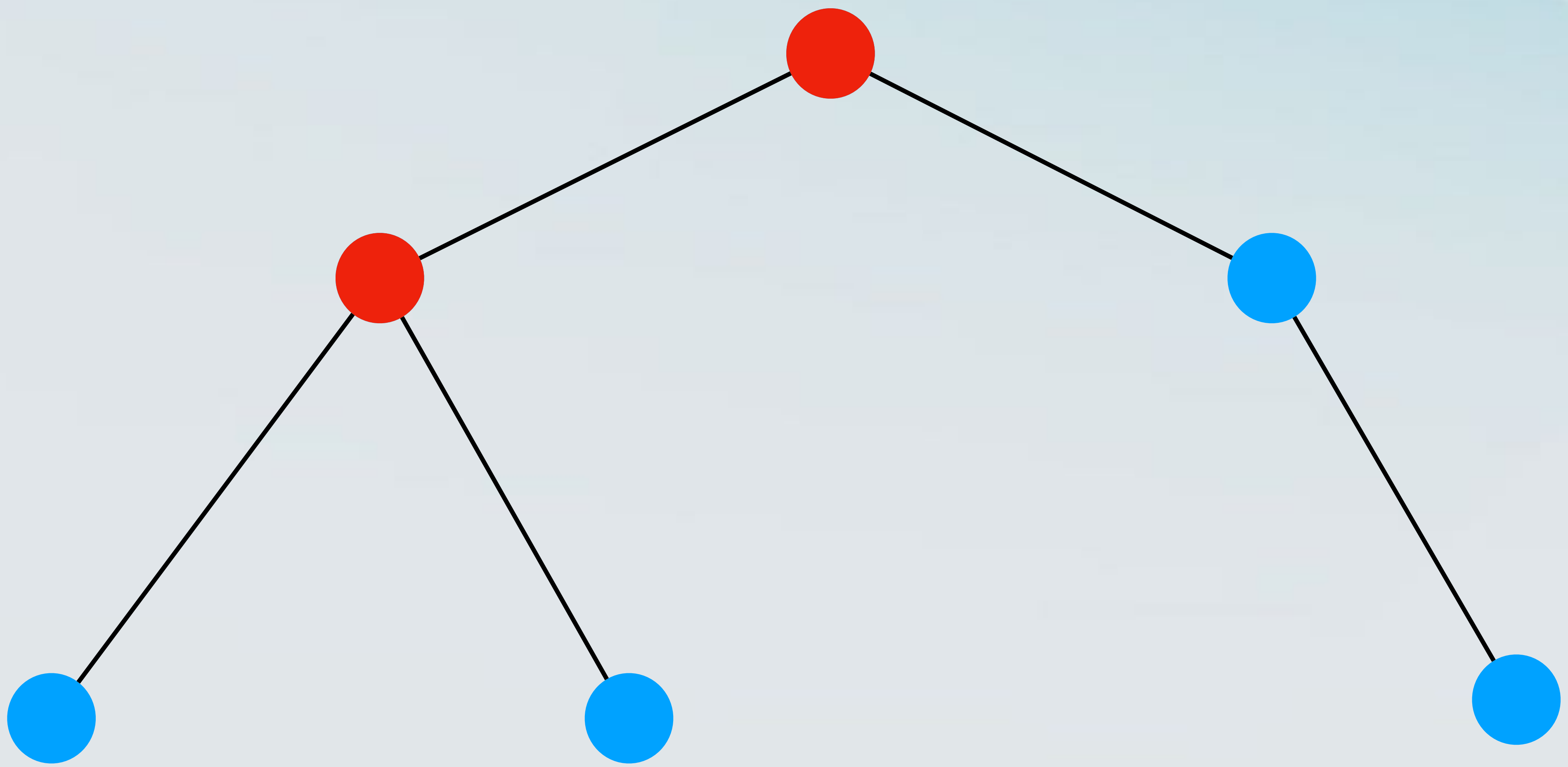
An idea on a tree



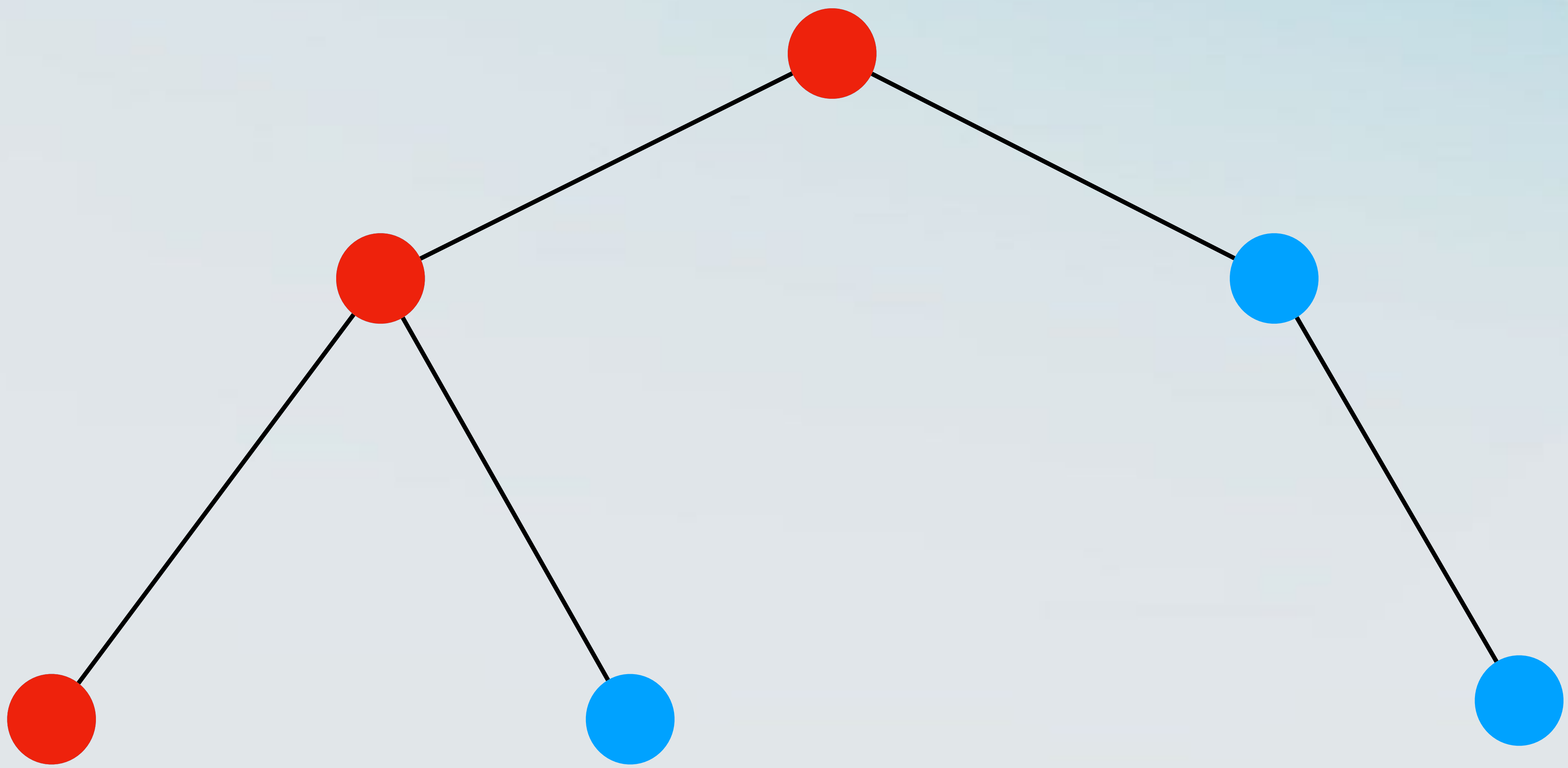
An idea on a tree



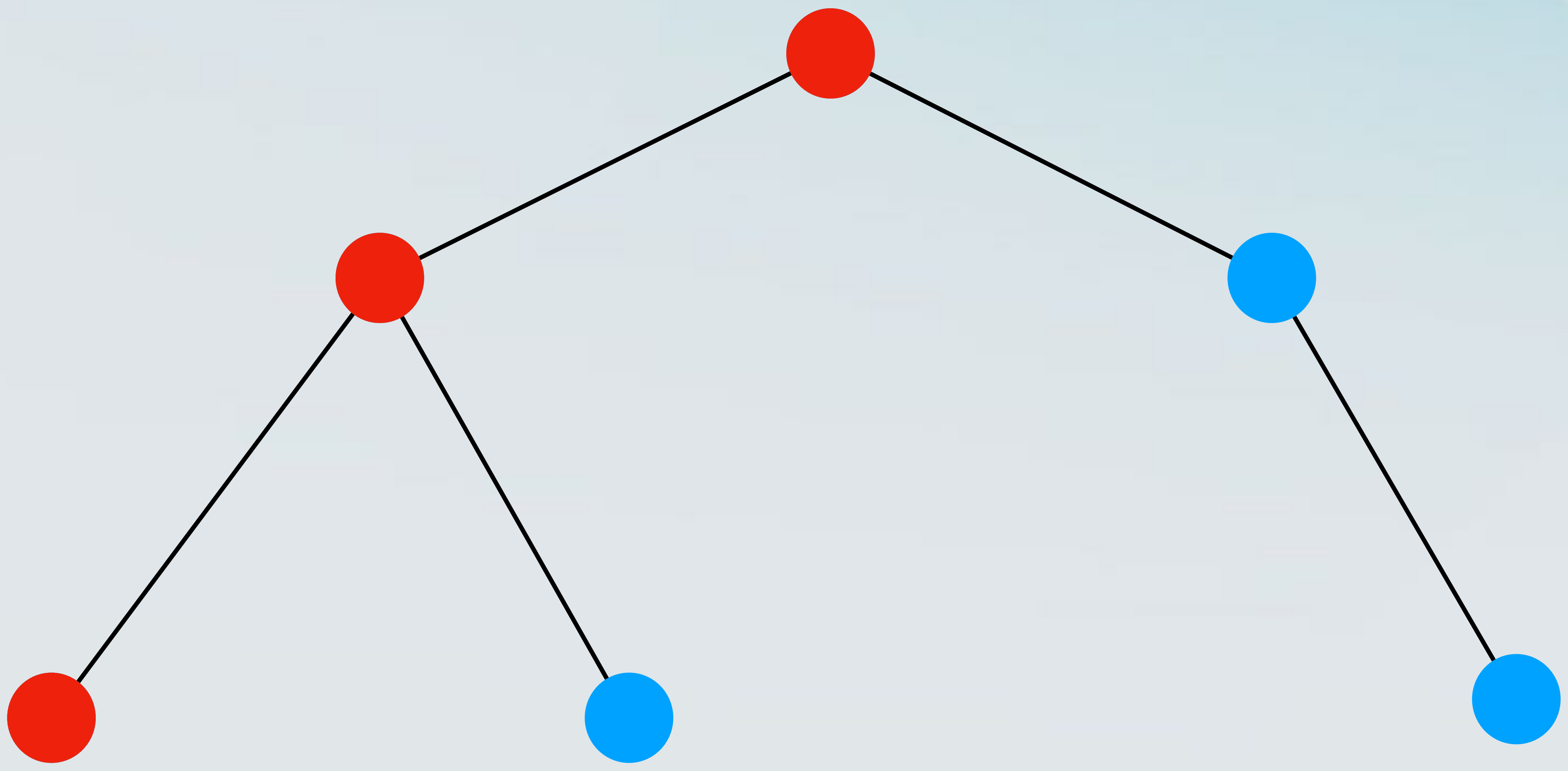
An idea on a tree



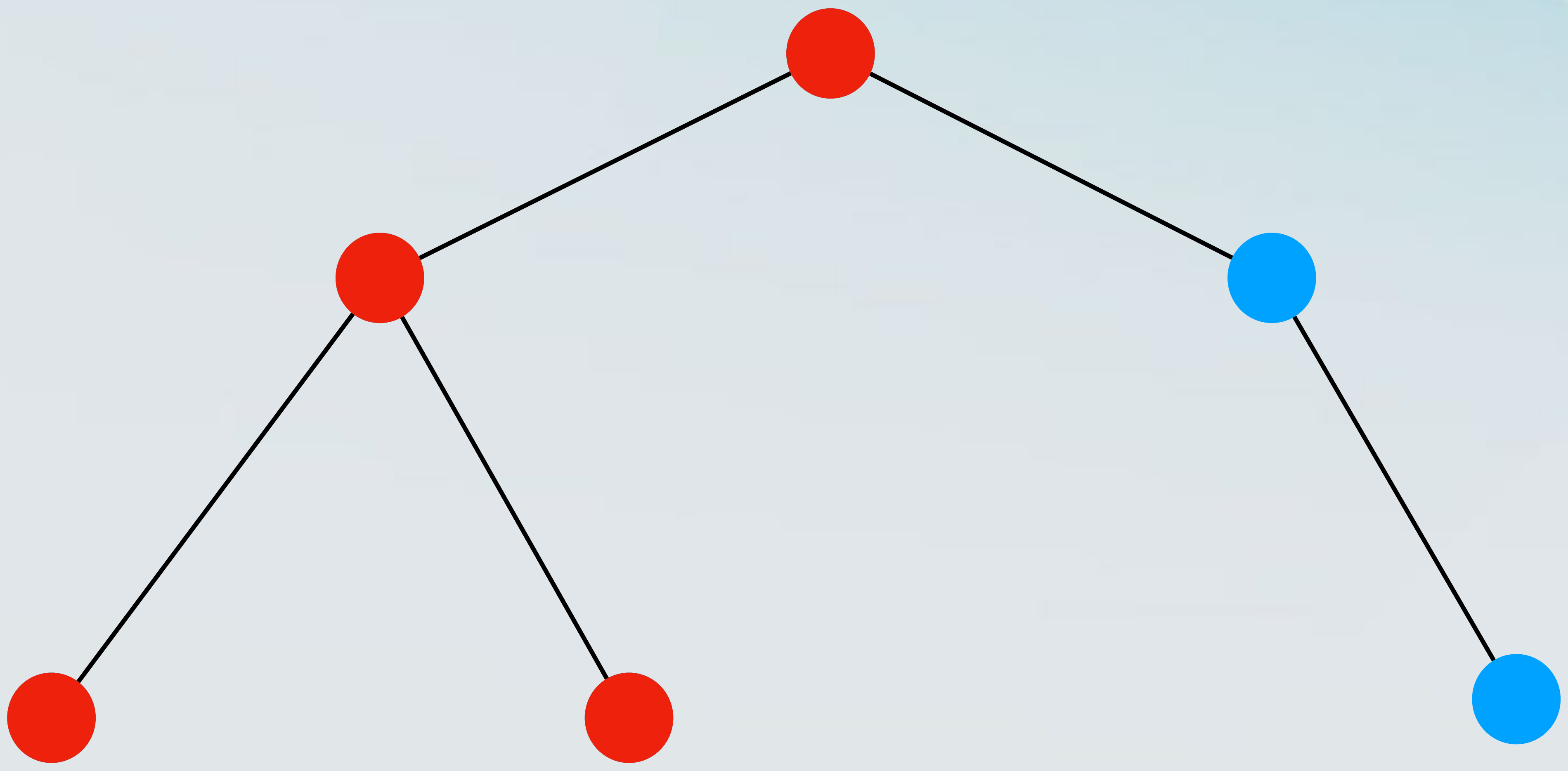
An idea on a tree



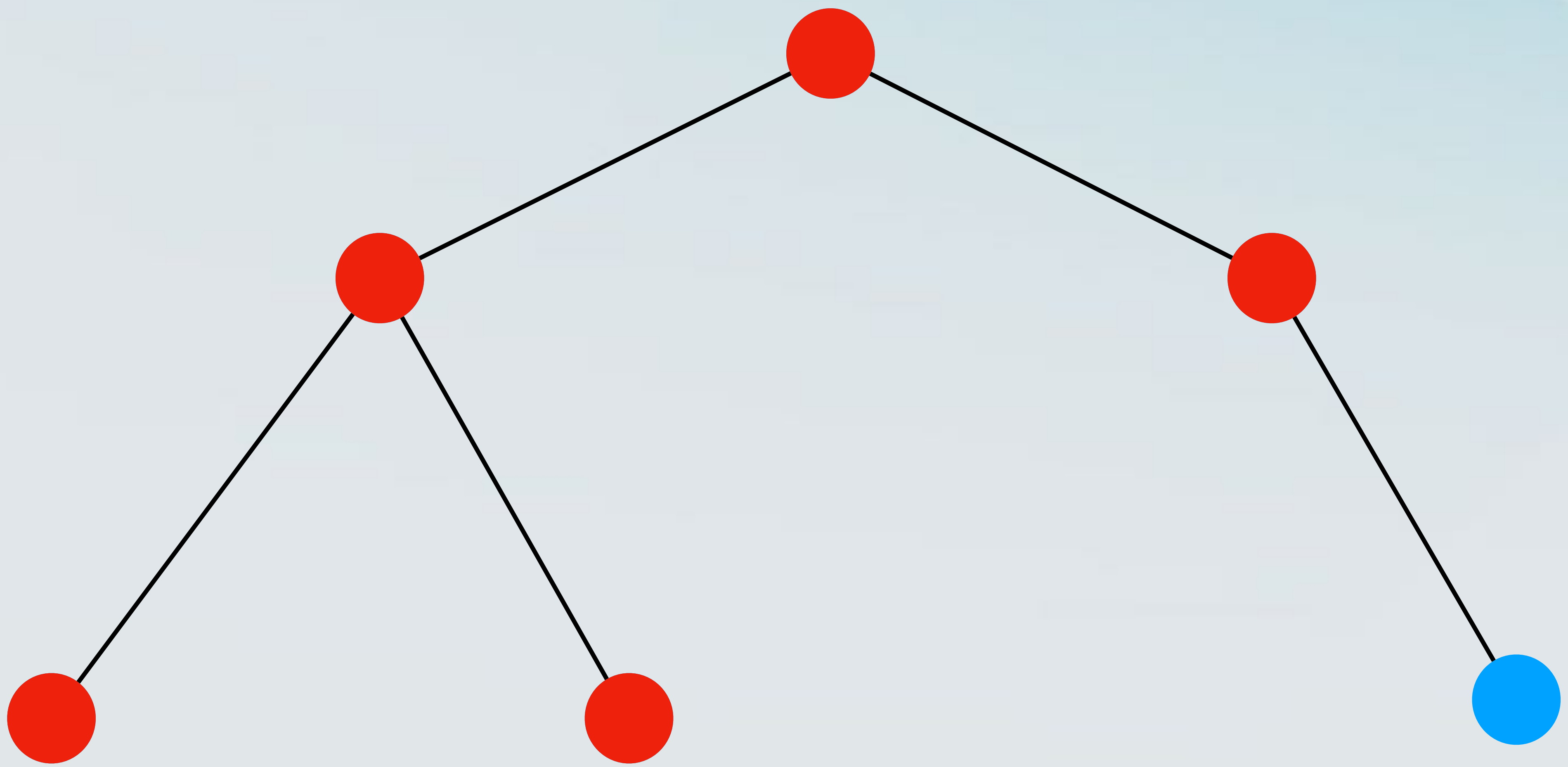
An idea on a tree



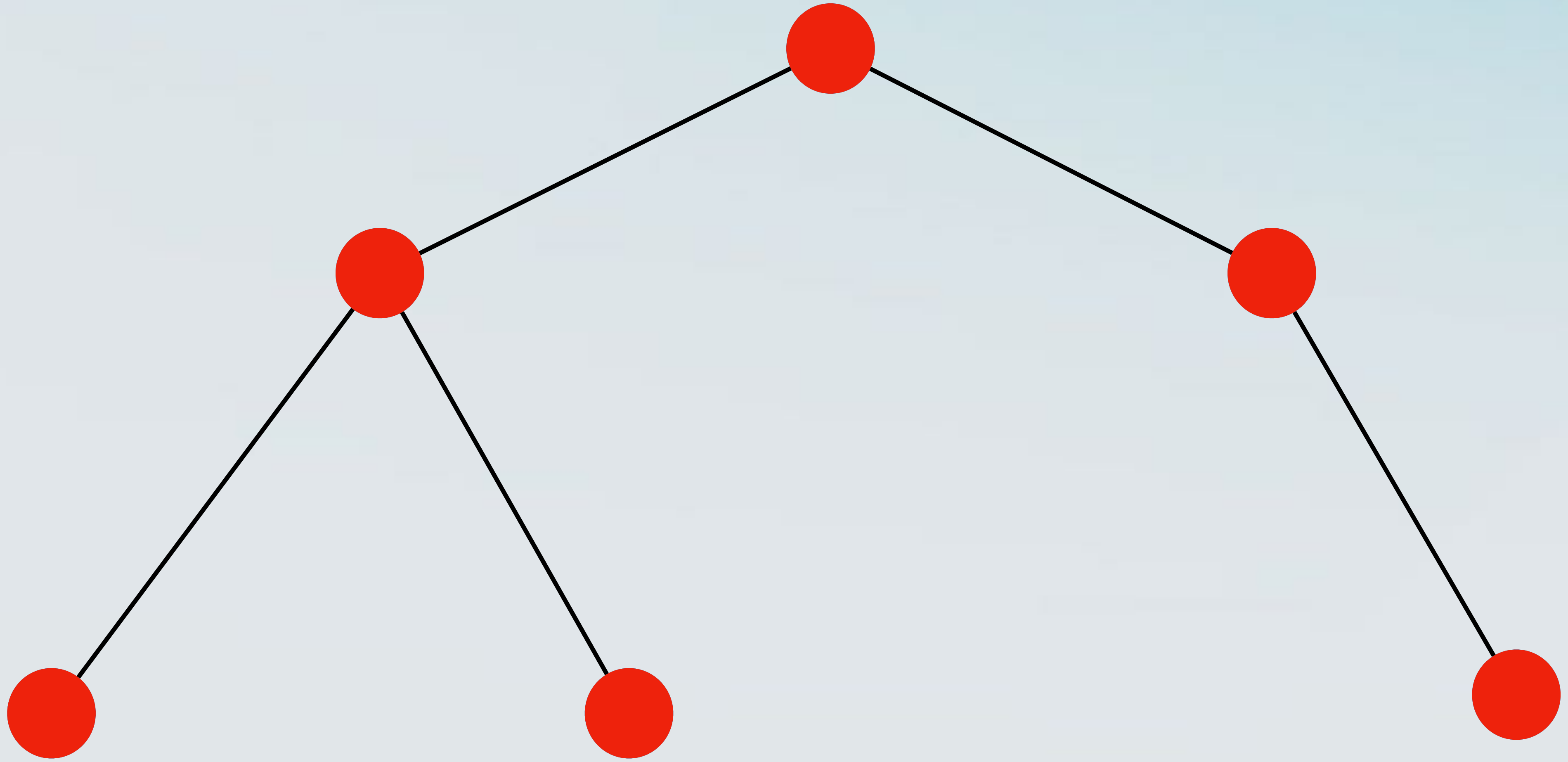
An idea on a tree



An idea on a tree



An idea on a tree



Graph Traversal

Graph Traversal

- We would like to go over all the possible nodes of an (undirected) graph.

Graph Traversal

- We would like to go over all the possible nodes of an (undirected) graph.
- There are different ways of doing that.

Graph Traversal

- We would like to go over all the possible nodes of an (undirected) graph.
- There are different ways of doing that.
- Two systematic ways:

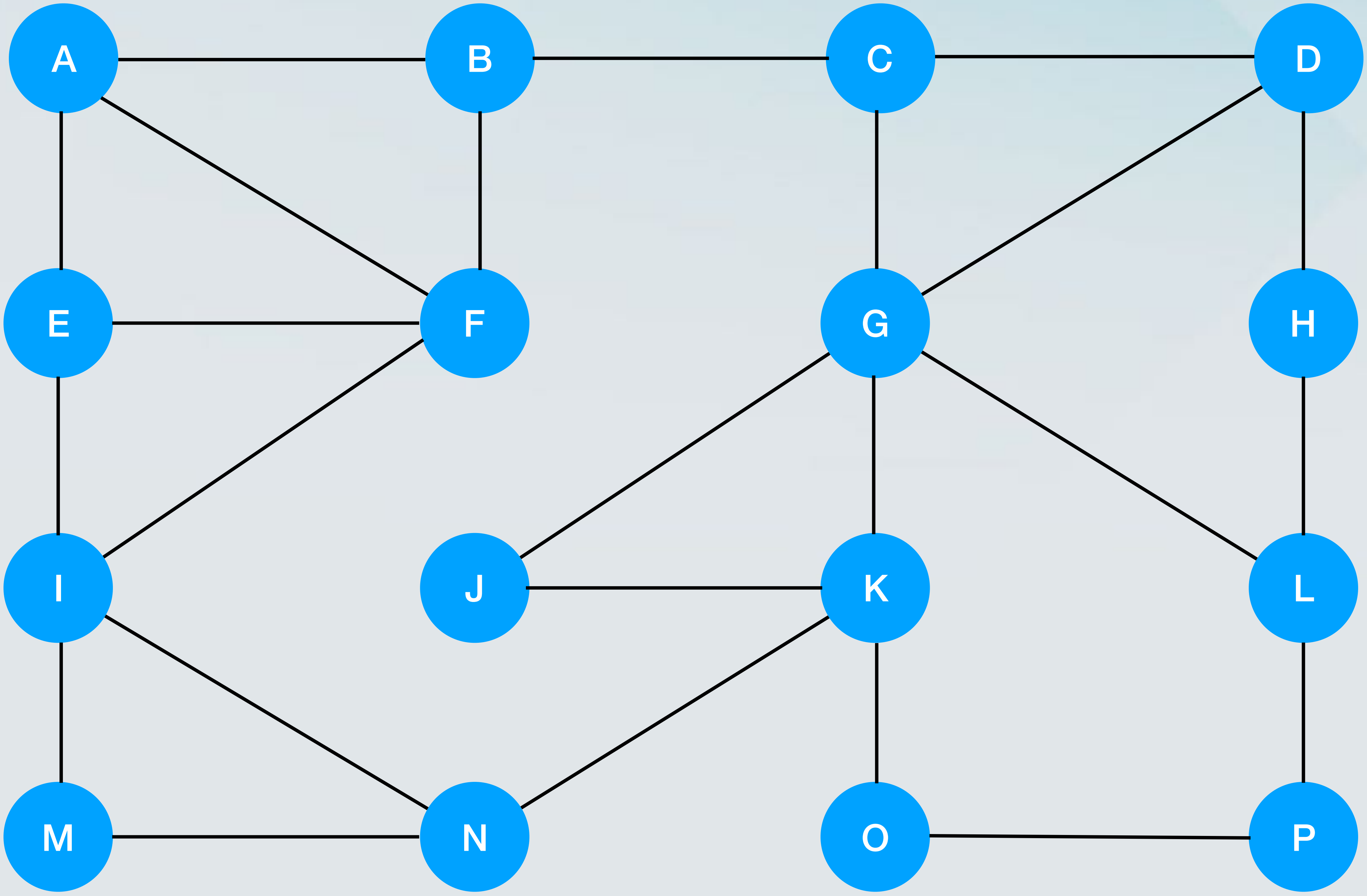
Graph Traversal

- We would like to go over all the possible nodes of an (undirected) graph.
- There are different ways of doing that.
- Two systematic ways:
  - Depth-First Search

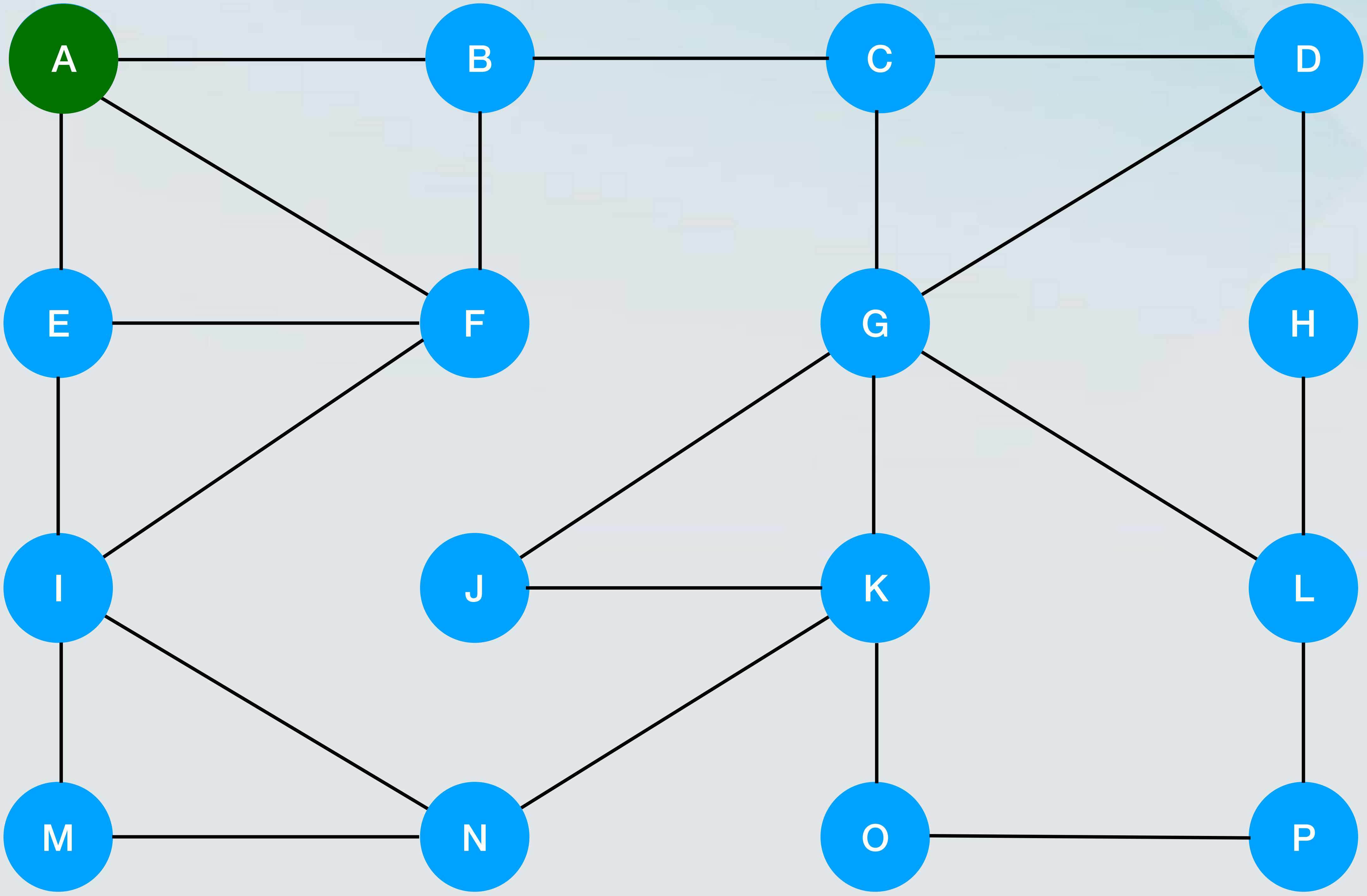
Graph Traversal

- We would like to go over all the possible nodes of an (undirected) graph.
- There are different ways of doing that.
- Two systematic ways:
  - Depth-First Search
  - Breadth-First Search

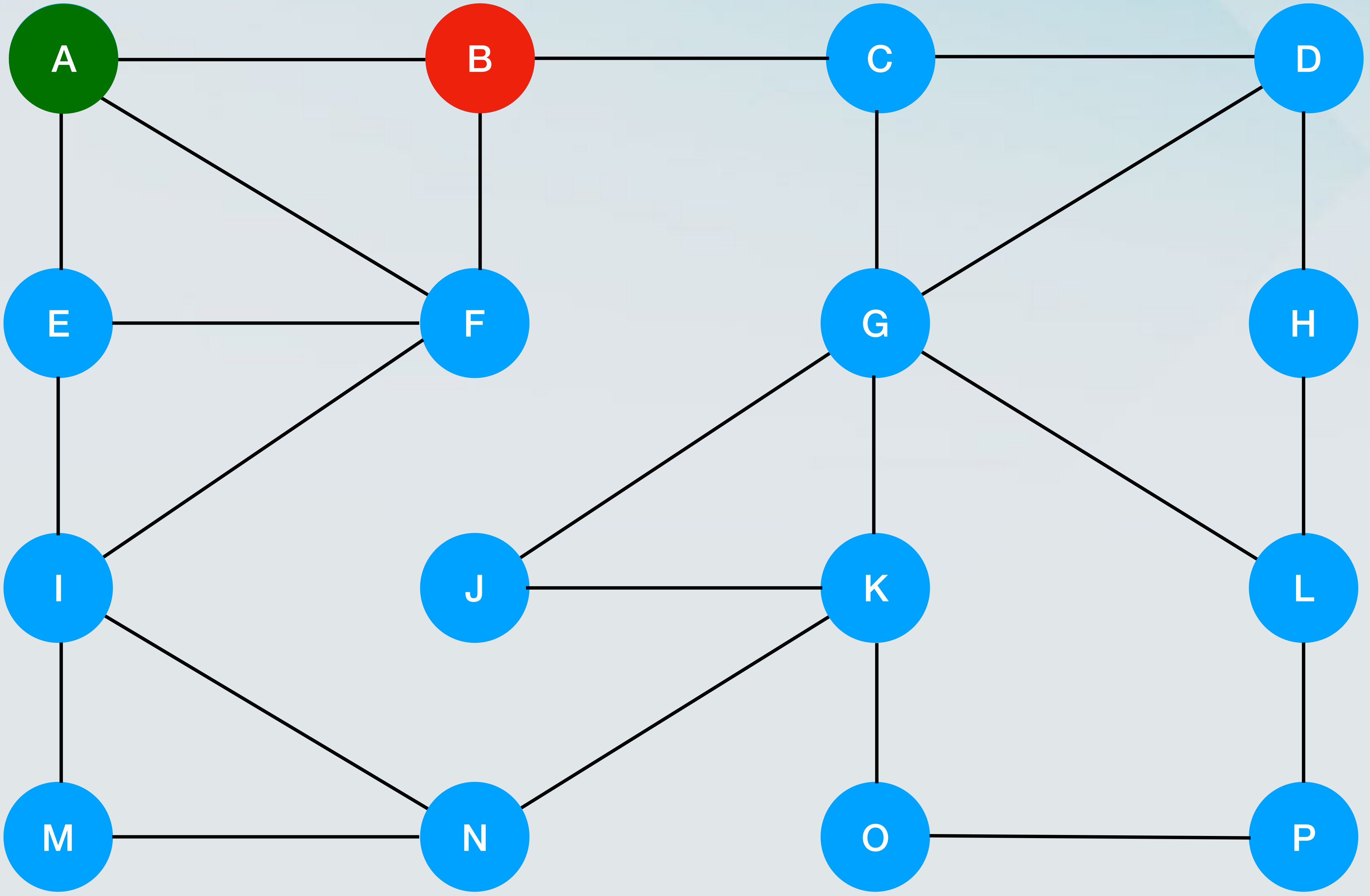
Depth-First Search



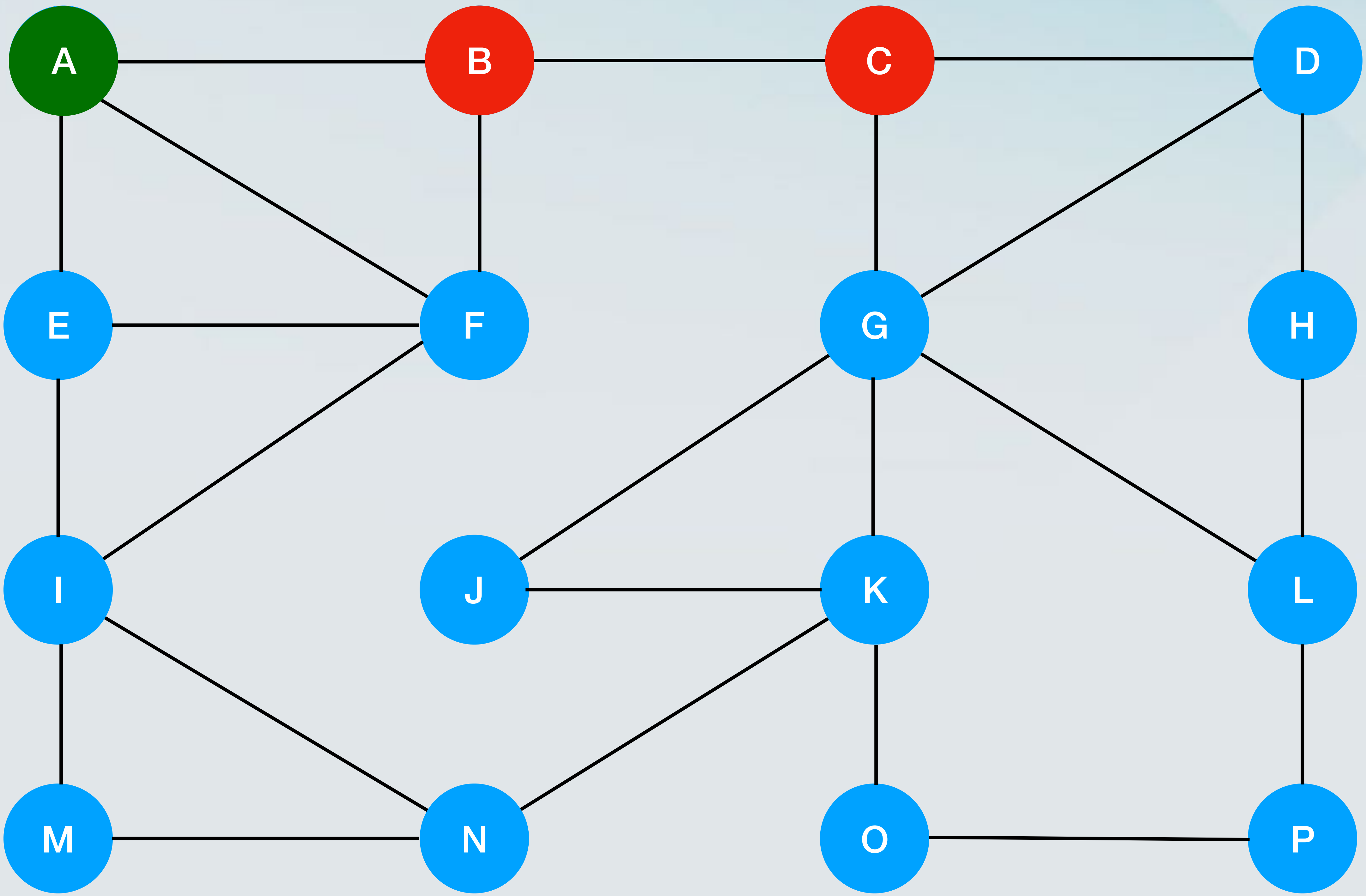
Depth-First Search



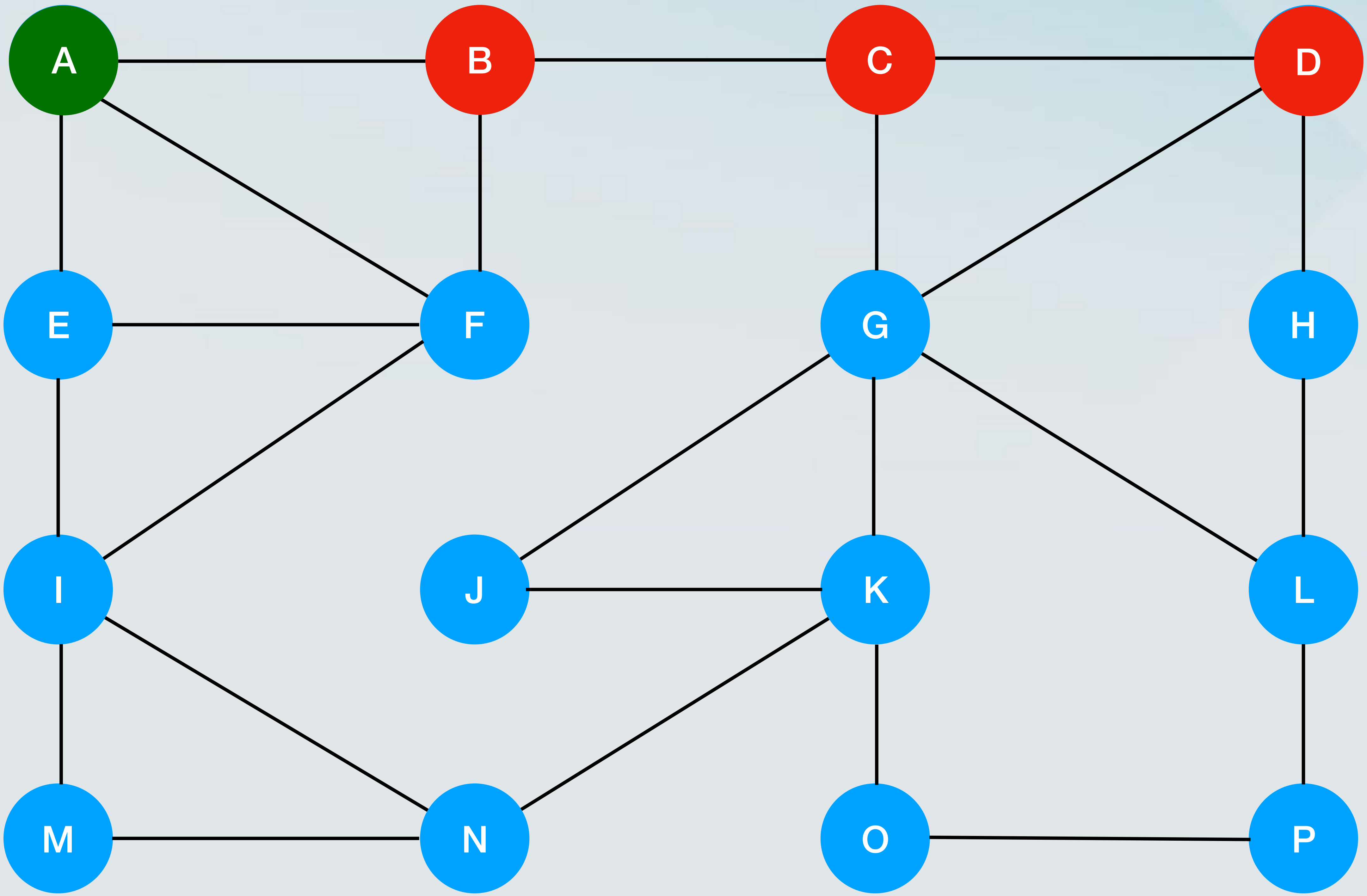
Depth-First Search



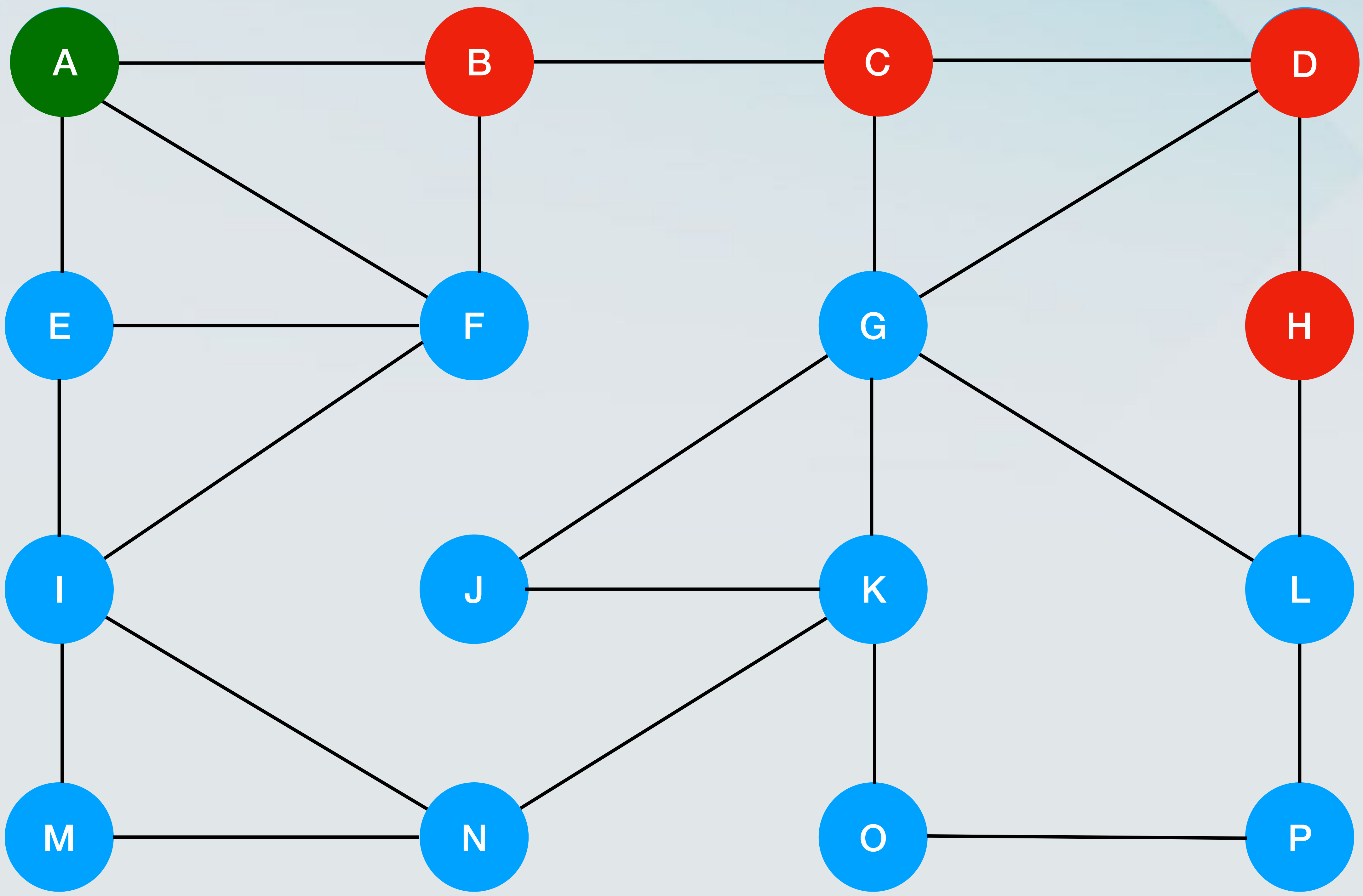
Depth-First Search



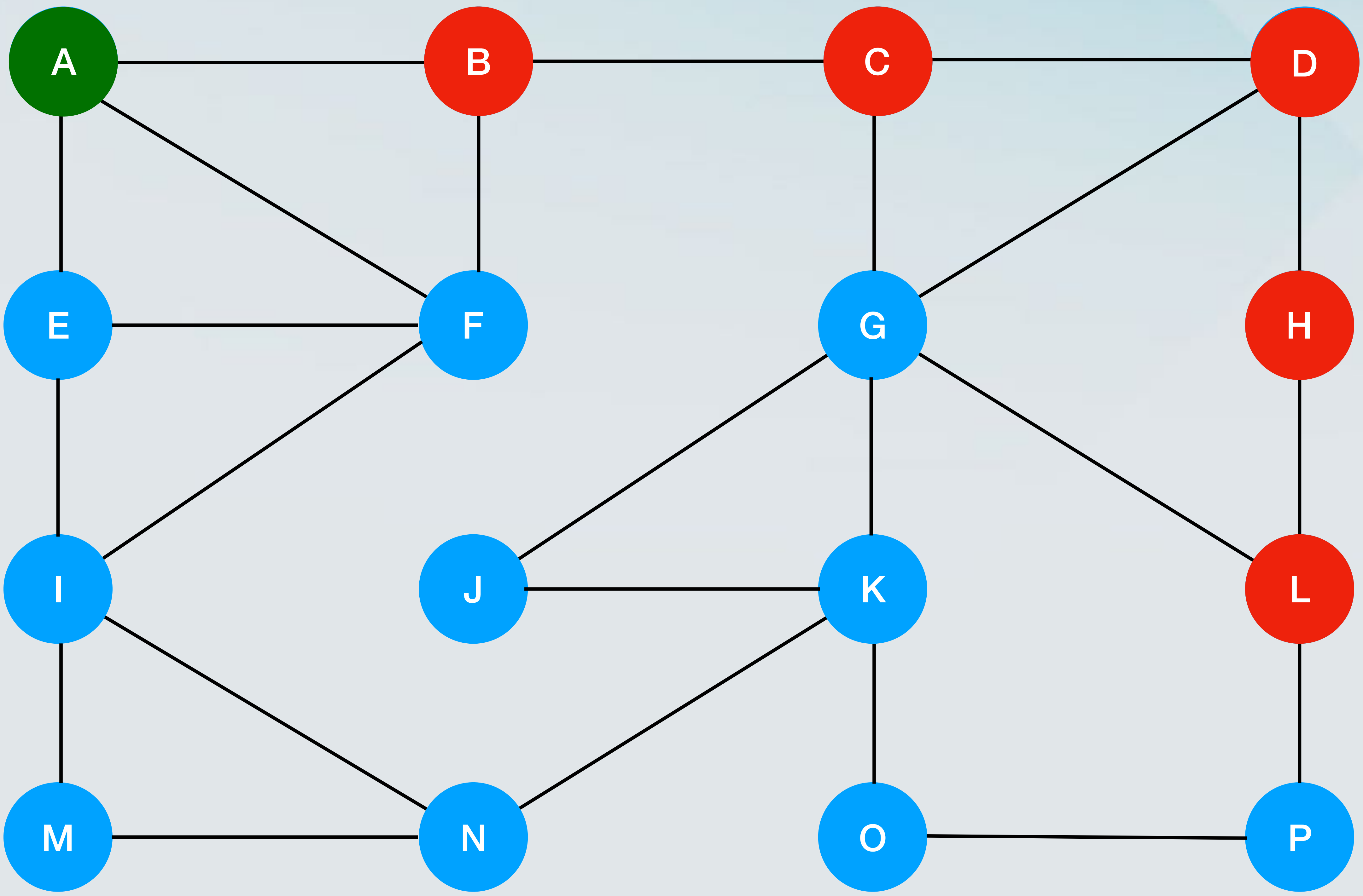
Depth-First Search



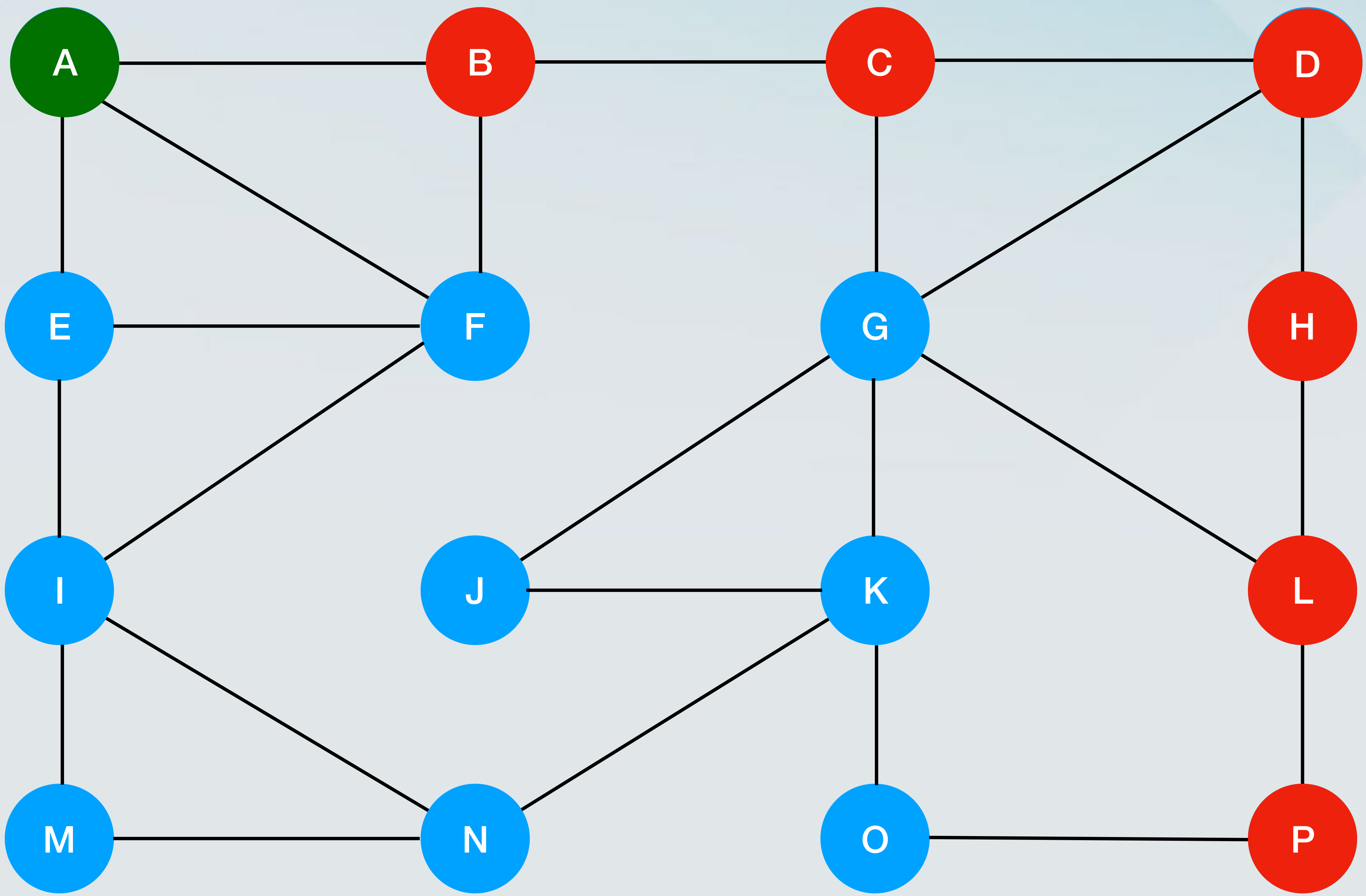
Depth-First Search



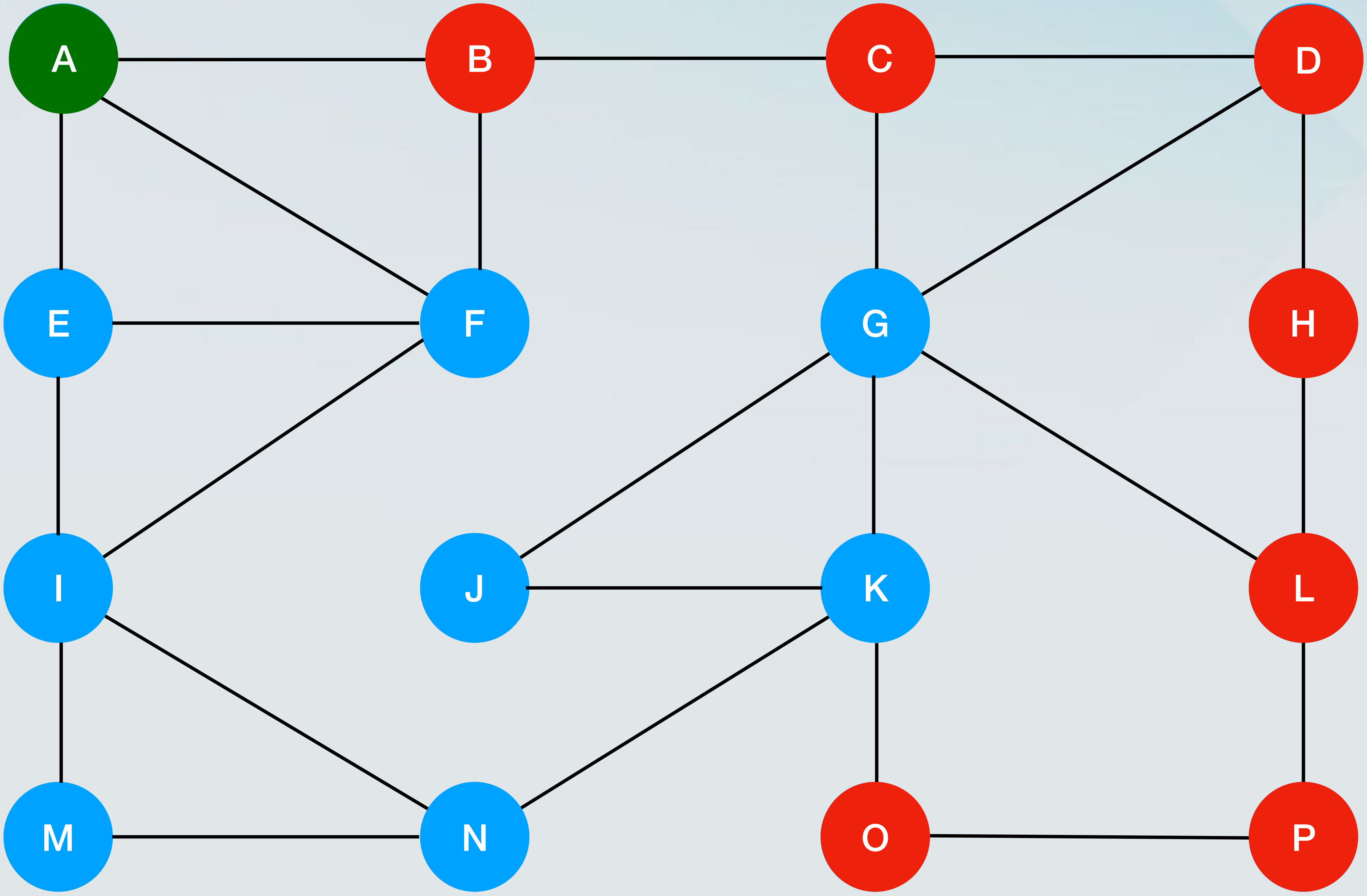
Depth-First Search



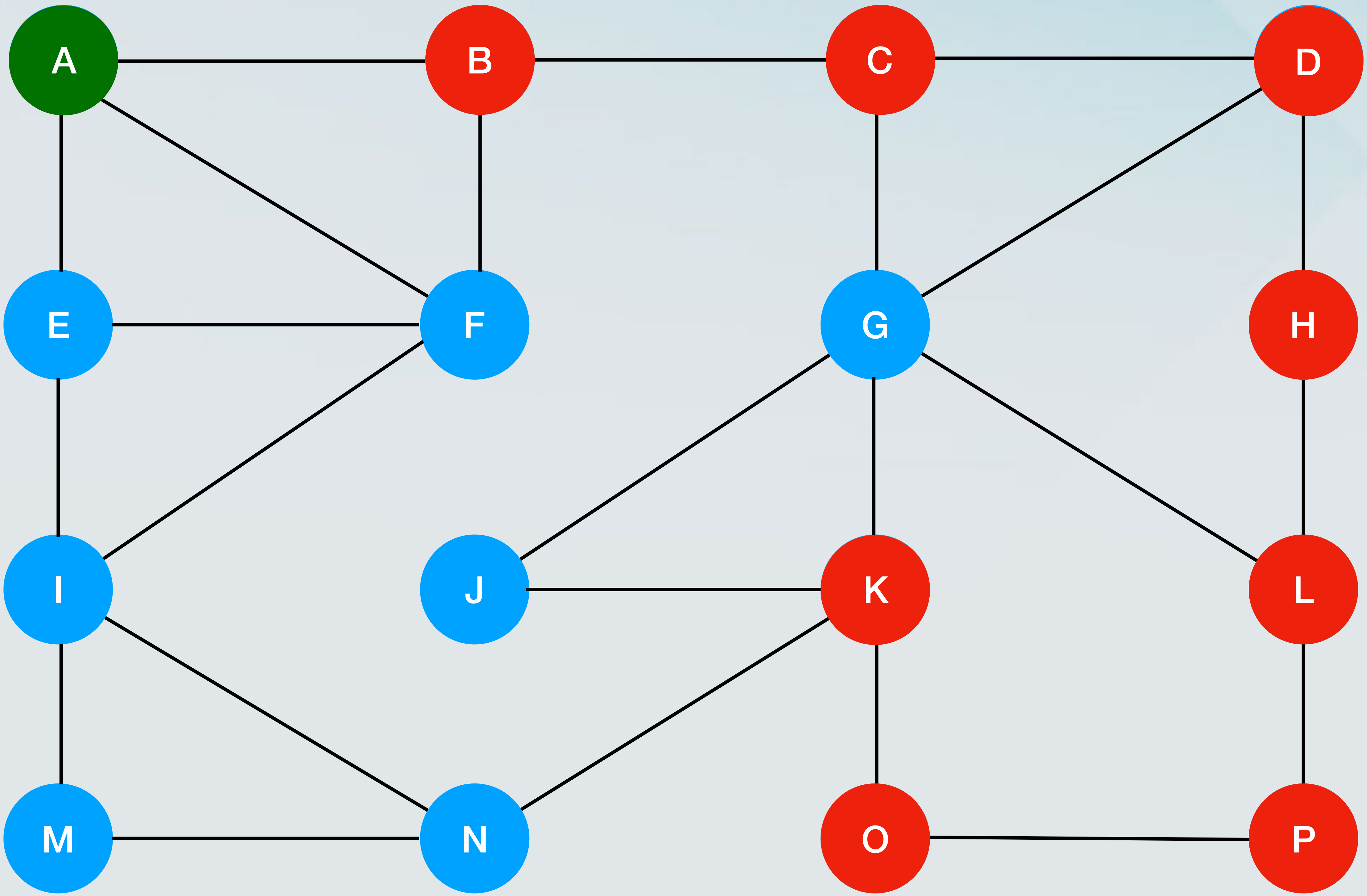
Depth-First Search



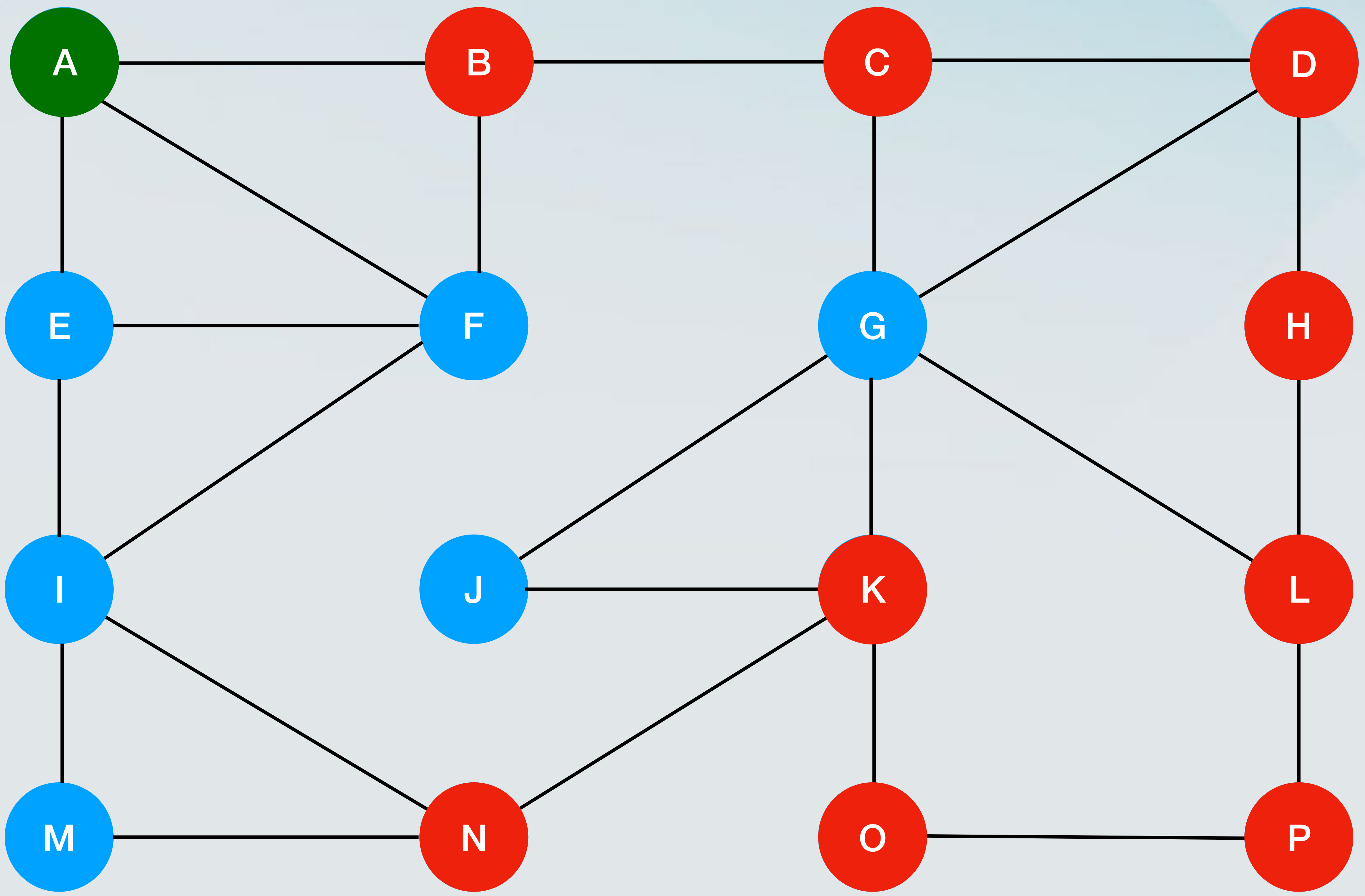
Depth-First Search



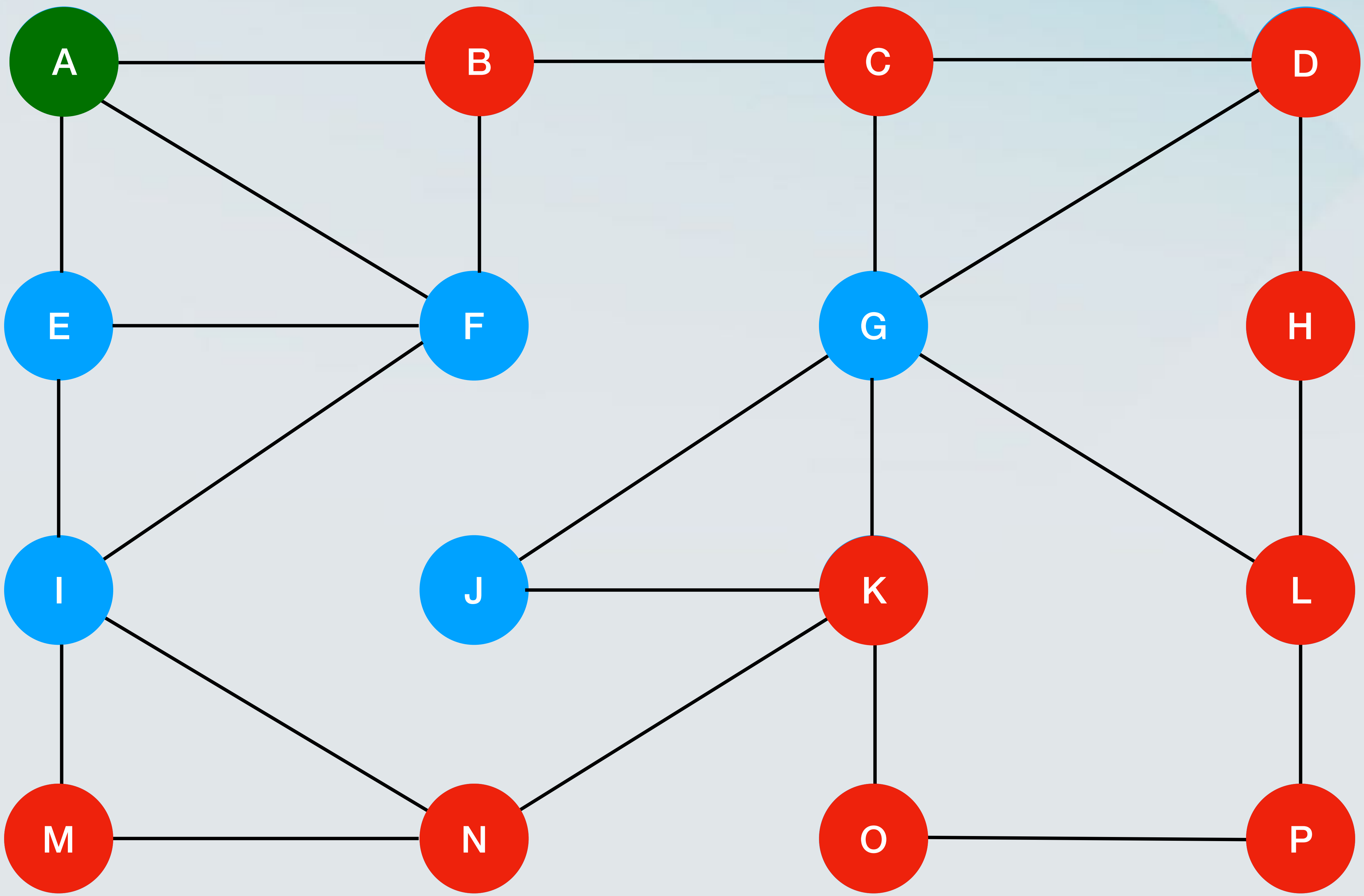
Depth-First Search



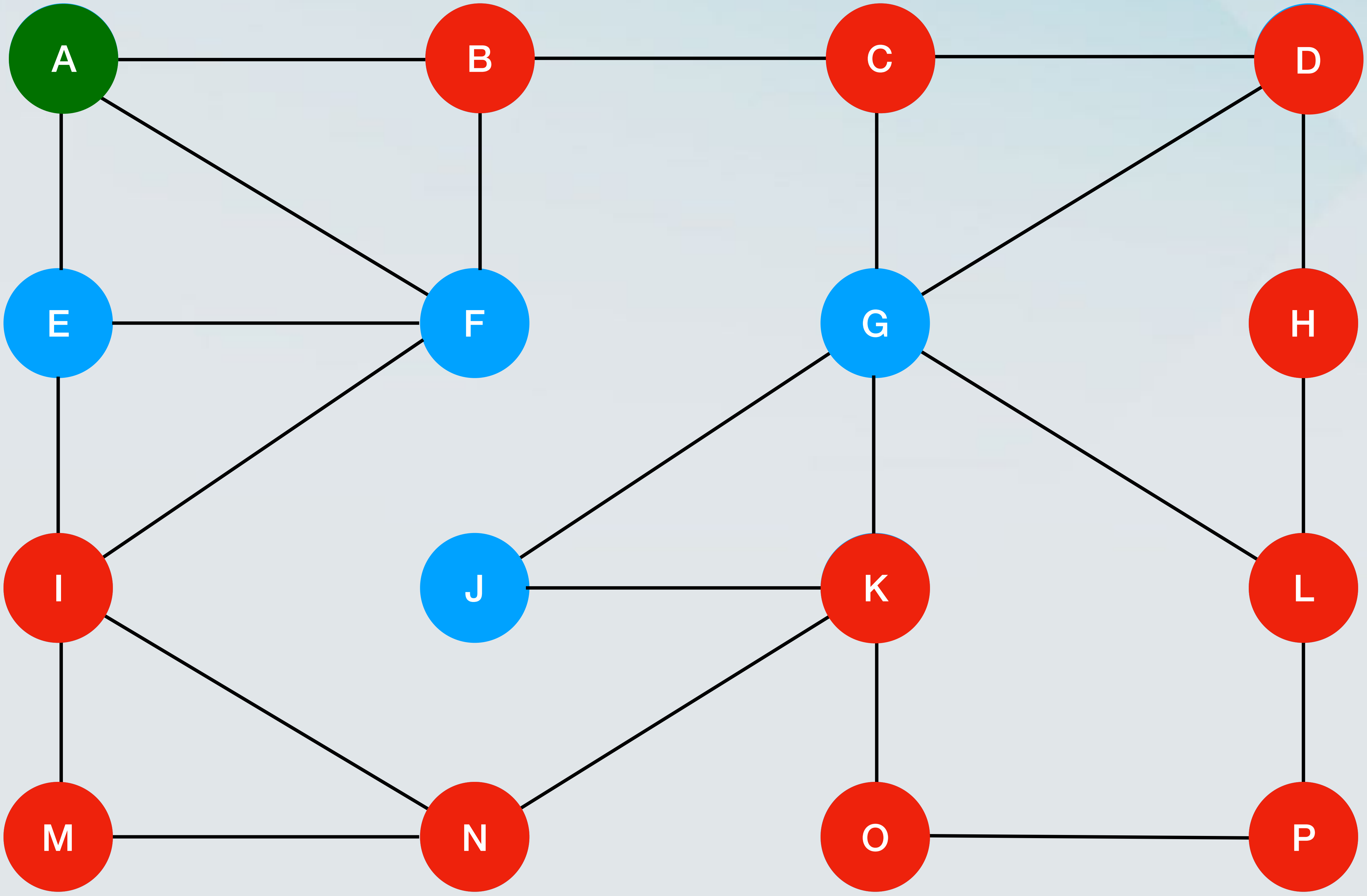
Depth-First Search



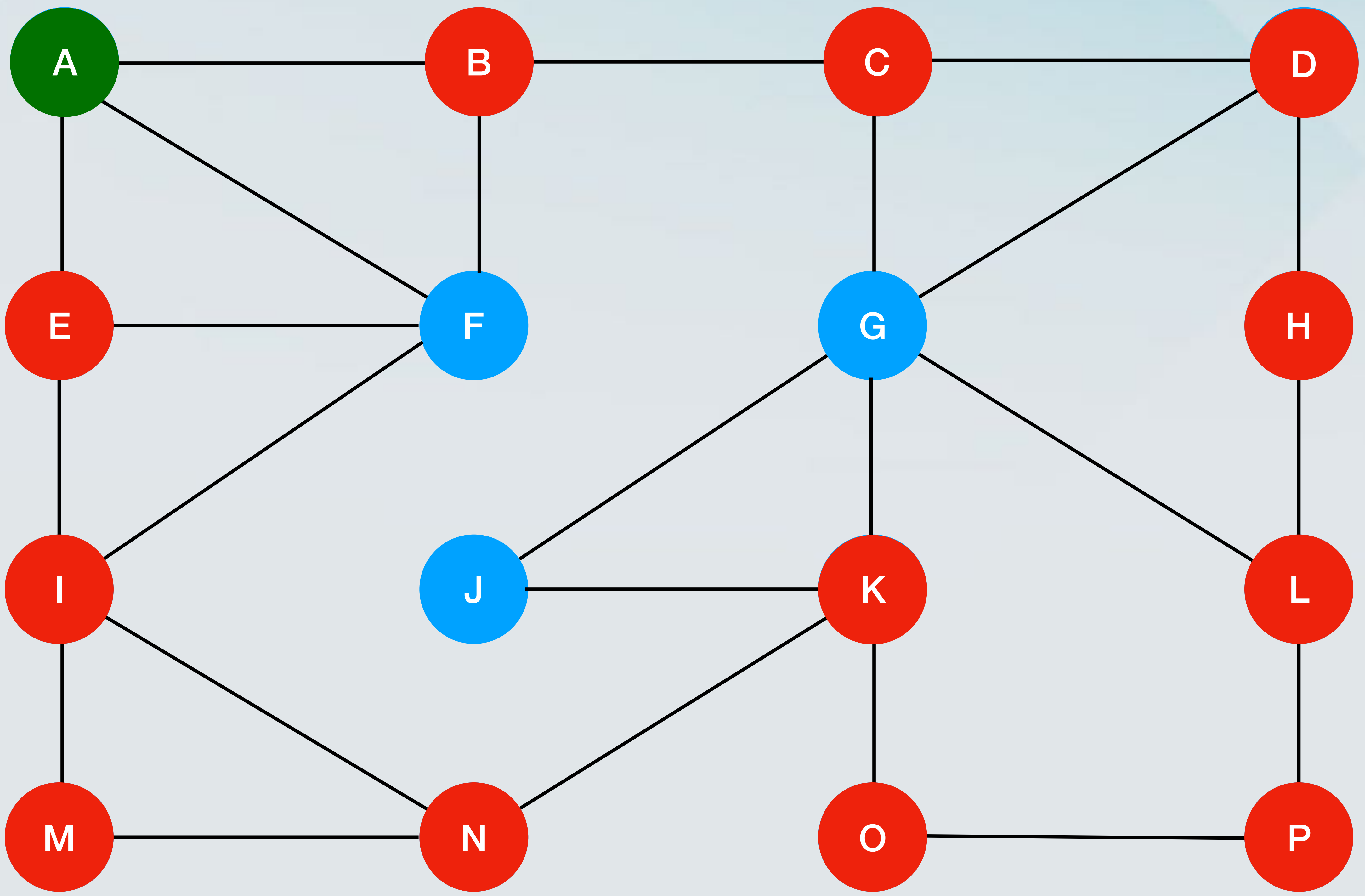
Depth-First Search



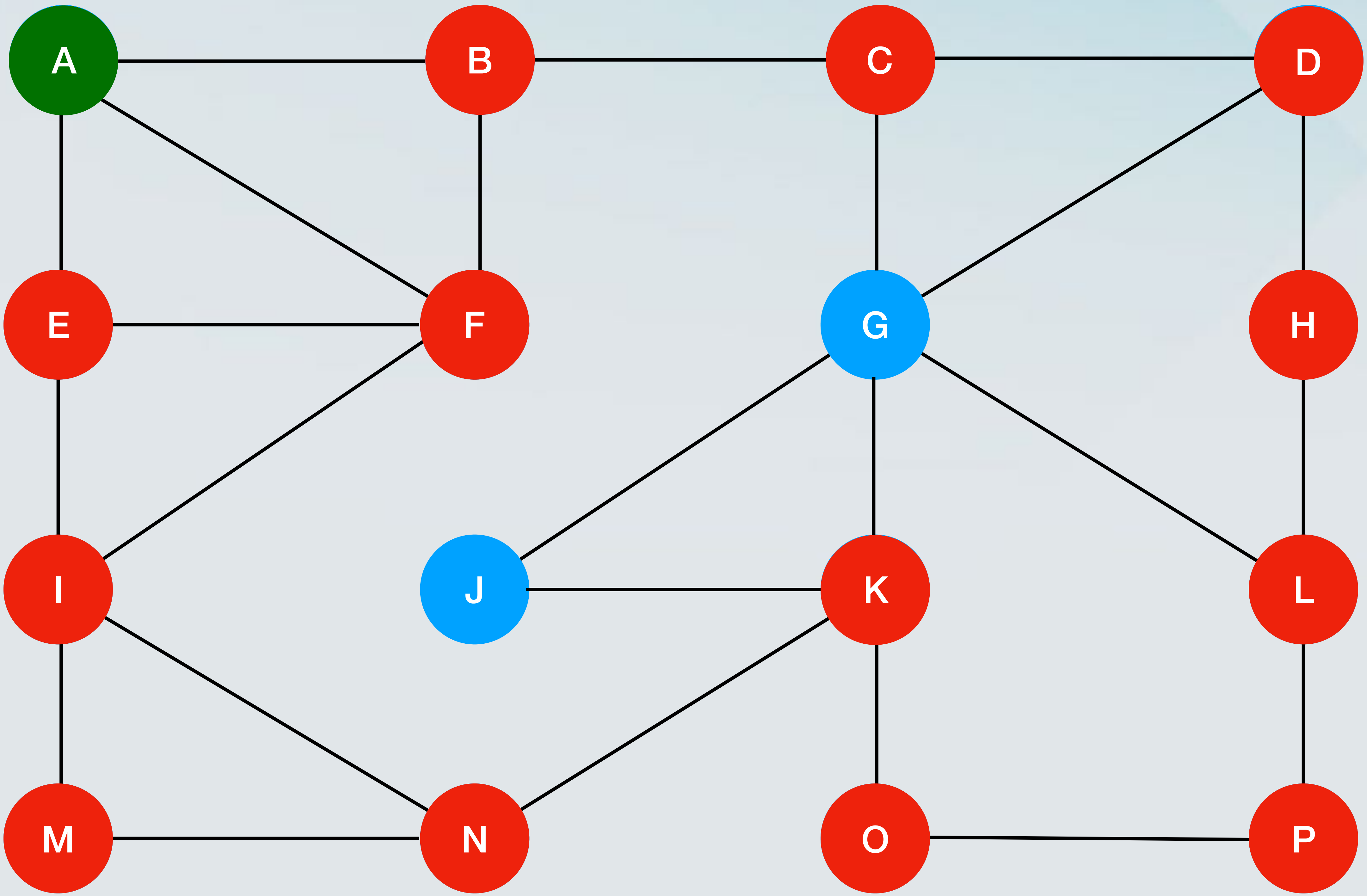
Depth-First Search



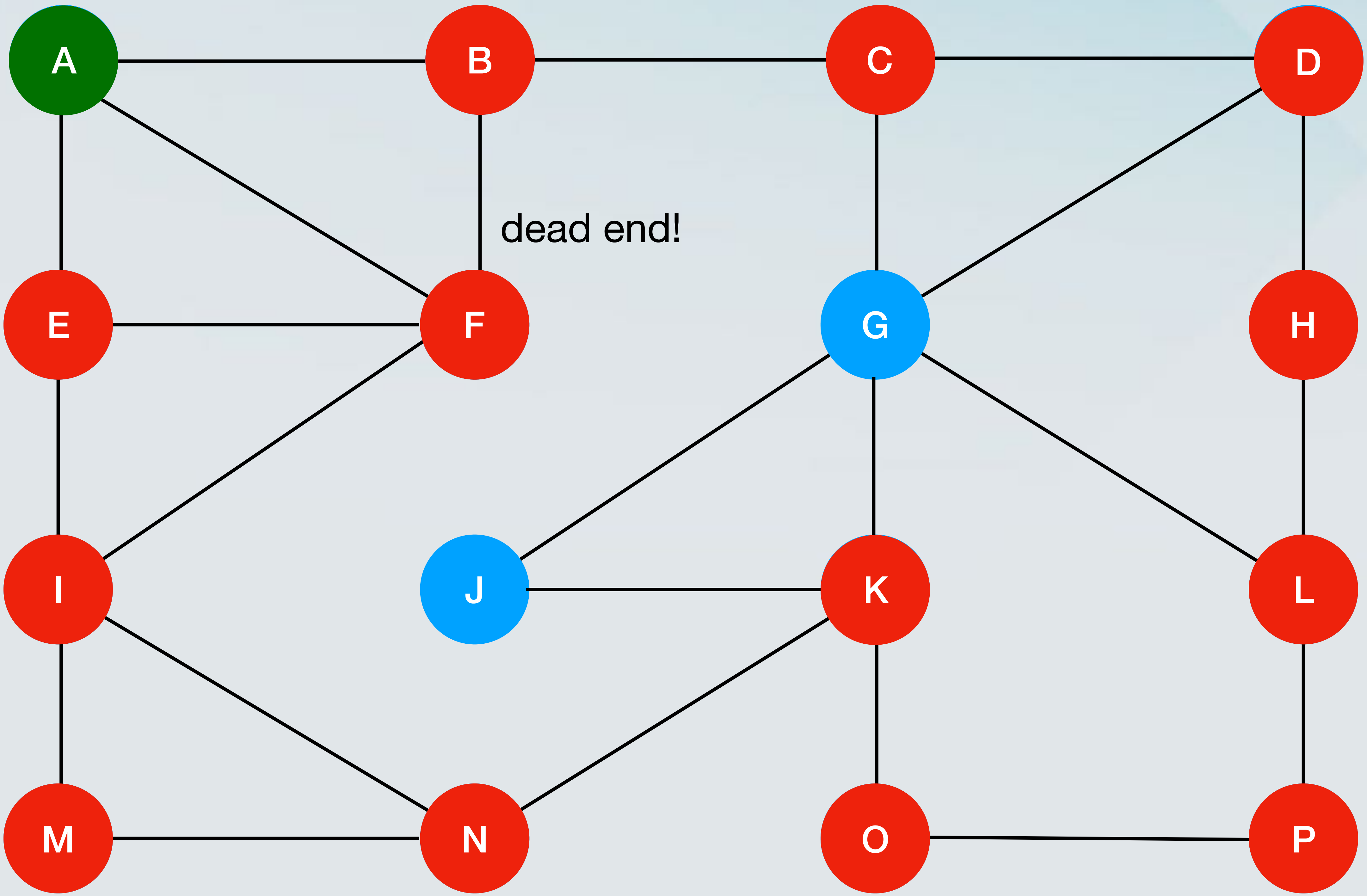
Depth-First Search



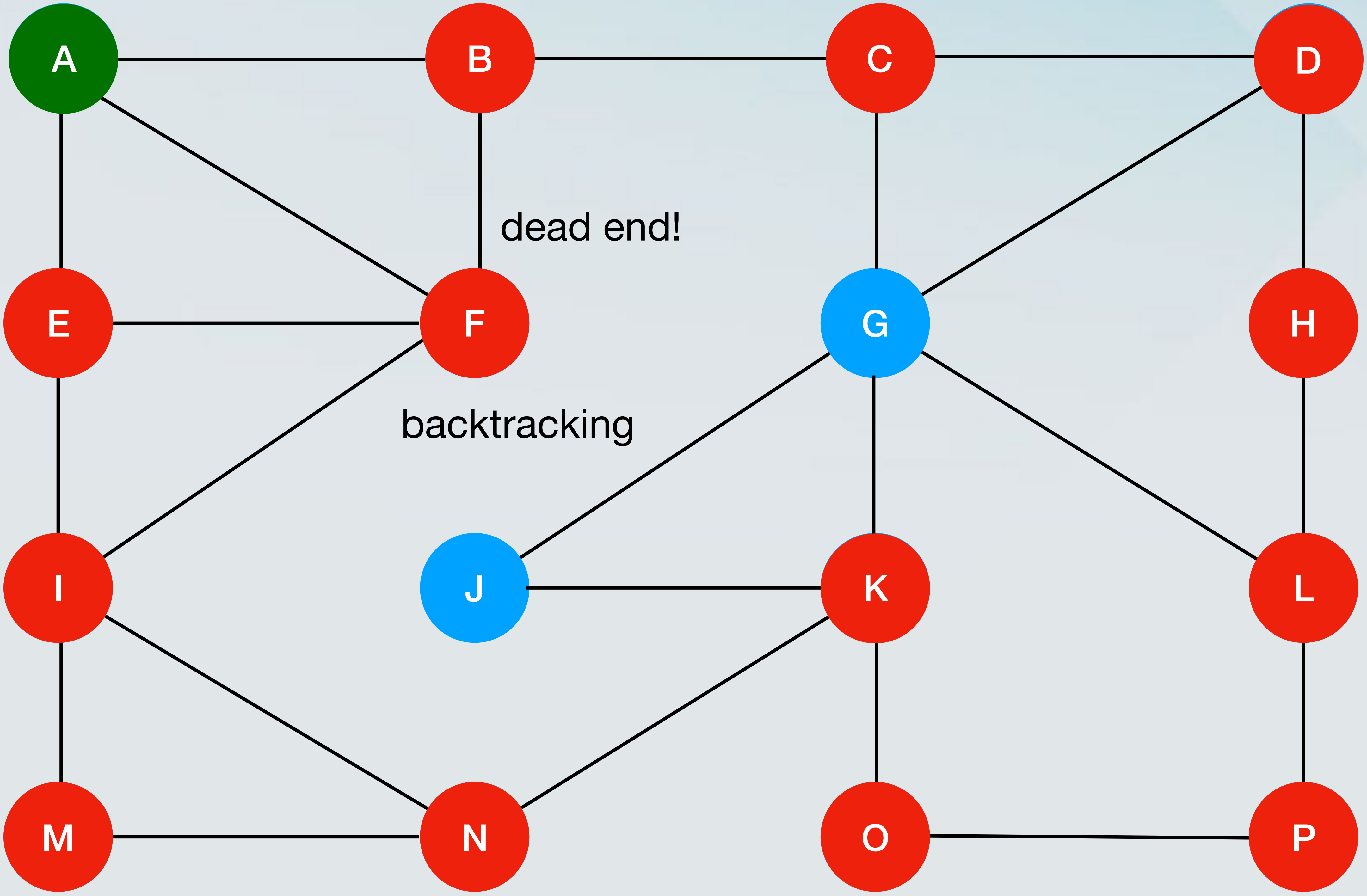
Depth-First Search



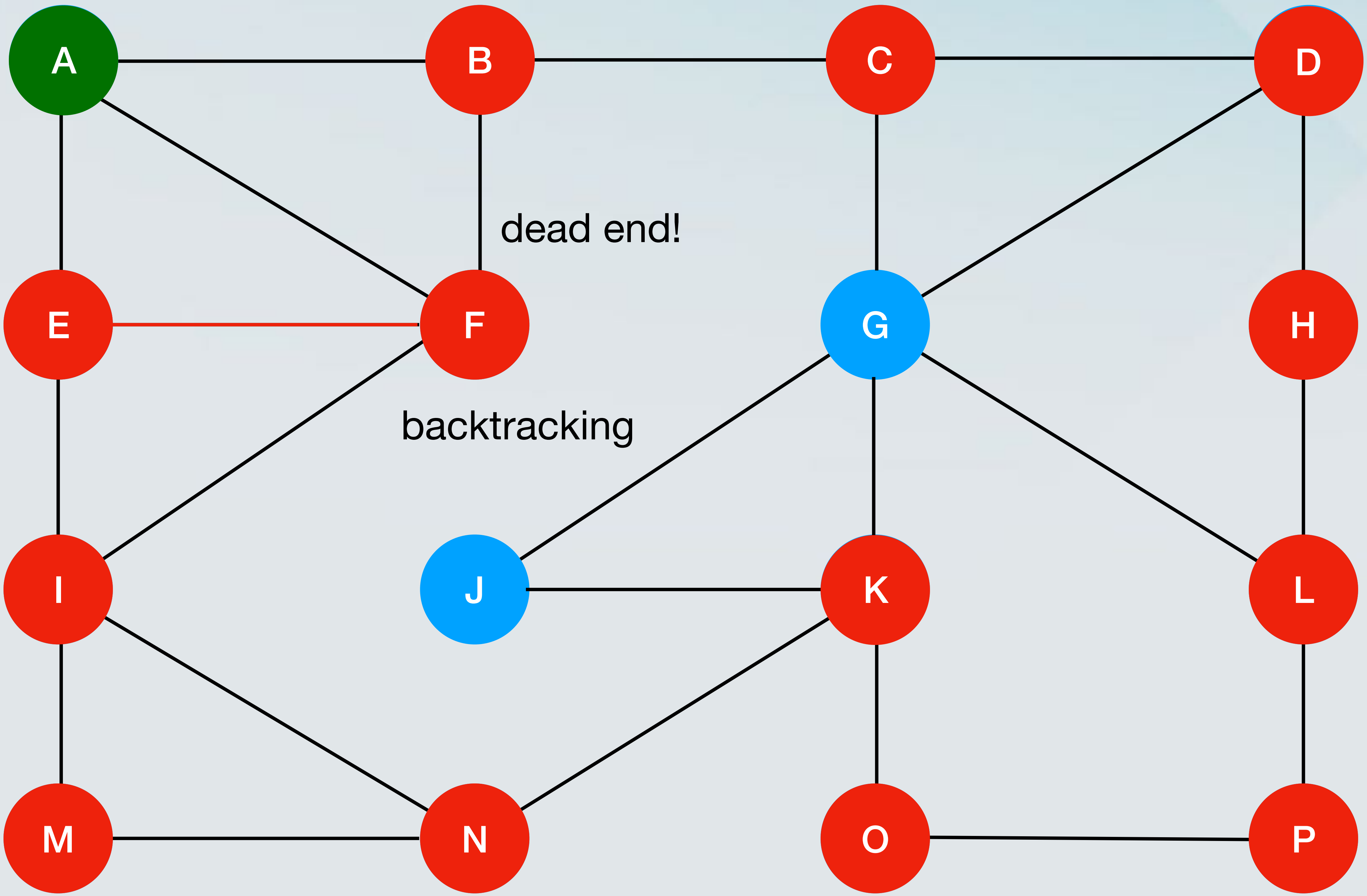
Depth-First Search



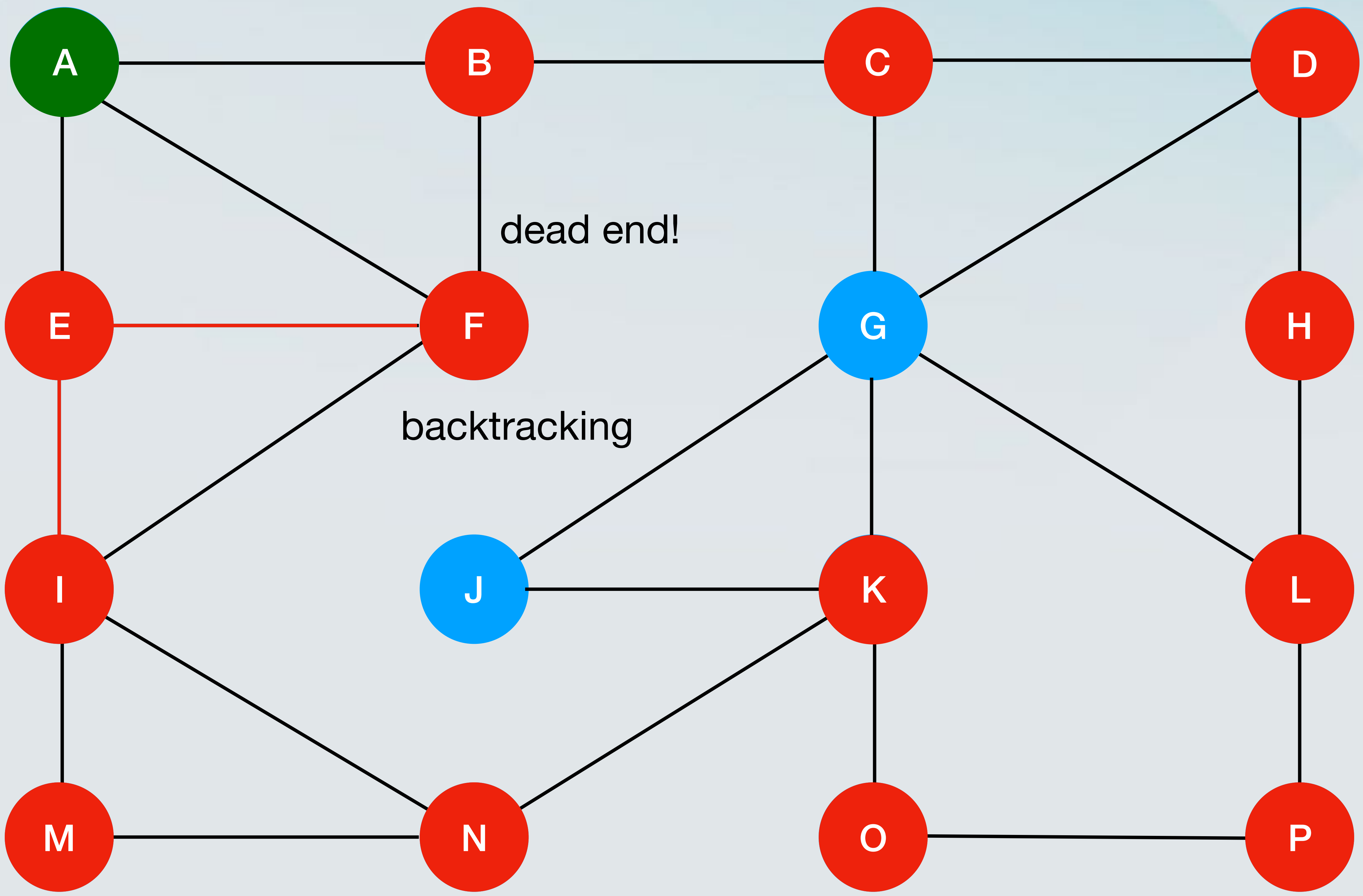
Depth-First Search



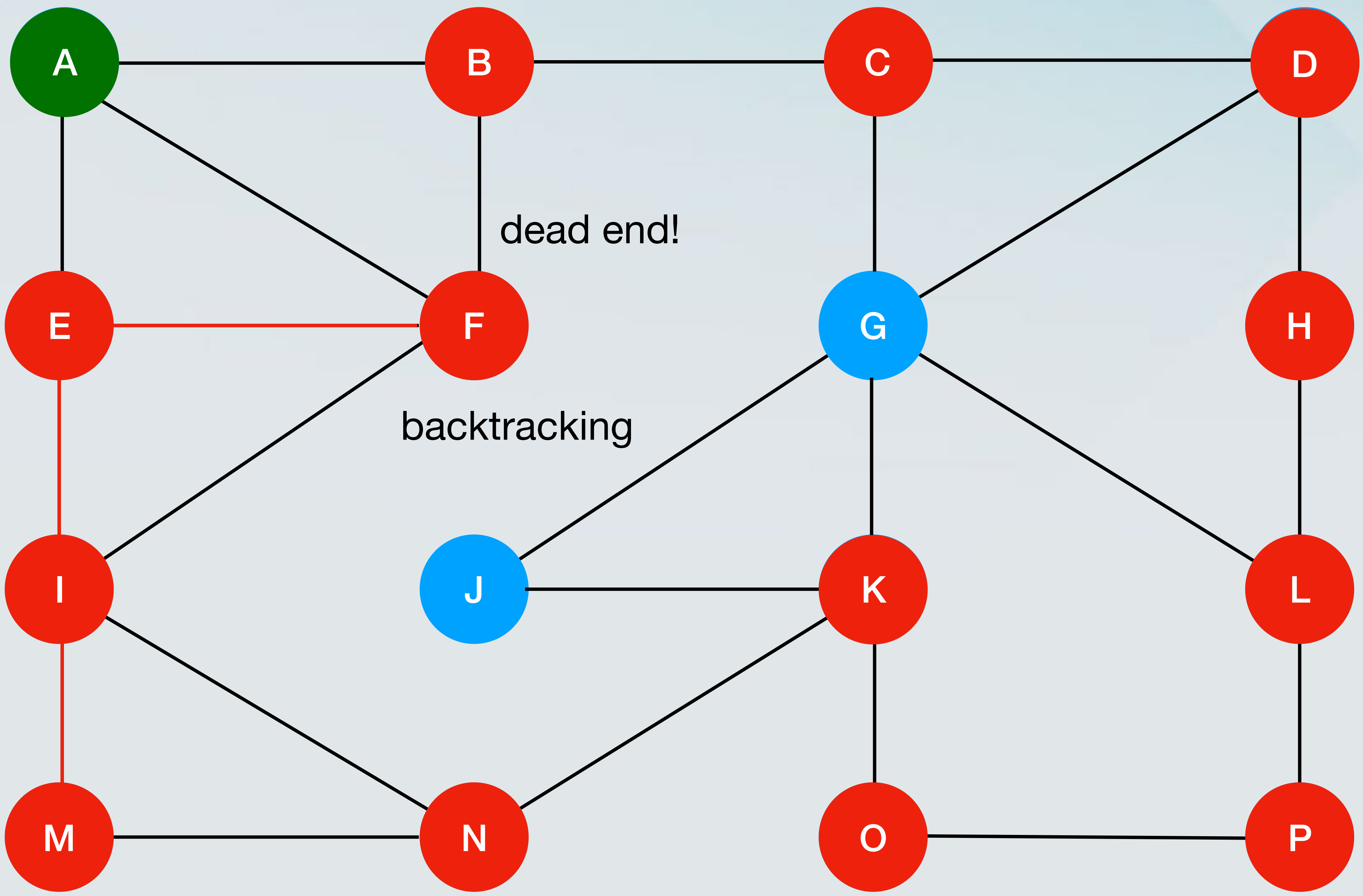
Depth-First Search



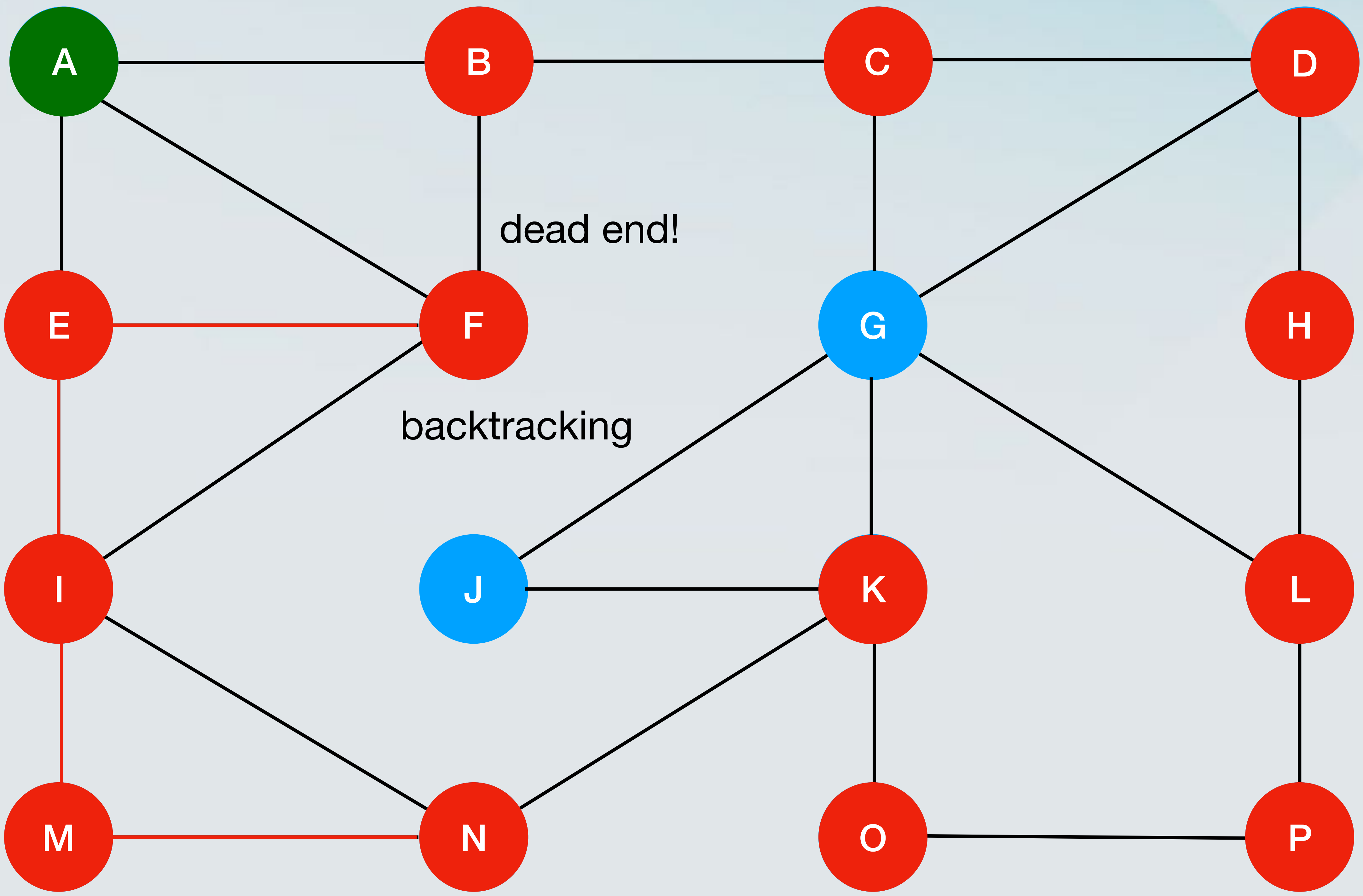
Depth-First Search



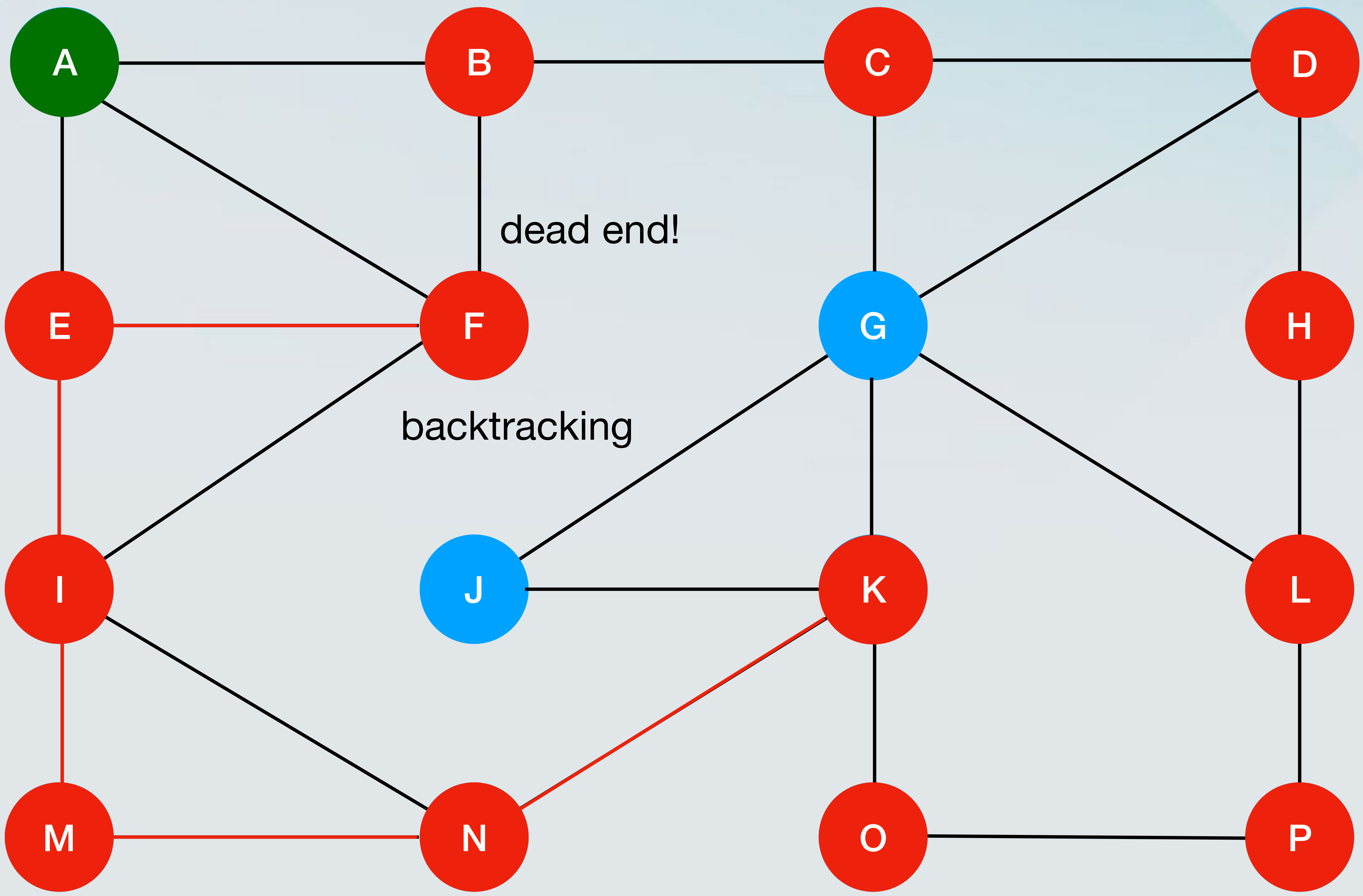
Depth-First Search



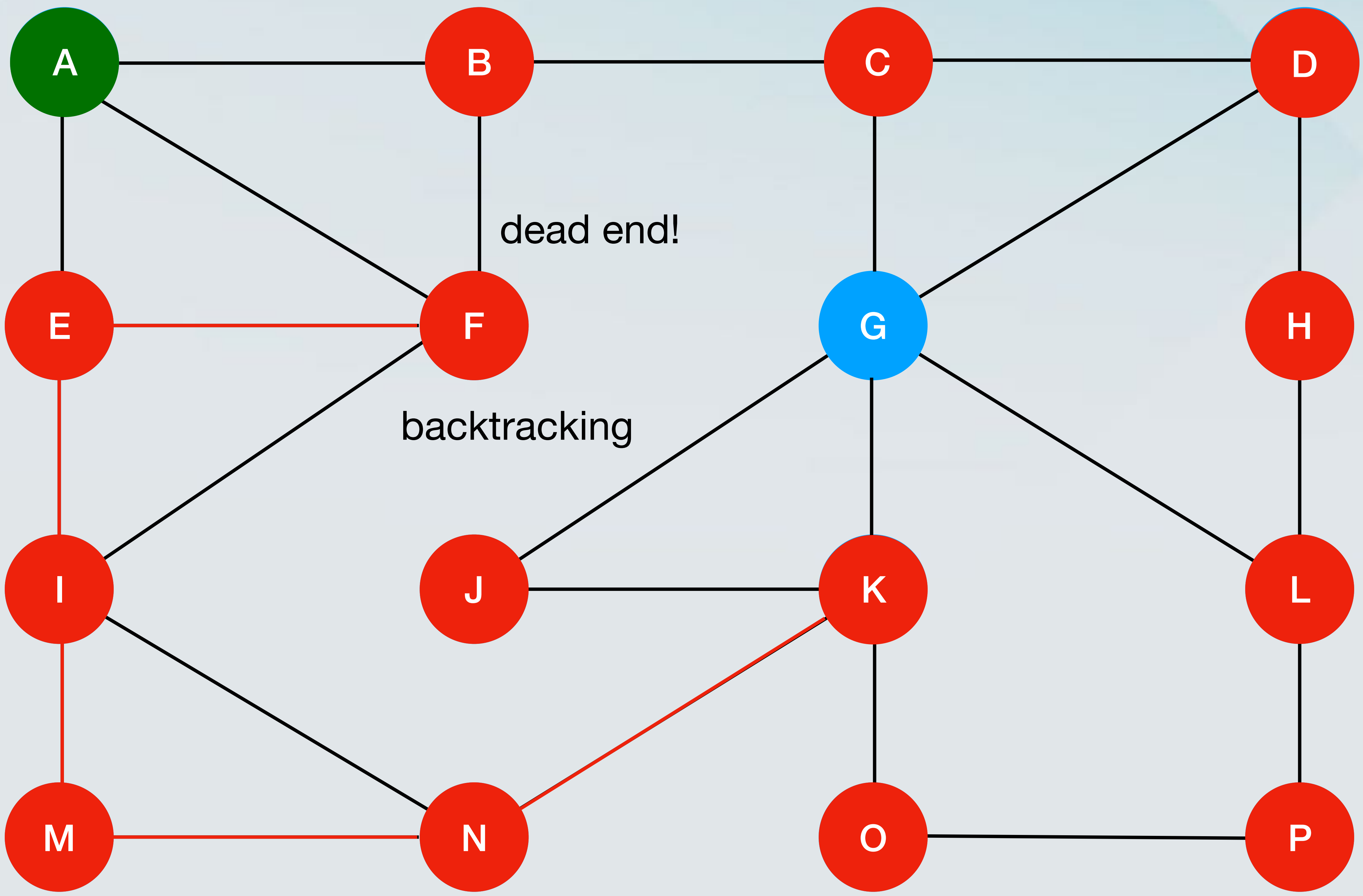
Depth-First Search



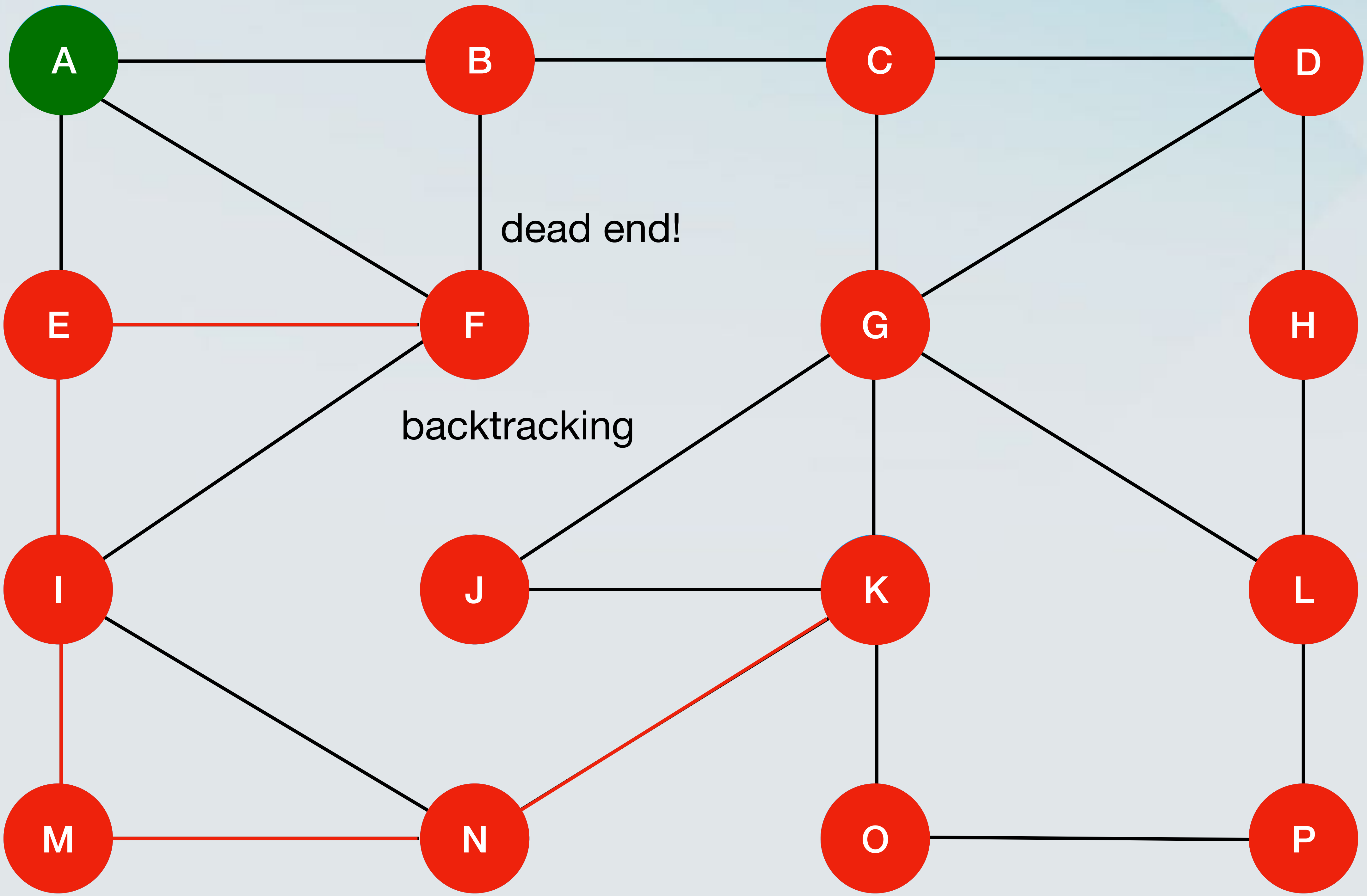
Depth-First Search



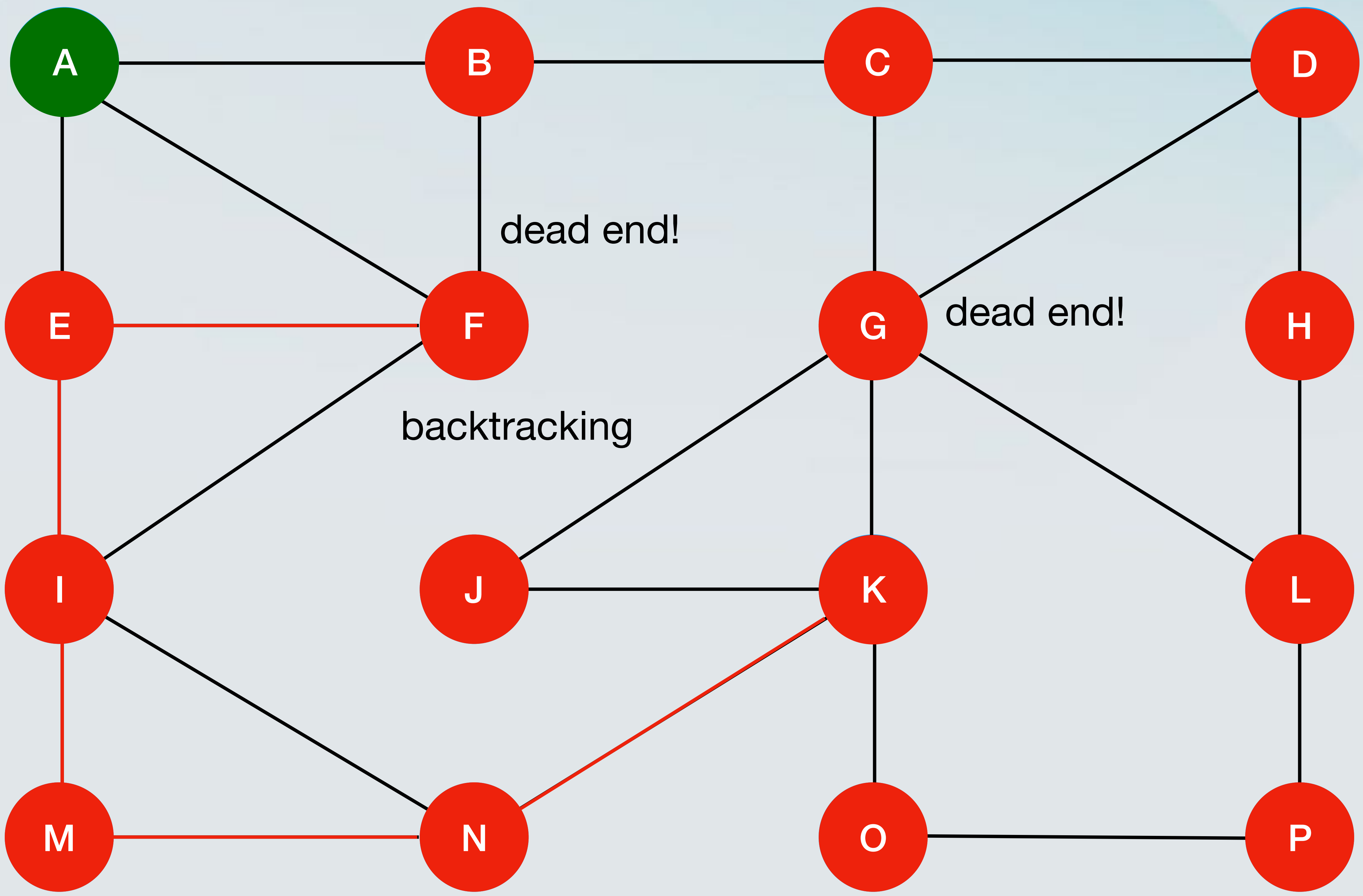
Depth-First Search



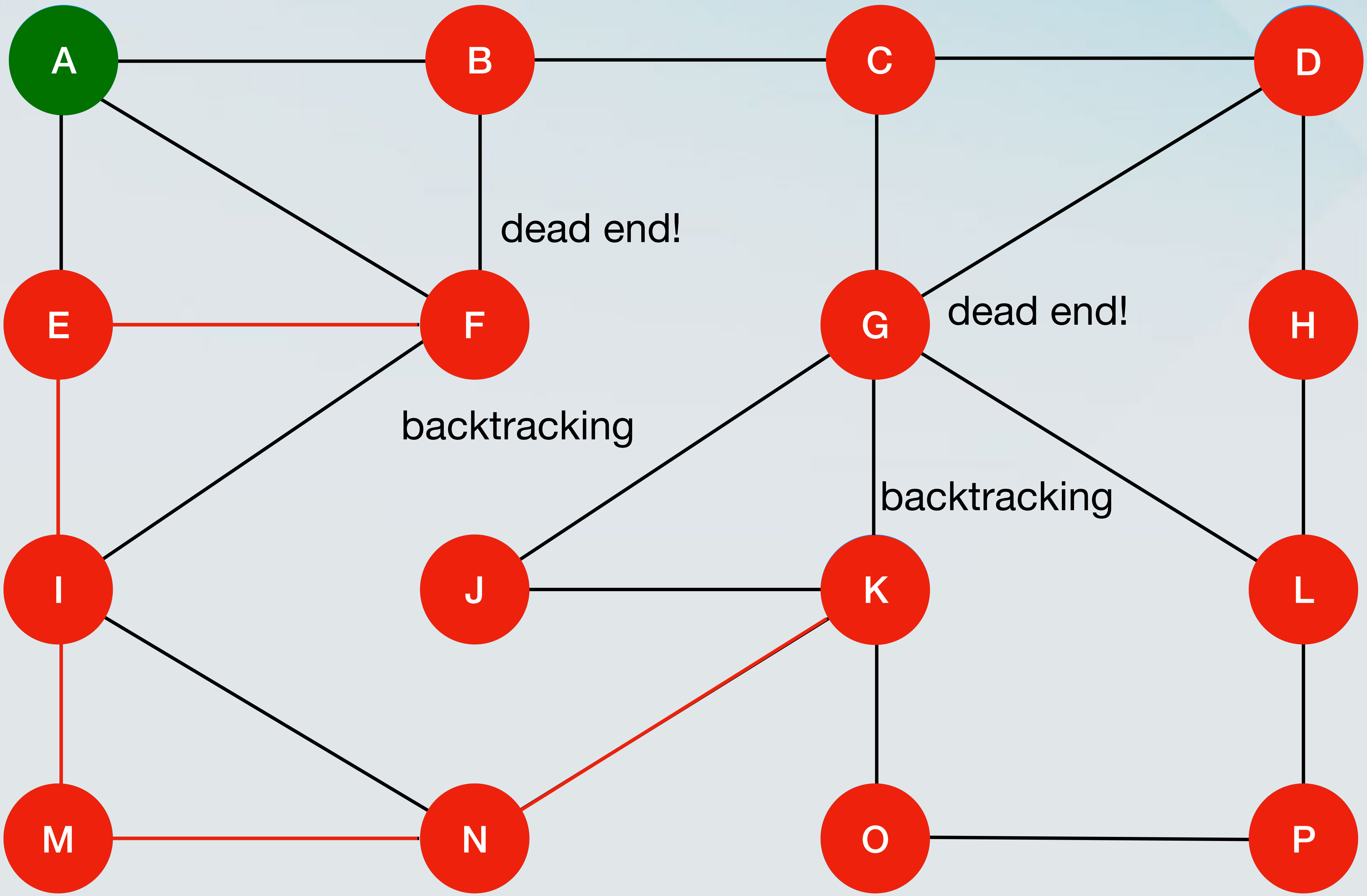
Depth-First Search



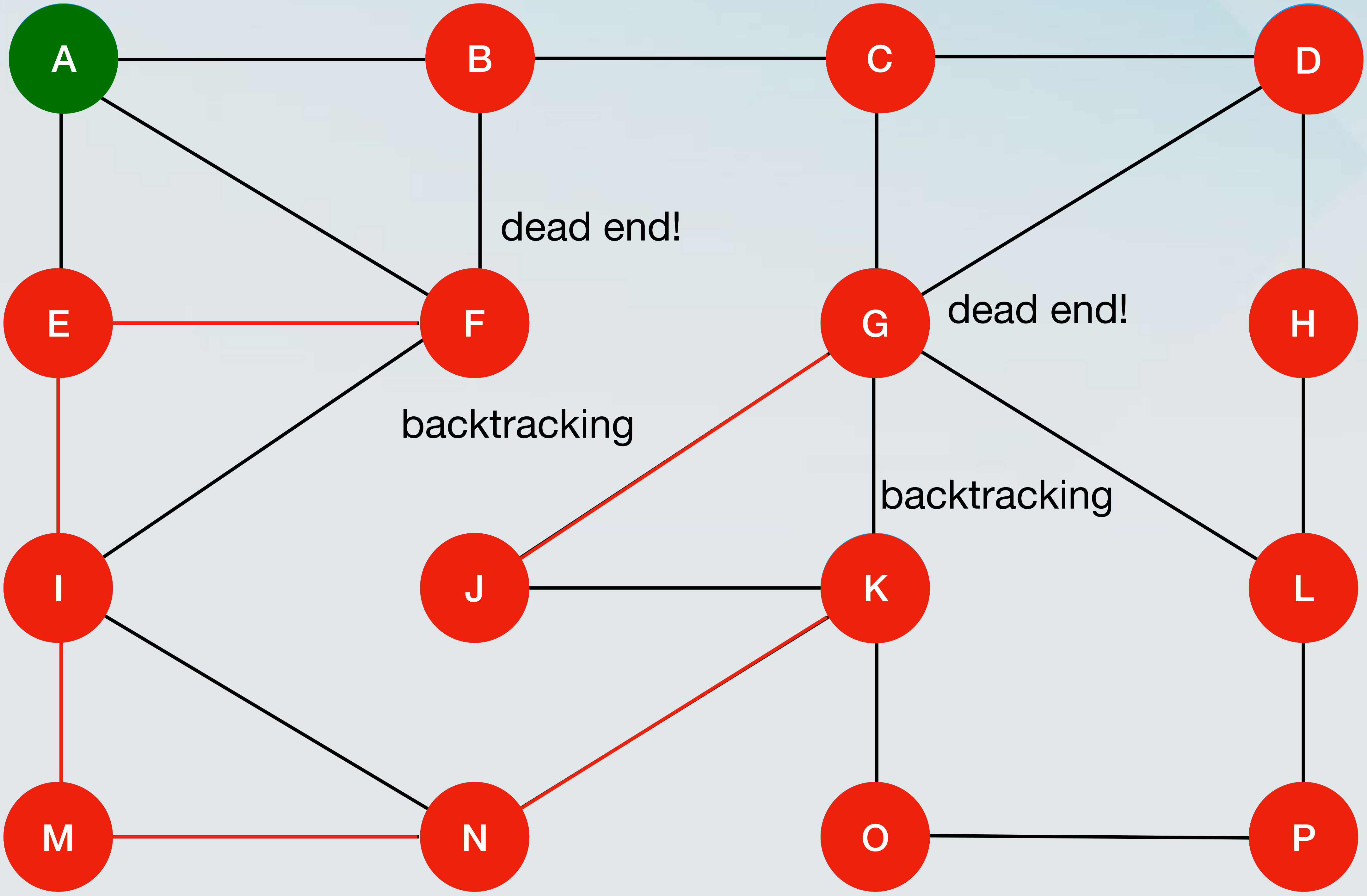
Depth-First Search



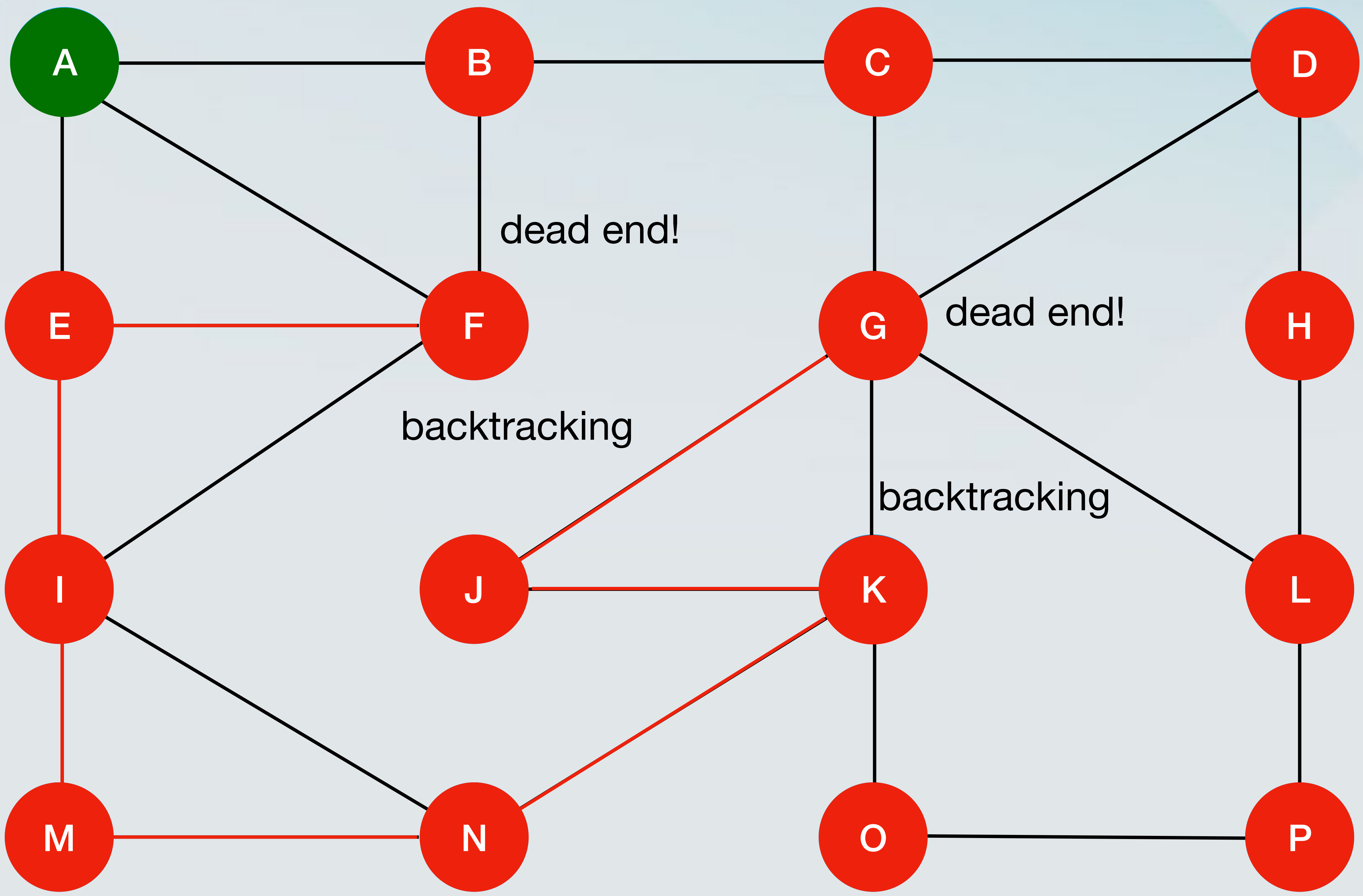
Depth-First Search



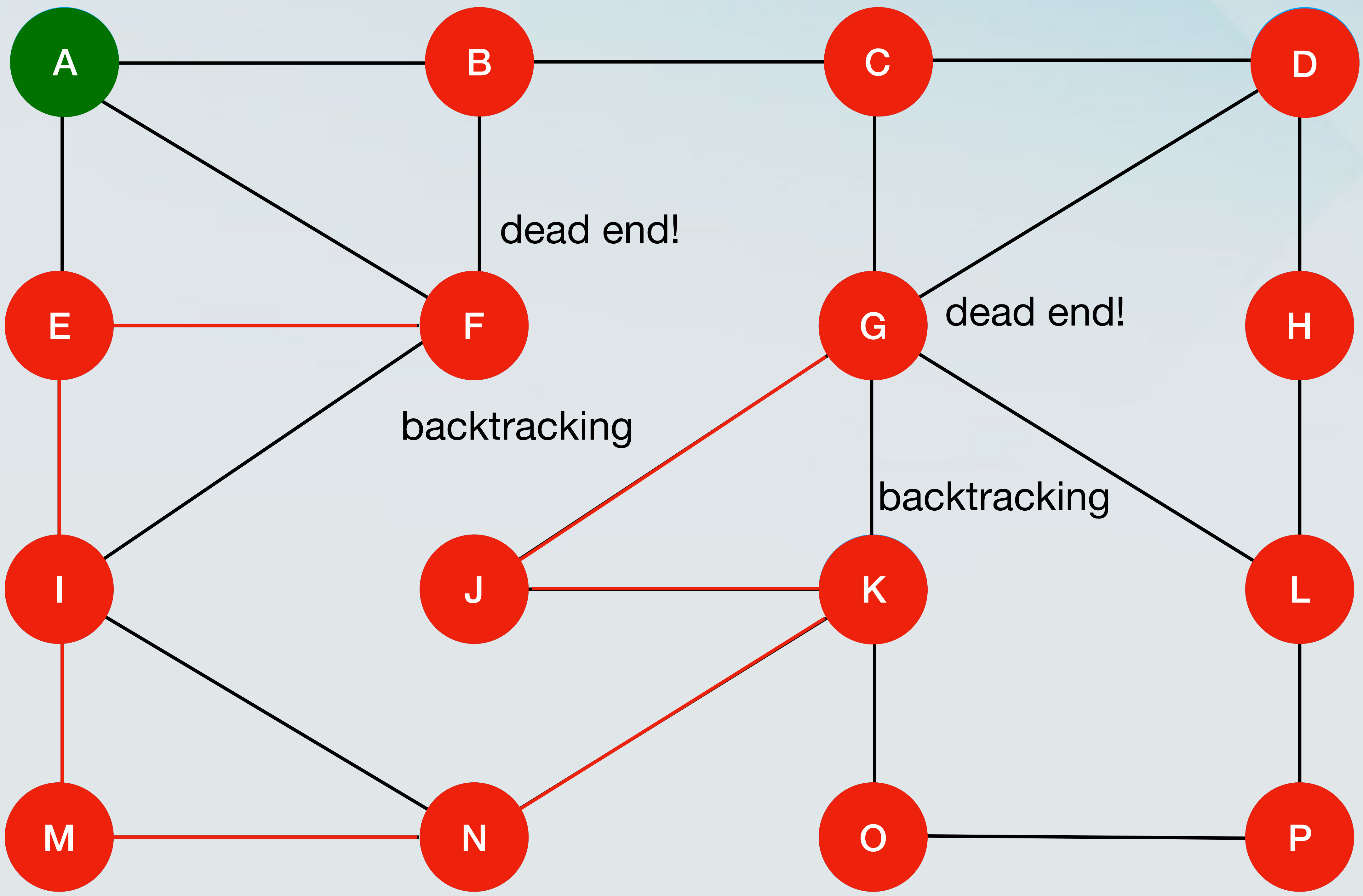
Depth-First Search



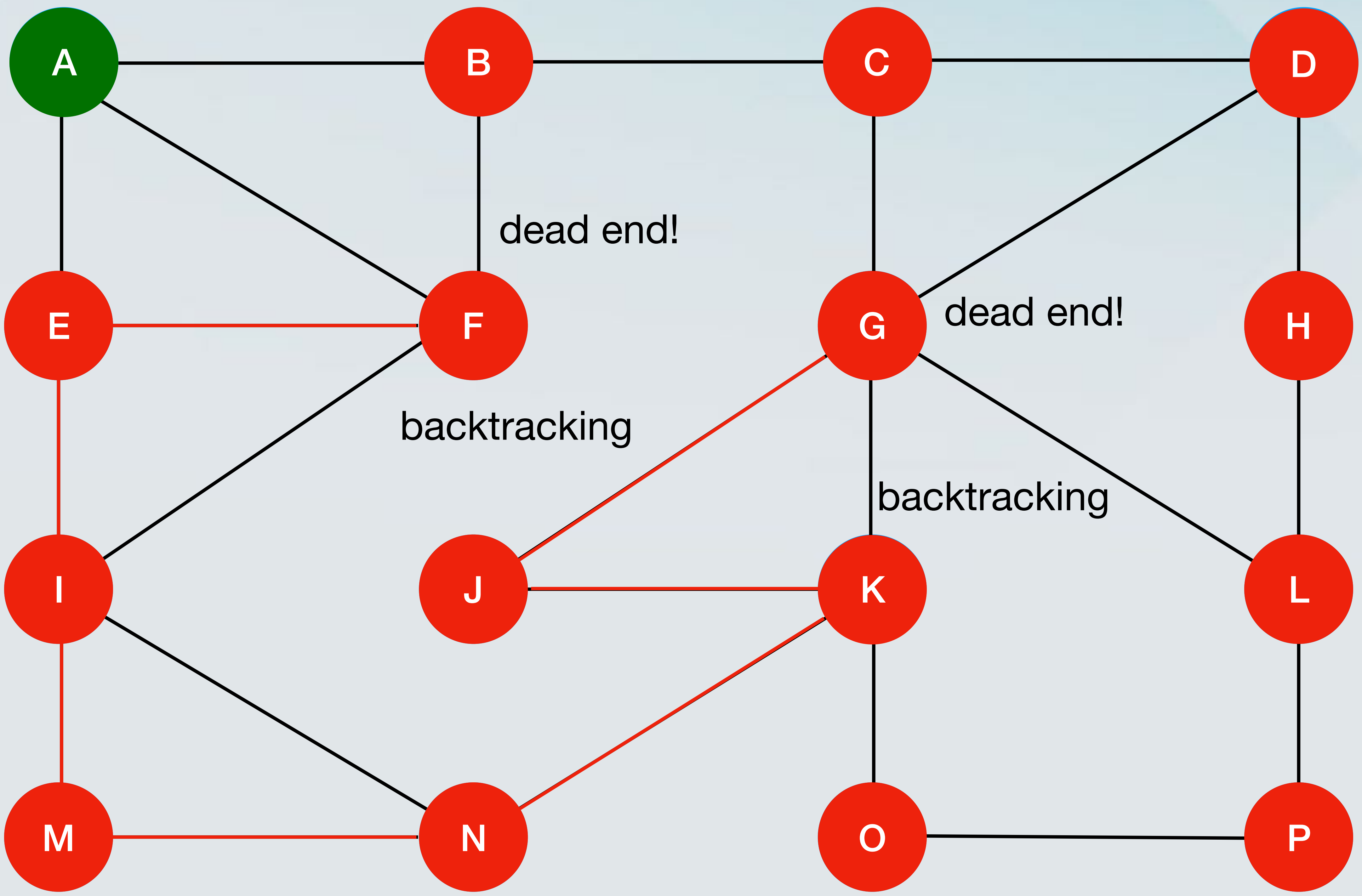
Depth-First Search



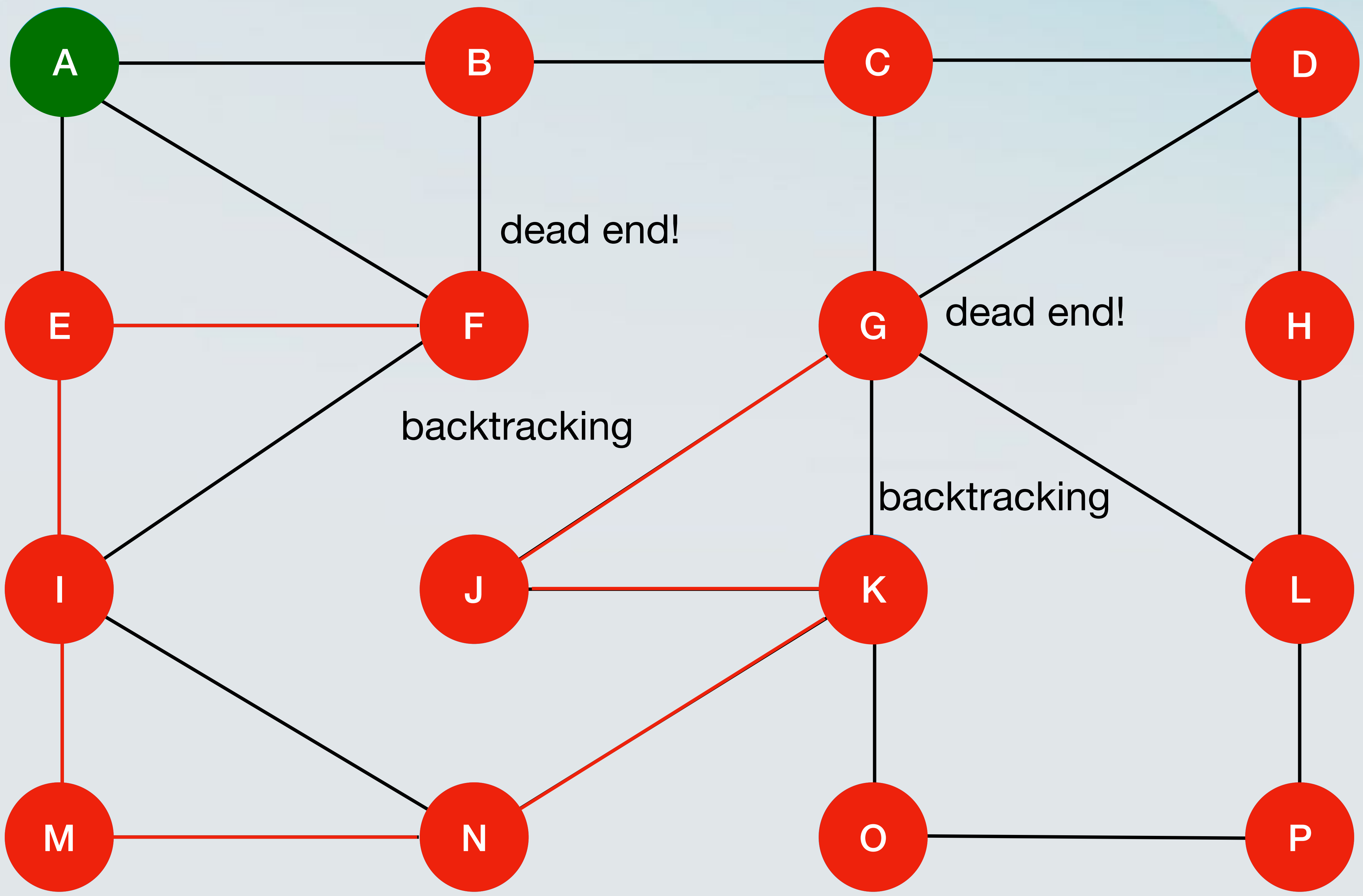
Depth-First Search



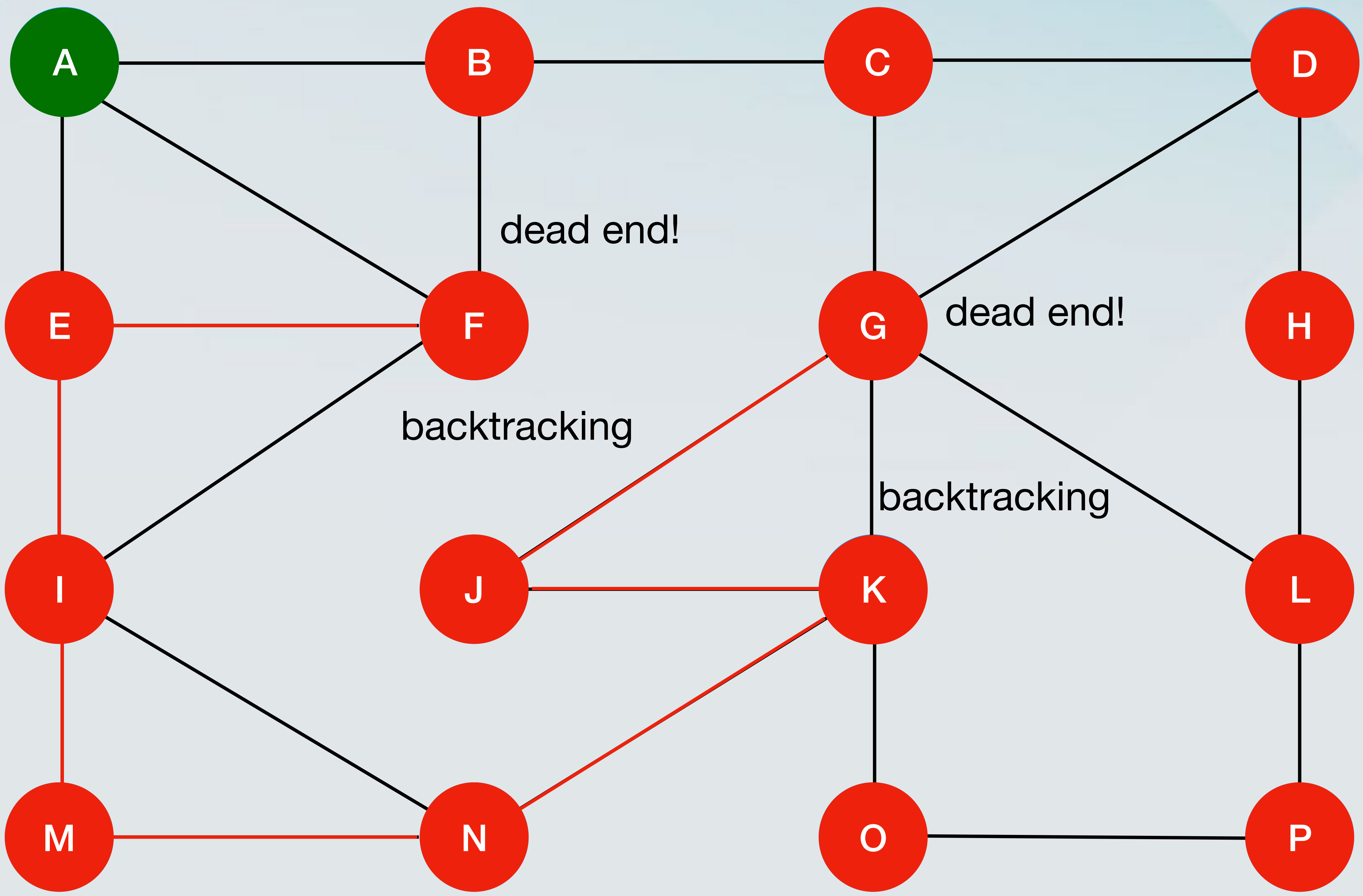
Depth-First Search



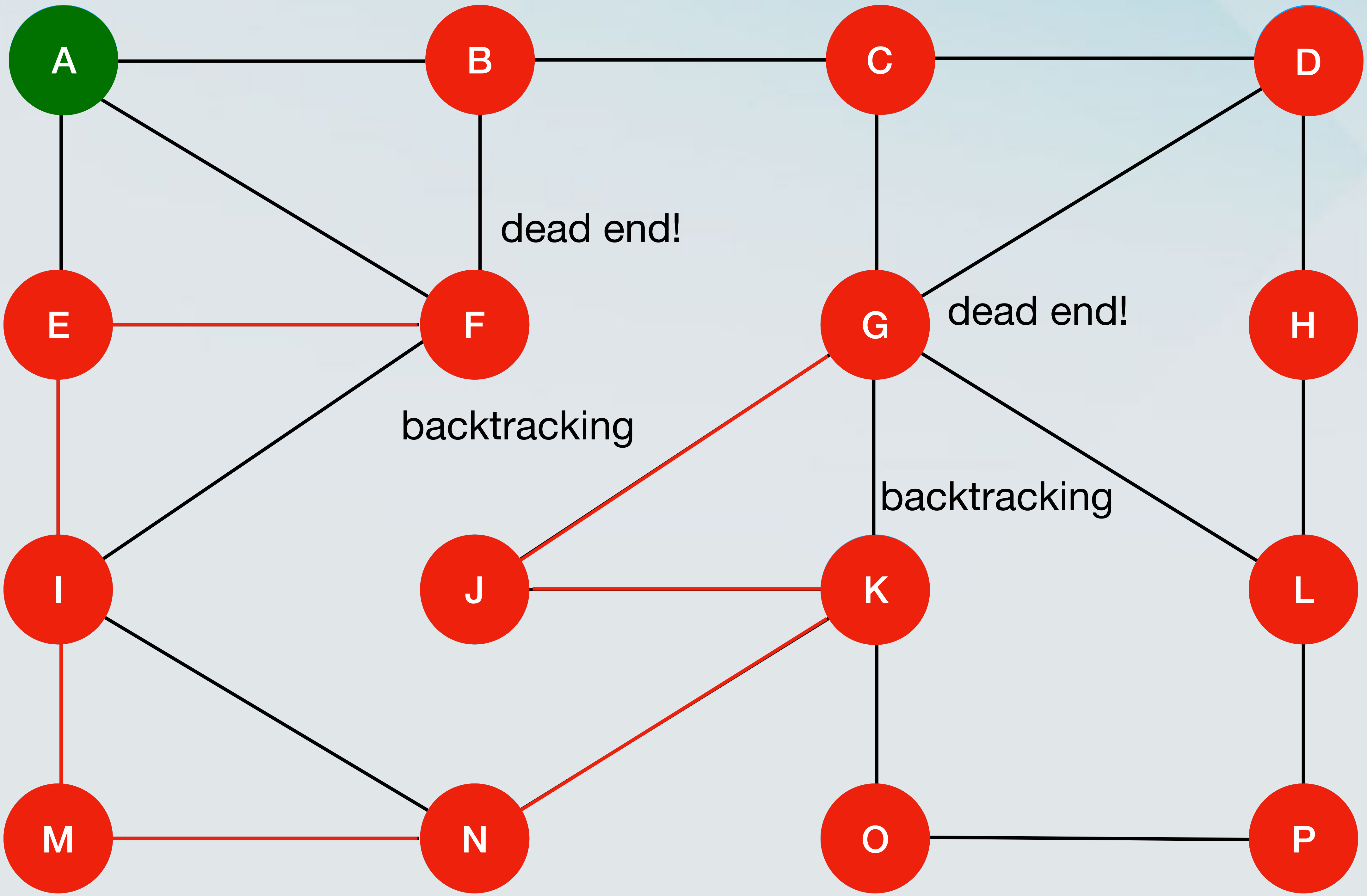
Depth-First Search



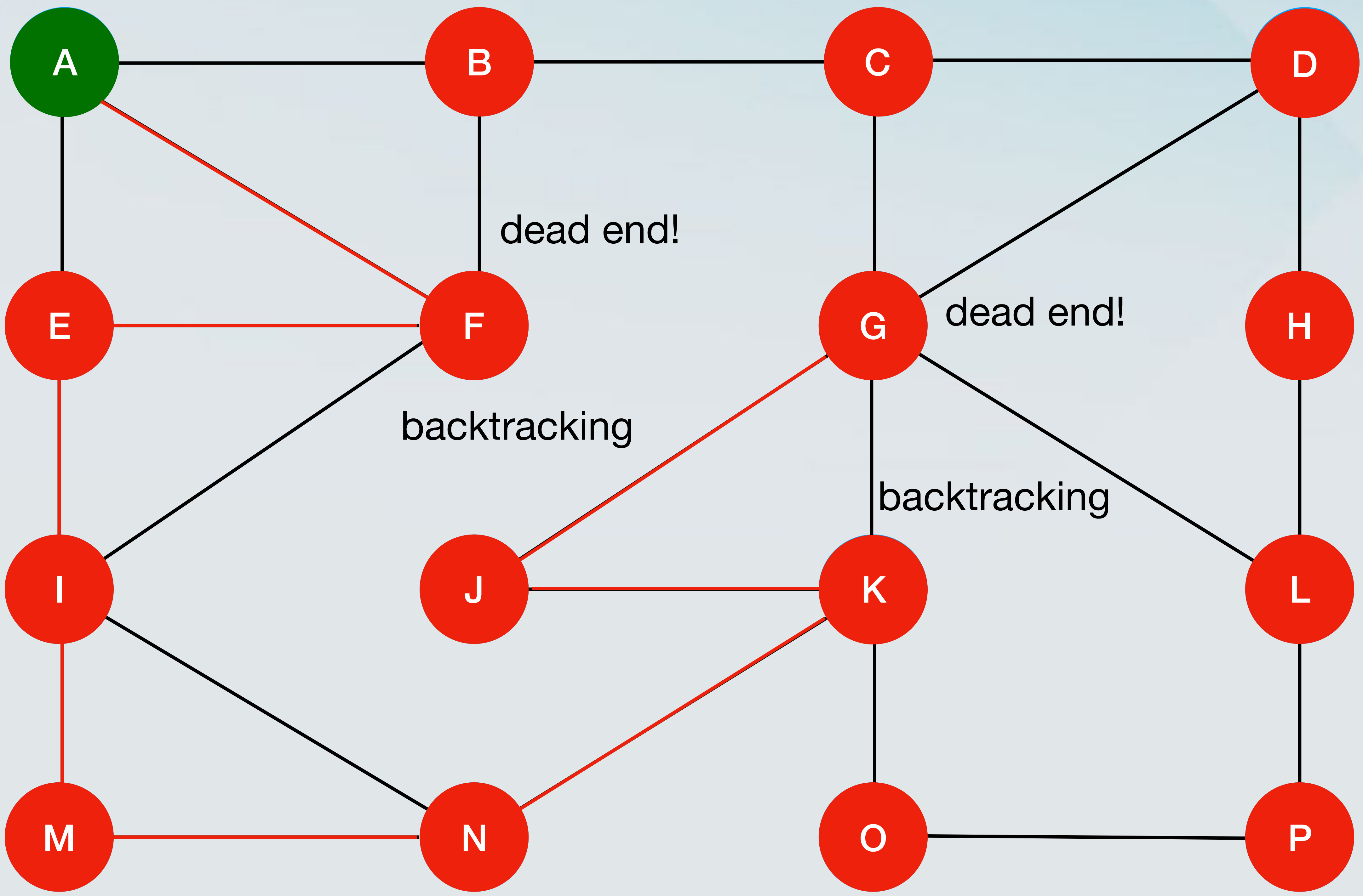
Depth-First Search



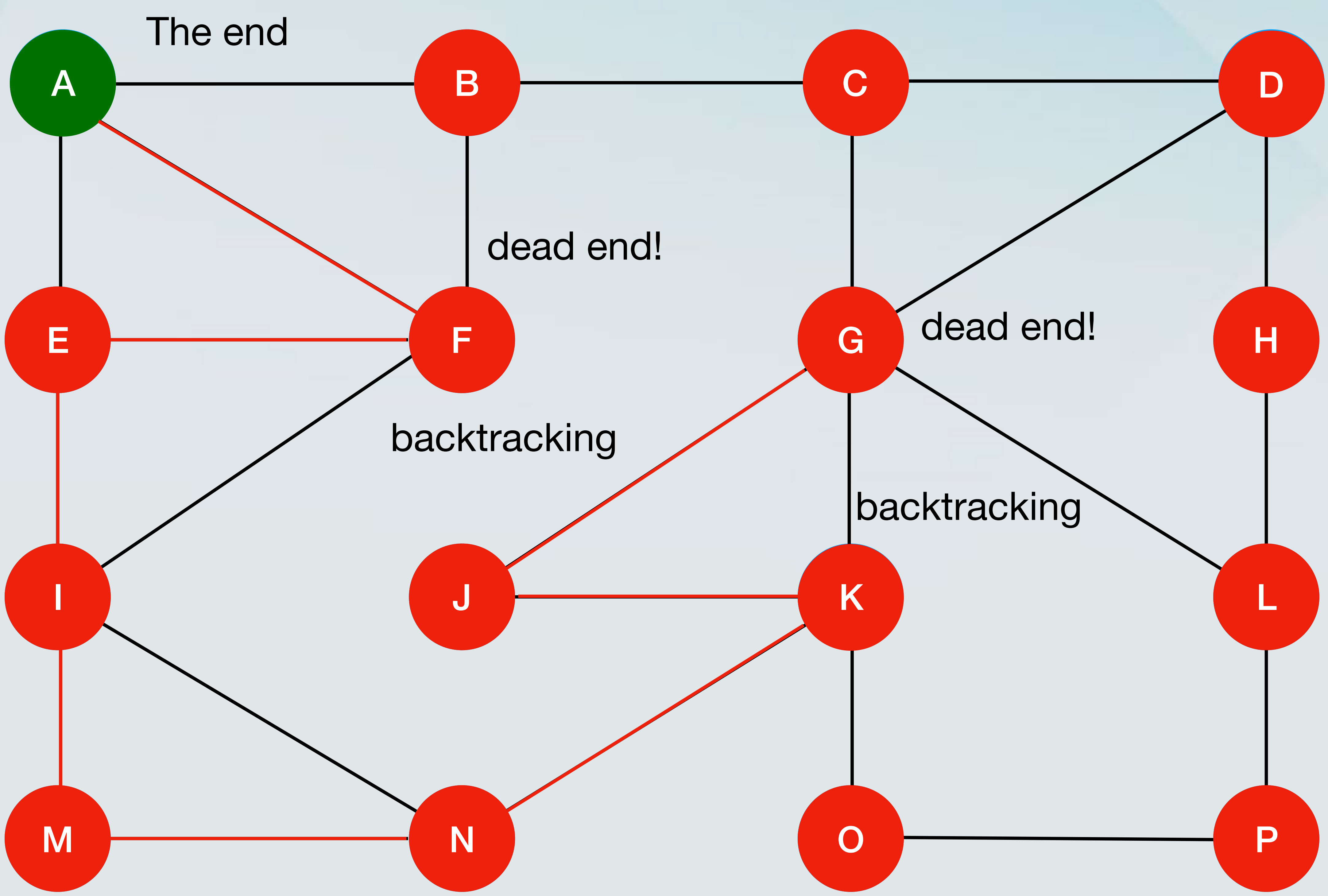
Depth-First Search



Depth-First Search



Depth-First Search



In words

- We wander through a labyrinth with a string and a can of red paint.
- We start at a node **s** and we tie the end of our string to **s**. We paint node **s** as visited.
- We will let **u** denote our *current vertex*. We initialise **u** = **s**
- We travel along an arbitrary edge (**u**,**v**).
    - If the (**u**,**v**) leads to a visited vertex, we return to **u.**
    - Otherwise, we paint **v** as visited, and we set **u** = **v**
    - Then, we return to the beginning of the step.
- Once we get to a dead end (all neighbours have been visited), we *backtrack* to the previously visited vertex v. We set **u** = **v** and repeat the previous steps.
- When we backtrack back to **s**, we terminate the process.

Visualising Depth-First Search

Visualising Depth-First Search

- Orient the edges along the direction in which they are visited during the traversal.

Visualising Depth-First Search

- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.

Visualising Depth-First Search

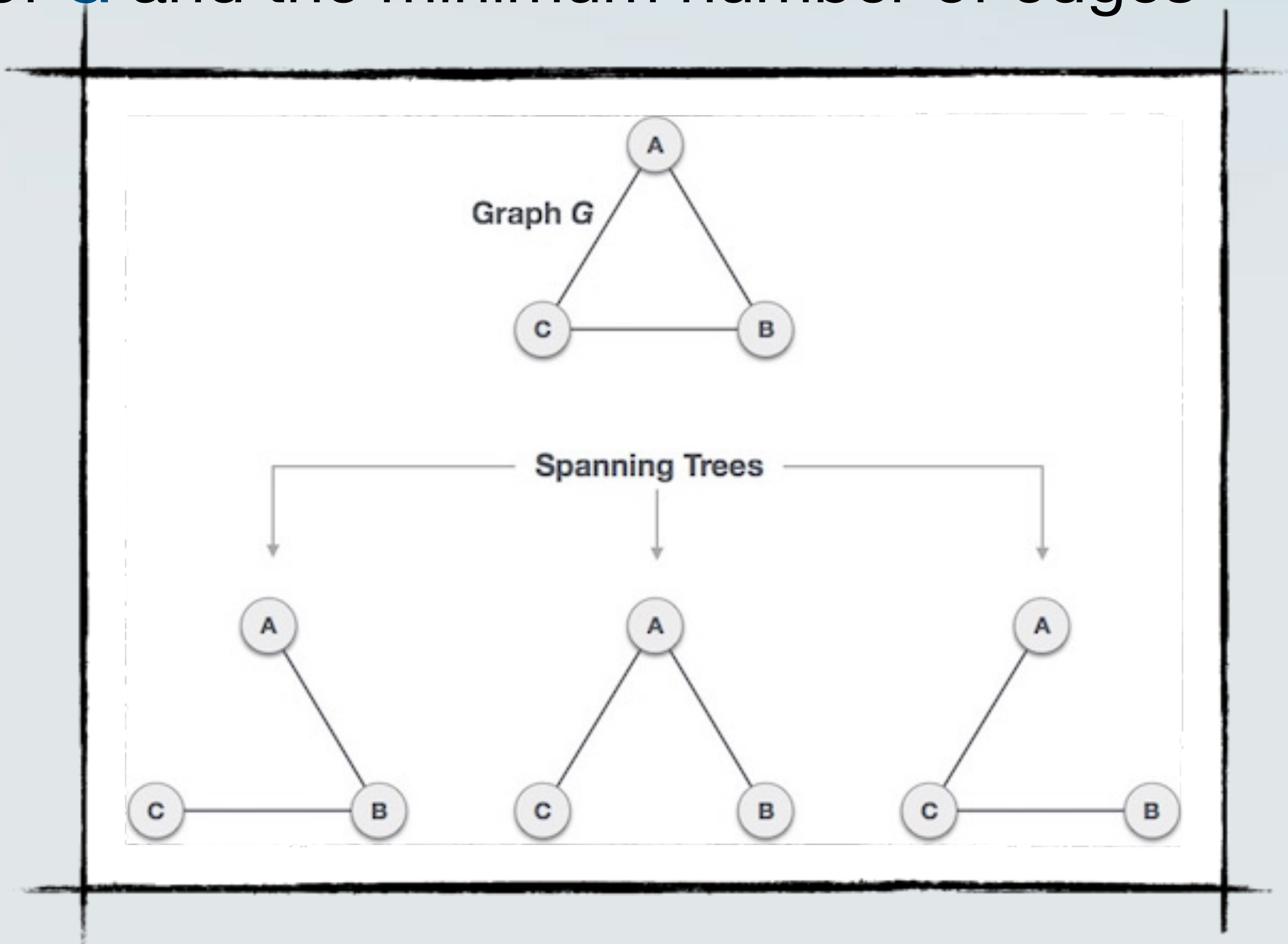
- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.
  - Some edges are *back edges*, because they lead to visited vertices.

Visualising Depth-First Search

- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.
  - Some edges are *back edges*, because they lead to visited vertices.
- The discovery edges form a **spanning tree** of the **connected component** of the starting vertex **s**.

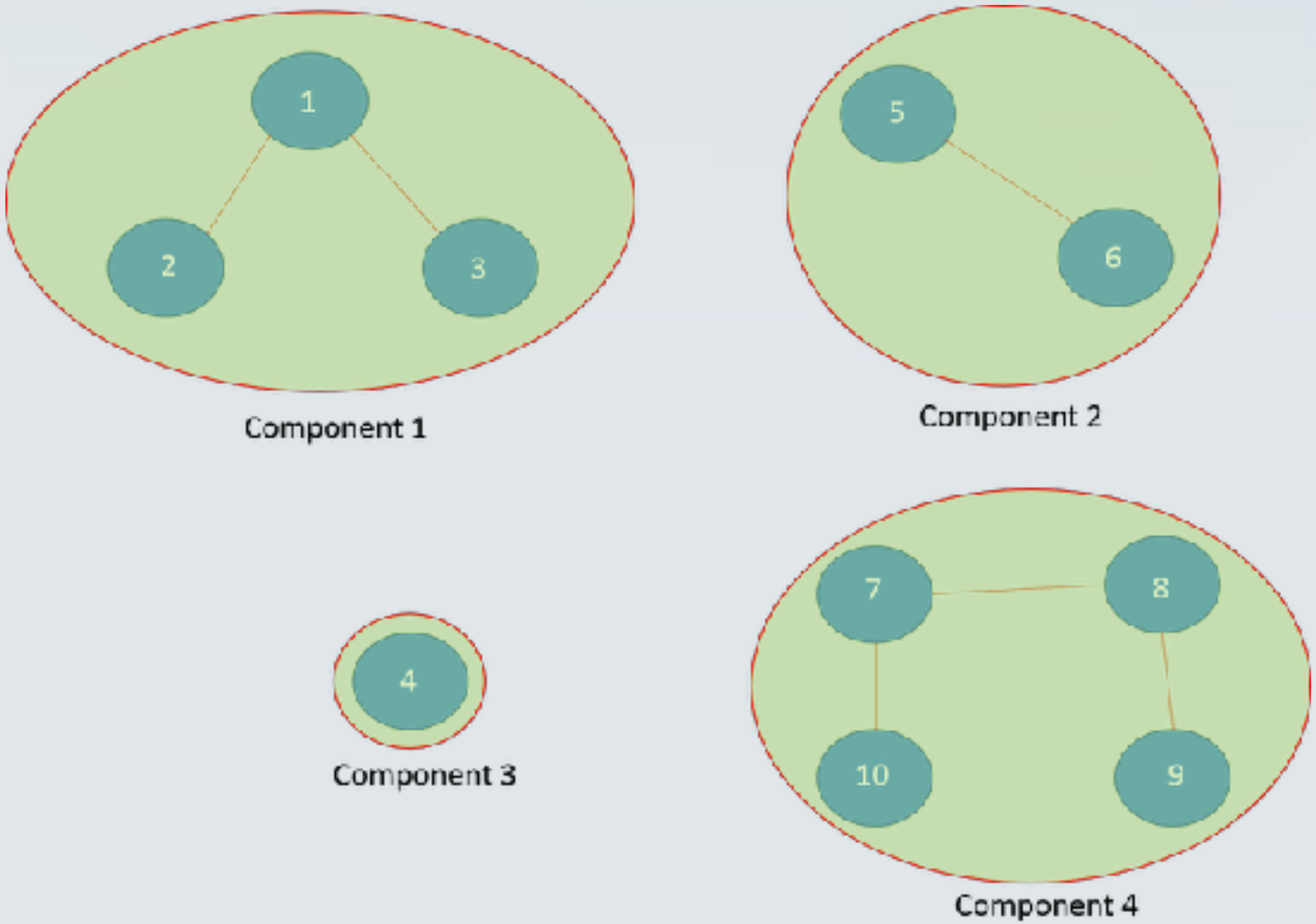
Definitions

- A **spanning tree** of a graph **G** is a tree containing all the nodes of **G** and the minimum number of edges



Definitions

- A **connected component** of a graph **G** is subgraph such that any two vertices are connected via some path.

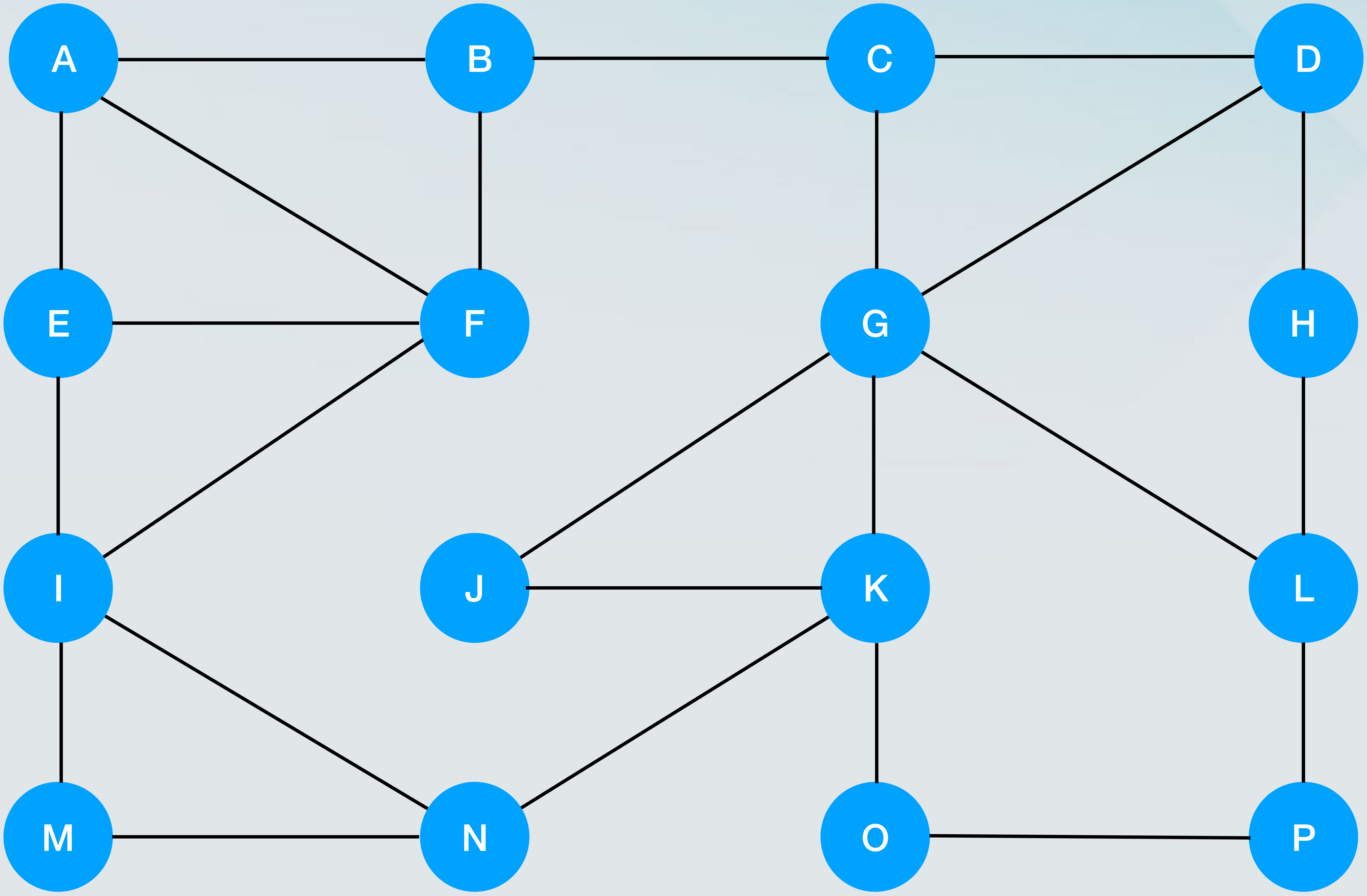


Depth-First Search Pseudocode

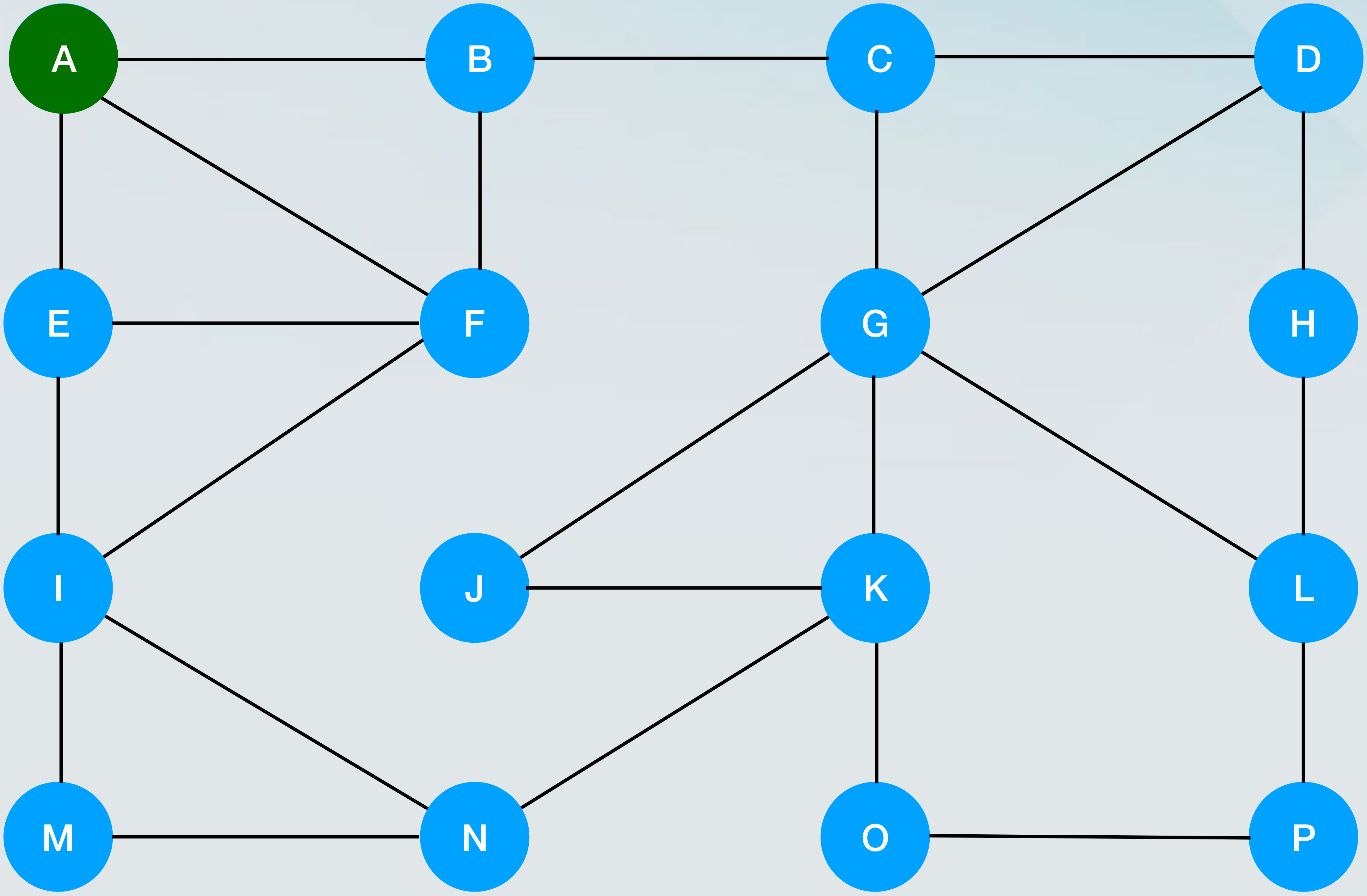
```
Algorithm DFS(G,v)

  for all edges e incident to v.   /* all edges that have v as one of their endpoints */
      if edge e is unexplored
          Let u be the other endpoint of e
          If vertex u is unexplored
              Label e as a discovery edge
              DFS(G,u)
          Else
              Label e as a back edge
```

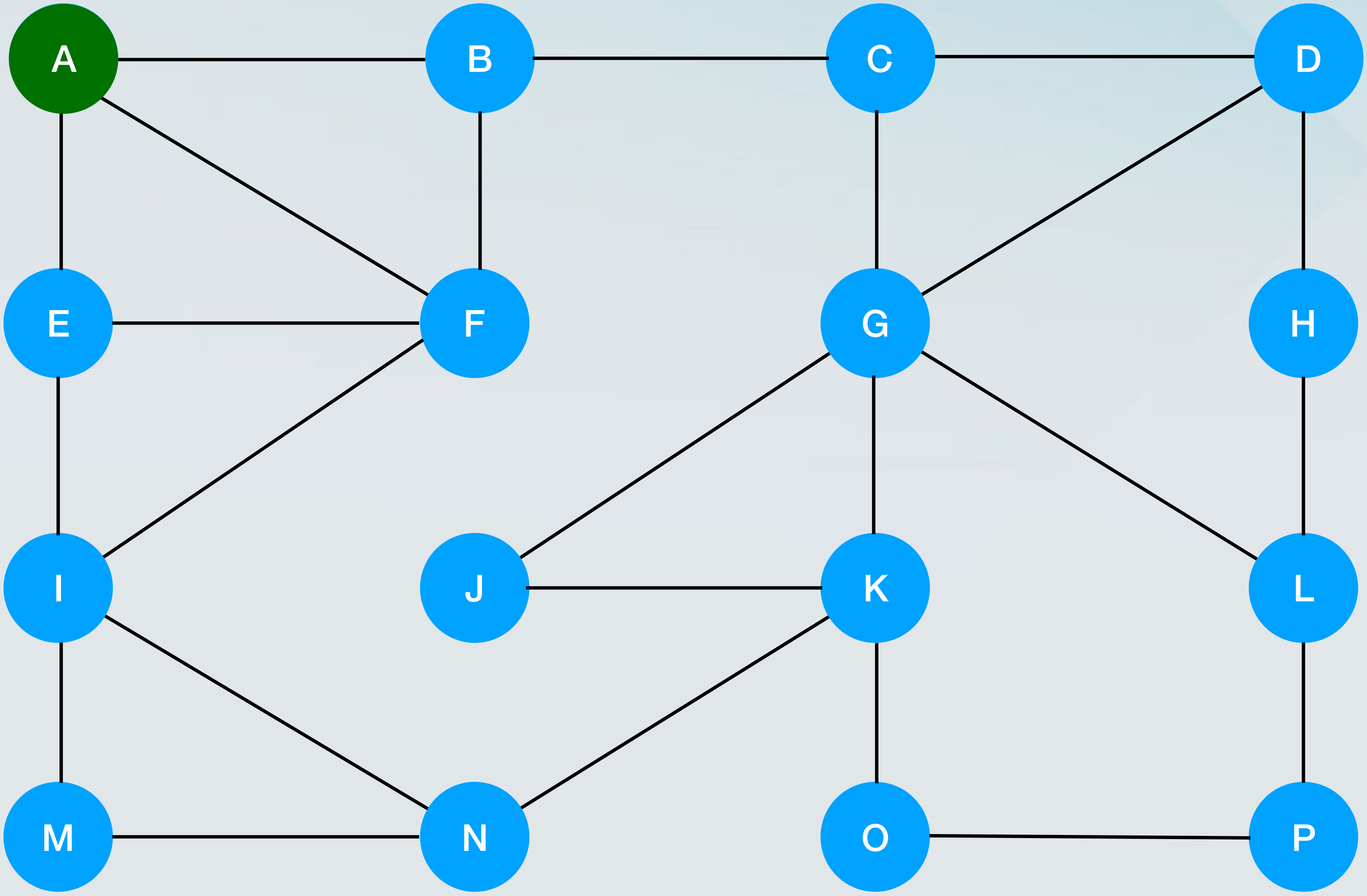
Depth-First Search



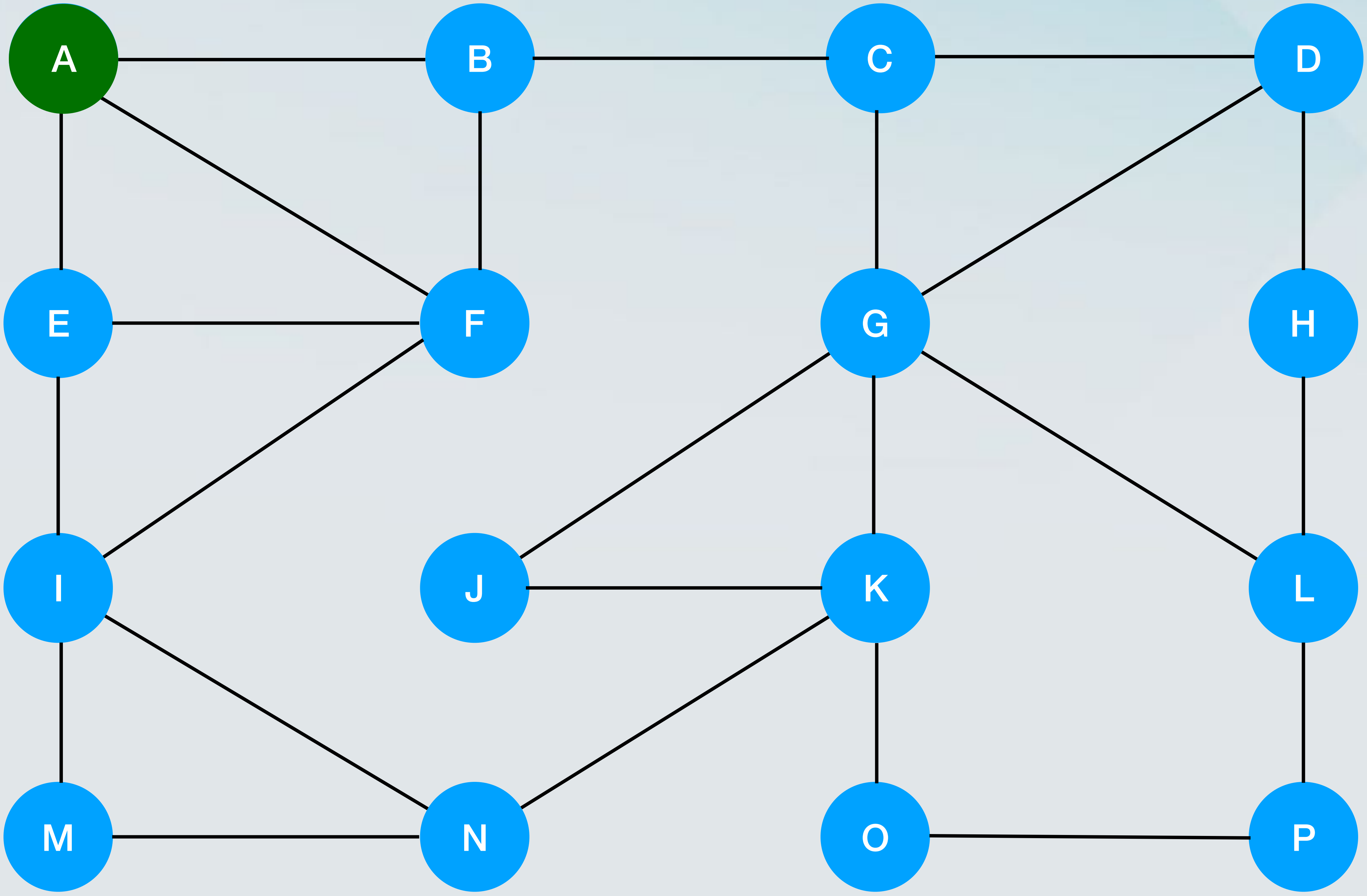
Depth-First Search



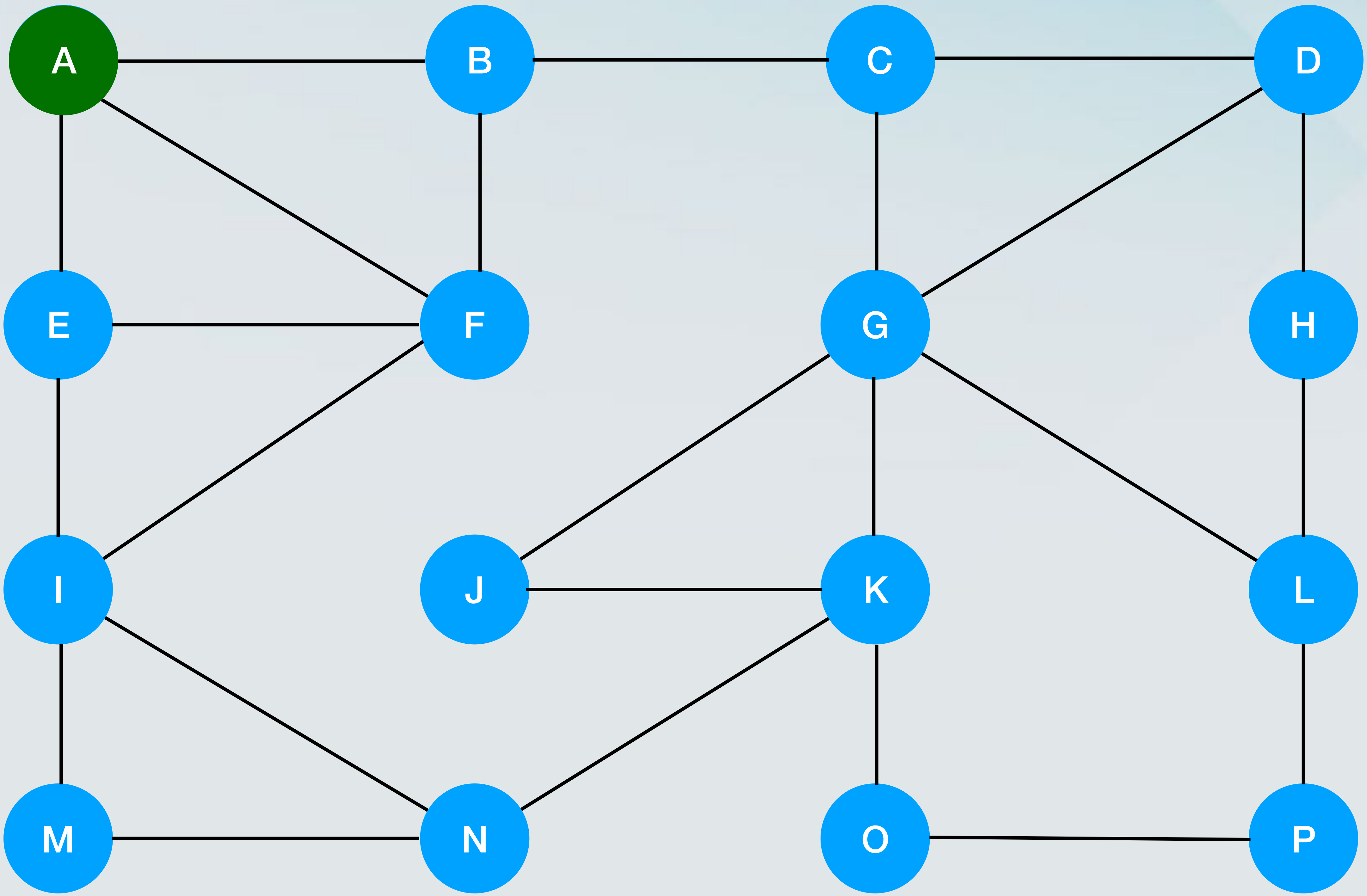
Depth-First Search



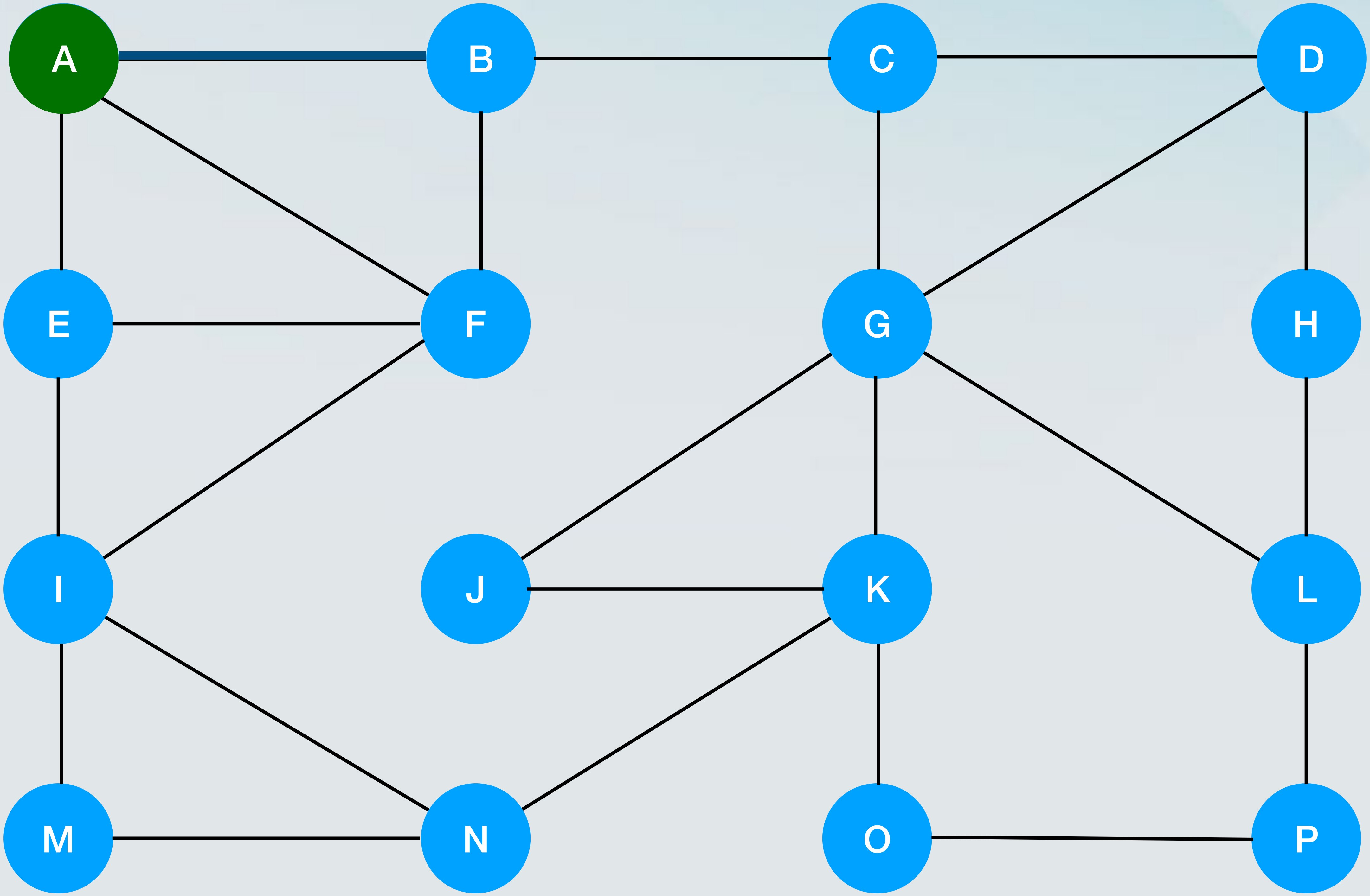
Depth-First Search



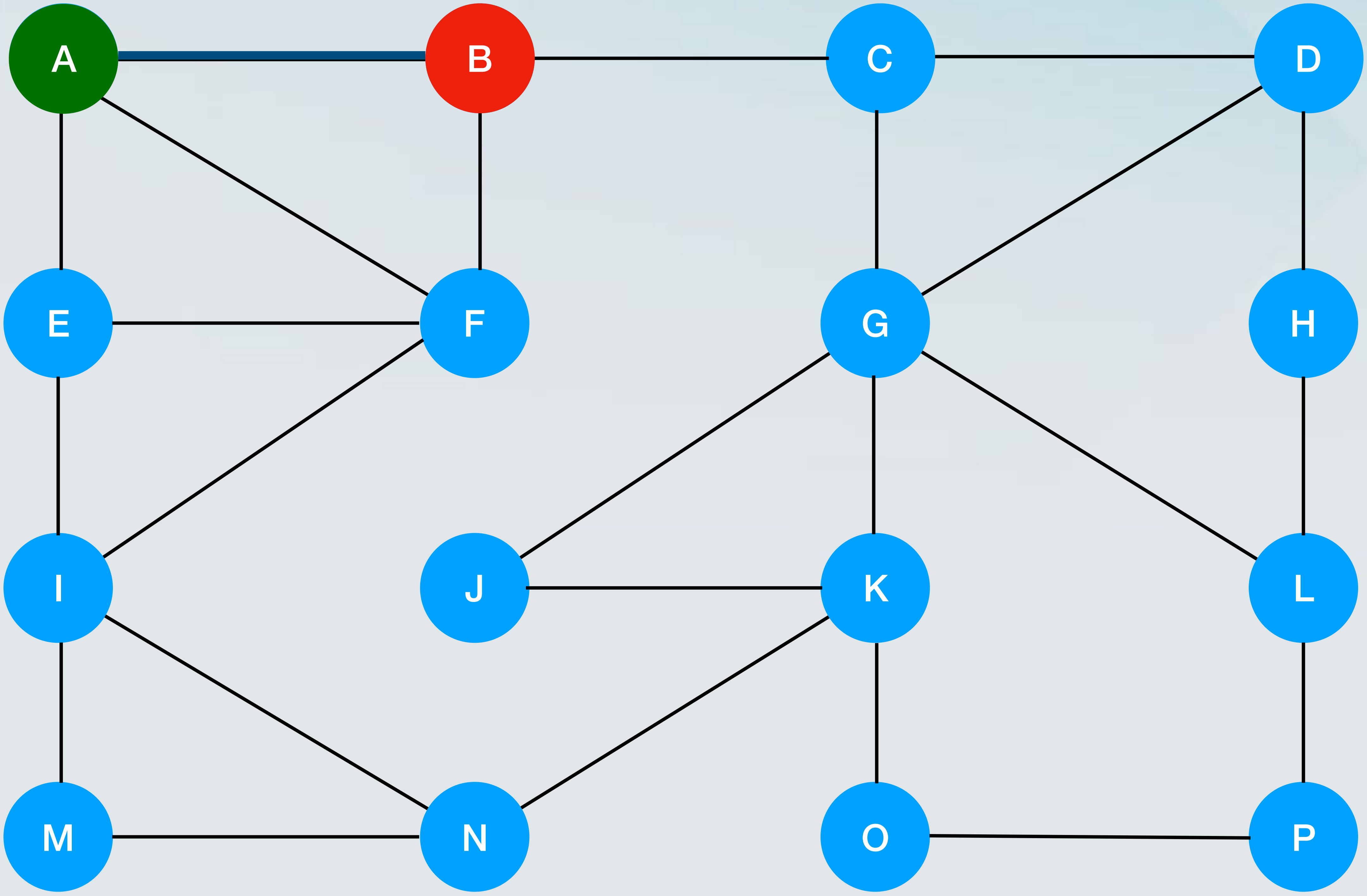
Depth-First Search



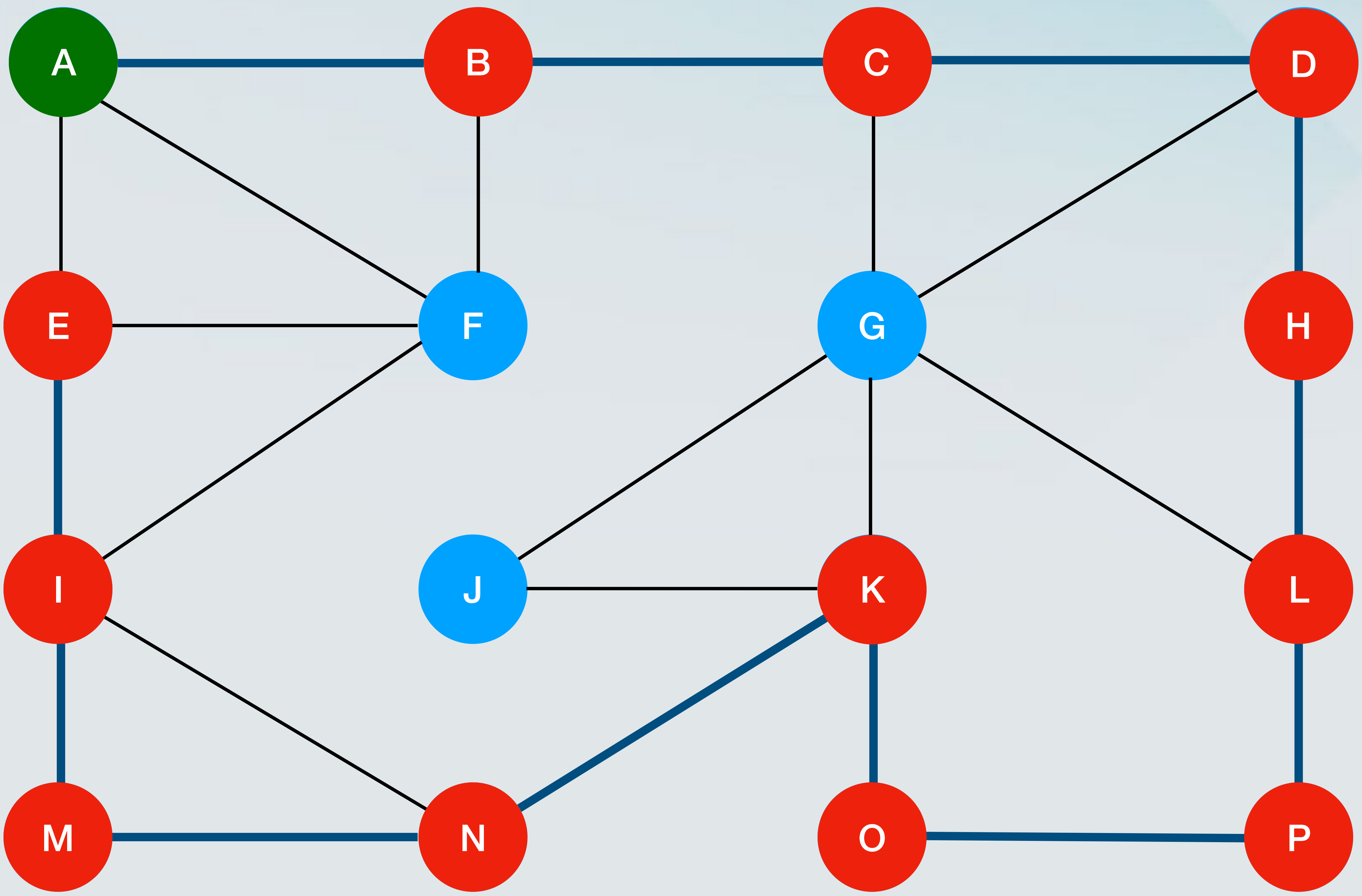
Depth-First Search



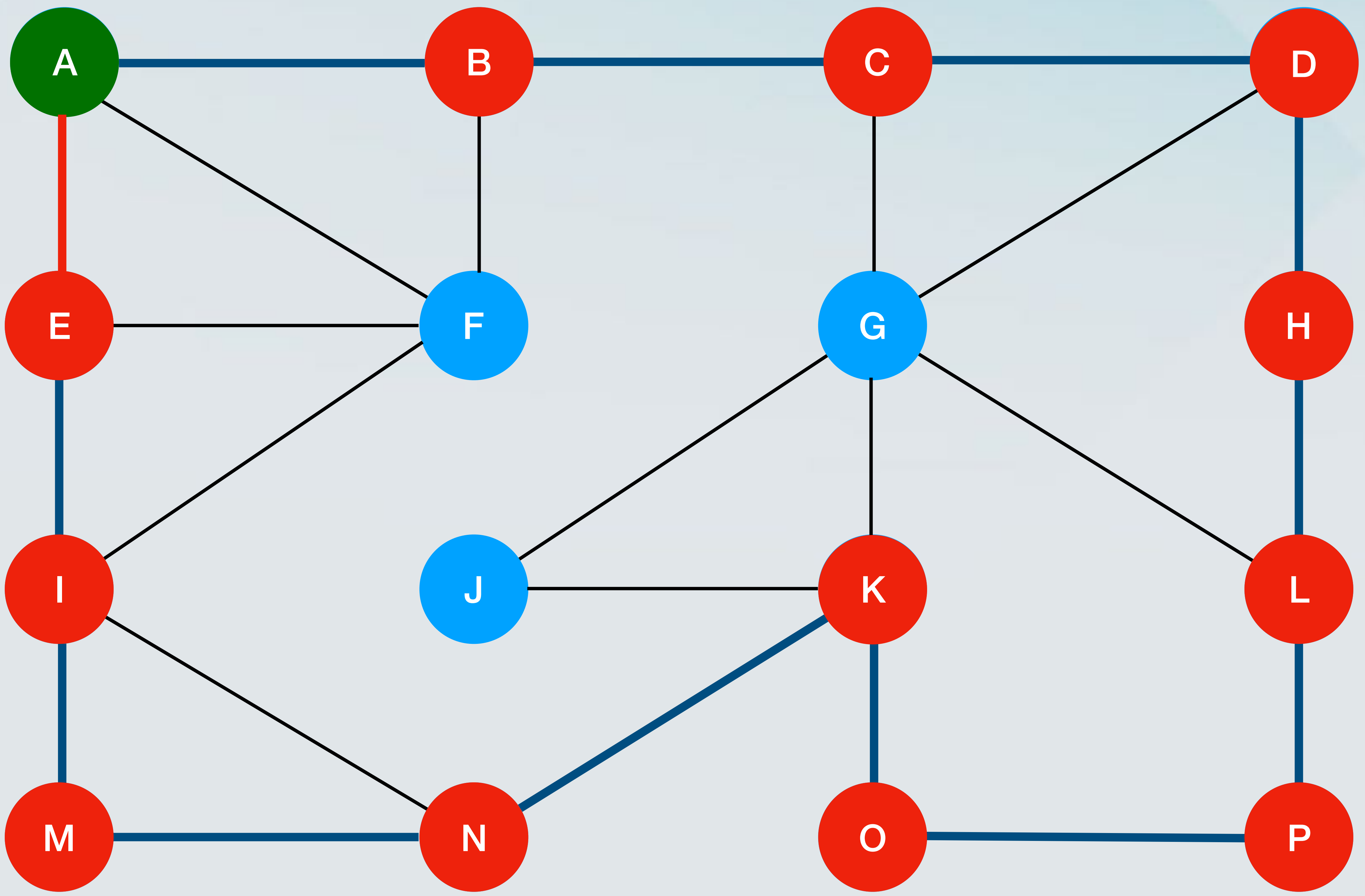
Depth-First Search



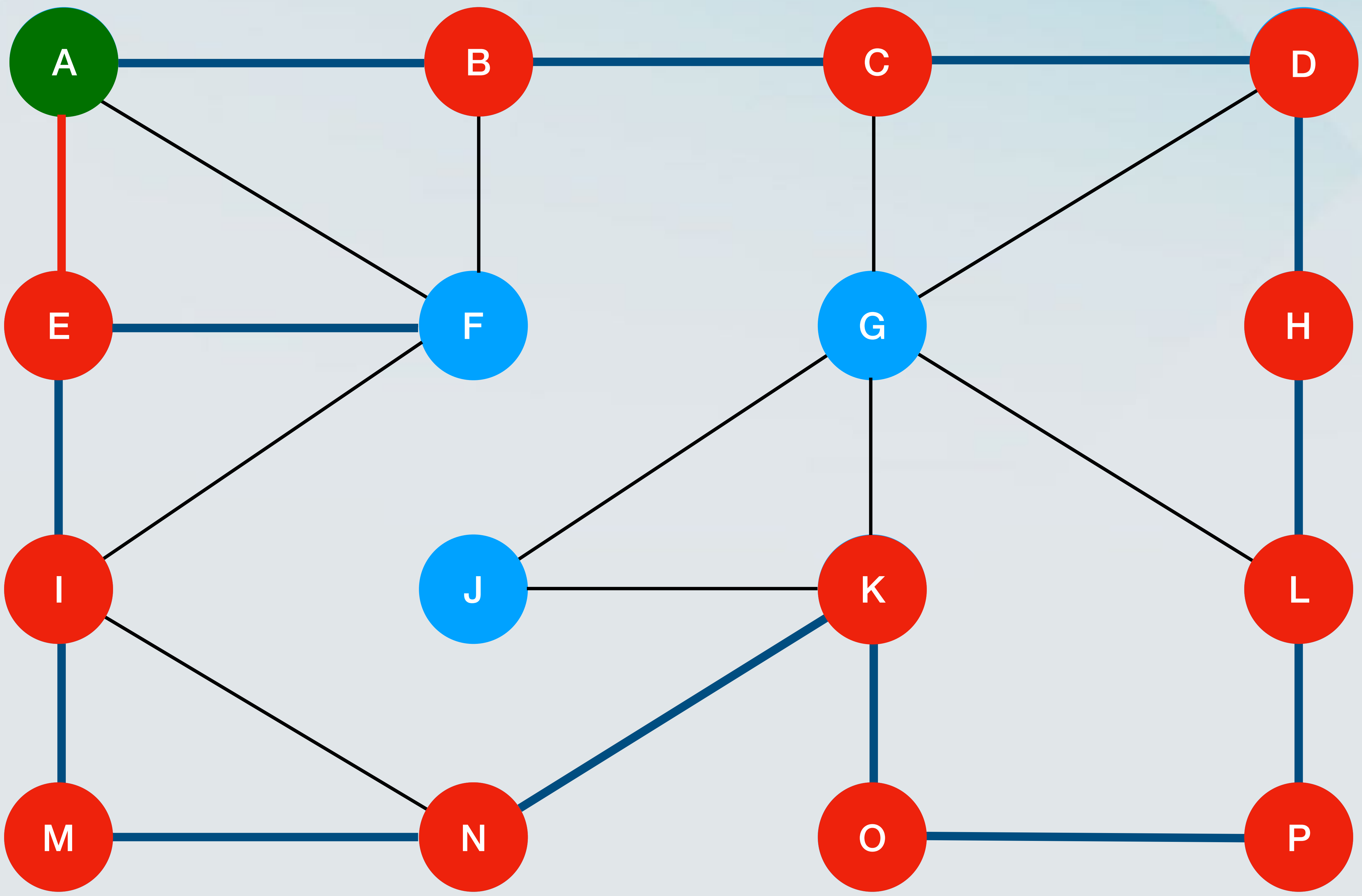
Depth-First Search



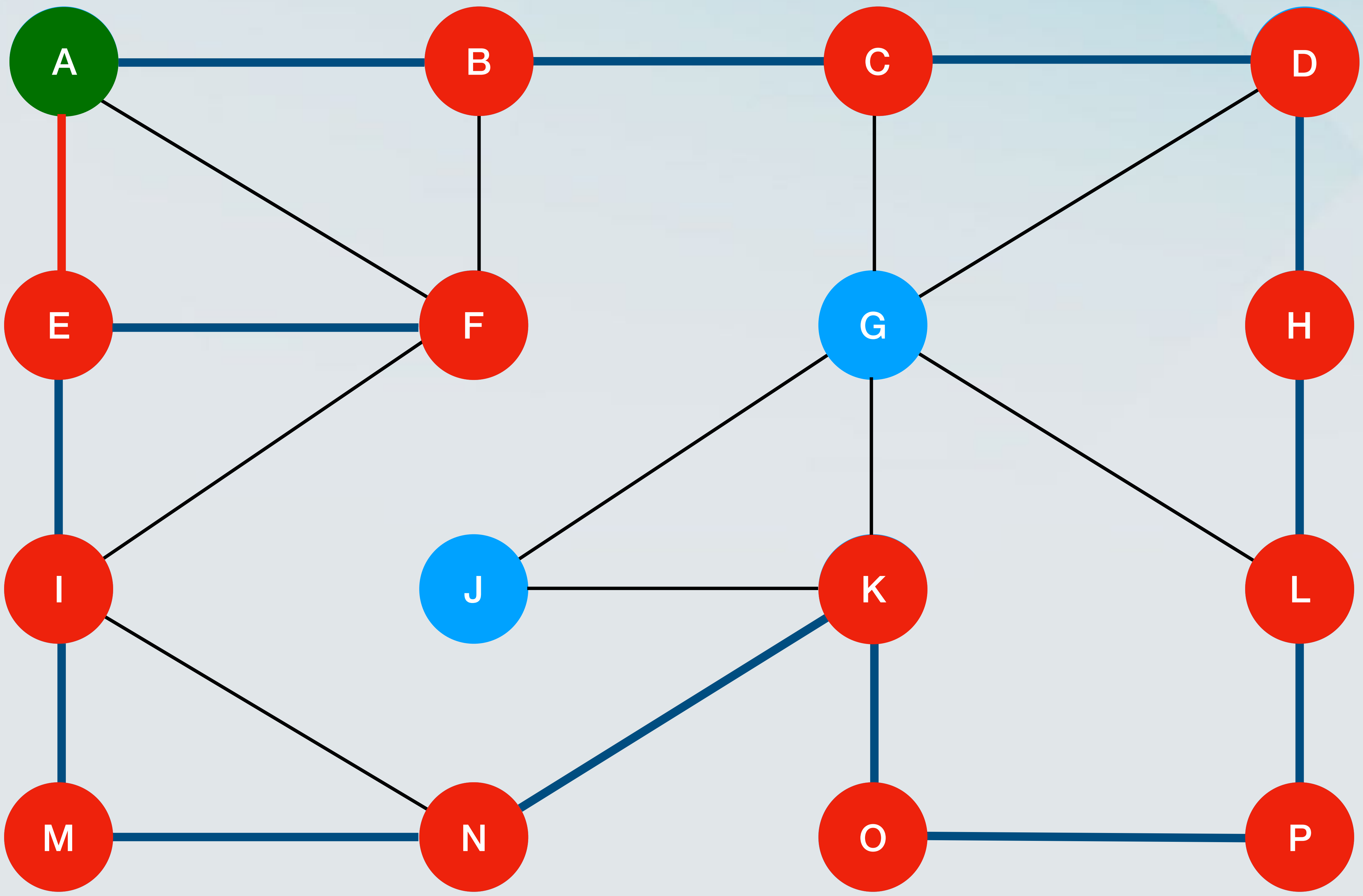
Depth-First Search



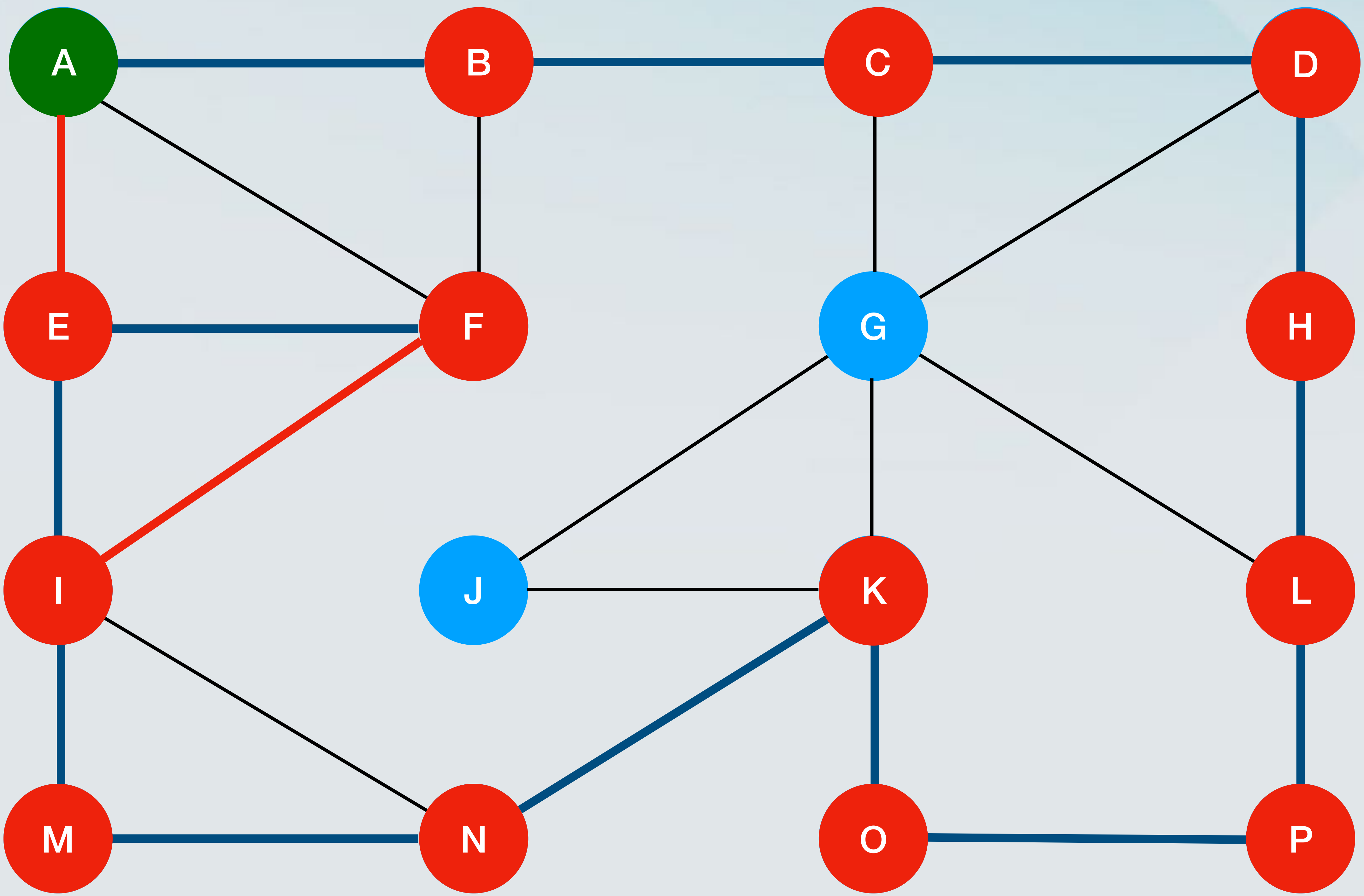
Depth-First Search



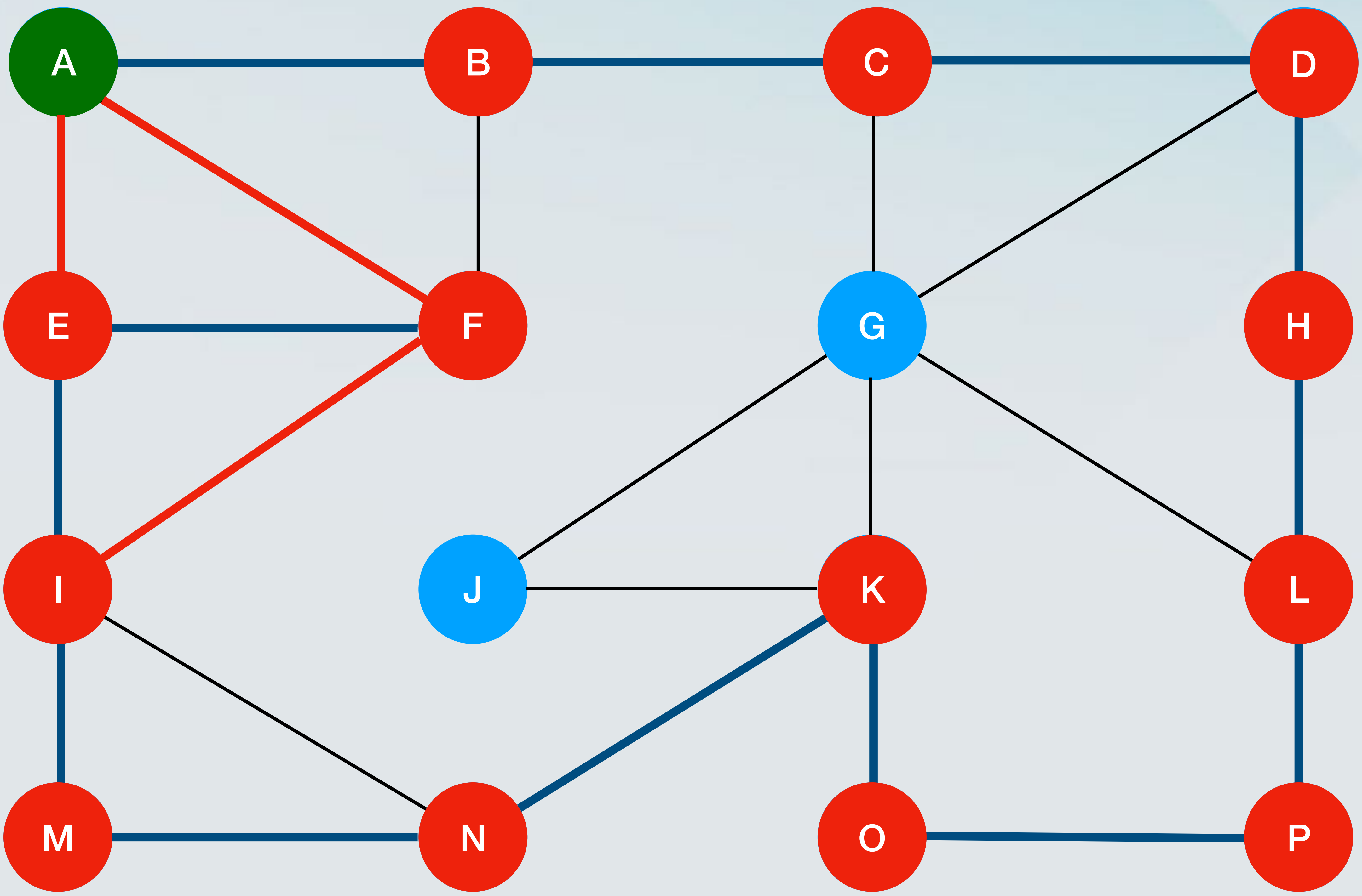
Depth-First Search



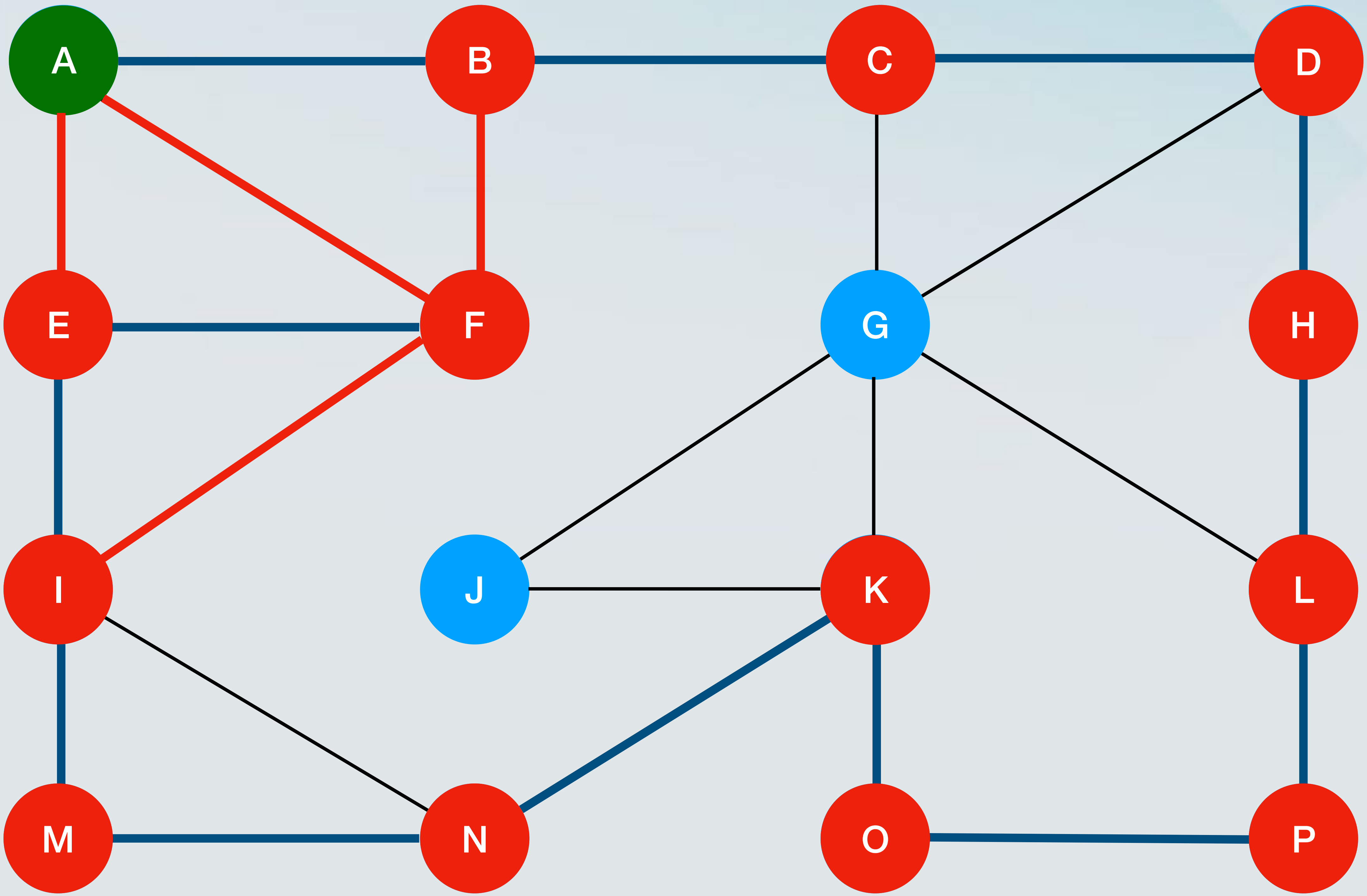
Depth-First Search



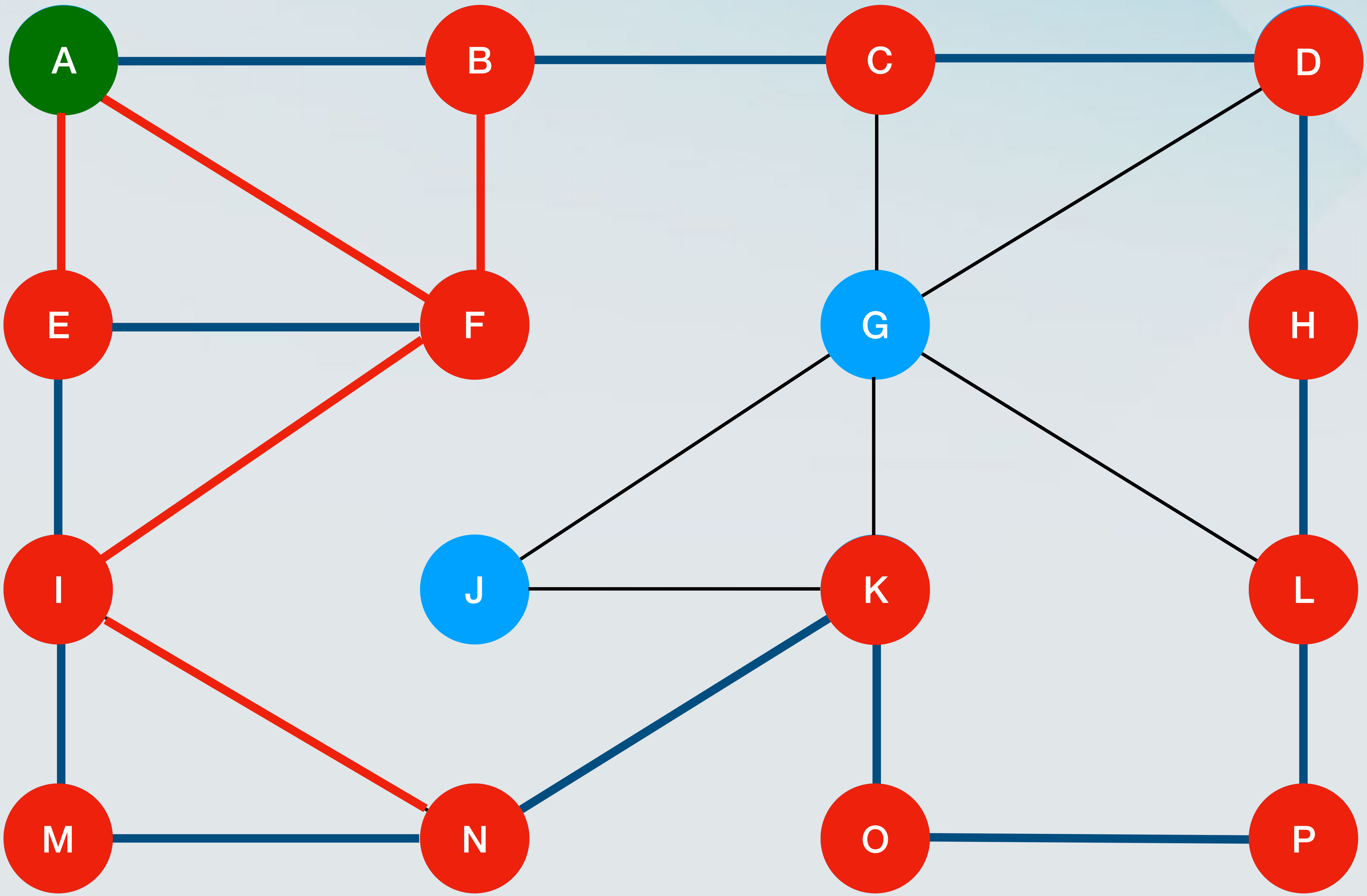
Depth-First Search



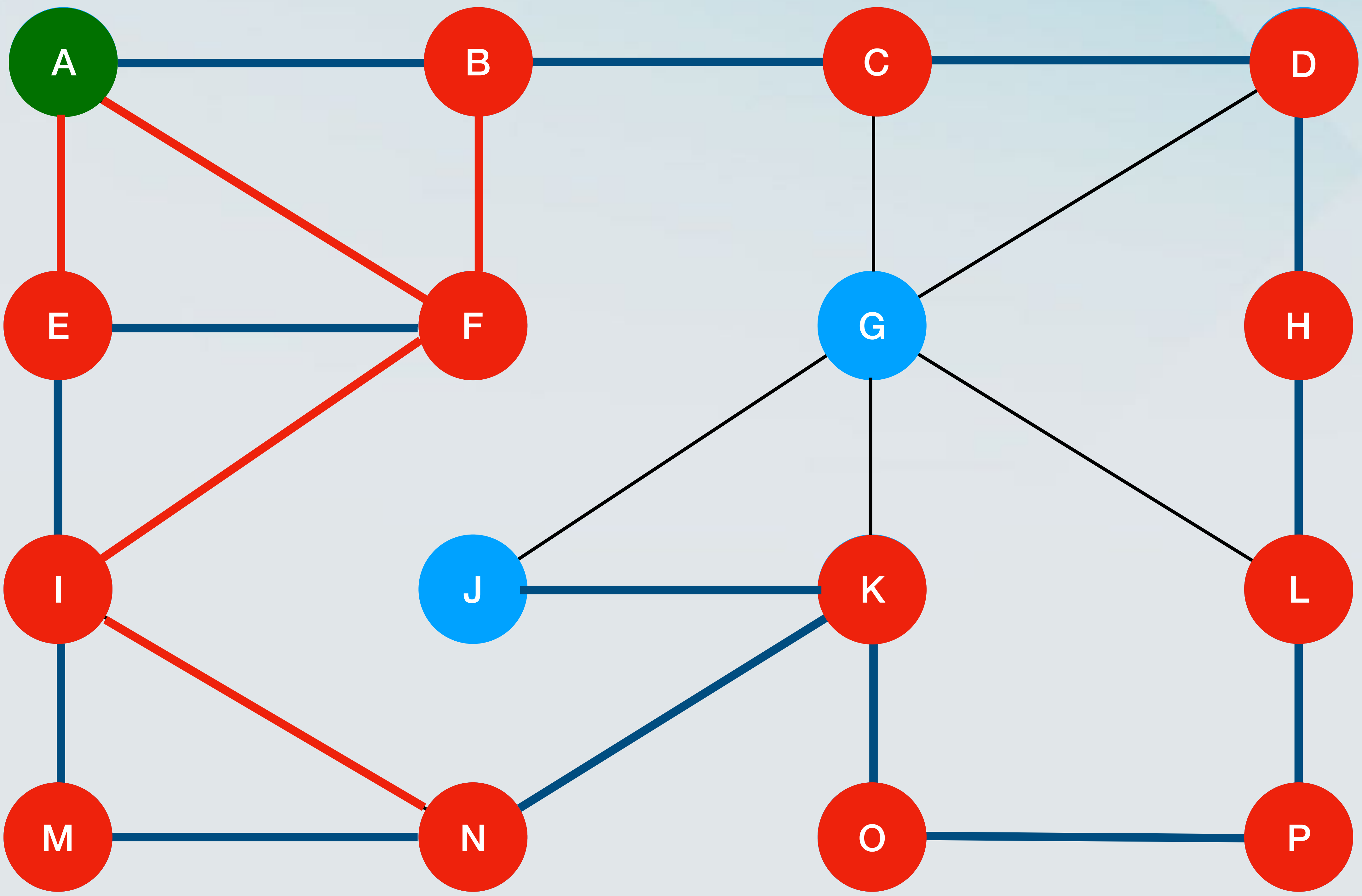
Depth-First Search



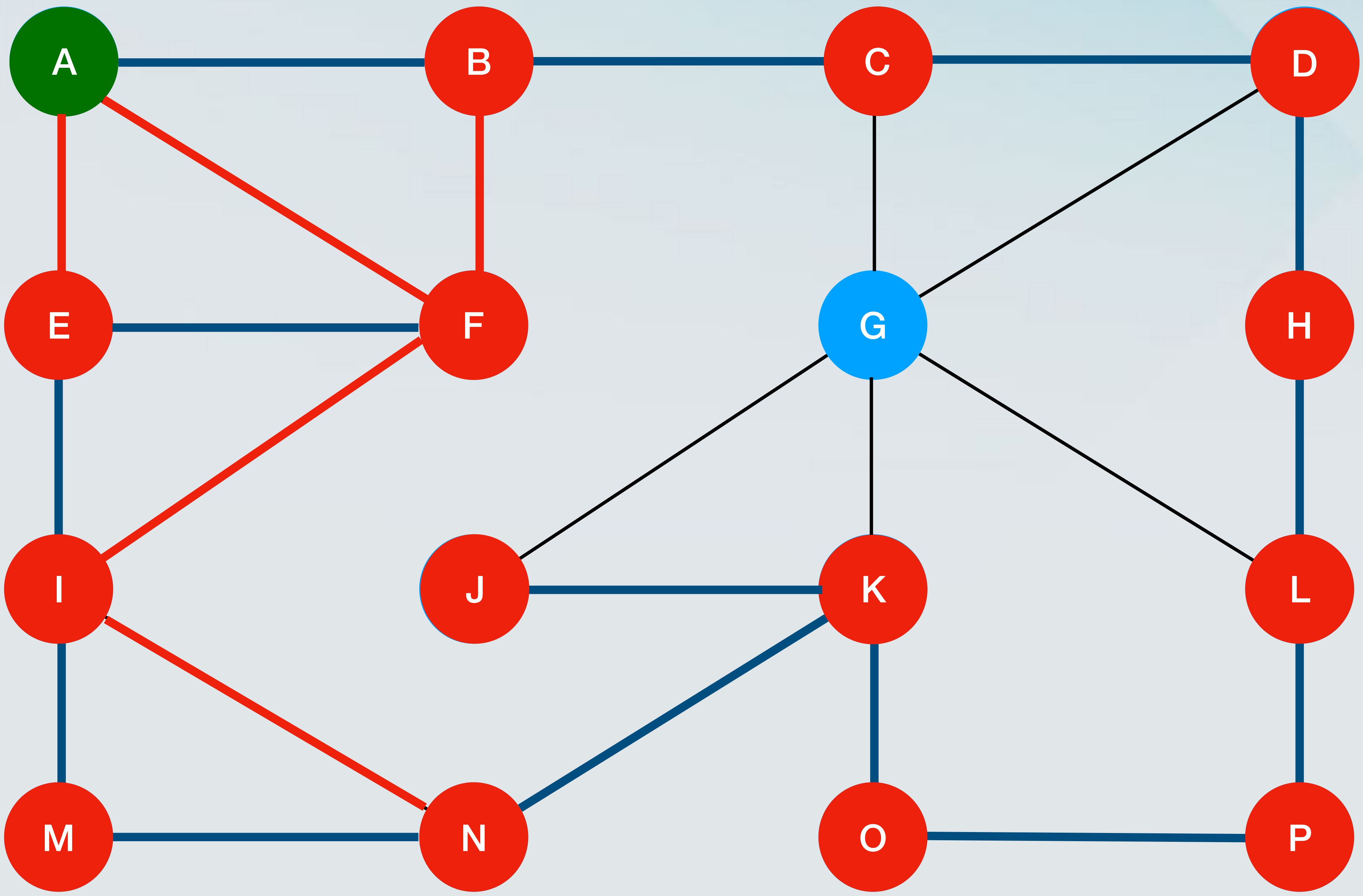
Depth-First Search



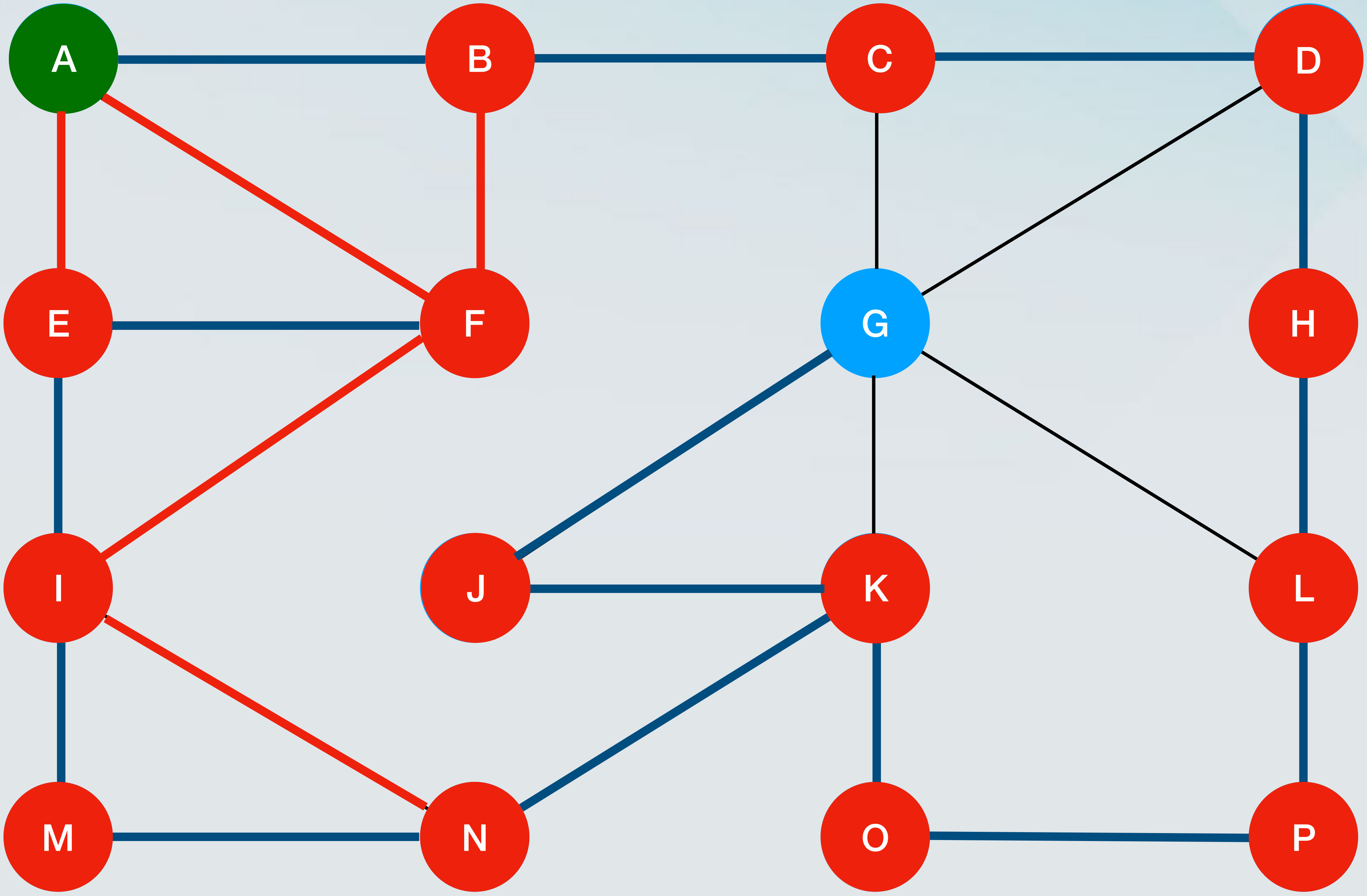
Depth-First Search



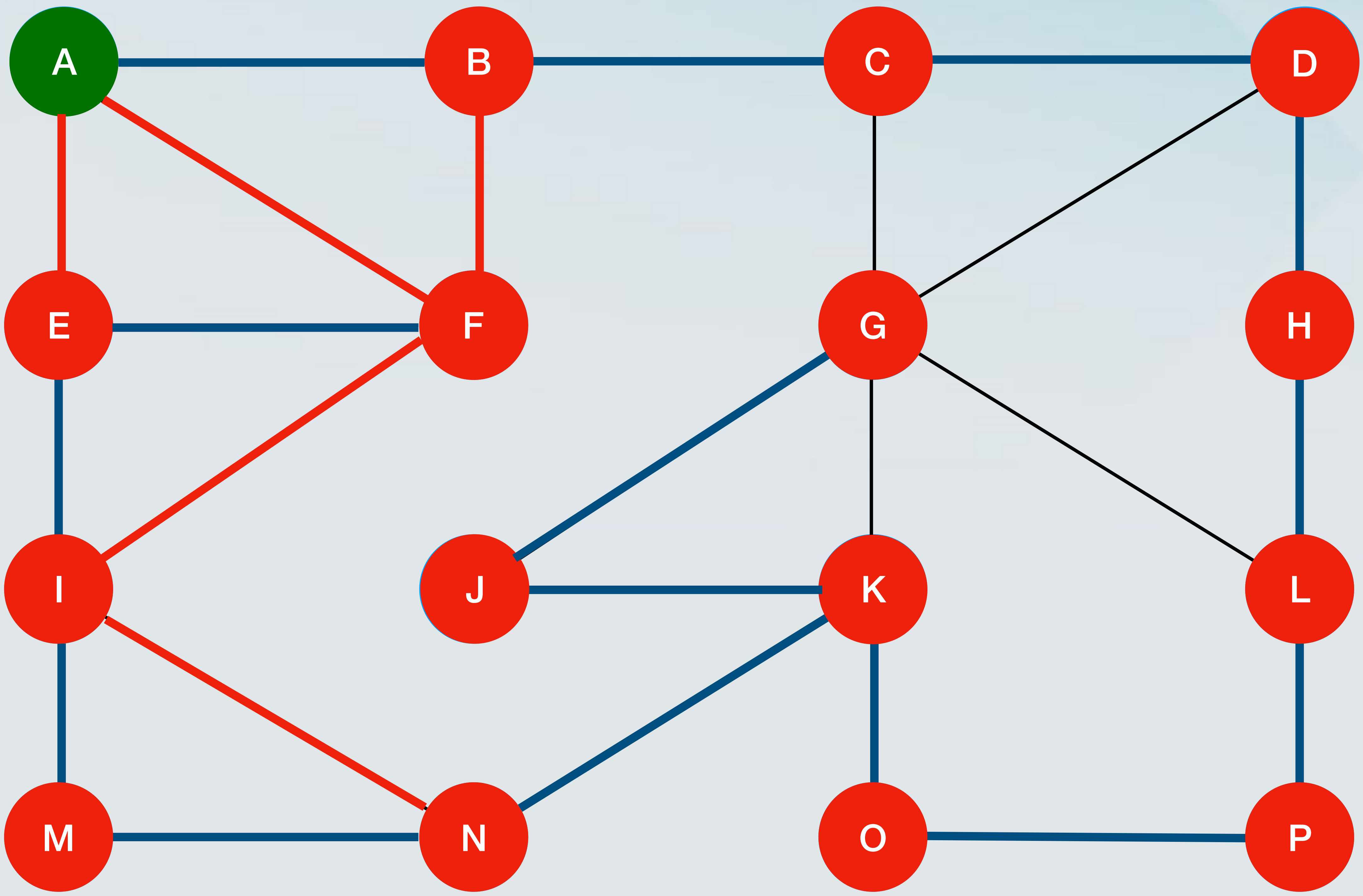
Depth-First Search



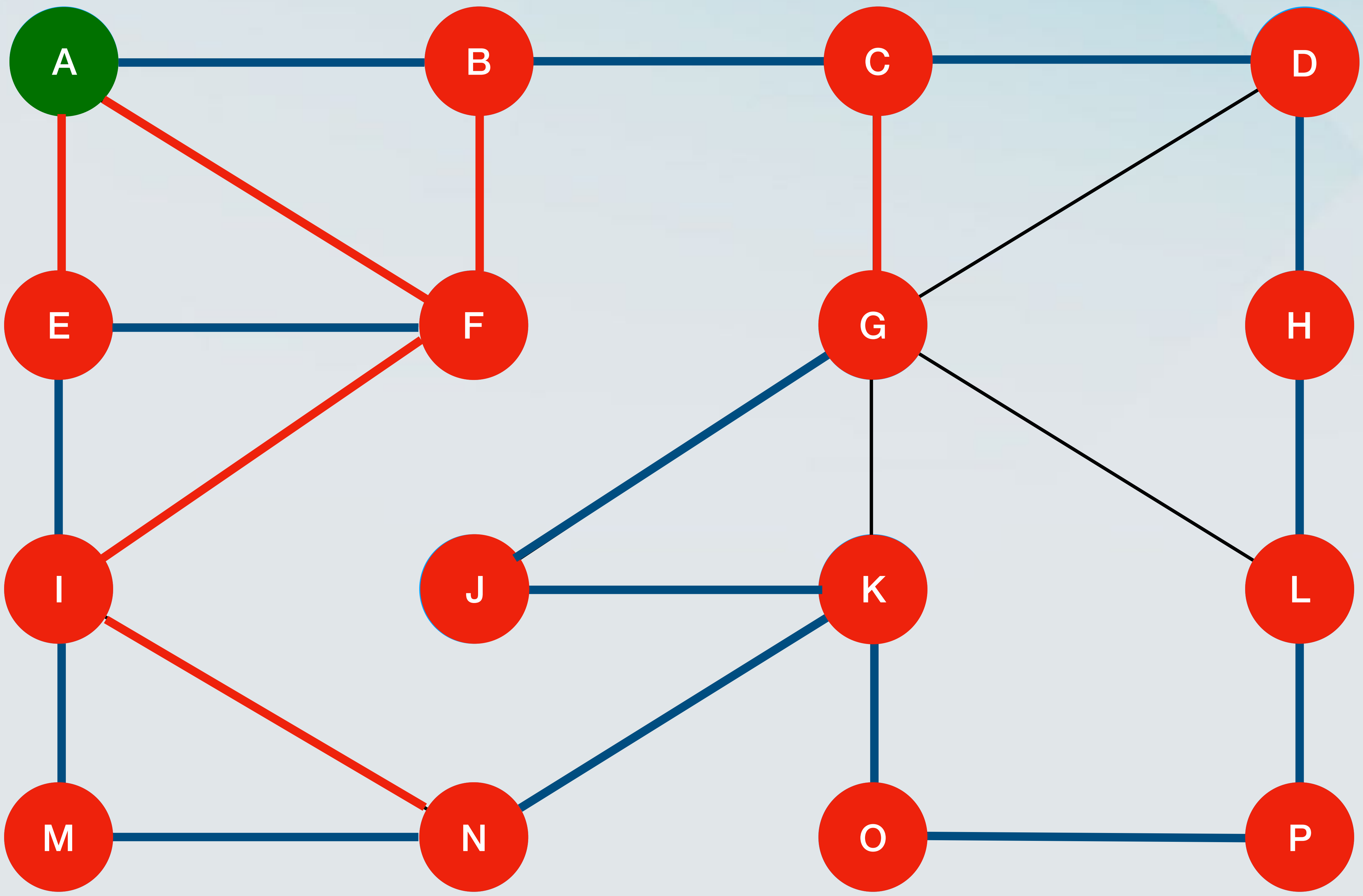
Depth-First Search



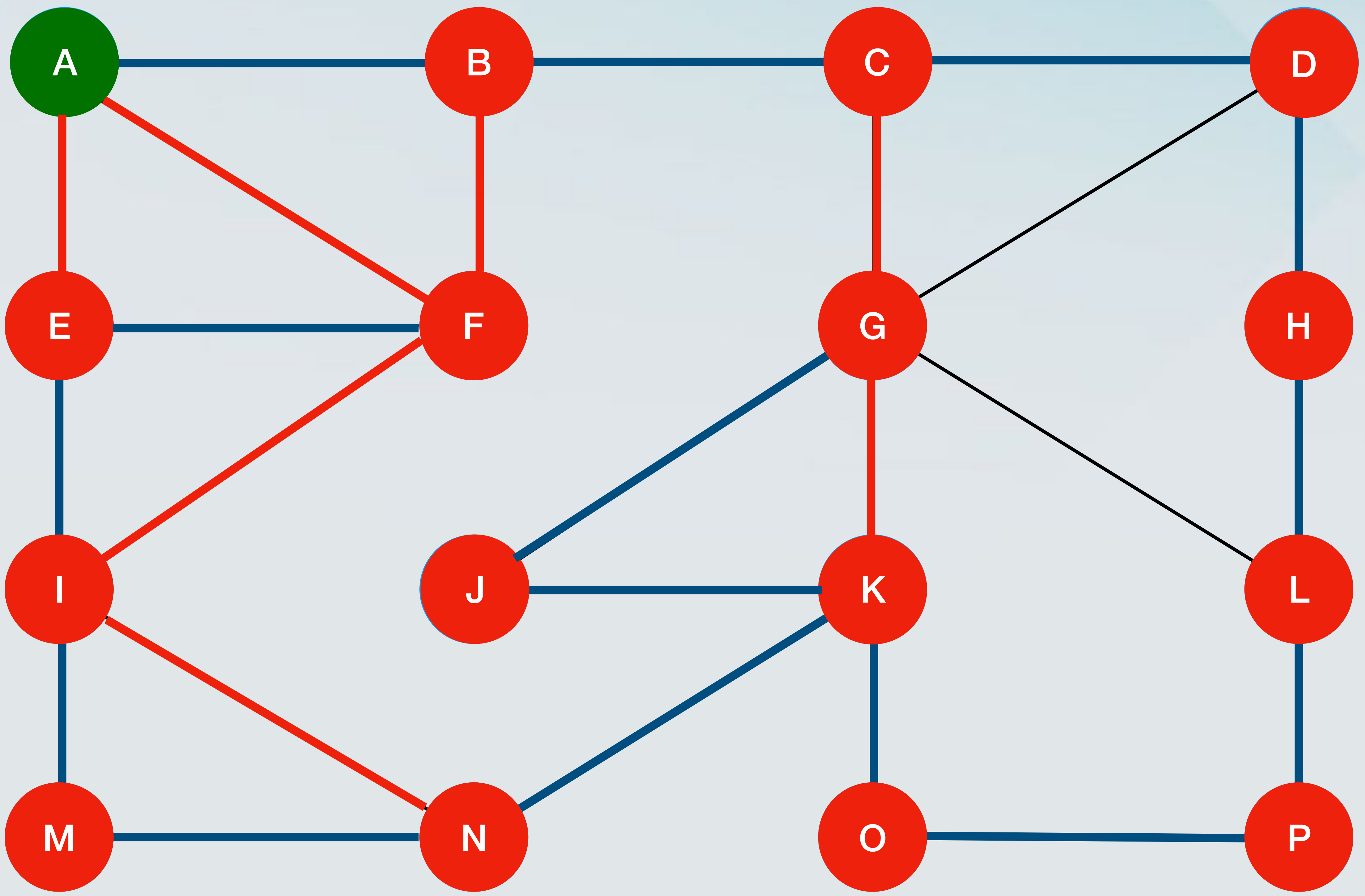
Depth-First Search



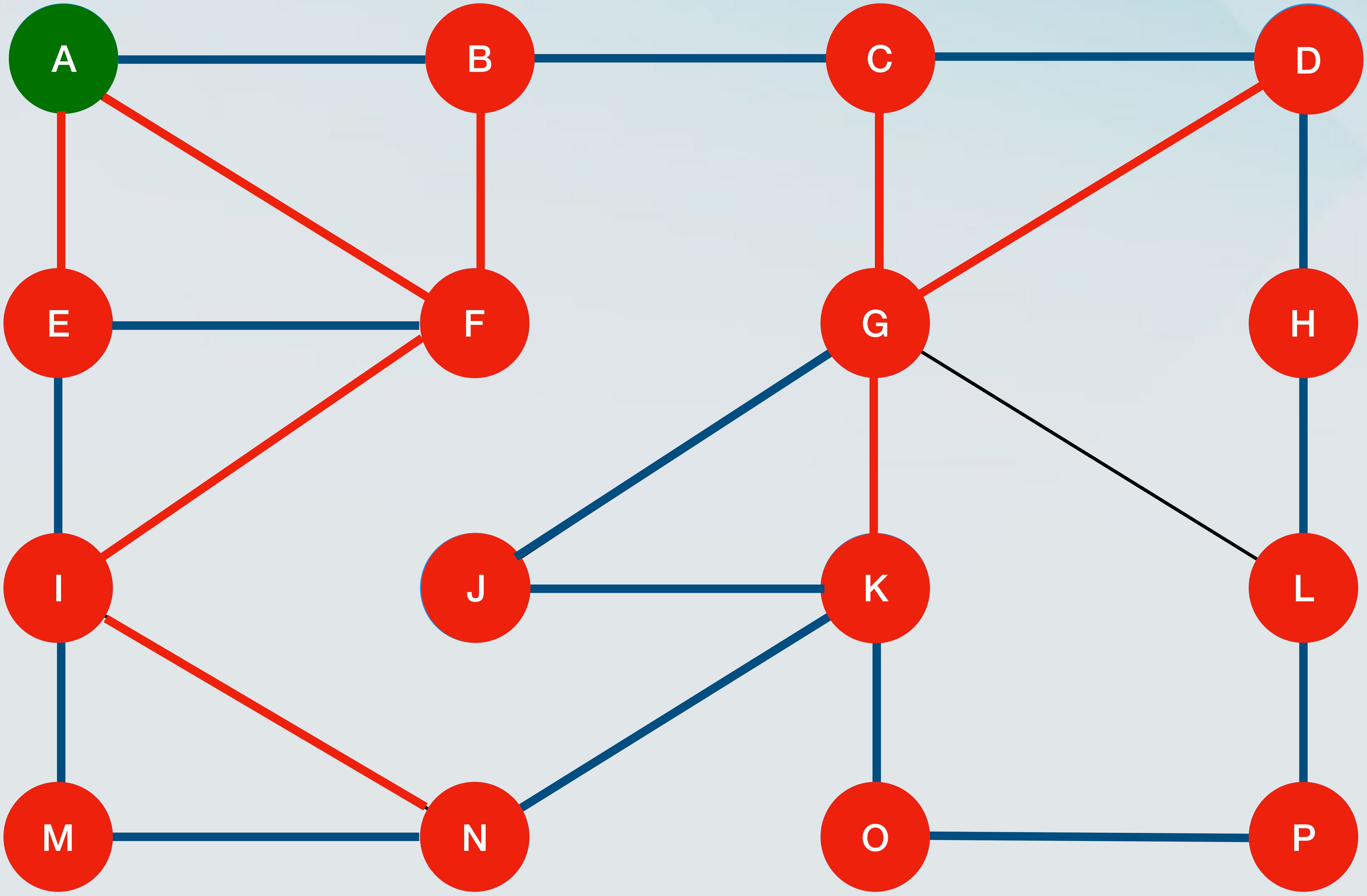
Depth-First Search



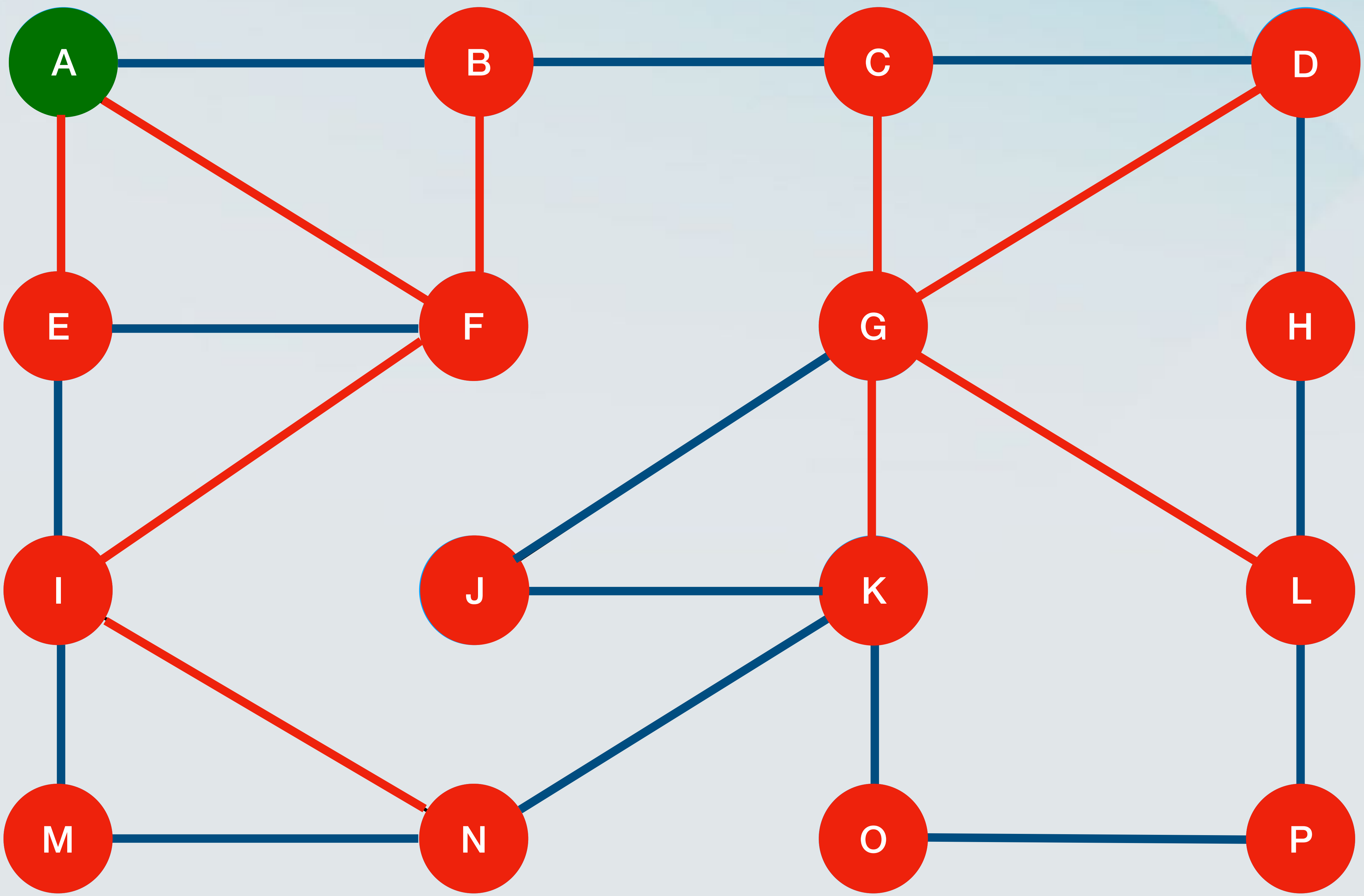
Depth-First Search



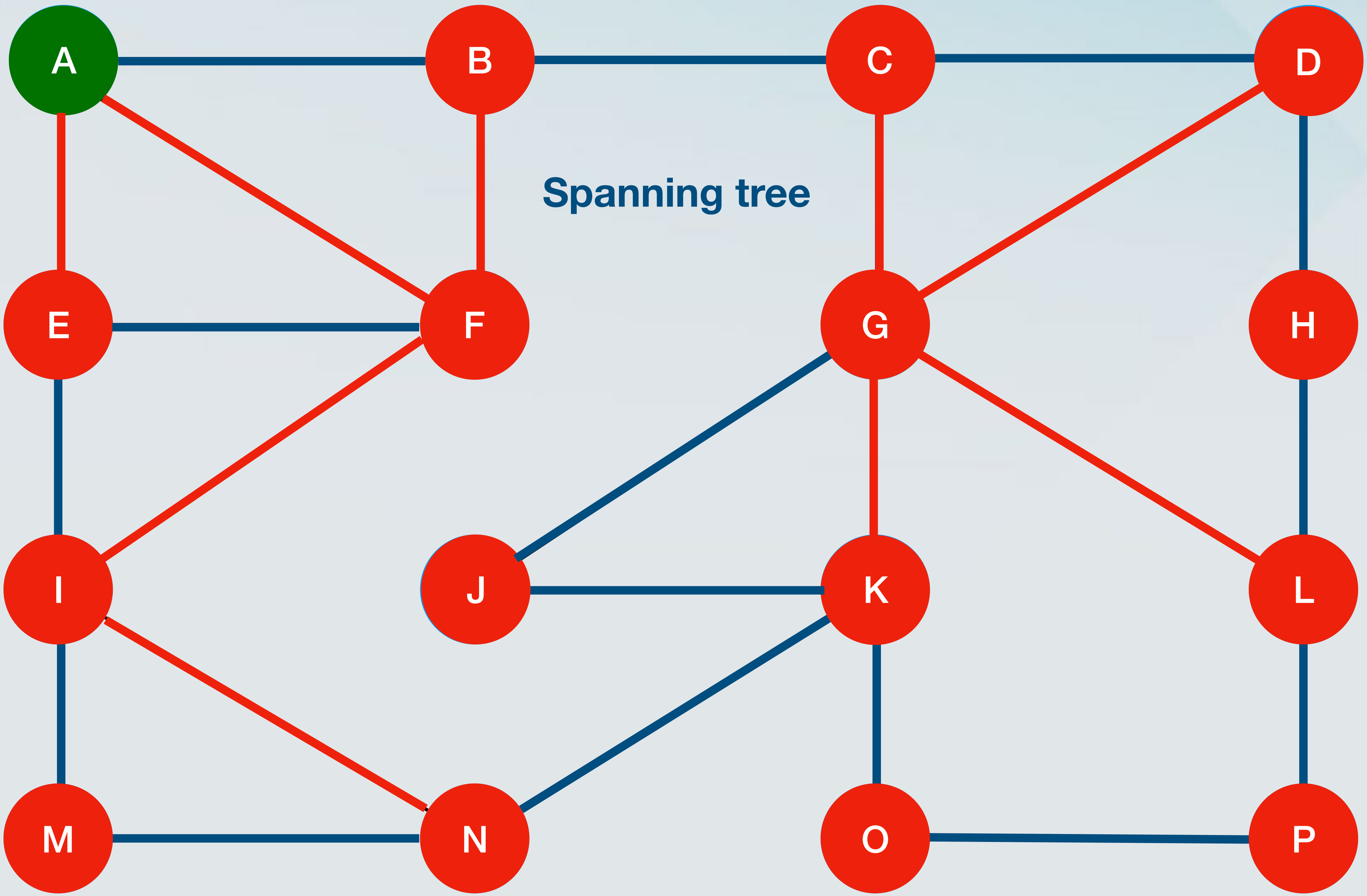
Depth-First Search



Depth-First Search



Depth-First Search



Implementing DFS

- We need the following properties:
  - We can find all incident edges to a vertex **v** in O(deg(**v**)) time.
  - Given one endpoint of an edge **e**, we can find the other endpoint in O(1) time.
  - We have a way of marking nodes or edges as "explored", and to test if a node or edge has been "explored" in O(1) time. **In other words, we never examine any edge twice!**

Properties of DFS

Properties of DFS

- For simplicity, assume that the graph is **connected.**

Properties of DFS

- For simplicity, assume that the graph is **connected.**
- DFS visits all nodes of the graph.

Properties of DFS

- For simplicity, assume that the graph is **connected.**
- DFS visits all nodes of the graph.
    - Quick proof: Assume by contradiction that some node **v** is unvisited and let **w** be the first unvisited node on some path from **s** to **v**. Since **w** was the first unvisited node, some neighbour **u** of w has been visited. But then, the edge **(u,w)** was explored and **w** was visited.

Properties of DFS

- For simplicity, assume that the graph is **connected.**
- DFS visits all nodes of the graph.
    - Quick proof: Assume by contradiction that some node **v** is unvisited and let **w** be the first unvisited node on some path from **s** to **v**. Since **w** was the first unvisited node, some neighbour **u** of w has been visited. But then, the edge **(u,w)** was explored and **w** was visited.
- The discovery edges form a spanning tree.

Properties of DFS

- For simplicity, assume that the graph is **connected.**
- DFS visits all nodes of the graph.
  - Quick proof: Assume by contradiction that some node **v** is unvisited and let **w** be the first unvisited node on some path from **s** to **v**. Since **w** was the first unvisited node, some neighbour **u** of w has been visited. But then, the edge **(u,w)** was explored and **w** was visited.
- The discovery edges form a spanning tree.
  - We only mark edges as discovered when we go to unvisited nodes. We can never have a cycle of discovered edges.

Running time of DFS

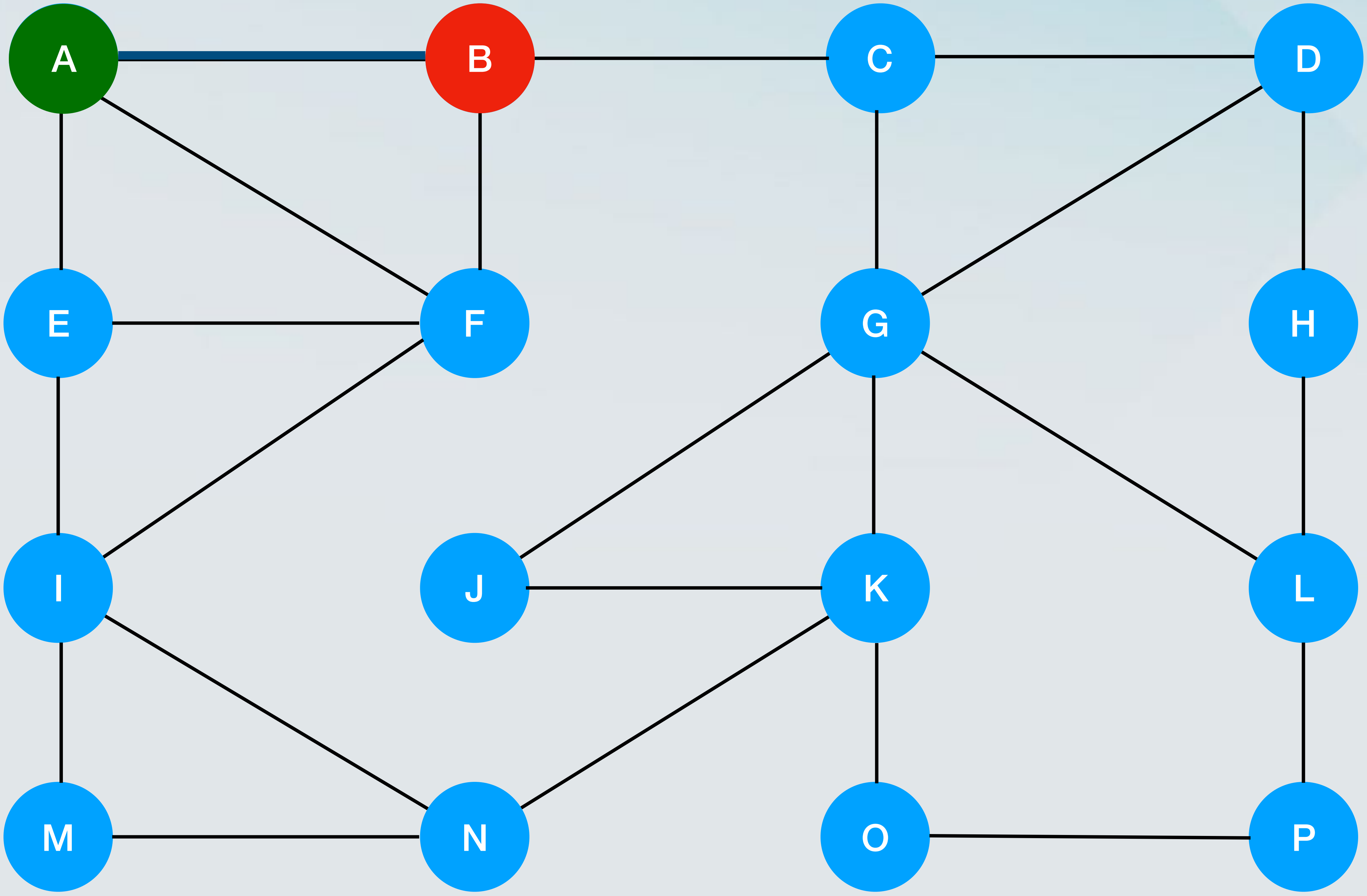
- DFS is called on each node exactly once.

Depth-First Search Pseudocode

```
Algorithm DFS(G,v)

    for all edges e incident to v.   /* all edges that have v as one of their endpoints */
        if edge e is unexplored
            Let u be the other endpoint of e
            If vertex u is unexplored
                Label e as a discovery edge
                DFS(G,u)
            Else
                Label e as a back edge
```

Depth-First Search



Running time of DFS

- DFS is called on each node exactly once.
- Every edge is examined exactly twice.
    - Once from each of its endpoint vertices.

Depth-First Search Pseudocode

```
Algorithm DFS(G,v)

for all edges e incident to v.   /* all edges that have v as one of their endpoints */
    if edge e is unexplored
        Let u be the other endpoint of e
        If vertex u is unexplored
            Label e as a discovery edge
            DFS(G,u)
        Else
            Label e as a back edge
```

Running time of DFS

- DFS is called on each node exactly once.
- Every edge is examined exactly twice.
    - Once from each of its endpoint vertices.
- Therefore, DFS runs in time **O(n+m)**.

Implementing DFS

- We need the following properties:
  - We can find all incident edges to a vertex **v** in O(deg(**v**)) time.
  - Given one endpoint of an edge **e**, we can find the other endpoint in O(1) time.
  - We have a way of marking nodes or edges as "explored", and to test if a vertex of edges has been "explored" in O(1) time. **In other words, we never examine any edge twice!**

Implementing DFS

The first two properties are satisfied by the Adjacency List representation!

- We need the following properties:
  - We can find all incident edges to a vertex **v** in O(deg(**v**)) time.
  - Given one endpoint of an edge **e**, we can find the other endpoint in O(1) time.
  - We have a way of marking nodes or edges as "explored", and to test if a vertex of edges has been "explored" in O(1) time. **In other words, we never examine any edge twice!**

Marking nodes

- We will need to following data structures
  - An Adjacency List for the graph, with a *.next* pointer, which goes through the neighbours of a vertex in order of appearance. (**v**.*next* gives the next neighbour).
  - A stack **S** (data structure where elements are put on top of each other).
  - An array explored[*1,…n*] where we will store the explored elements.

Marking nodes

Marking nodes

Is s.*next* in explored?

Marking nodes

Is s.*next* in explored? **No**

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

Marking nodes

Is *s.next* in explored?   No

u = *s.next*

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

Mark u as explored

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

Mark u as explored

Is u.*next* in explored?

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

Mark u as explored

Is u.*next* in explored?   No

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

Mark u as explored

Is u.*next* in explored?   No

v = u.*next*

Marking nodes

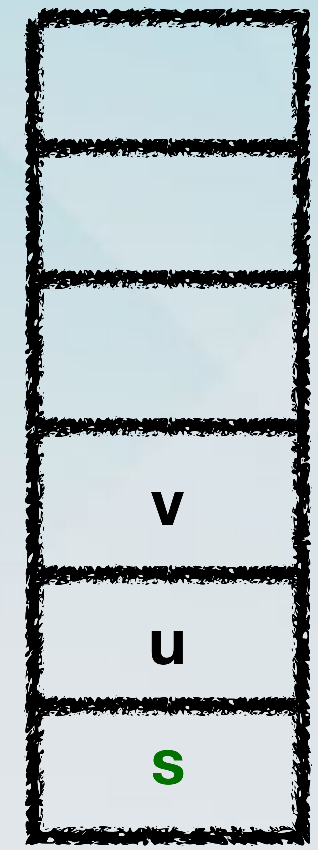
Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*



Marking nodes

Is s.*next* in explored? No

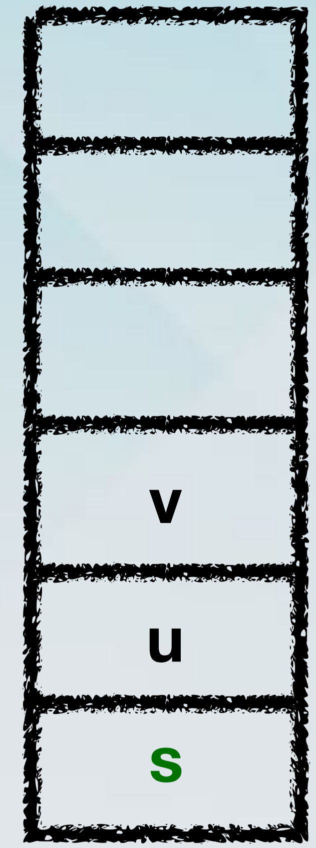
u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored



Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored?

Marking nodes

Is s.*next* in explored?   No

u = s.*next*

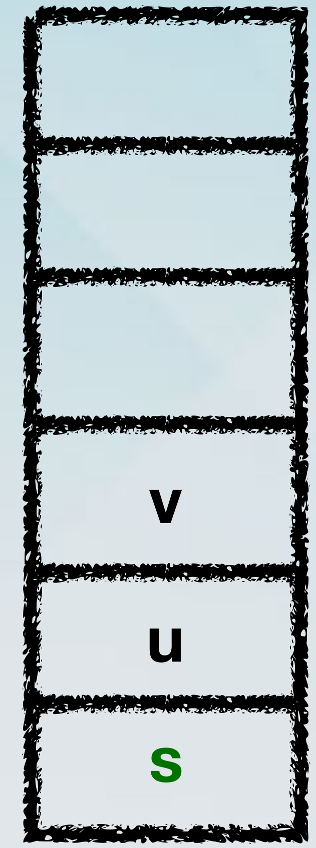
Mark u as explored

Is u.*next* in explored?   No

v = u.*next*

Mark v as explored

Is v.*next* in explored?   Yes



Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored?

Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty

Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty

Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

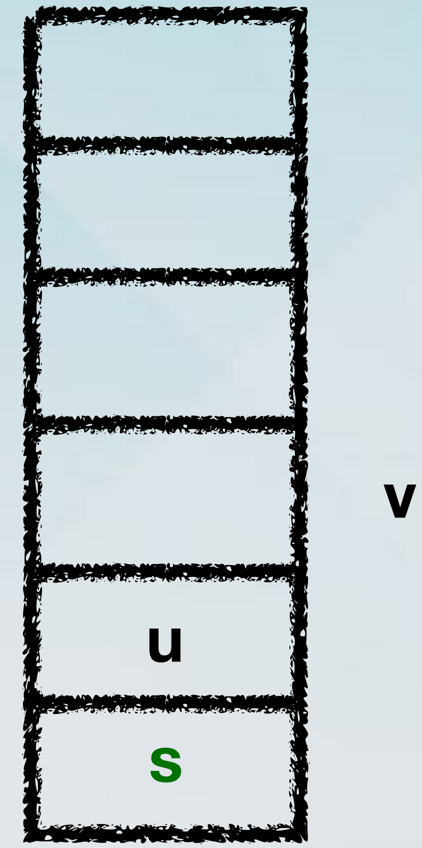
Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty

Remove v from the stack



Marking nodes

Is s.*next* in explored? No

Remove v from the stack

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

e=(u,v) is added to the spanning tree of DFS

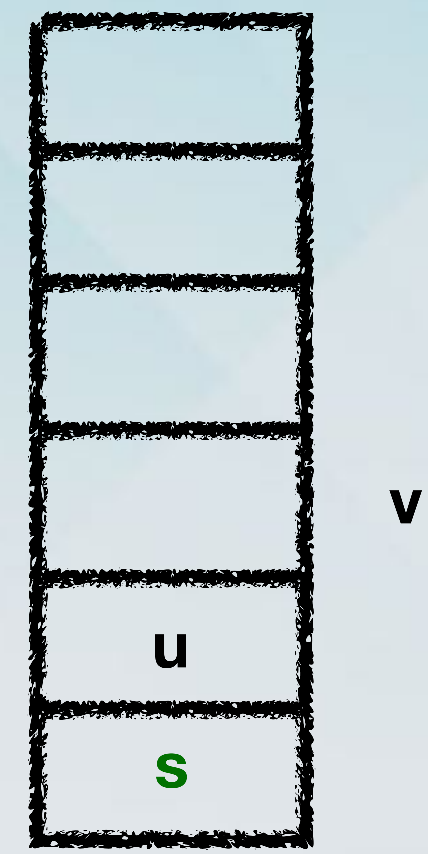
Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty



Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

e=(u,v) is added to the spanning tree of DFS

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty

Remove v from the stack

Continue with the top element of the stack

Marking nodes

Is s.*next* in explored? No

Remove v from the stack

u = s.*next*

Mark u as explored

We are about to consider the same neighbour!

Is u.*next* in explored? No

v = u.*next*

e=(u,v) is added to the spanning tree of DFS

Mark v as explored

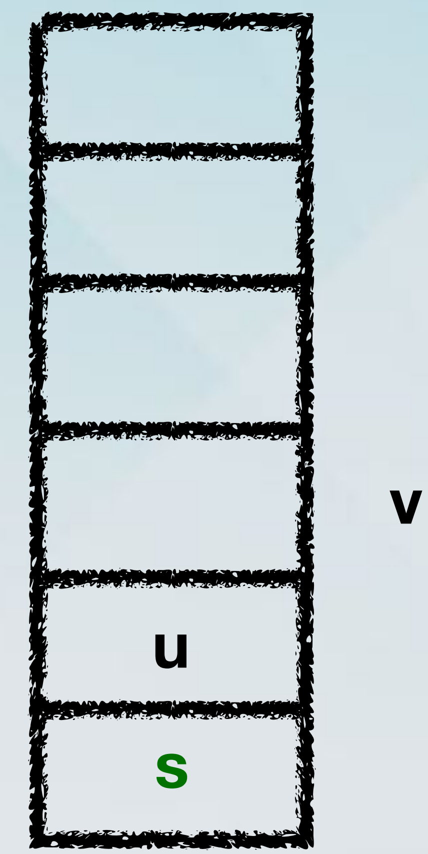
Continue with the top element of the stack

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty



Marking nodes

Is s.*next* in explored? No

u = s.*next*

Mark u as explored

Is u.*next* in explored? No

v = u.*next*

Mark v as explored

Is v.*next* in explored? Yes

Apply v.*next* once more to get the next neighbour

Is v.*next* in explored? Yes

When the neighbour set is empty

Remove v from the stack

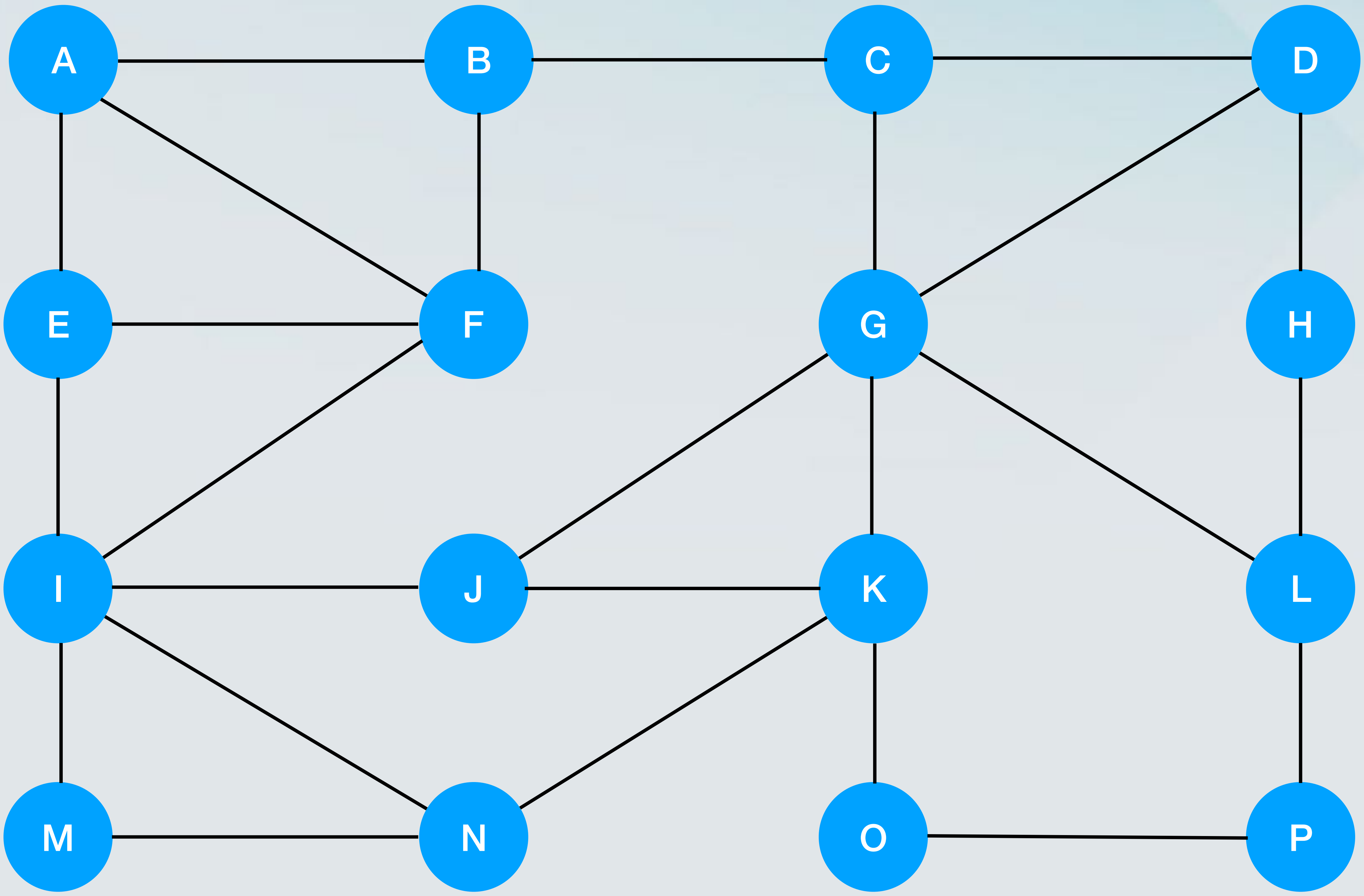
We are about to consider the same neighbour!

e=(u,v) is added to the spanning tree of DFS

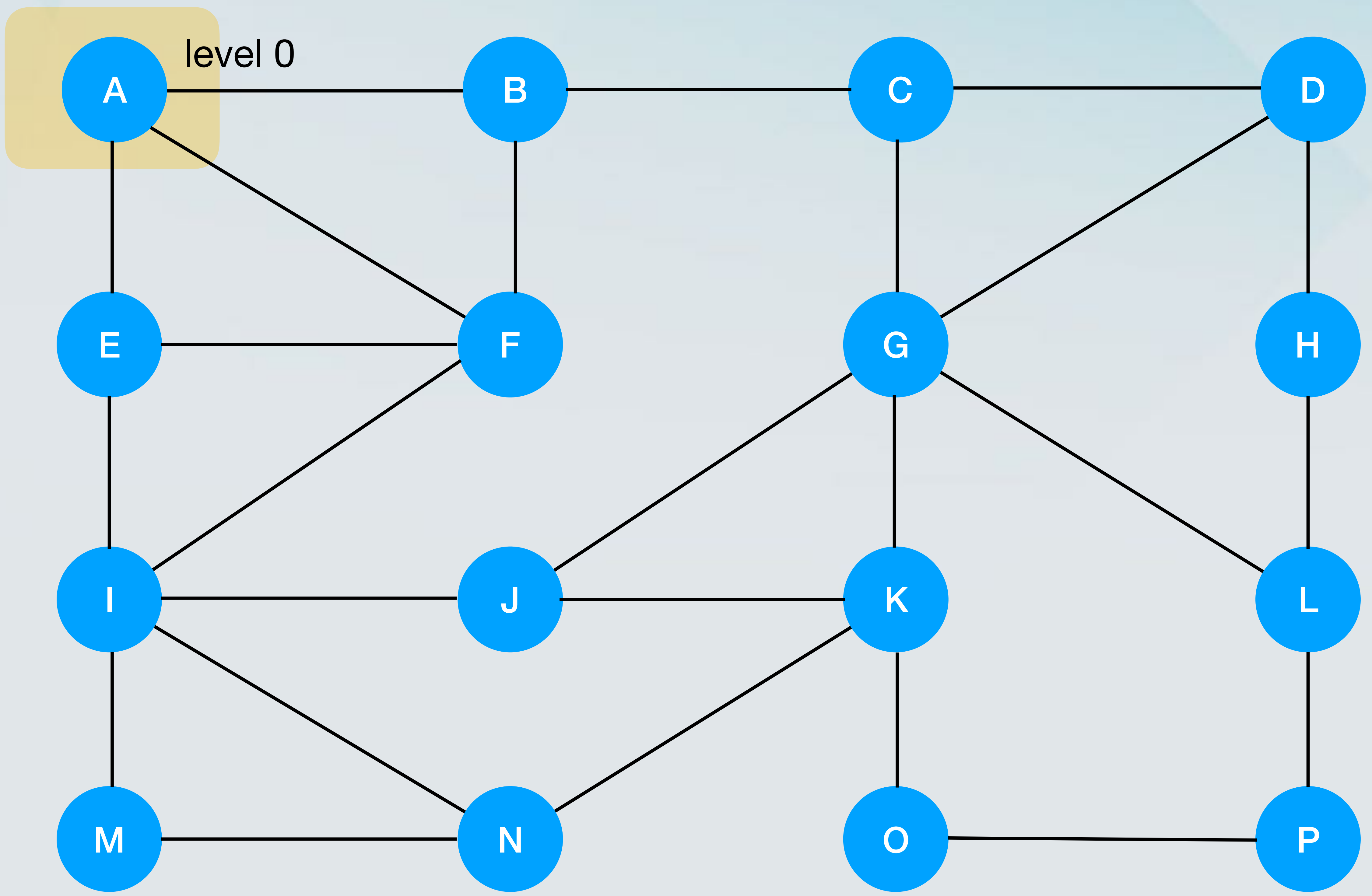
Continue with the top element of the stack

We remove the neighbours we have already considered.

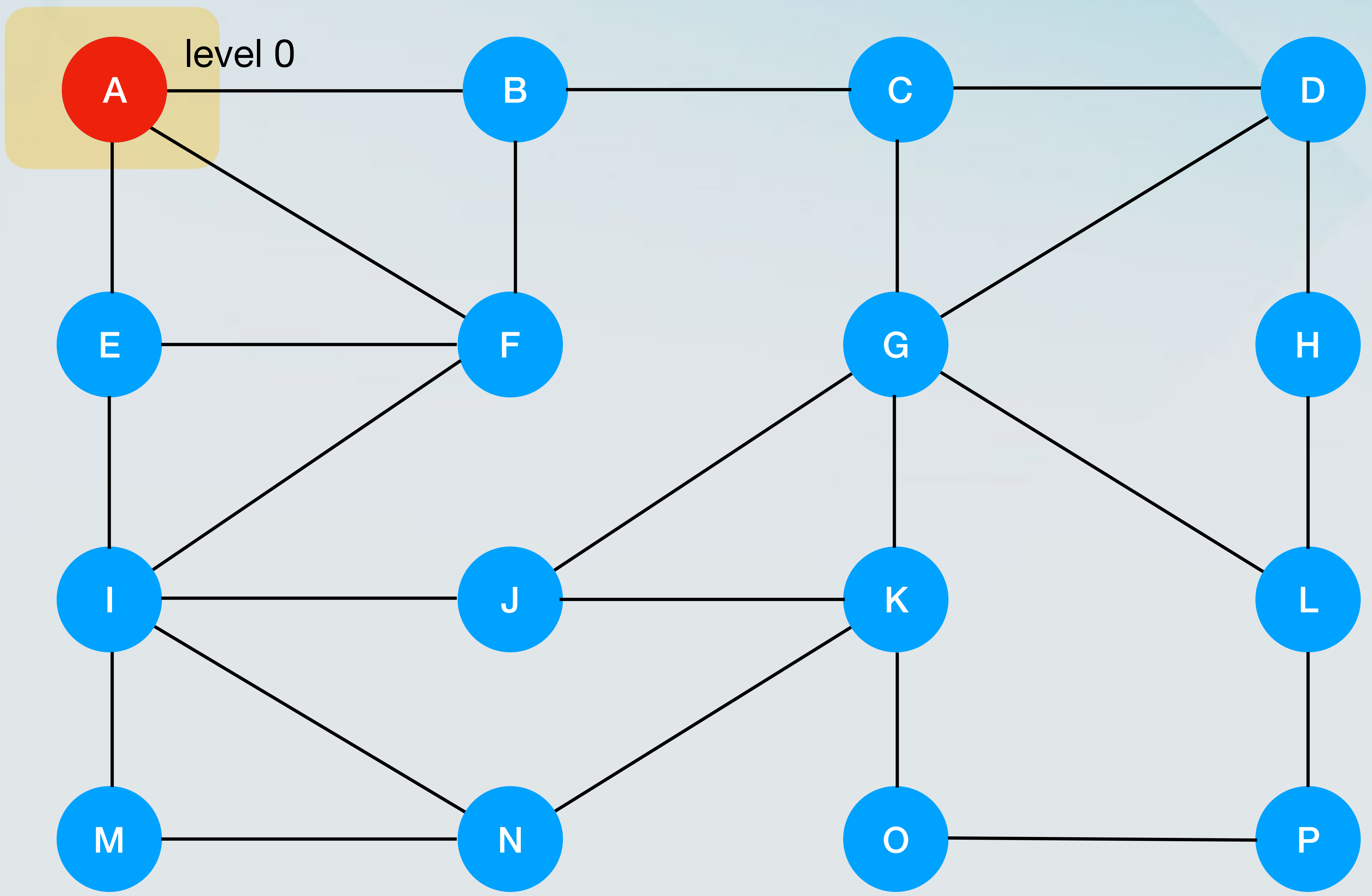
Breadth-First Search



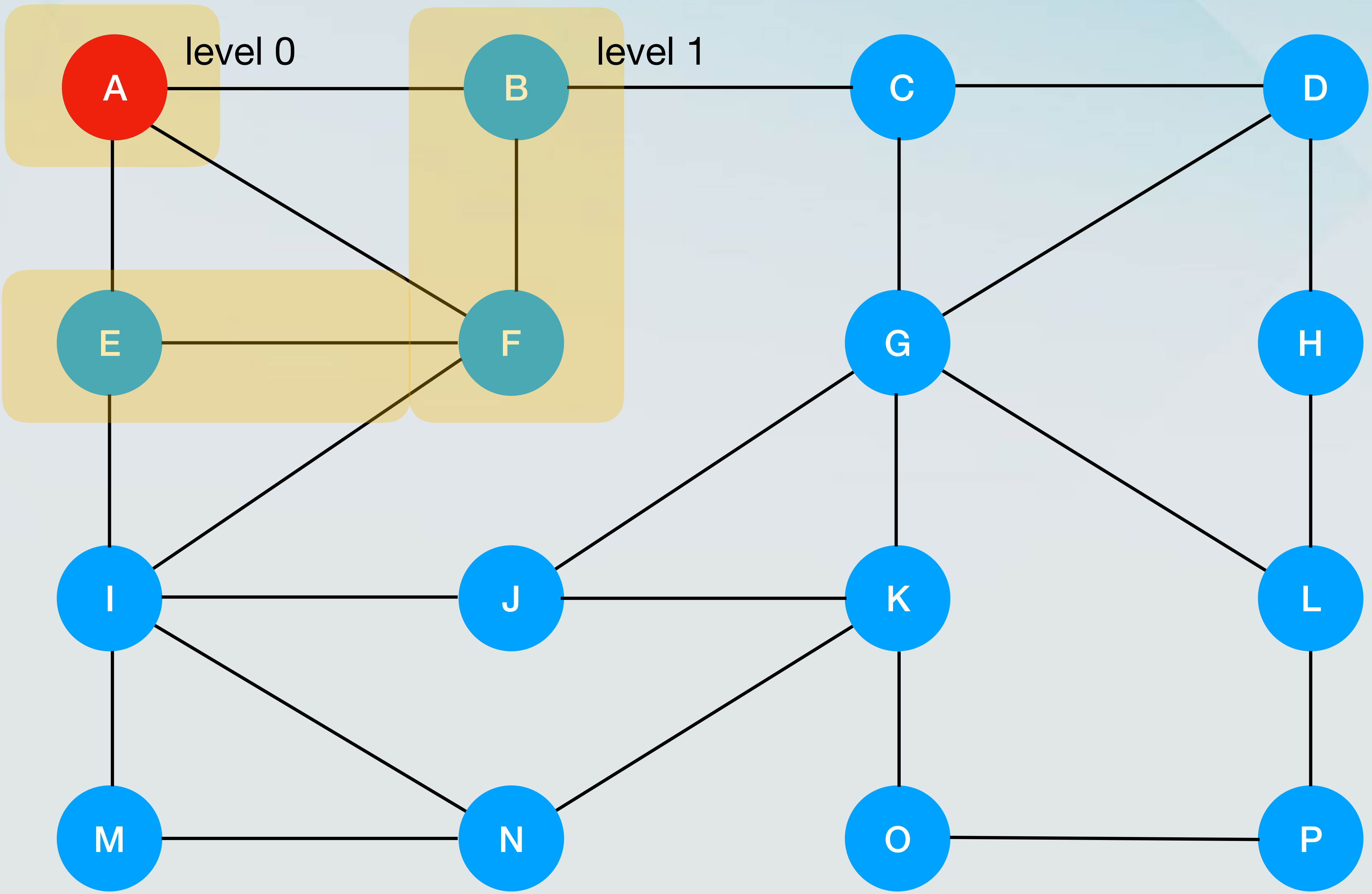
Breadth-First Search



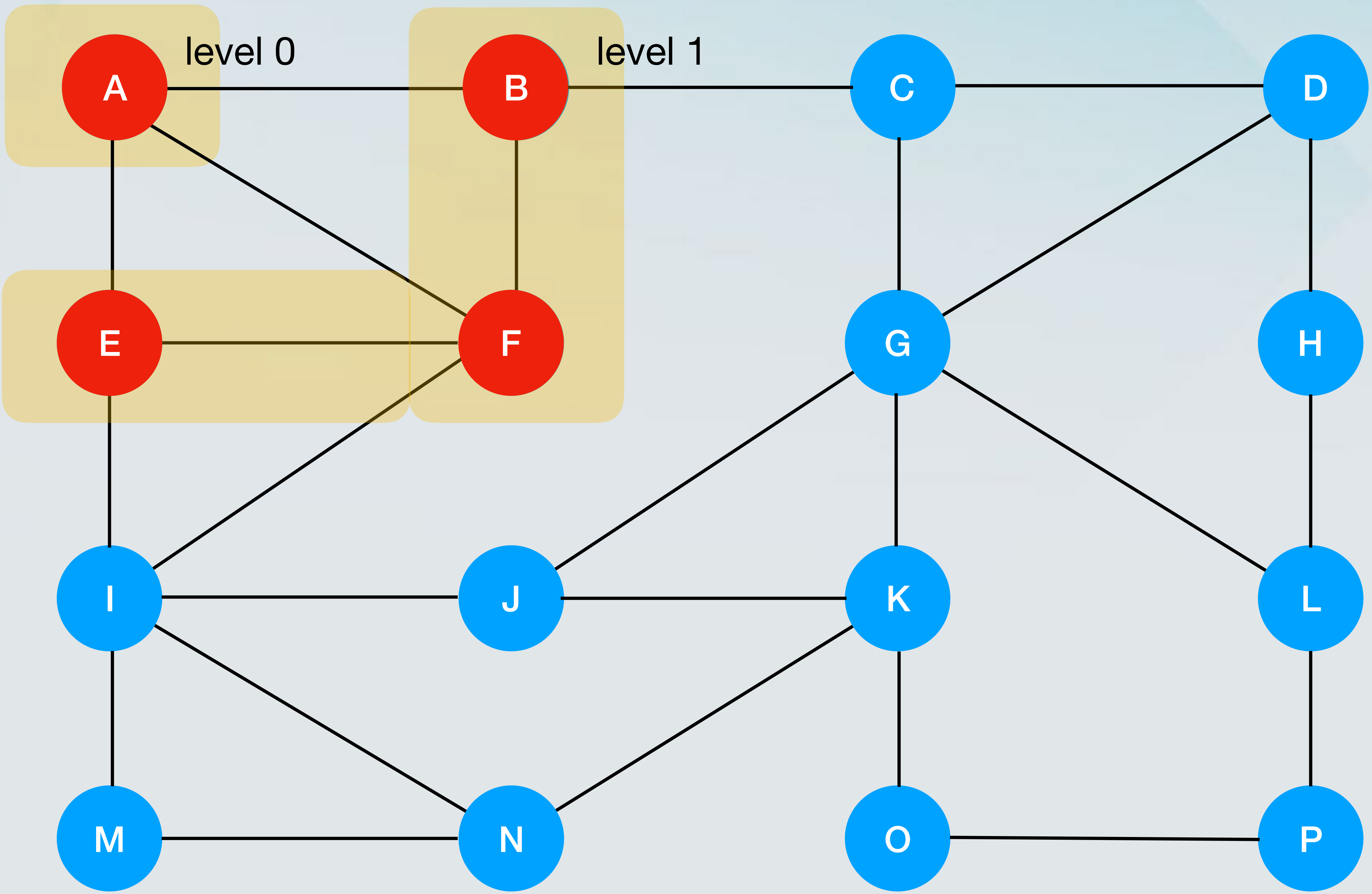
Breadth-First Search



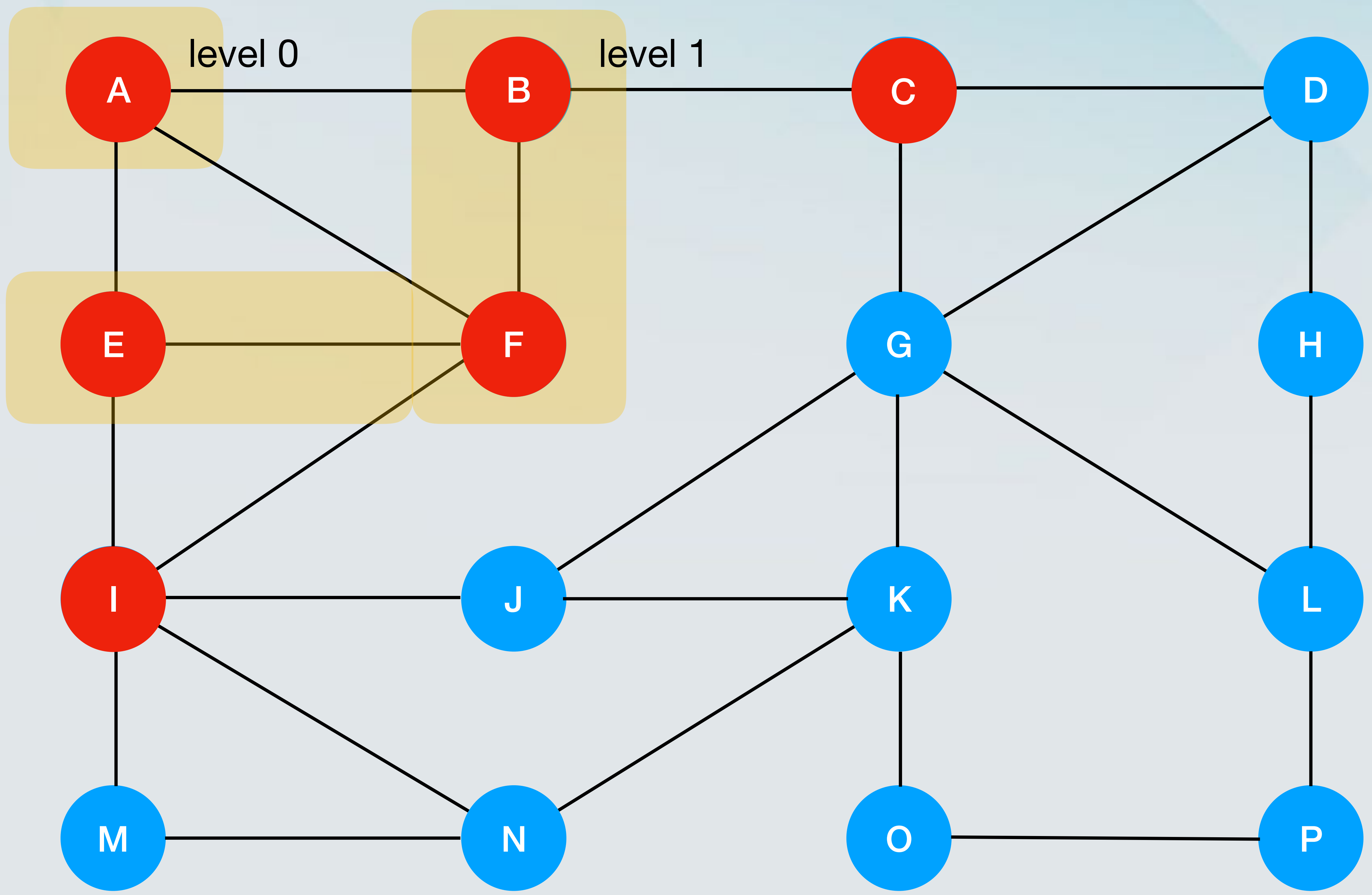
Breadth-First Search



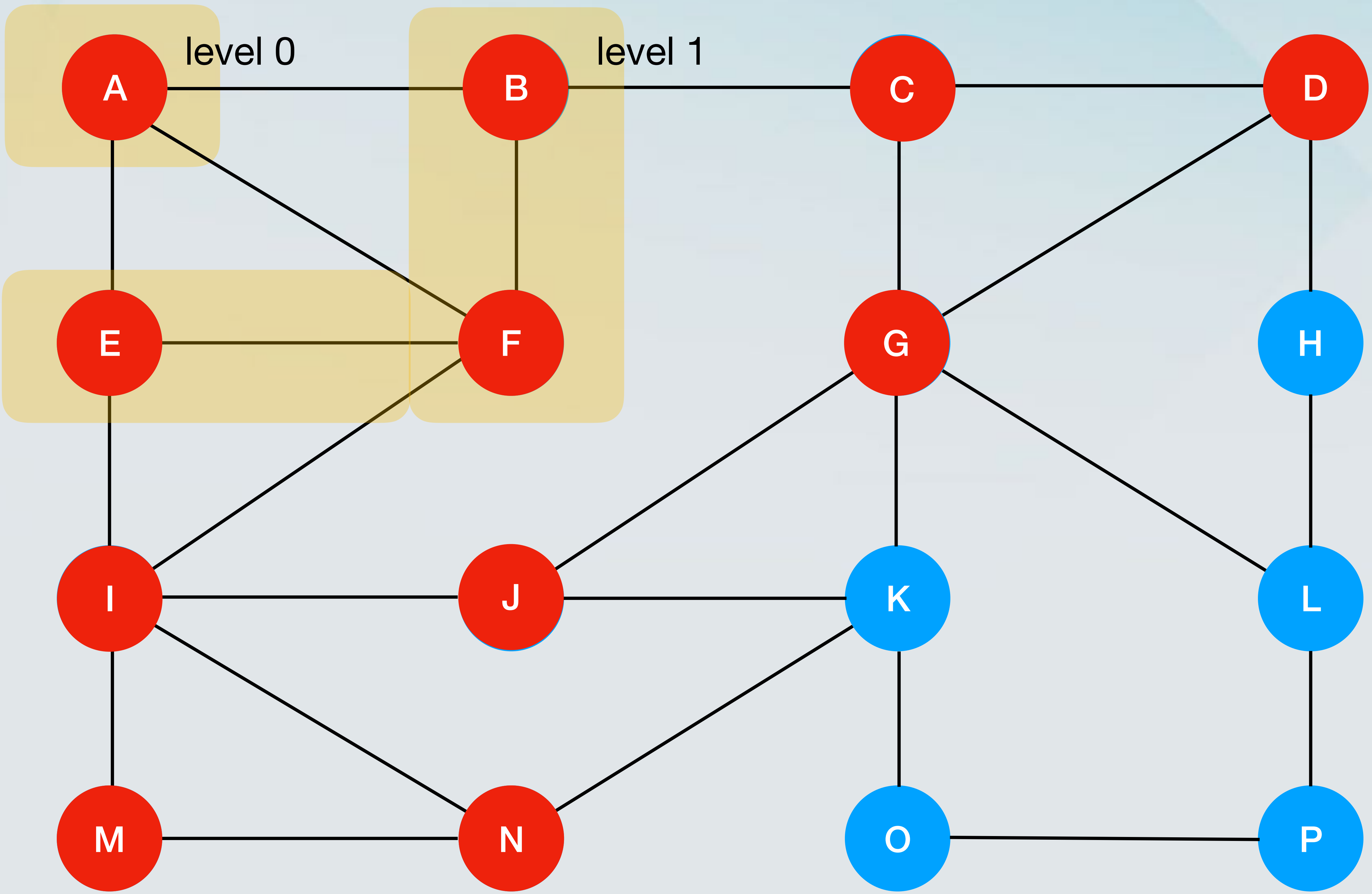
Breadth-First Search



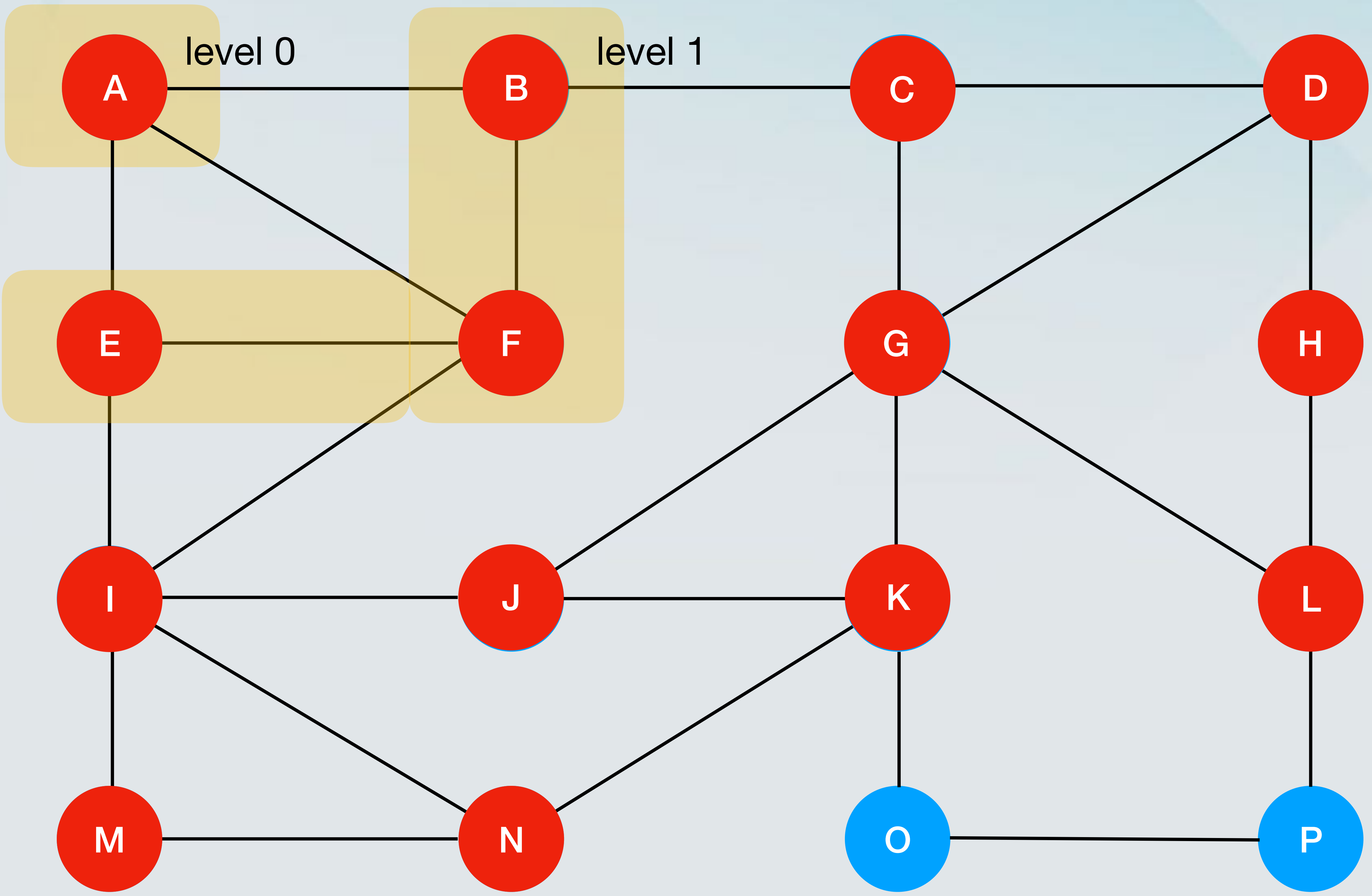
Breadth-First Search



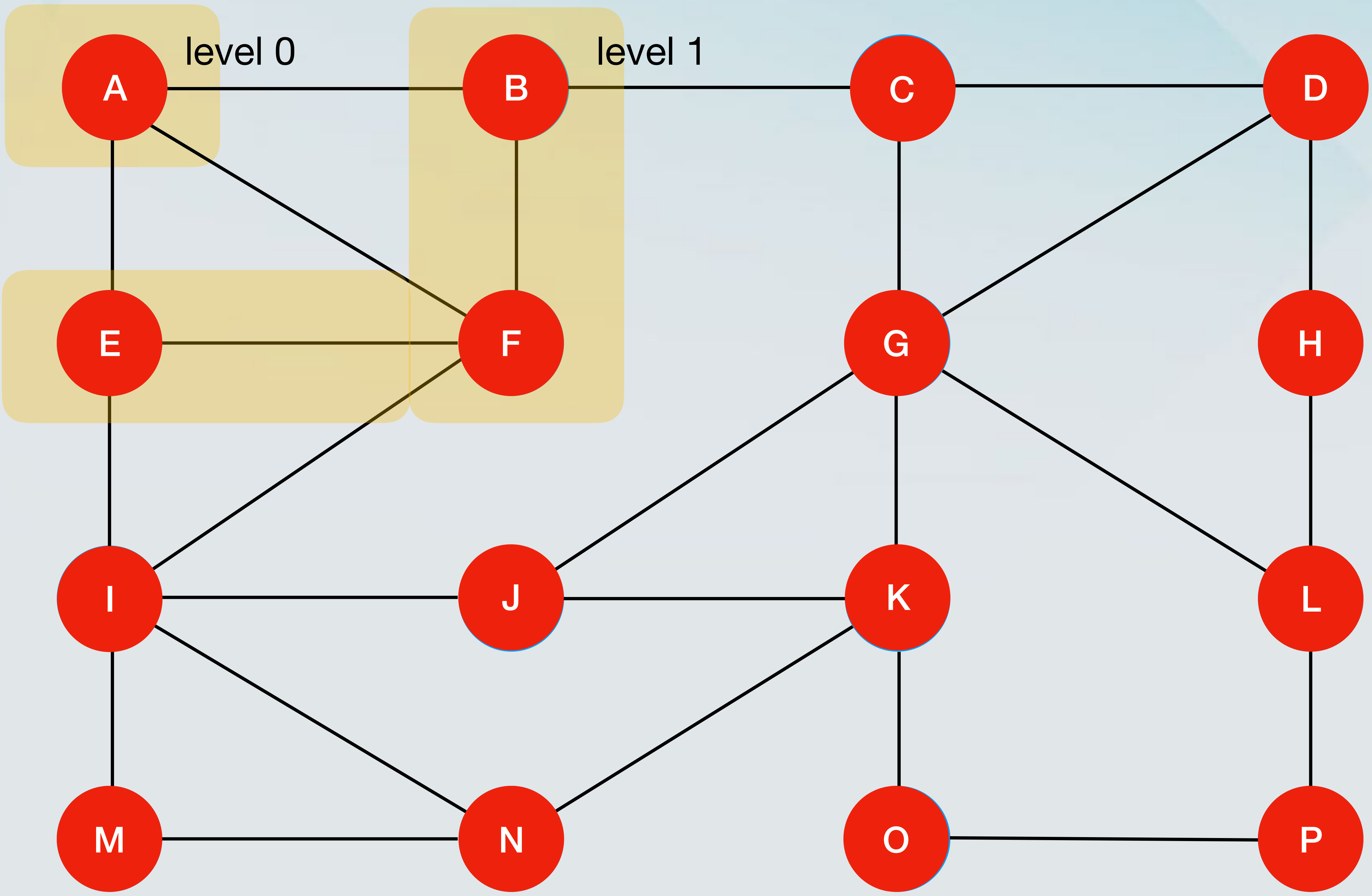
Breadth-First Search



Breadth-First Search



Breadth-First Search



Simple idea

- Start from the starting vertex **s** which is at *level 0* and consider it explored.
- For any node at *level i*, put all of its unexplored neighbours in *level i+1* and consider them explored.
- Terminate at *level j*, when none of the nodes of the level has any neighbours which are unexplored.

Visualising Breadth-First Search

Visualising Breadth-First Search

- Orient the edges along the direction in which they are visited during the traversal.

Visualising Breadth-First Search

- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.

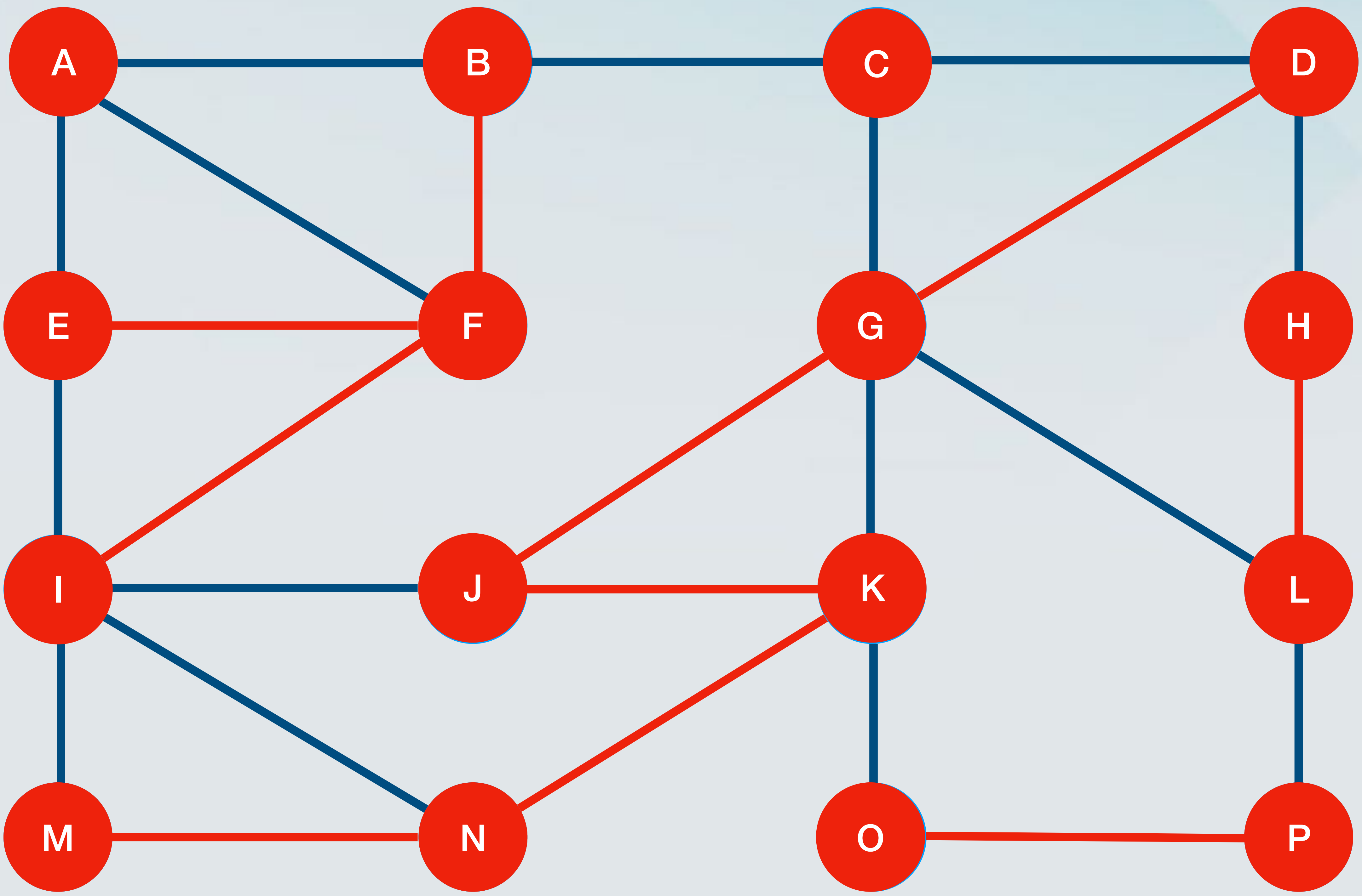
Visualising Breadth-First Search

- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.
  - Some edges are *cross edges*, because they lead to visited vertices.

Visualising Breadth-First Search

- Orient the edges along the direction in which they are visited during the traversal.
  - Some edges are *discovery edges*, because they lead to unvisited vertices.
  - Some edges are *cross edges*, because they lead to visited vertices.
- The discovery edges form a **spanning tree** of the **connected component** of the starting vertex **s**.

Breadth-First Search



Breadth-First Search Pseudocode

```
Algorithm BFS(G,s)

Initialise empty list L0
Insert s into L0

Set i=0
While Li is not empty
      Initialise empty list Li+1
      for each node v in Li
         for all edges e incident to v
           if edge e is unexplored
             let w be the other endpoint of e
             if node w is unexplored
                label e as discovery edge
                insert w into Li+1
             else
                label e as cross edge
i = i+1
```

Properties of BFS

- For simplicity, assume that the graph is **connected.**
- The traversal visits all vertices of the graph.
- The *discovery edges* form a spanning tree.
- The path of the spanning tree from **s** to a node **v** at level i has i edges, and this is the shortest path.
- If **e**=(**u**,**v**) is a *cross edge*, then the **u** and **v** differ by at most one level.

Running time of BFS

- In every iteration, we consider nodes on different levels.
  - Therefore nodes are not considered twice.
- Every edge is examined at most twice.
- Therefore, BFS runs in time **O(n+m)**.

DFS vs BFS

DFS vs BFS

- Which one is better?

DFS vs BFS

- Which one is better?
- Depends on what we use it for.

DFS vs BFS

- Which one is better?
- Depends on what we use it for.
- Stay tuned.