Advanced Algorithmic Techniques (COMP523)

Graph Algorithms #2

Recap and plan

Recap and plan

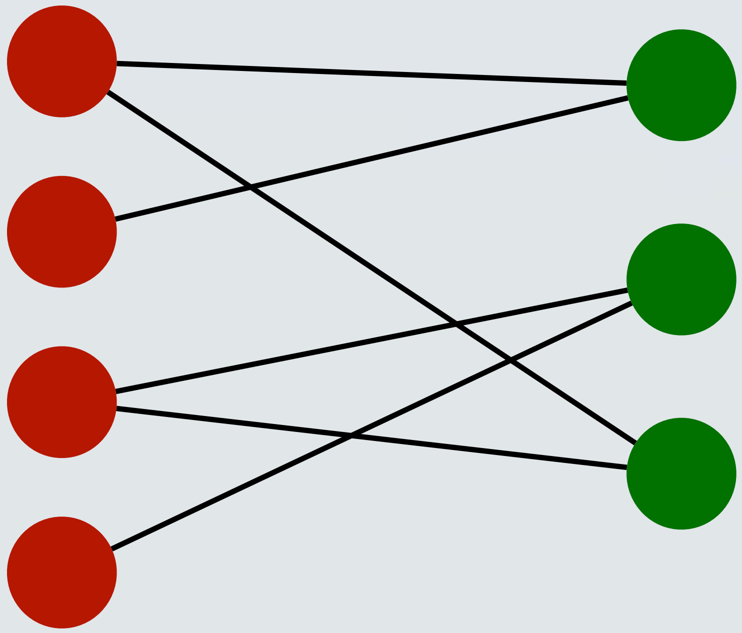
- **Last lecture:**
  - Graph definitions
  - Graph representations
  - Depth-First Search, Breadth-First Search

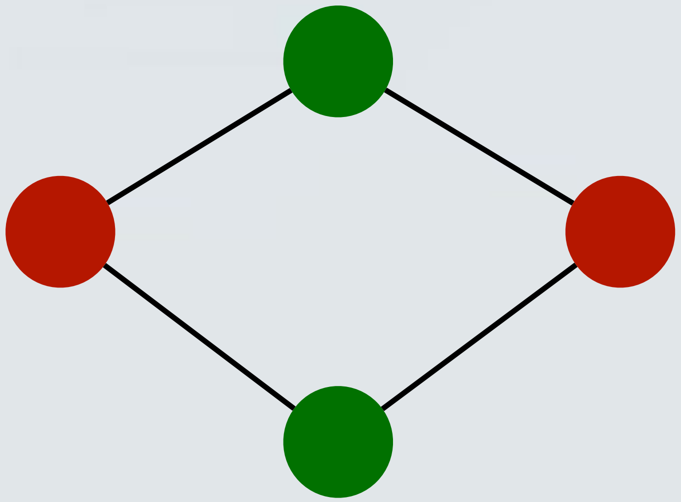
Recap and plan

- **Last lecture:**
  - Graph definitions
  - Graph representations
  - Depth-First Search, Breadth-First Search
- **This lecture:**
  - Testing bipartiteness
  - DFS and BFS on directed graphs
  - Testing connectivity

Bipartite graphs

- A graph G=(V,E) is bipartite *if any only if* it can be partitioned into sets A and B such that each edge has one endpoint in A and one endpoint in B.
  - Often, we write G=(A U B,E).





Alternative definitions

- A graph G=(V,E) is bipartite *if any only if* its nodes can be coloured with 2 colours (say red and green), such that every vertex has one red endpoint and one green endpoint.
- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.

No odd cycles

No odd cycles

- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.

No odd cycles

- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.
    - **=>** Assume that G is bipartite

No odd cycles

- A graph $G=(V,E)$ is bipartite *if any only if* it does not contain any cycles of odd length.
  - => Assume that G is bipartite
  - Suppose that G does contain an odd cycle (proof by contradiction), $C = u_1\ u_2\ u_3\ \ldots\ u_n\ u$ for some u in A *(wlog),* or alternatively, for some u that is red.

No odd cycles

- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.
  - => Assume that G is bipartite
  - Suppose that G does contain an odd cycle (proof by contradiction), $C = u_1\ u_2\ u_3\ \dots\ u_n\ u$ for some u in A *(wlog),* or alternatively, for some u that is red.
  - Because G is bipartite, u_2 must be green, and then u_3 must be red, and so on.

No odd cycles

- A graph $G=(V,E)$ is bipartite *if any only if* it does not contain any cycles of odd length.
  - **=>** Assume that G is bipartite
  - Suppose that G does contain an odd cycle (proof by contradiction), $C = u_1\ u_2\ u_3\ \ldots\ u_n\ u$ for some u in A *(wlog),* or alternatively, for some u that is red.
  - Because G is bipartite, u_2 must be green, and then u_3 must be red, and so on.
  - Generally, we observe that for all k in {1,2, … ,n}, u_k is red if k is odd and green if k is even.

No odd cycles

- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.
  - **=>** Assume that G is bipartite
  - Suppose that G does contain an odd cycle (proof by contradiction), $C = u_1\ u_2\ u_3\ \ldots\ u_n\ u$ for some u in A *(wlog),* or alternatively, for some u that is red.
  - Because G is bipartite, u_2 must be green, and then u_3 must be red, and so on.
  - Generally, we observe that for all k in {1,2, … ,n}, u_k is red if k is odd and green if k is even.
  - By assumption, n is odd, so it must be red. But then u cannot be red, because G is bipartite.

Alternative definitions

- A graph G=(V,E) is bipartite *if any only if* its nodes can be coloured with 2 colours (say red and green), such that every vertex has one red endpoint and one green endpoint.
- A graph G=(V,E) is bipartite *if any only if* it does not contain any cycles of odd length.
- Sometimes, these alternatives definitions are also called "characterisations".

Testing bipartiteness

Testing bipartiteness

- Given a graph $G=(V,E)$, decide if it is bipartite or not.

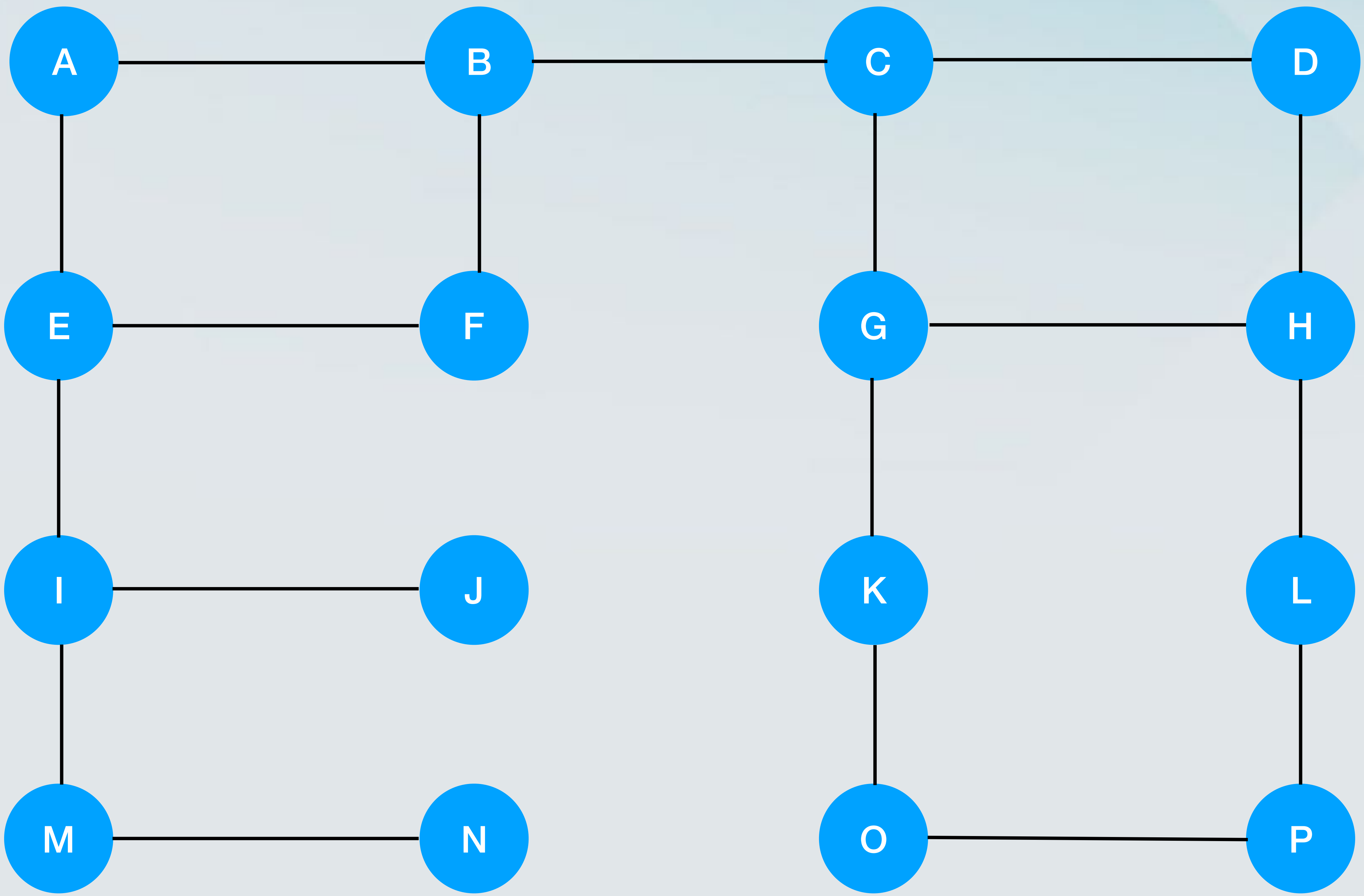
Testing bipartiteness

- Given a graph G=(V,E), decide if it is bipartite or not.
- Given a a graph G=(V,E) decide if it is 2-colourable or not.

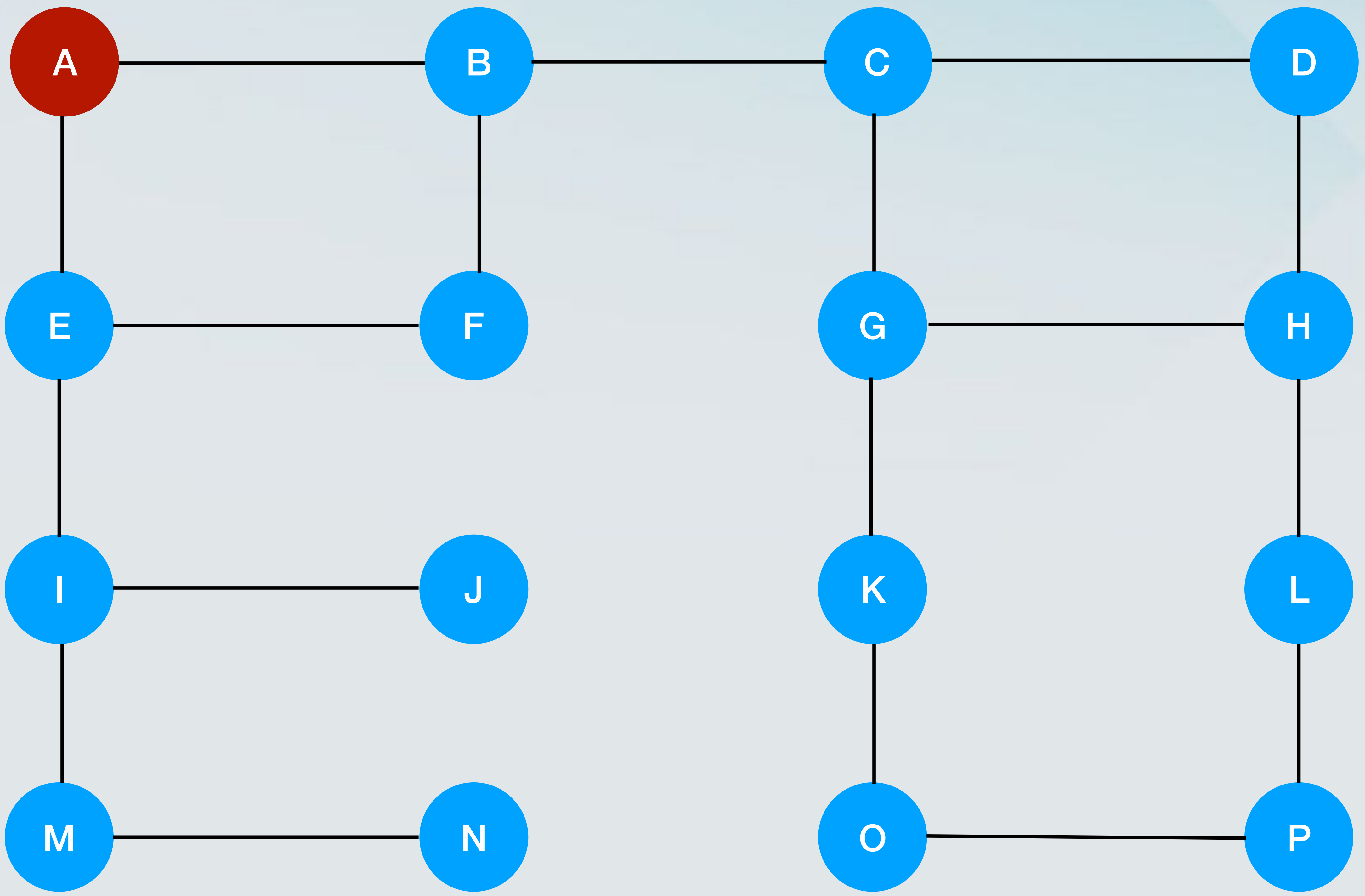
Testing bipartiteness

- Given a graph G=(V,E), decide if it is bipartite or not.
- Given a a graph G=(V,E) decide if it is 2-colourable or not.
- Given a a graph G=(V,E) decide if it is contains cycles of odd length or not.

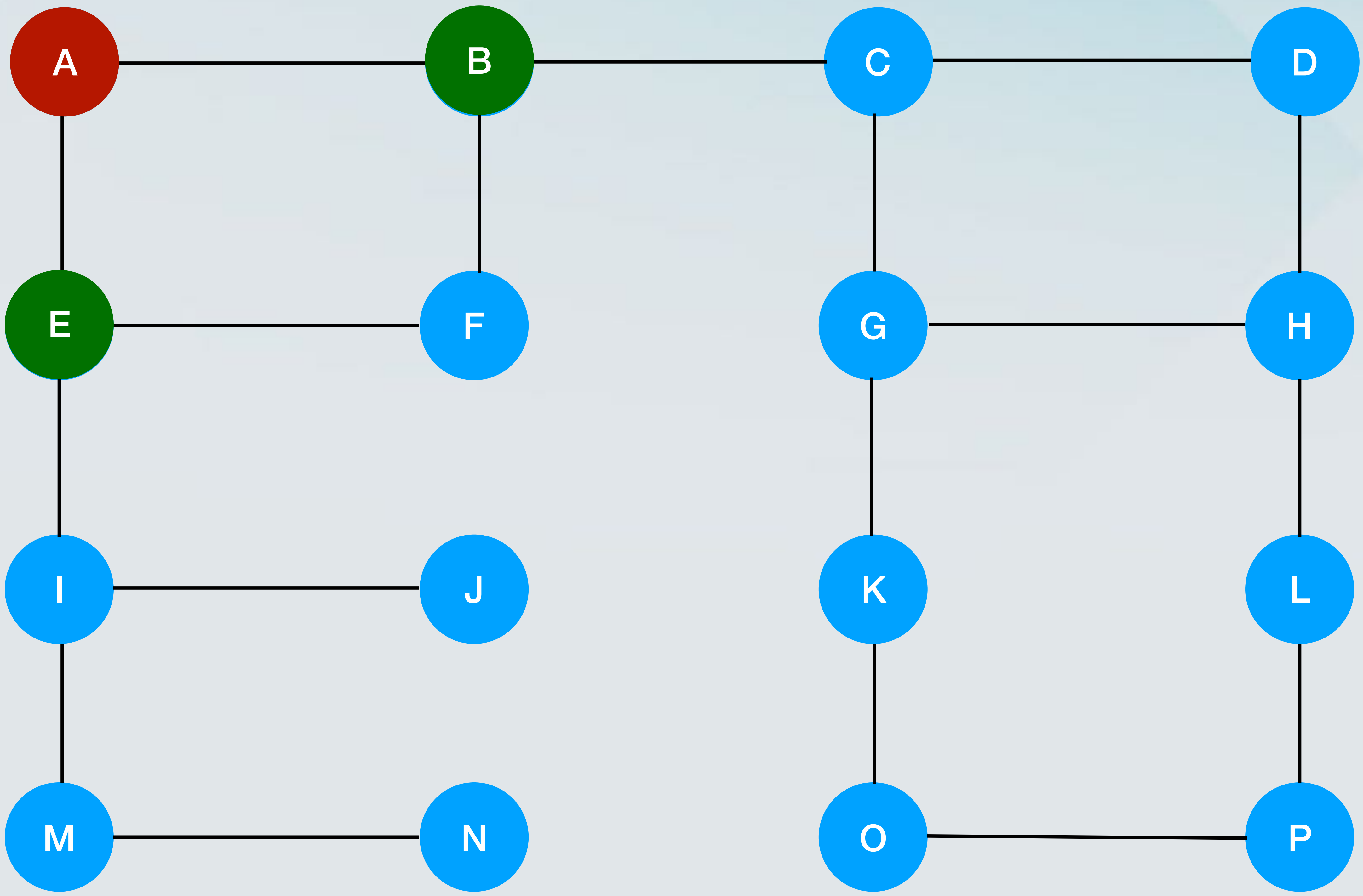
Colouring the nodes



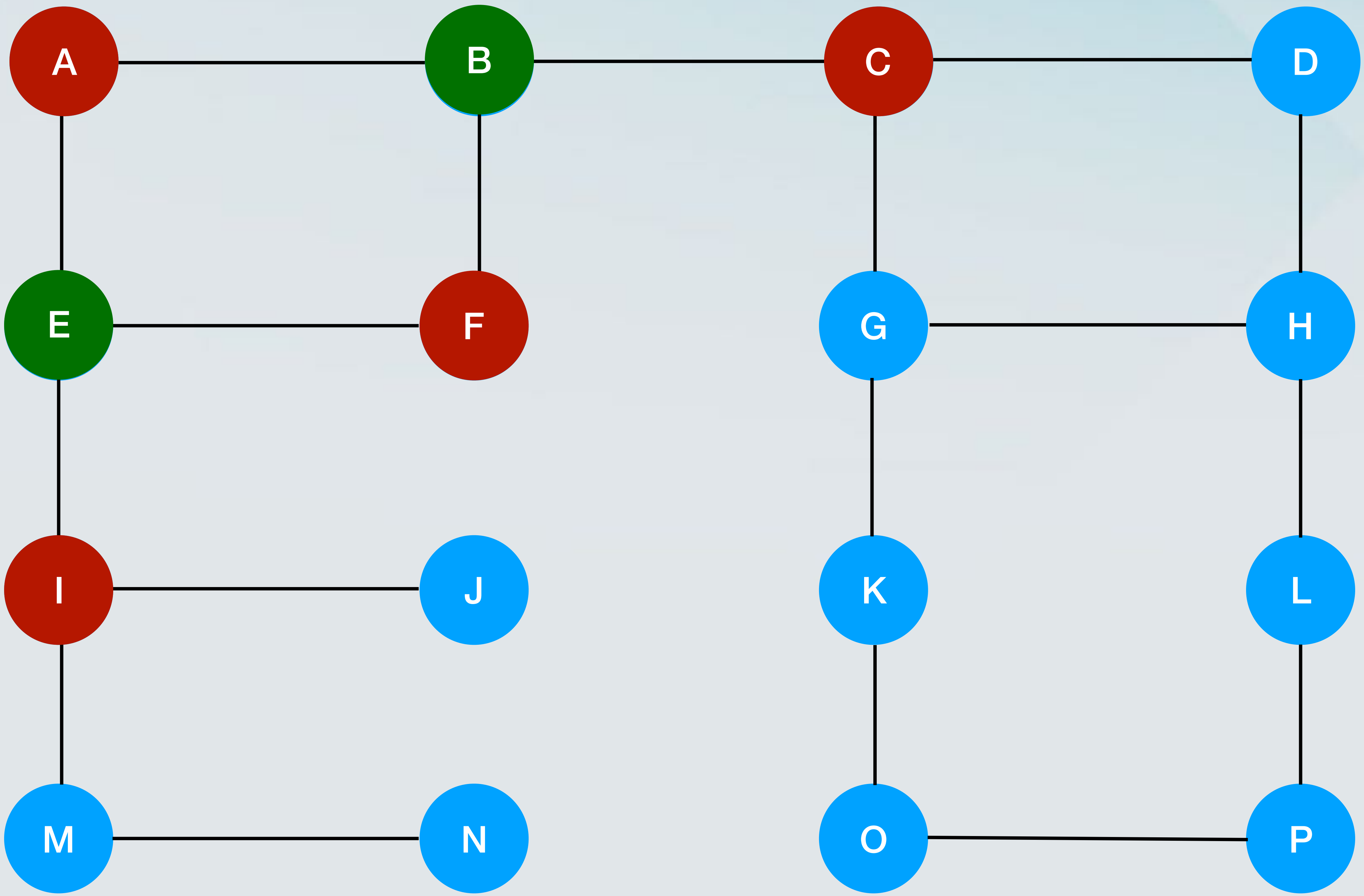
Colouring the nodes



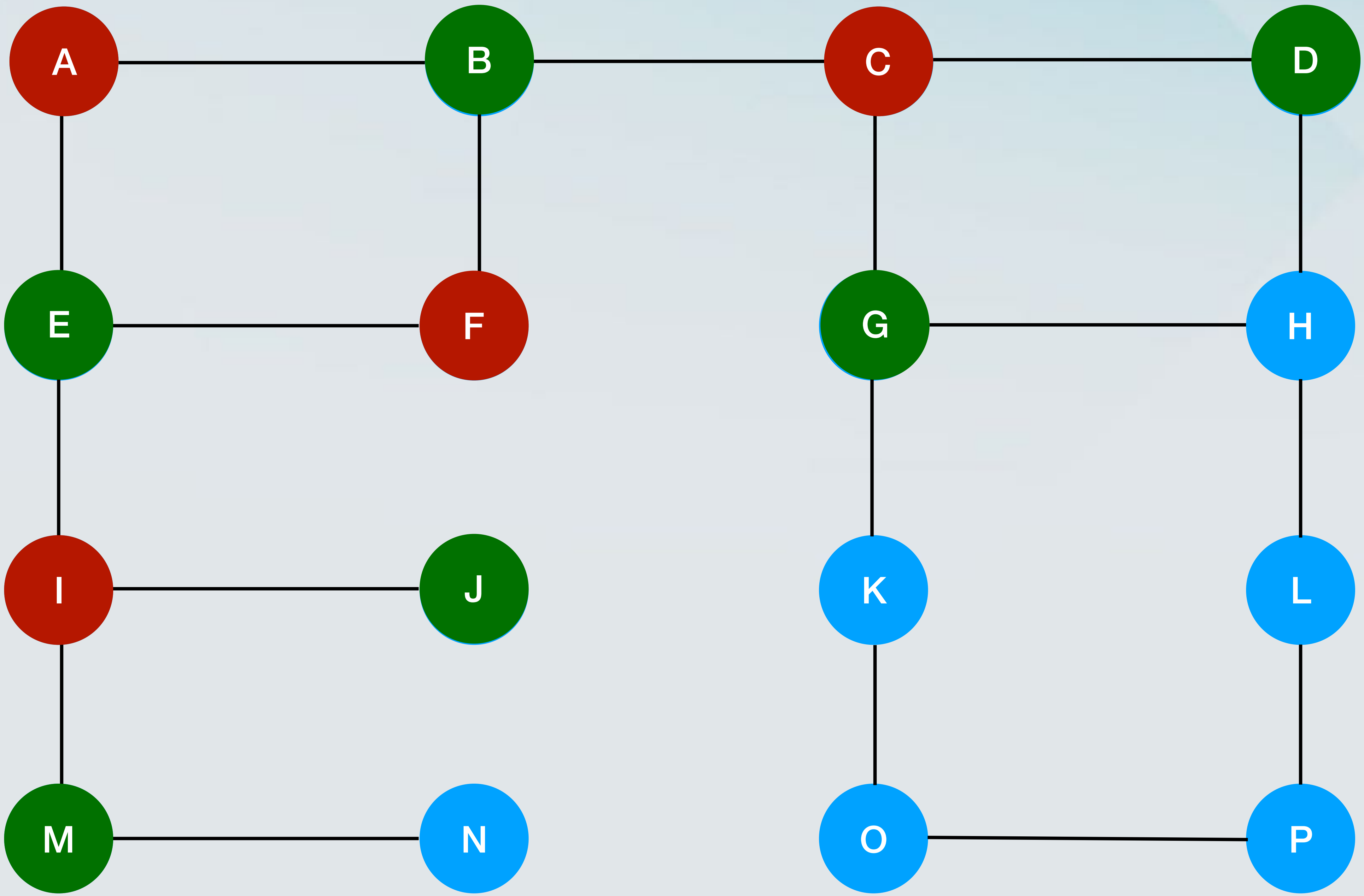
Colouring the nodes



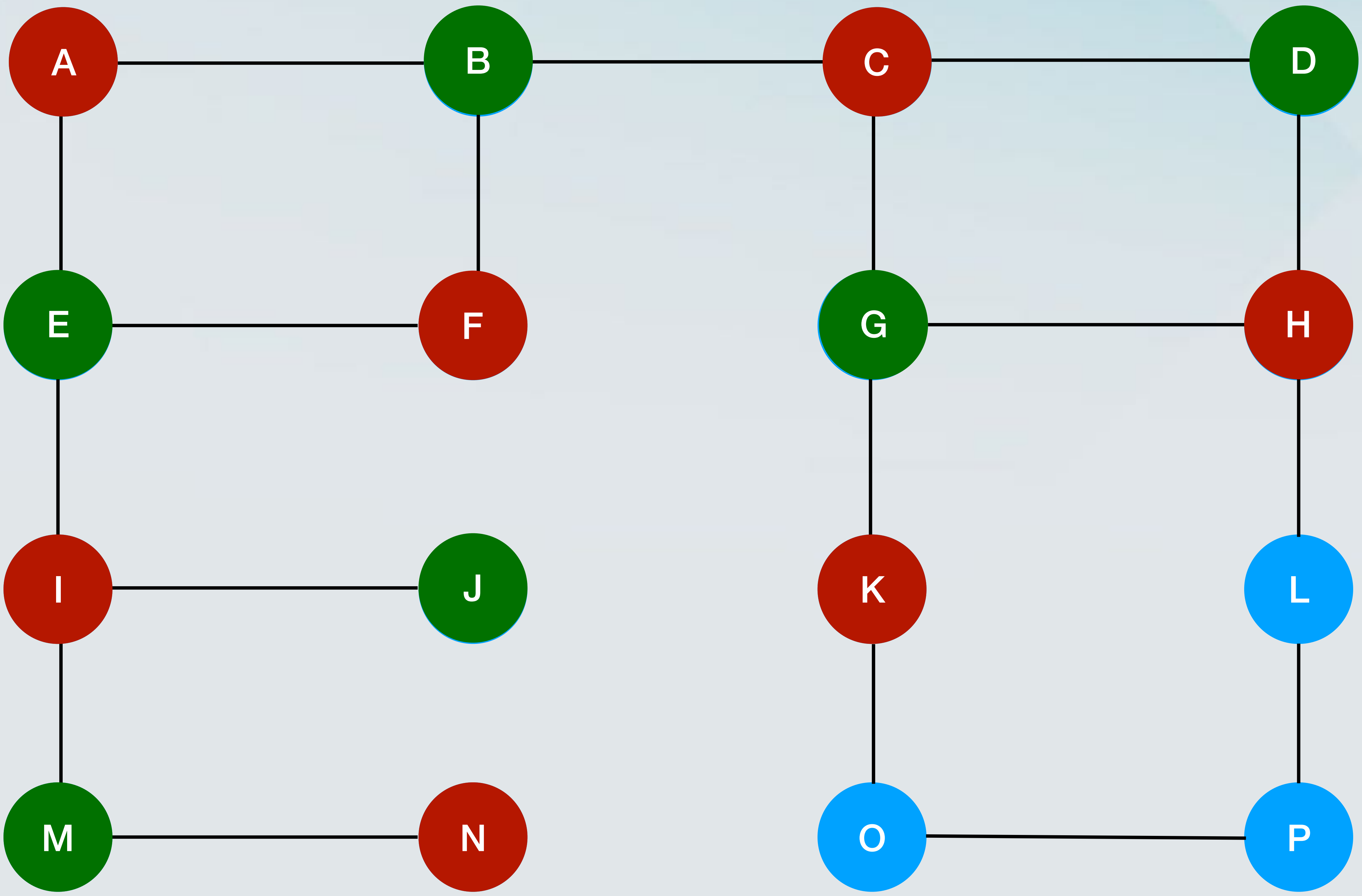
Colouring the nodes



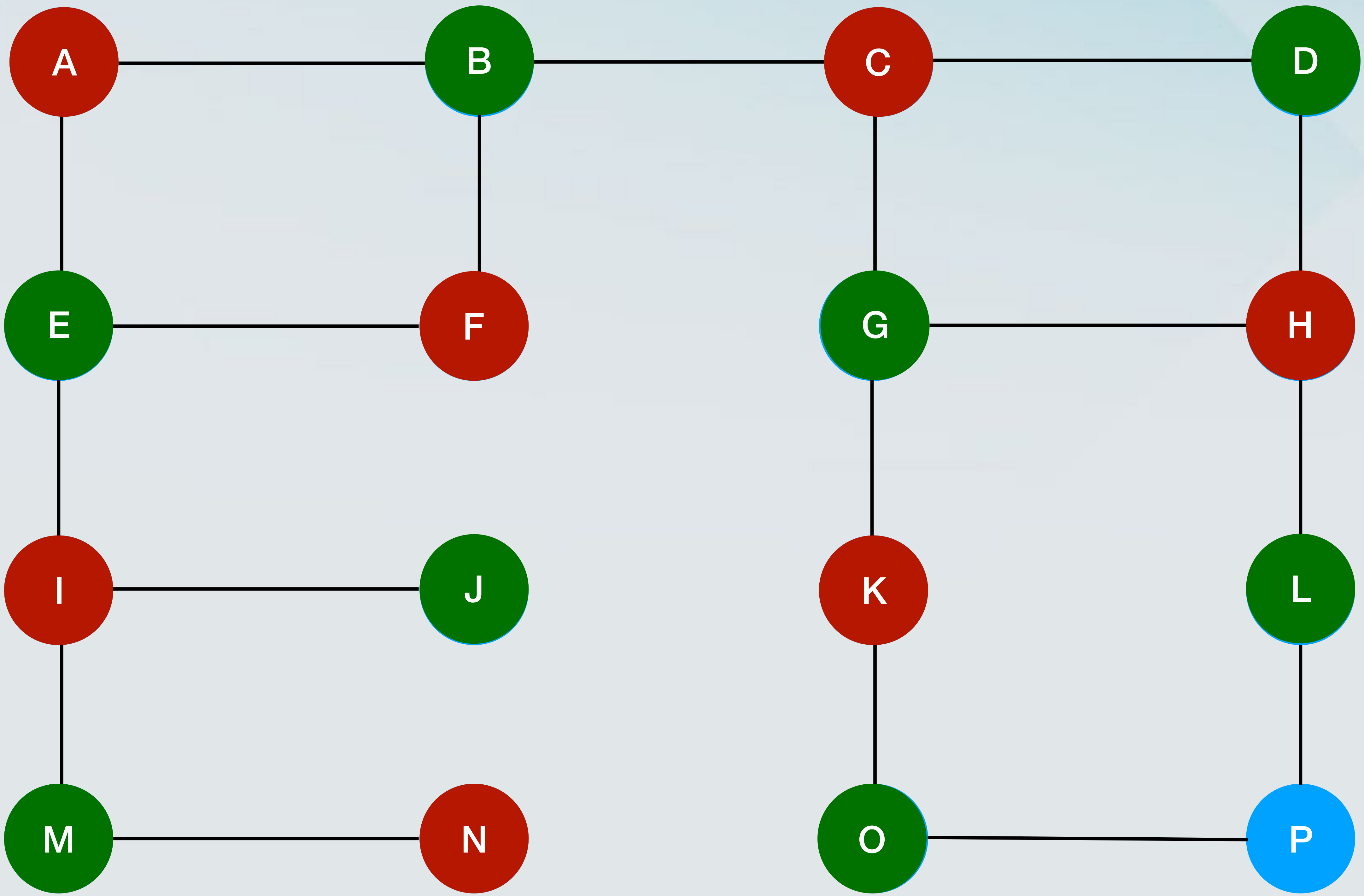
Colouring the nodes



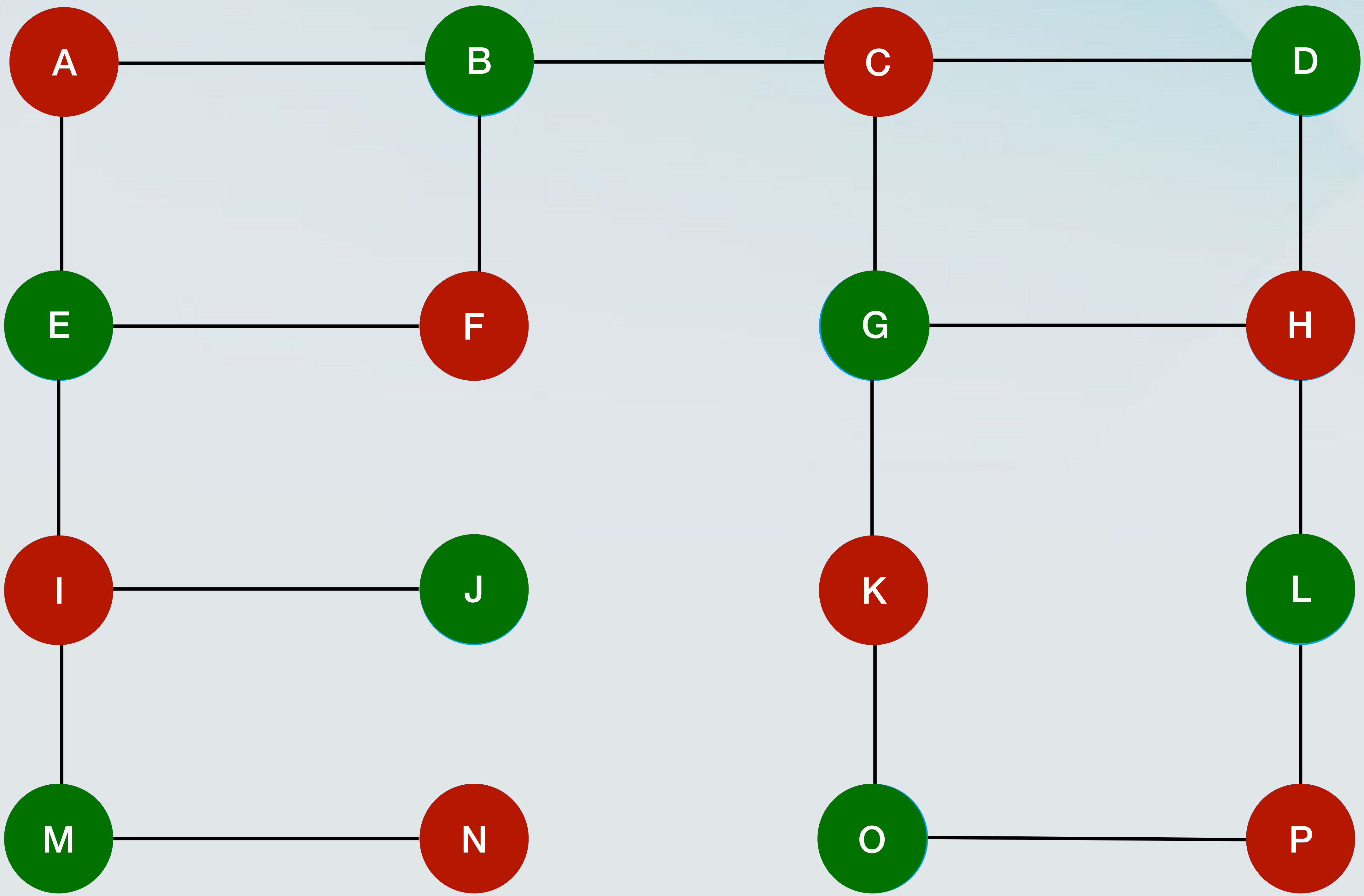
Colouring the nodes



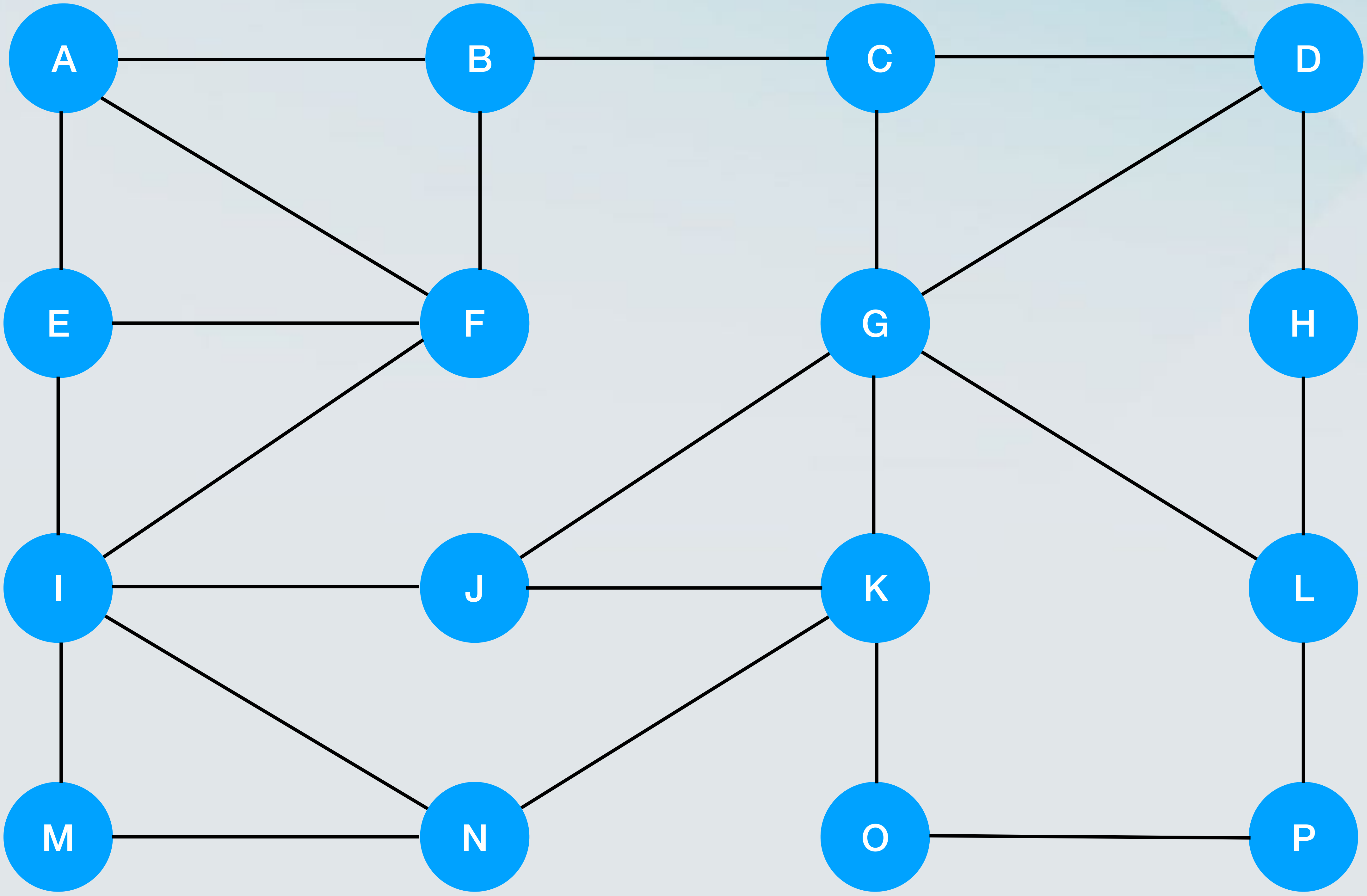
Colouring the nodes



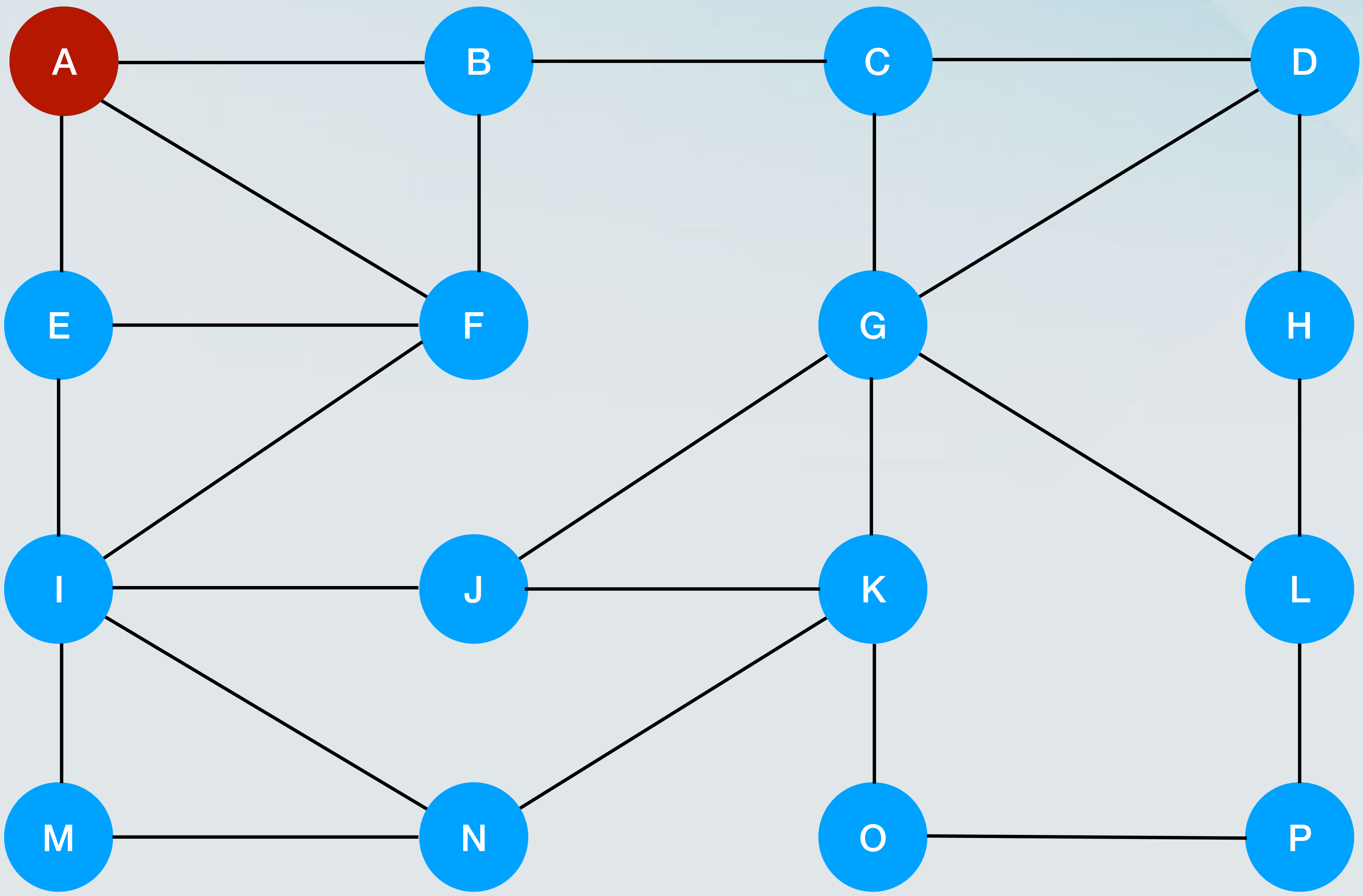
Colouring the nodes



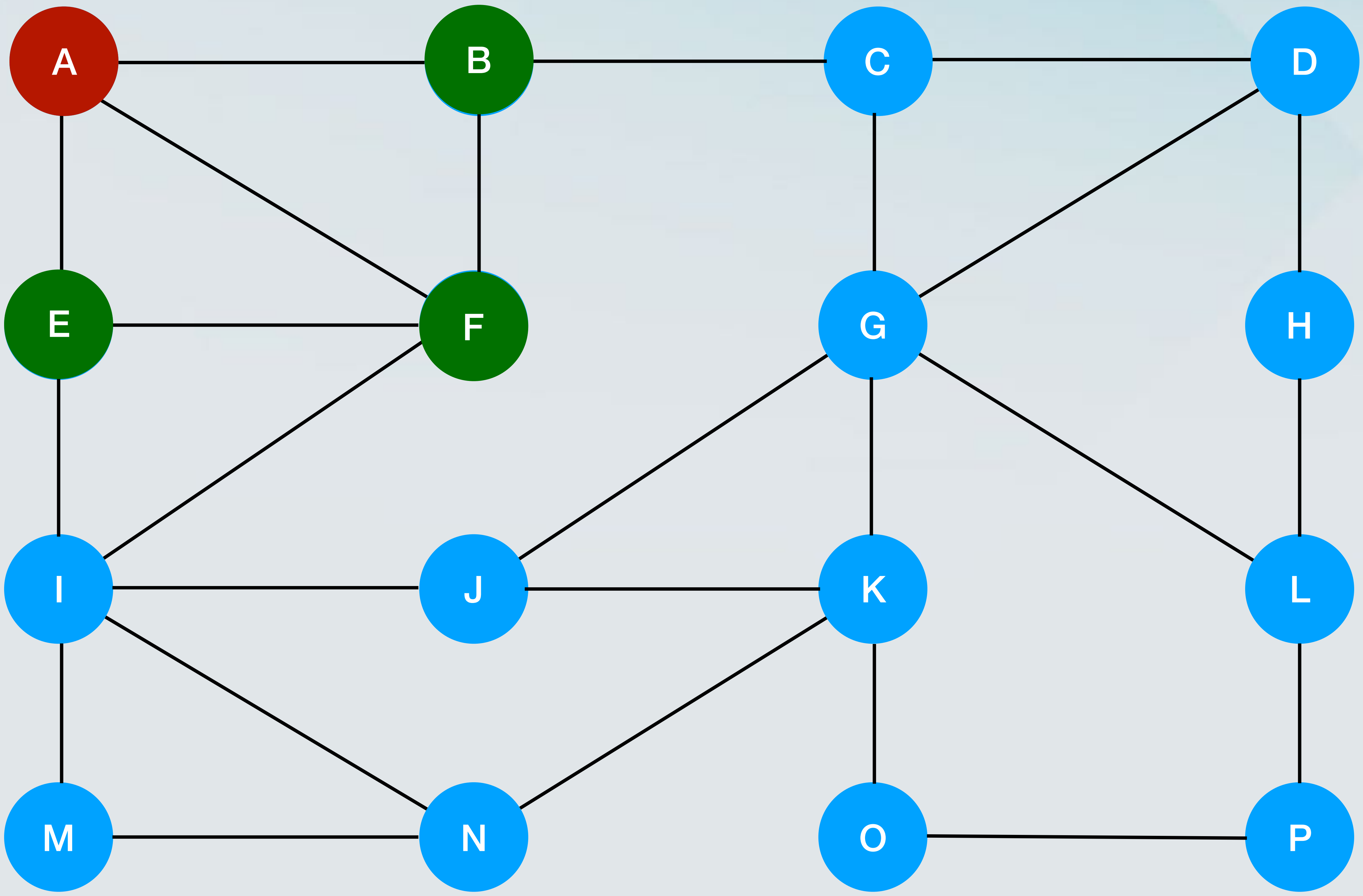
Colouring the nodes



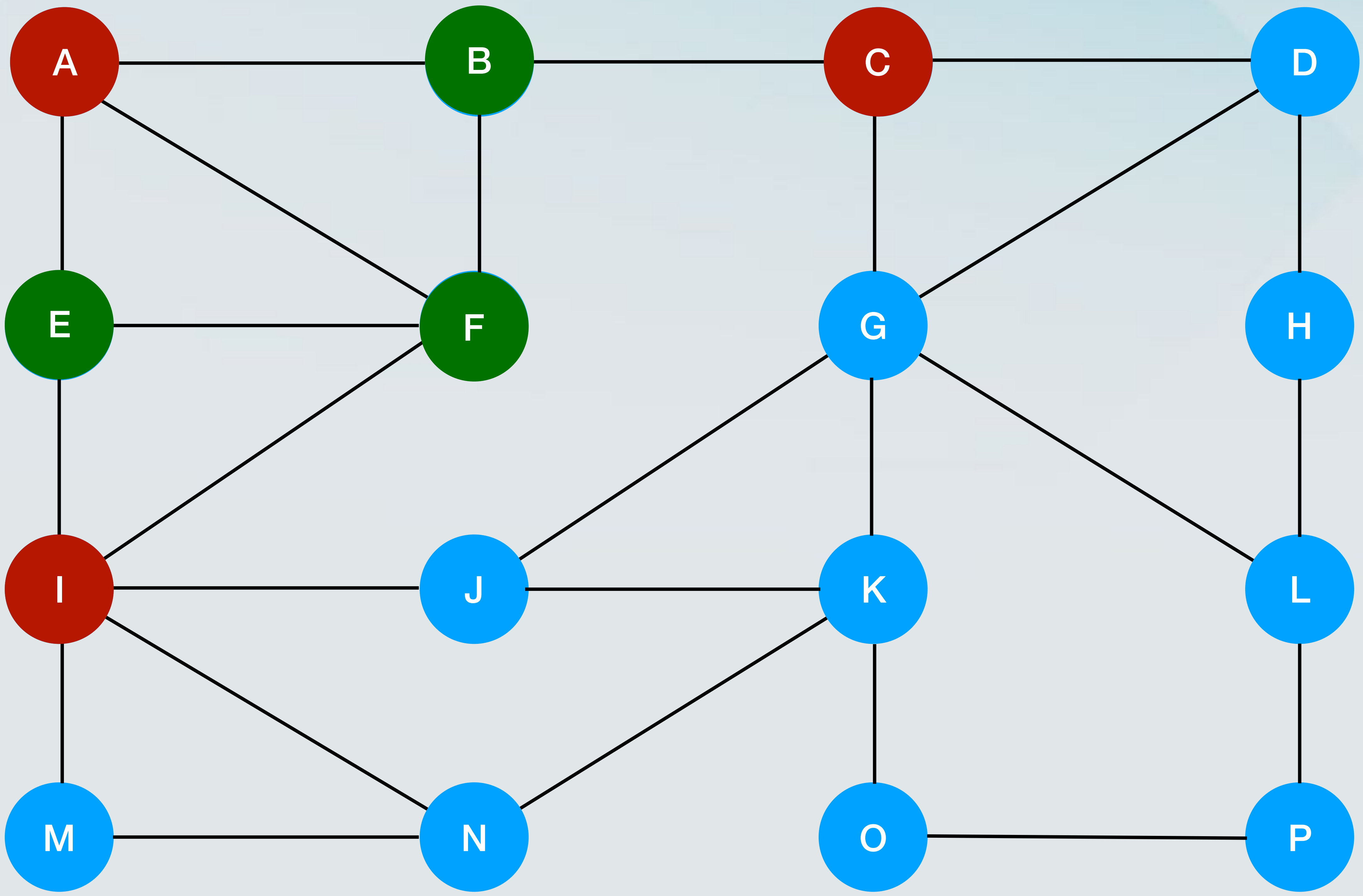
Colouring the nodes



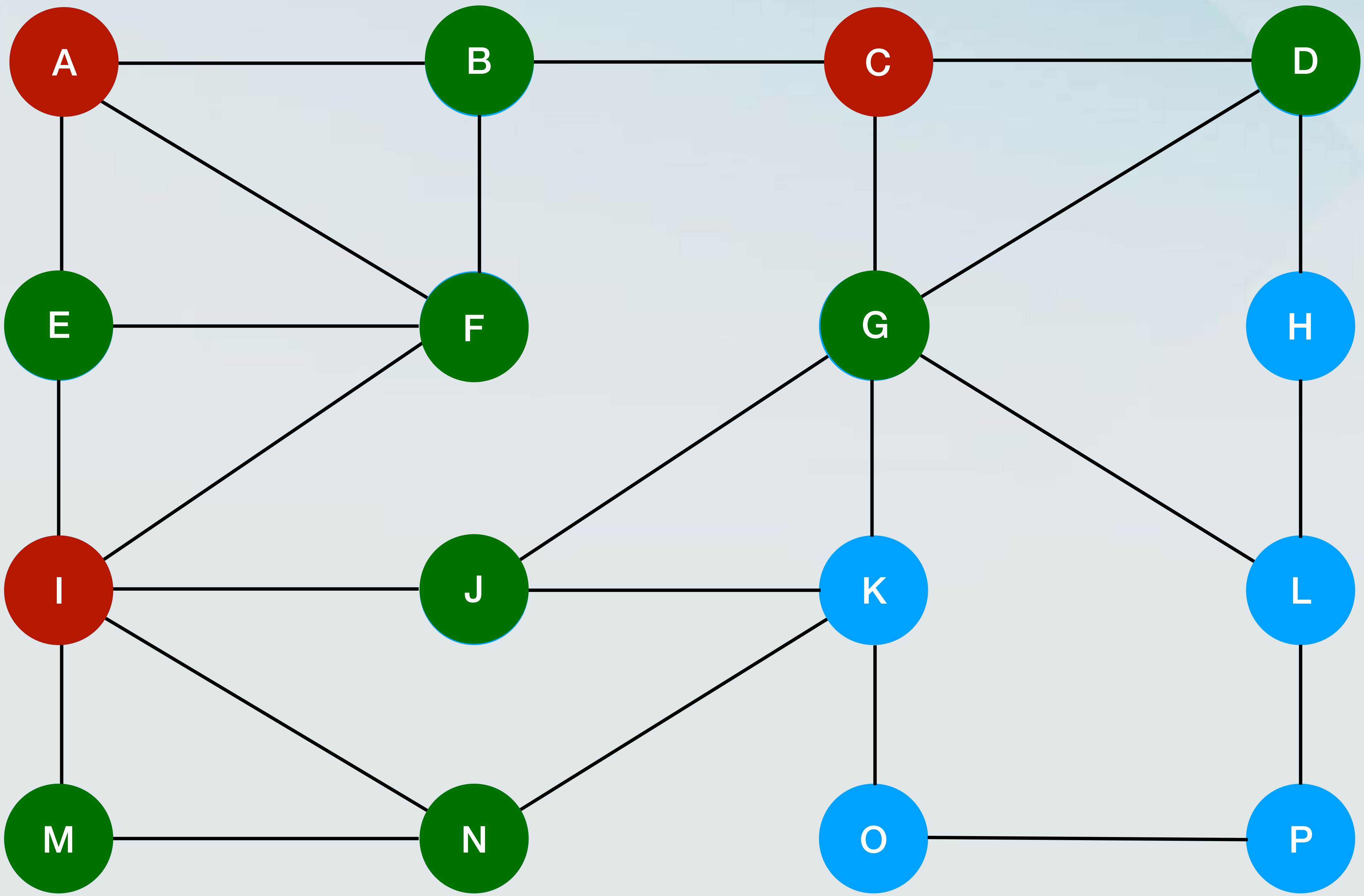
Colouring the nodes



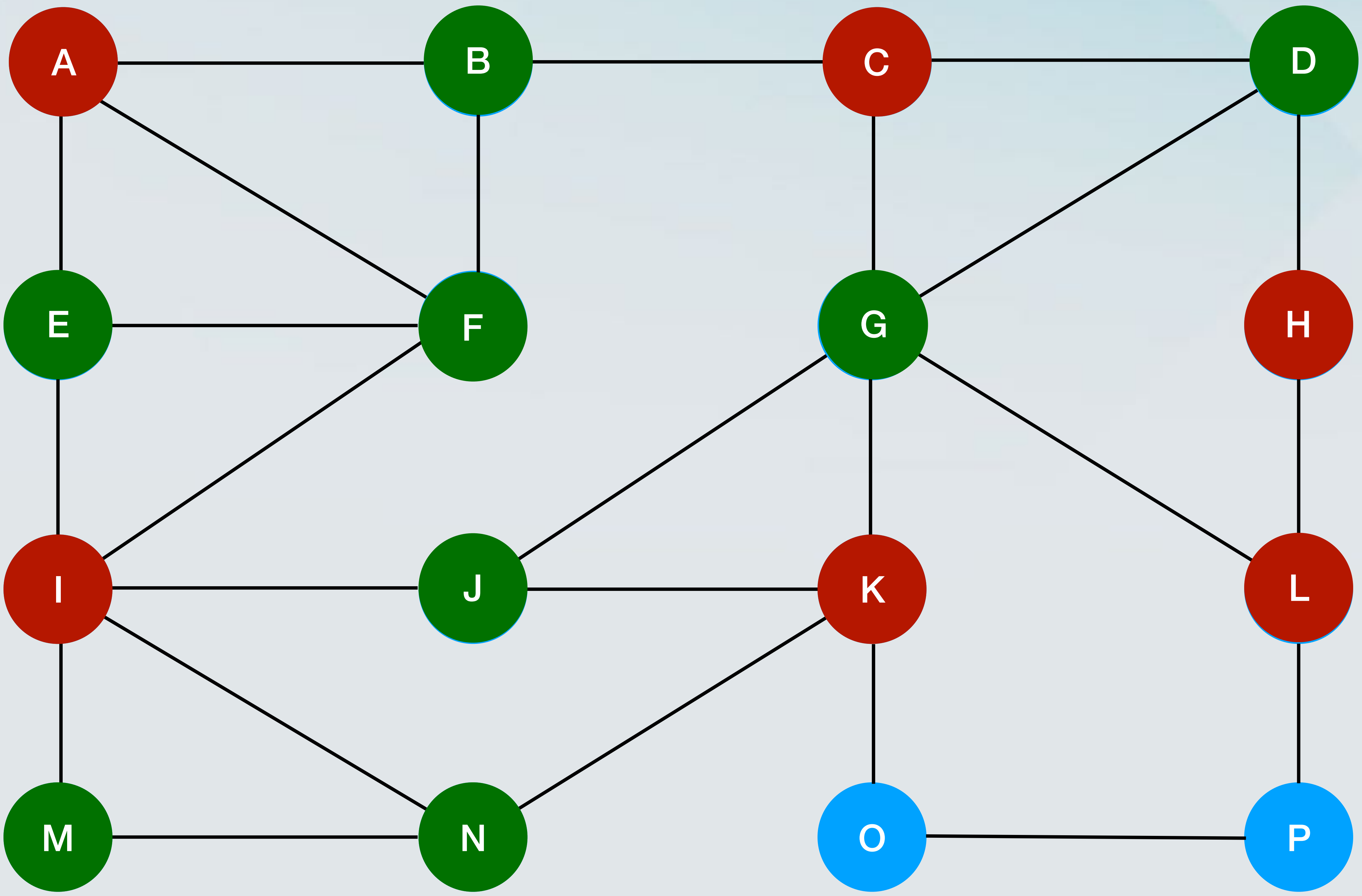
Colouring the nodes



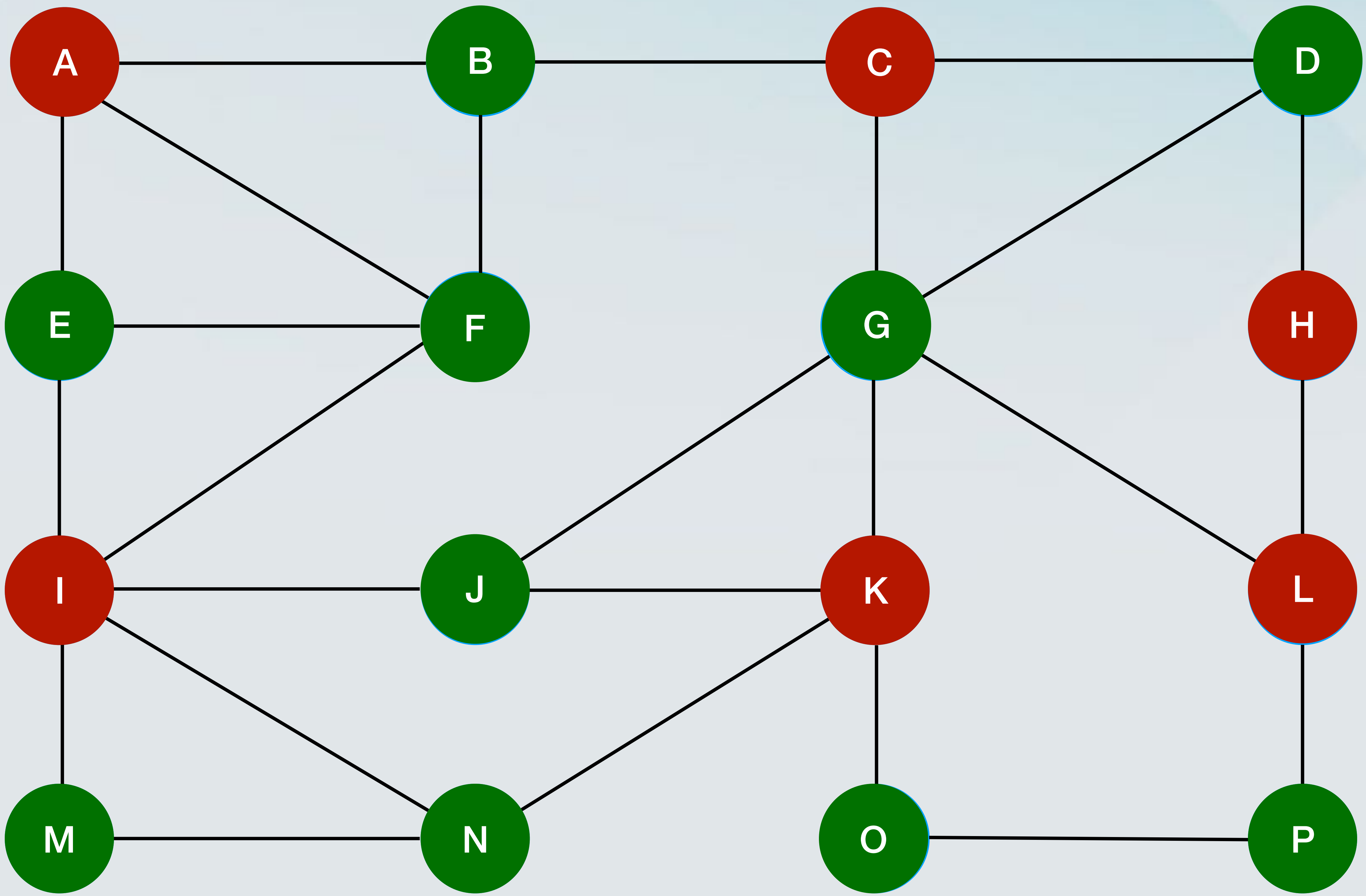
Colouring the nodes



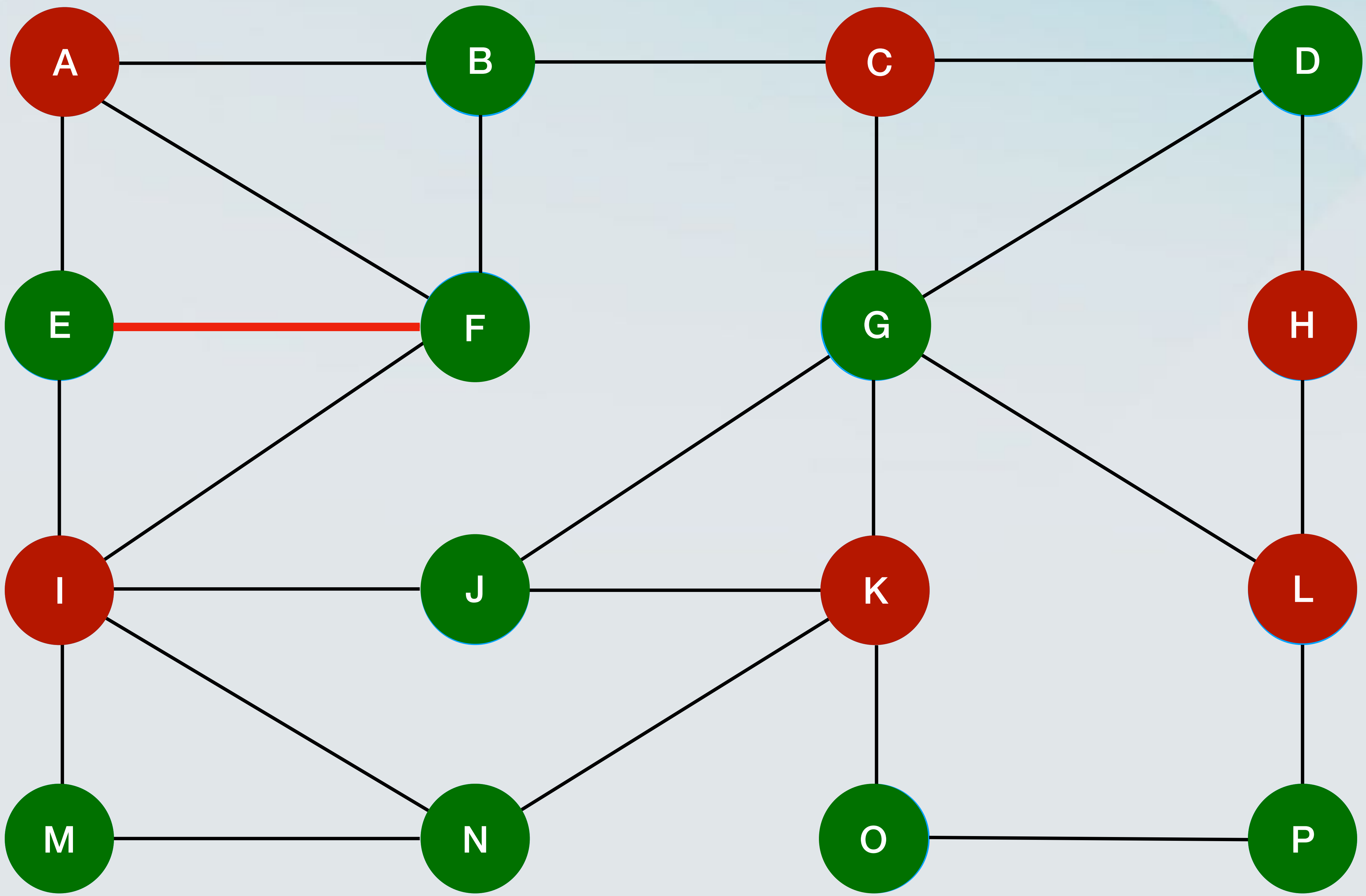
Colouring the nodes



Colouring the nodes



Colouring the nodes



Colouring the nodes

- Does this remind you of something?
  - It is essentially BFS!
  - We label the nodes of *level 1* red, the nodes of *level 2* green, and so on.
- Implementation:
  - Add a check for odd/even and assign a colour accordingly.
  - In the end, check all edges to see if they have endpoints of the same colour.

Breadth-First Search Pseudocode

```
Algorithm BFS(G,s)

Initialise empty list L0
Initialise colour list C
Insert s into L0
Set C[s] = red

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
                If i+1 is odd, set C[w] = red, else set C[w] = green
             else
                label e as cross edge
i = i+1

For all edges e=(u,v) in G
     if C[u] = C[v] return "not bipartite"
Return "bipartite"
```

Running time

Running time

- What did we add?

Running time

- What did we add?
    - A colour assignment for the starting node.

Running time

- What did we add?
  - A colour assignment for the starting node.
  - An odd/even check and a colour assignment for each node in the loop.

Running time

- What did we add?
    - A colour assignment for the starting node.
    - An odd/even check and a colour assignment for each node in the loop.
    - An extra loop for checking the edges of their graph for the colours of their endpoints.

Running time

- What did we add?
  - A colour assignment for the starting node.
  - An odd/even check and a colour assignment for each node in the loop.
  - An extra loop for checking the edges of their graph for the colours of their endpoints.
- How much more do we “pay” (asymptotically)?

Running time

- What did we add?
  - A colour assignment for the starting node.
  - An odd/even check and a colour assignment for each node in the loop.
  - An extra loop for checking the edges of their graph for the colours of their endpoints.
- How much more do we “pay” (asymptotically)?
  - Nothing!

Running time

- What did we add?
  - A colour assignment for the starting node.
  - An odd/even check and a colour assignment for each node in the loop.
  - An extra loop for checking the edges of their graph for the colours of their endpoints.
- How much more do we “pay” (asymptotically)?
  - Nothing!
- Running time **O(m+n)**.

Correctness

- We started at an arbitrary node s.
- Maybe we were lucky / unlucky?

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.

Properties of BFS

- If **e**=(**u**,**v**) is a *cross edge*, then the **u** and **v** differ by at most one level.

- If **e**=(**u**,**v**) is a *discovery edge*, then the **u** and **v** differ by at most one level.

Correctness

Correctness

- Suppose that G is **bipartite**. Then, all cycles must be of even length.

Correctness

- Suppose that G is **bipartite**. Then, all cycles must be of even length.
- Suppose *to the contrary* that the algorithm returns “*not bipartite*”.

Correctness

- Suppose that G is **bipartite**. Then, all cycles must be of even length.

- Suppose *to the contrary* that the algorithm returns “*not bipartite*”.

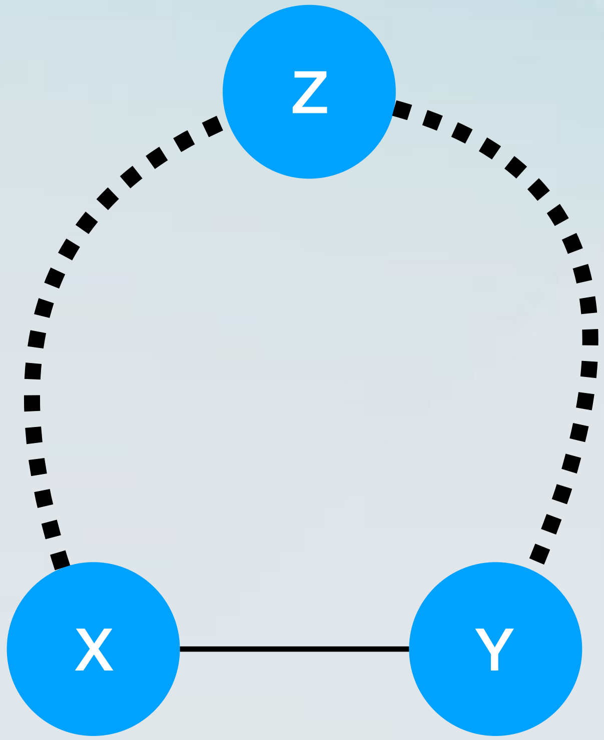
  - This means that it has found an edge e=(x,y) with endpoints of the same colour.

Correctness

- Suppose that G is **bipartite**. Then, all cycles must be of even length.
- Suppose *to the contrary* that the algorithm returns "*not bipartite*".
  - This means that it has found an edge e=(x,y) with endpoints of the same colour.
  - Since the endpoints of any edge can not differ by more than one layer and layers have alternating colours, x and y must be in the same layer.

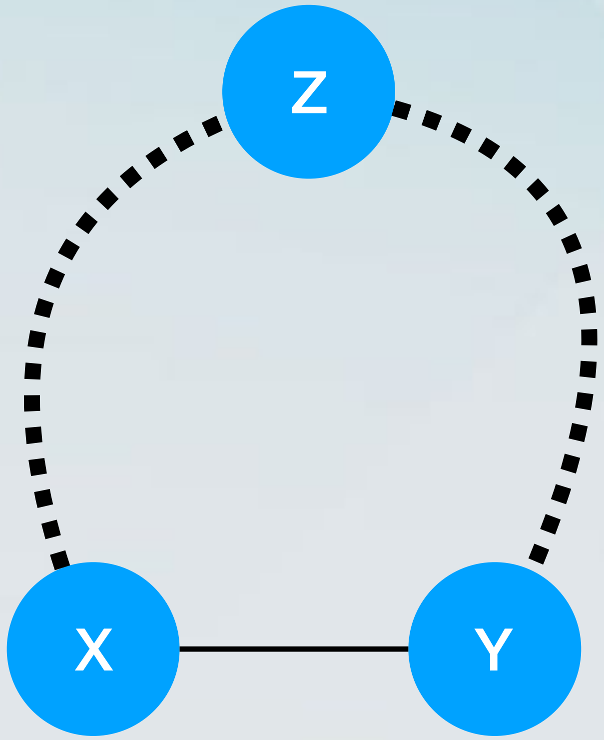
Correctness

- Consider the lowest common ancestor z of x and y in the BFS tree.



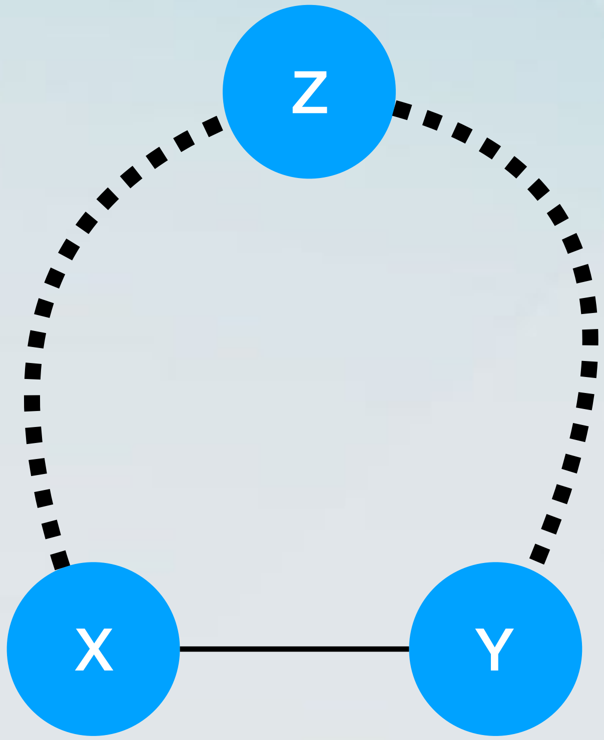
Correctness

- Consider the lowest common ancestor z of x and y in the BFS tree.
- Let L_i be the level of z and let L_j be the level of x and y



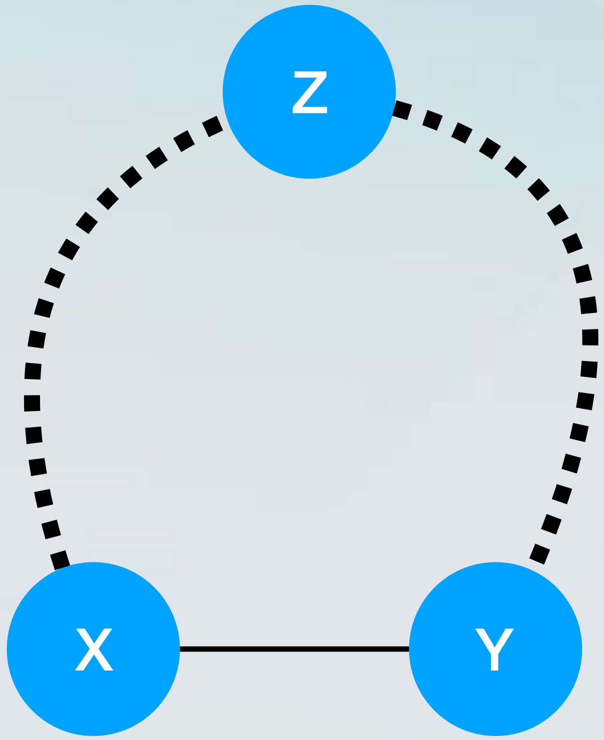
Correctness

- Consider the lowest common ancestor z of x and y in the BFS tree.
- Let L_i be the level of z and let L_j be the level of x and y
- Consider the cycle (z … x), (x,y), (y … z).



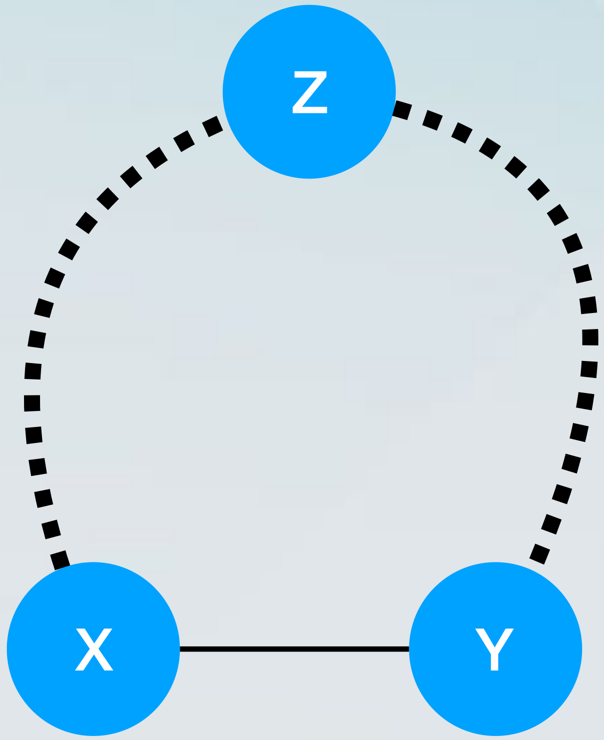
Correctness

- Consider the lowest common ancestor z of x and y in the BFS tree.
- Let $\mathbf{L_i}$ be the level of z and let $\mathbf{L_j}$ be the level of x and y
- Consider the cycle (z … x), (x,y), (y … z).
- Length: (j-i) + 1 + (j-i) (odd)



Correctness

- Consider the lowest common ancestor z of x and y in the BFS tree.
- Let L_i be the level of z and let L_j be the level of x and y
- Consider the cycle (z … x), (x,y), (y … z).
- Length: (j-i) + 1 + (j-i) (odd)
- **Contradiction!**



Correctness

Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.

Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns "*bipartite*".

Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns "*bipartite*".
    - This means that it has not found any edge e=(x,y) with endpoints of the same colour.

Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns “*bipartite*”.
  - This means that it has not found any edge e=(x,y) with endpoints of the same colour.
  - This also obviously means that there is no edge with endpoints in the same layer.

Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns "*bipartite*".
    - This means that it has not found any edge $e=(x,y)$ with endpoints of the same colour.
    - This also obviously means that there is no edge with endpoints in the same layer.
    - By the earlier discussion, all edges must have endpoints that lie in consecutive layers.

Correctness

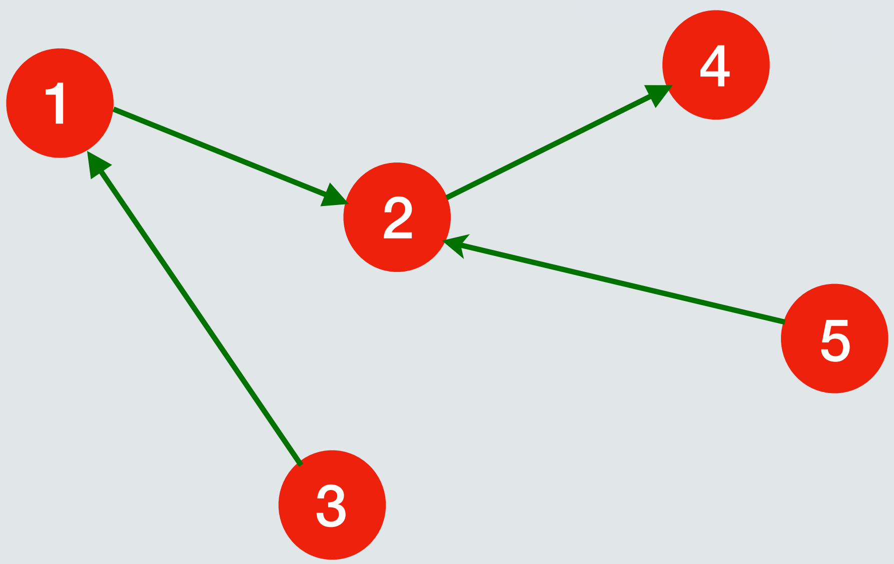
- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns "*bipartite*".
  - This means that it has not found any edge e=(x,y) with endpoints of the same colour.
  - This also obviously means that there is no edge with endpoints in the same layer.
  - By the earlier discussion, all edges must have endpoints that lie in consecutive layers.
  - Take any cycle (z, … , z). Since for every edge in this cycle there is a change of layer (from *j* to *j+1* or from *j+1* to *j*), the cycle must have even length.

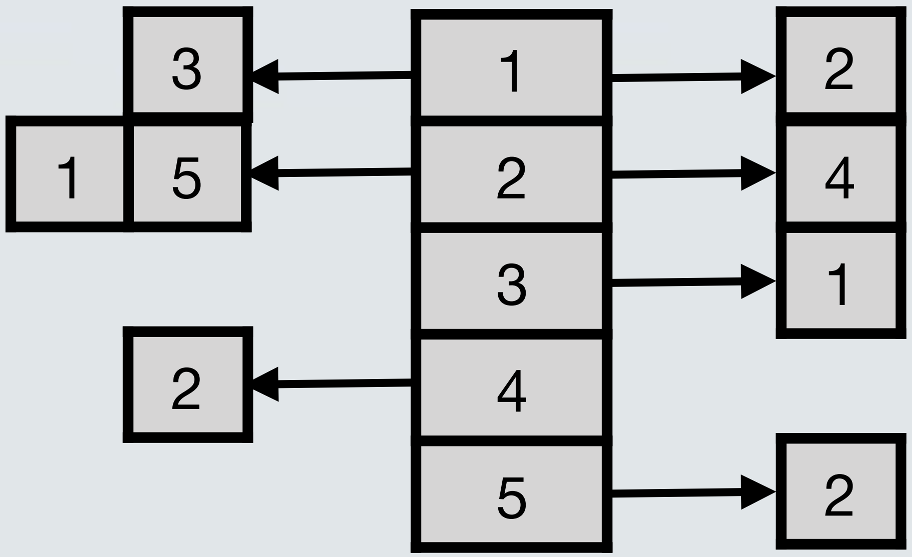
Correctness

- Suppose that G is **not bipartite**. Then, it must contain a cycle of odd length.
- Suppose *to the contrary* that the algorithm returns "*bipartite*".
  - This means that it has not found any edge $e=(x,y)$ with endpoints of the same colour.
  - This also obviously means that there is no edge with endpoints in the same layer.
  - By the earlier discussion, all edges must have endpoints that lie in consecutive layers.
  - Take any cycle $(z, \ldots, z)$. Since for every edge in this cycle there is a change of layer (from j to $j+1$ or from $j+1$ to j), the cycle must have even length.
  - **Contradiction!**

Directed graphs

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

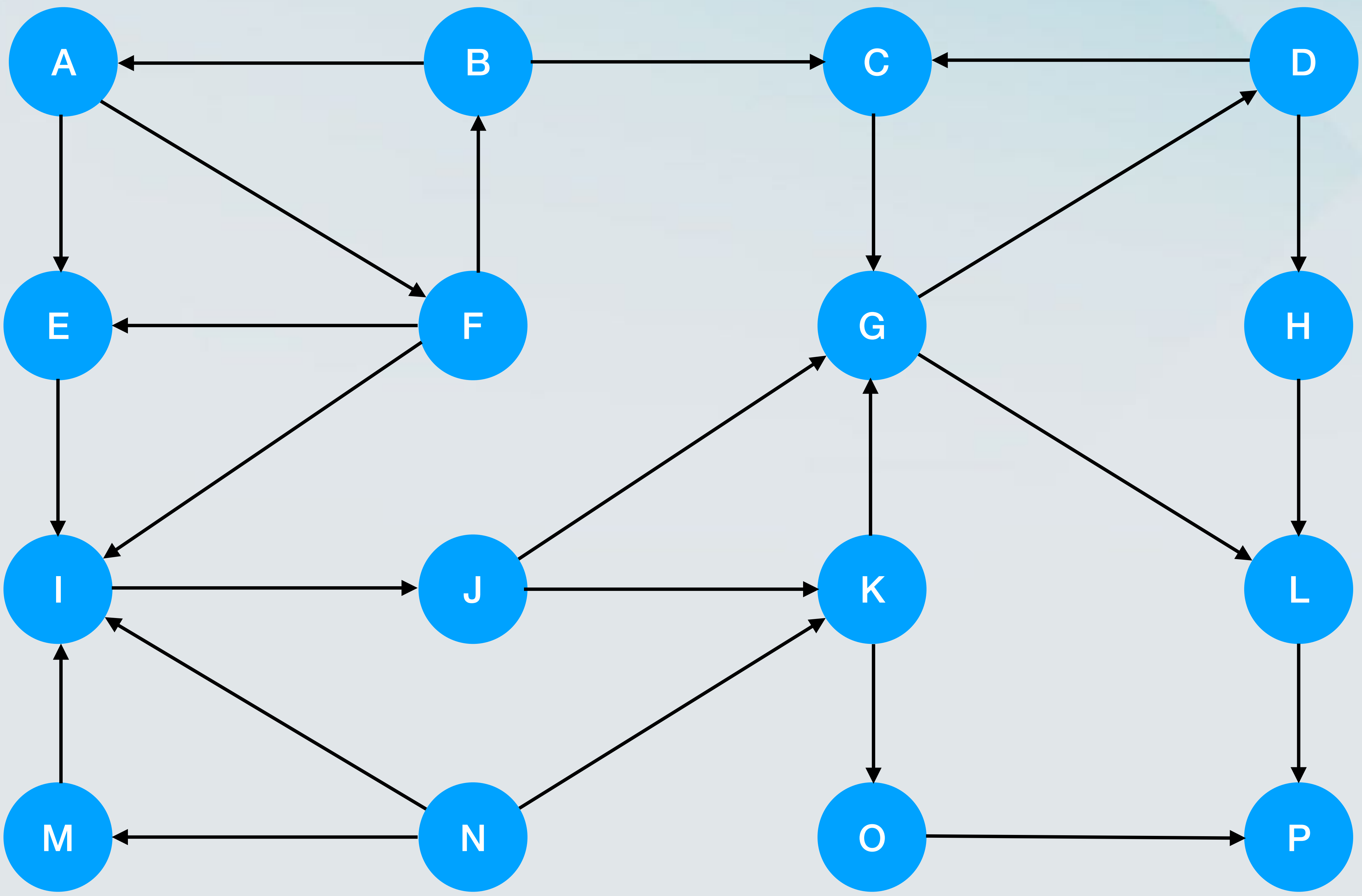




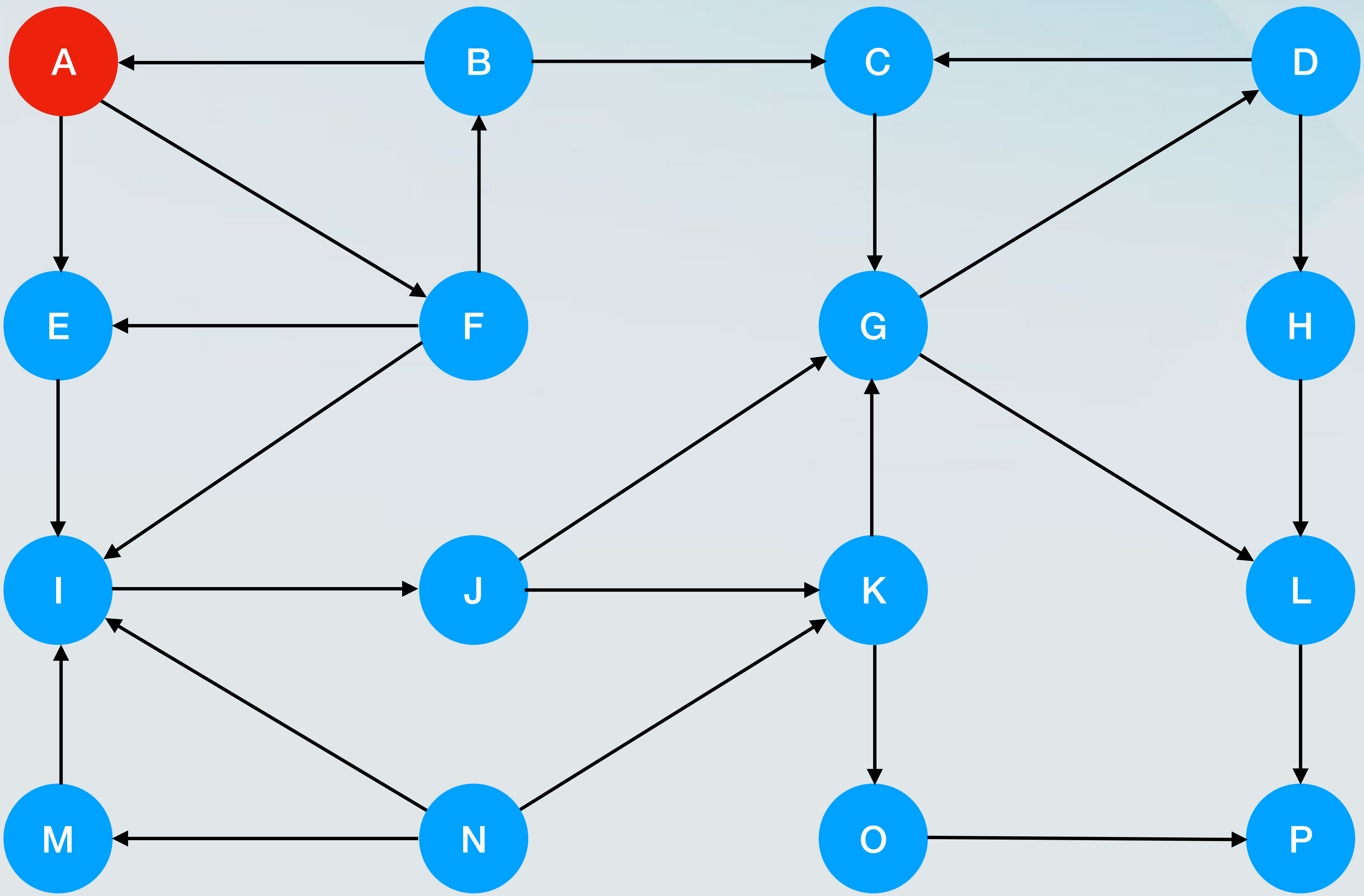
DFS and BFS on directed graphs

- Very similar to their version on undirected graphs.
- When we are at a node and we examine its neighbours, a neighbour is now only a node that we can reach with a directed edge.
- The running time is still **$O(n+m)$**.

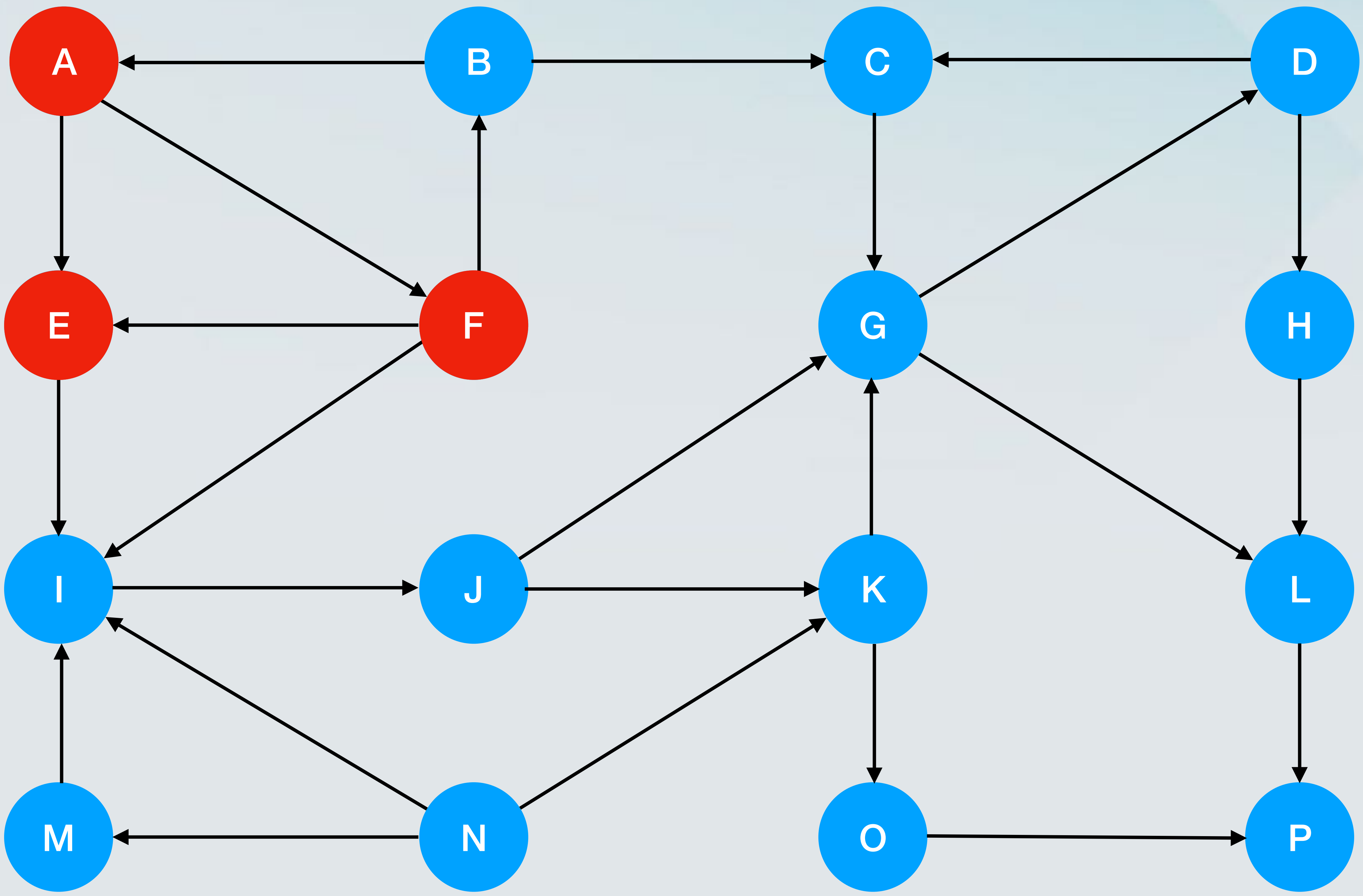
Breadth-First Search



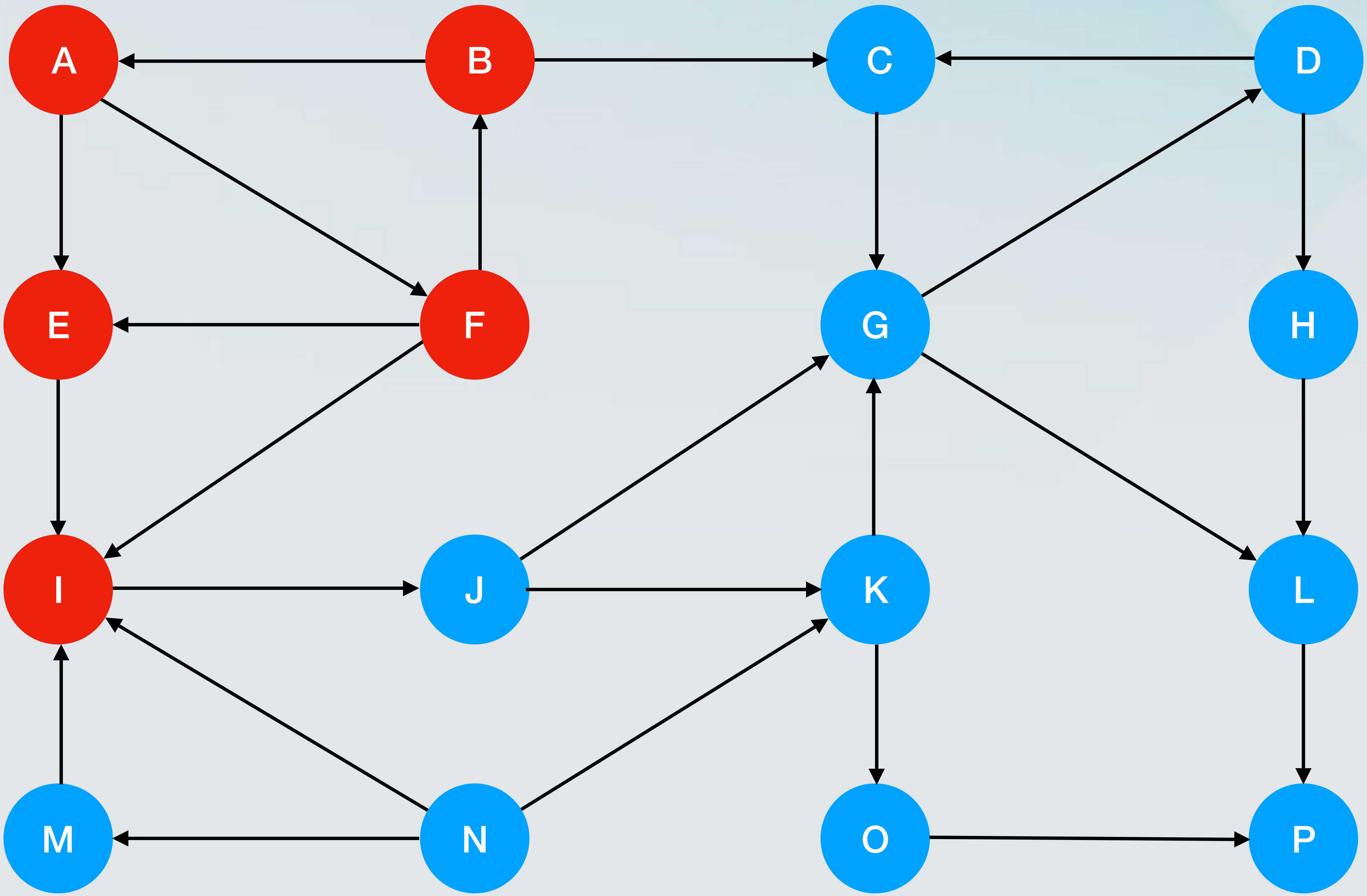
Breadth-First Search



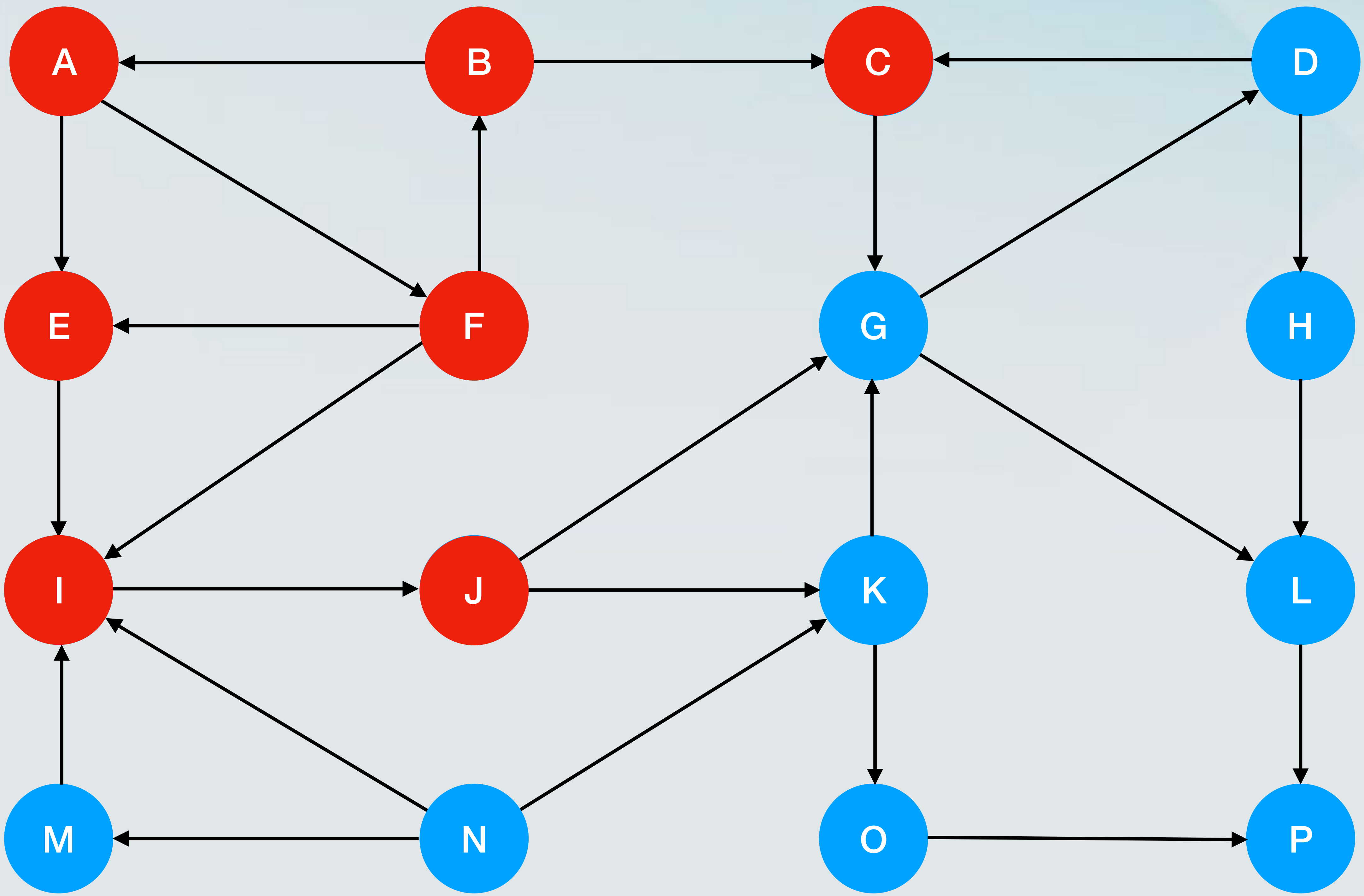
Breadth-First Search



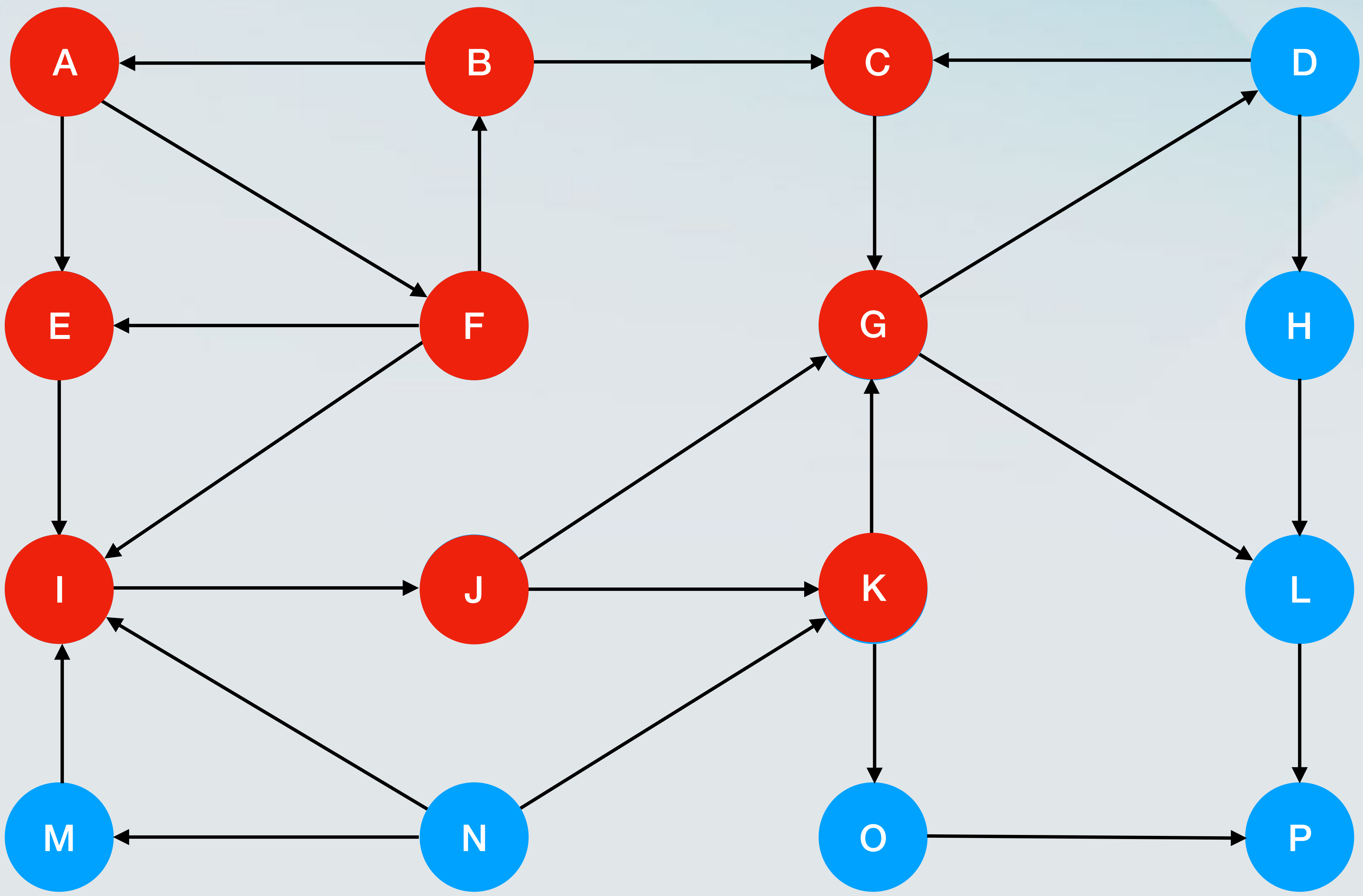
Breadth-First Search



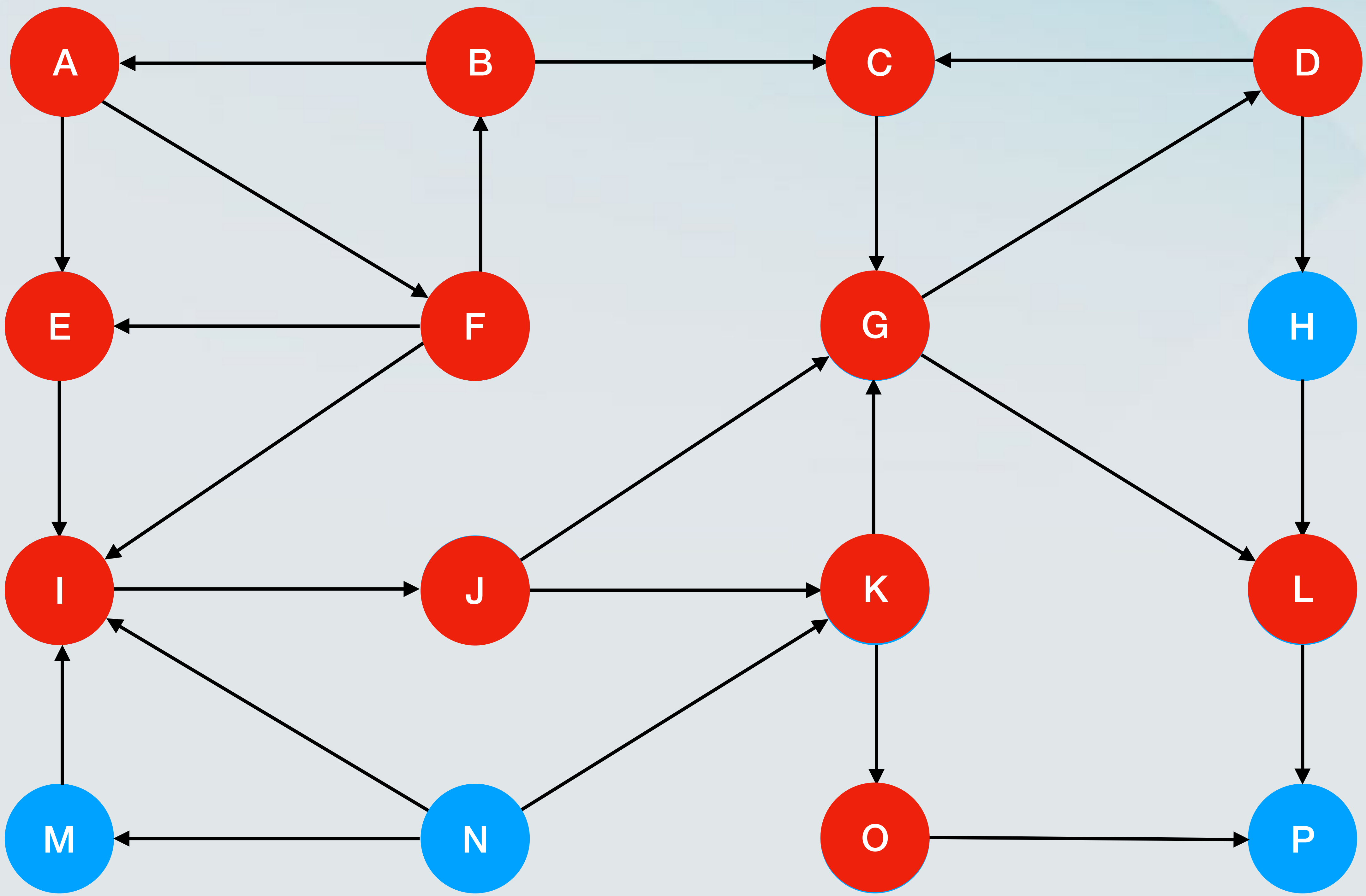
Breadth-First Search



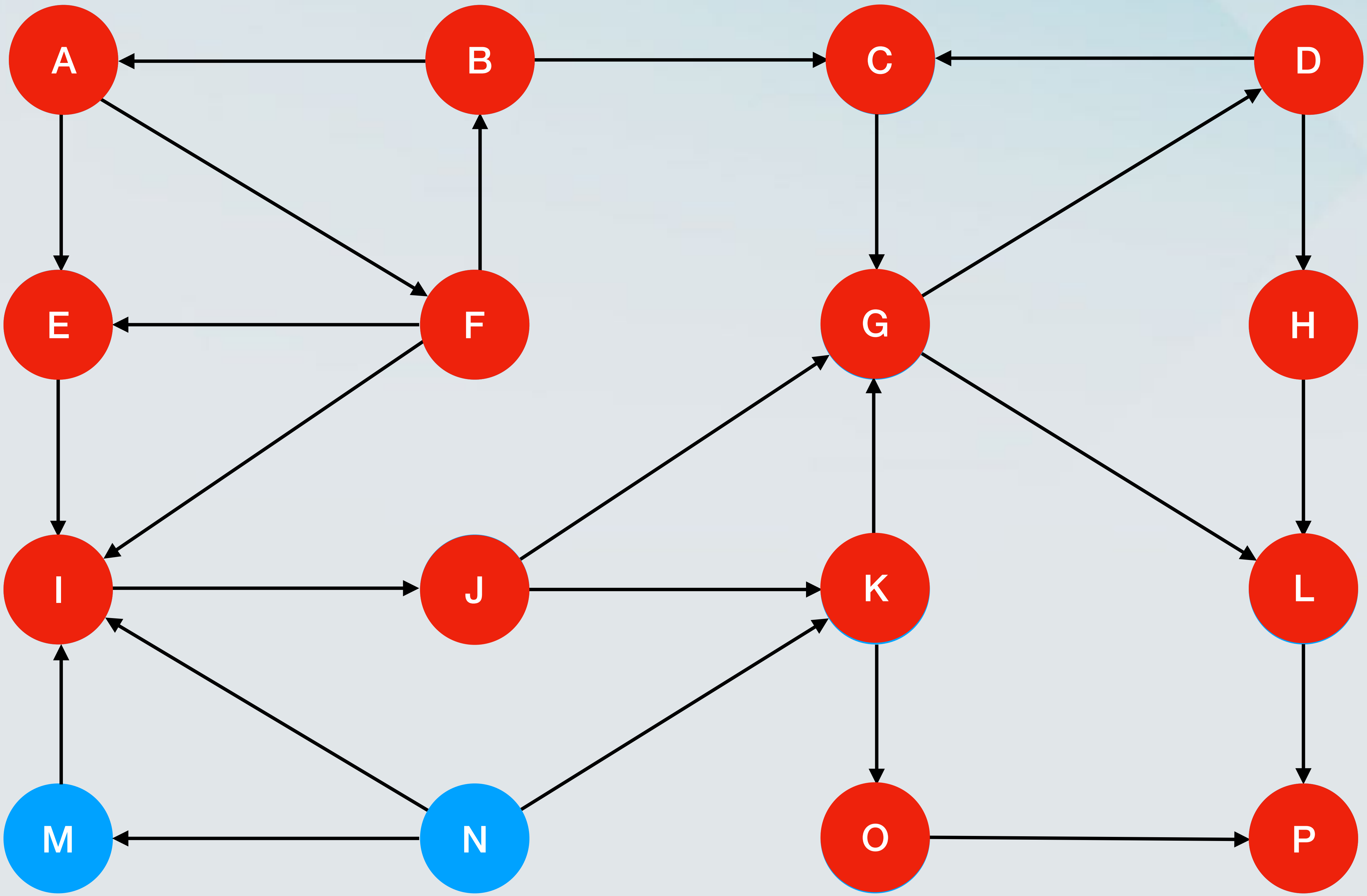
Breadth-First Search



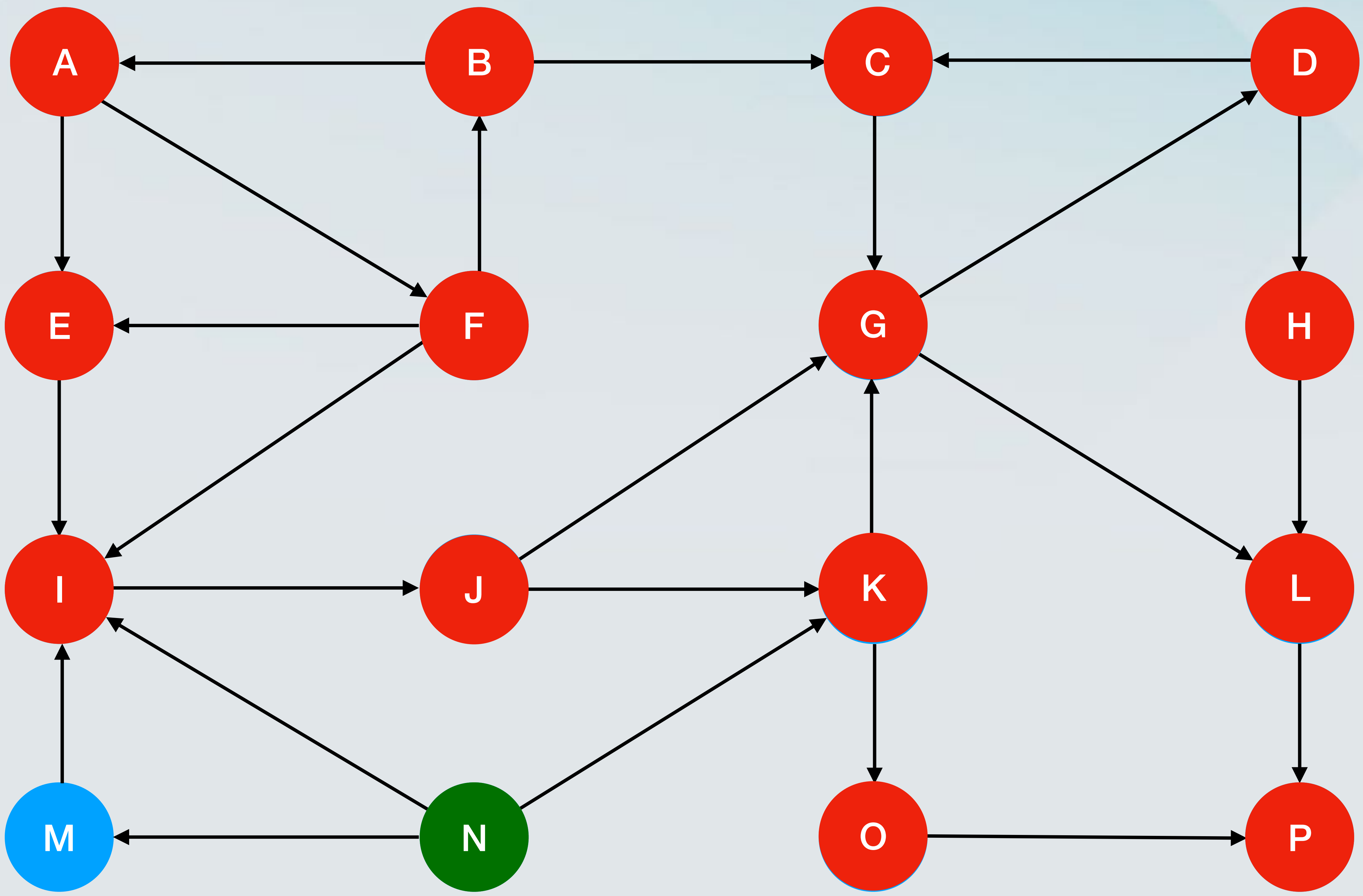
Breadth-First Search



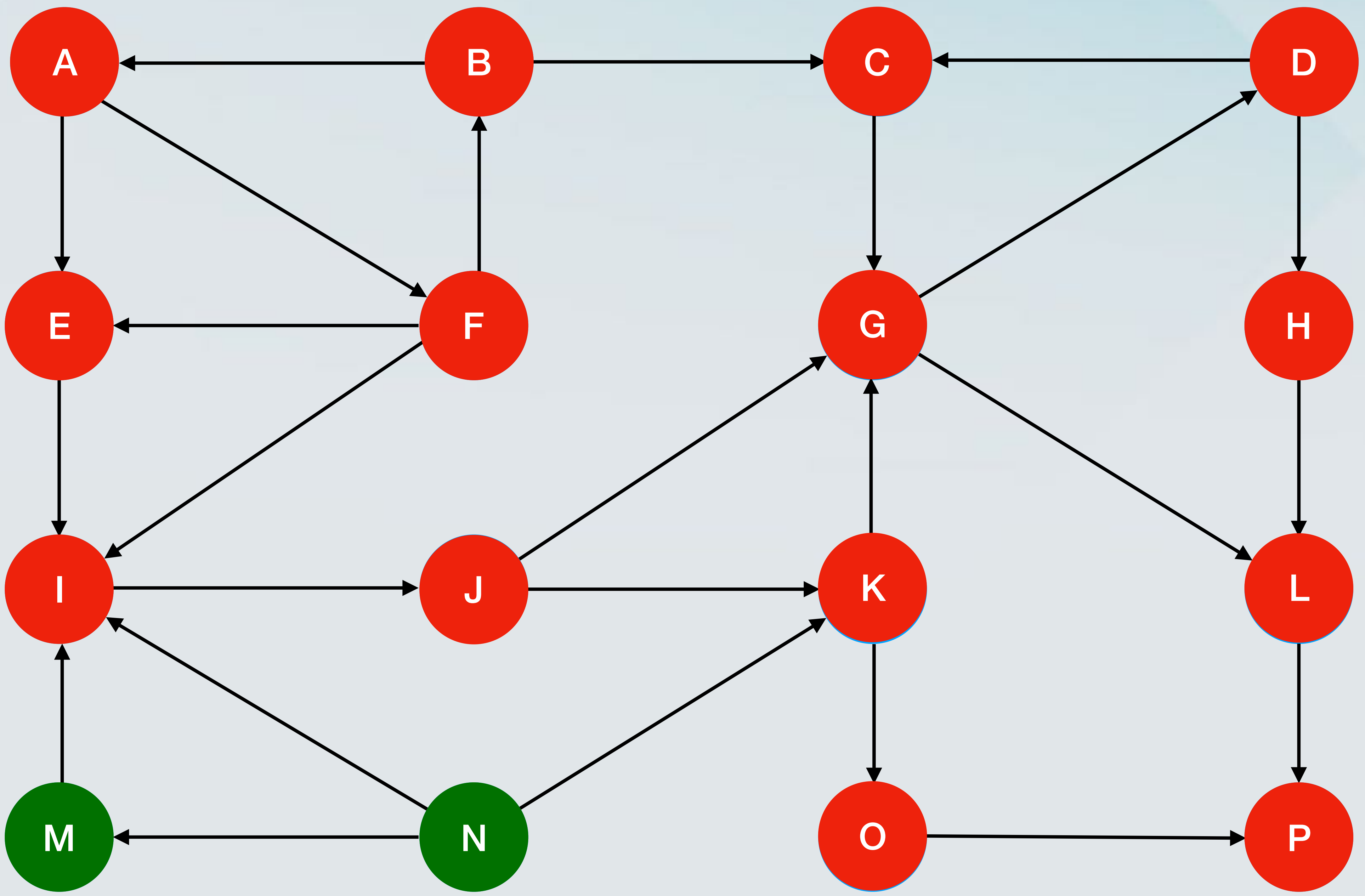
Breadth-First Search



Breadth-First Search



Breadth-First Search



Connectivity

- What BFS is computing is the set of nodes t such that there is a path from s to t.
- A path from s to t does not mean that there is path from t to s.
- (Weak) connectivity: If we ignored the directions for all edges, there would a path from any node to any node.
- Strong connectivity: For every two nodes u and v, there is a path from u to v and a path from v to u.
- **Question: Given a graph G=(V,E), is it strongly connected?**

Mutual reachability

- Two nodes u and v are mutually reachable, if there is path from u to v and a path from v to u in G.
- Strong connectivity: For every pair of nodes u and v, these nodes are mutually reachable.
- Transitivity: If u and v are mutually reachable and v and w are mutually reachable, then u and w are mutually reachable.

Testing strong connectivity

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.

- Pick any node s in V and run BFS(G,s) and BFS(G^{rev},s).

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.
- Pick any node s in V and run BFS(G,s) and BFS(G^{rev},s).
- If one of the two searches does not reach every node, then the graph is definitely not strongly connected.

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.

- Pick any node s in V and run BFS(G,s) and BFS(G^{rev},s).

- If one of the two searches does not reach every node, then the graph is definitely not strongly connected.

- Assume that both searches reach every node. This means that there is a path from s to any node u and a path from any node u to s.

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.
- Pick any node s in V and run BFS(G,s) and BFS(G^{rev},s).
- If one of the two searches does not reach every node, then the graph is definitely not strongly connected.
- Assume that both searches reach every node. This means that there is a path from s to any node u and a path from any node u to s.
  - For any node u, s and u are mutually reachable.

Testing strong connectivity

- Define the reverse graph G^{rev}, in which the nodes are the same and the edges are the same with reversed directions.

- Pick any node s in V and run BFS(G,s) and BFS(G^{rev},s).

- If one of the two searches does not reach every node, then the graph is definitely not strongly connected.

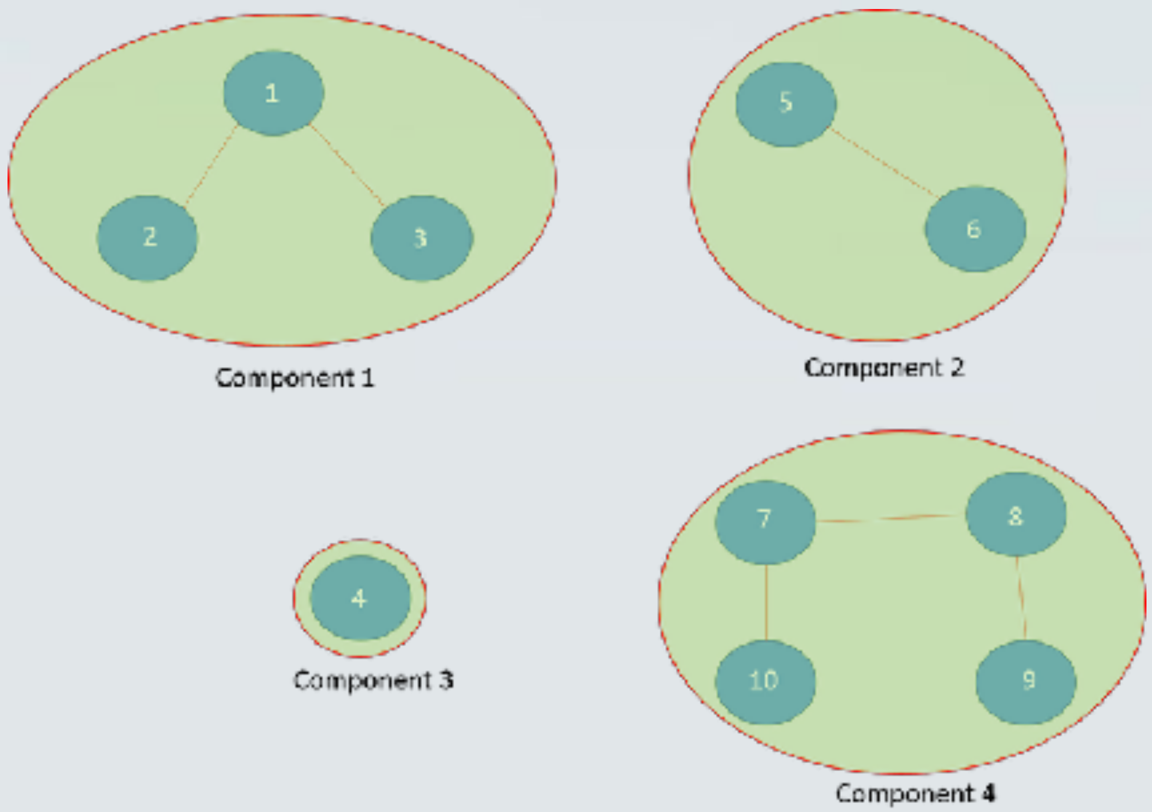
- Assume that both searches reach every node. This means that there is a path from s to any node u and a path from any node u to s.

  - For any node u, s and u are mutually reachable.

- Pick any other node v. Since s and v are also mutually reachable, by transitivity, v and u are mutually reachable and the graph is strongly connected.

Connected component

- A **connected component** of an *undirected* graph **G** is subgraph such that any two nodes are connected via some path.



Connected component

- A **connected component** of an *undirected* graph **G** is subgraph such that any two nodes are connected via some path.
- A strongly **connected component** of a *directed* graph **G** is subgraph such that any two nodes are mutually reachable.

Strongly connected components

- How do we find all strongly connected components of a graph G?
- We can run the "forward" and "backward" BFS for a node s and find the set of nodes that are mutually reachable from s.
  - This is the strongly connected component of s.
  - But BFS might produce different connected components, depending on how we visit the nodes.
  - We need a consistent way of visiting them in the "forward" and in the "backward" pass.

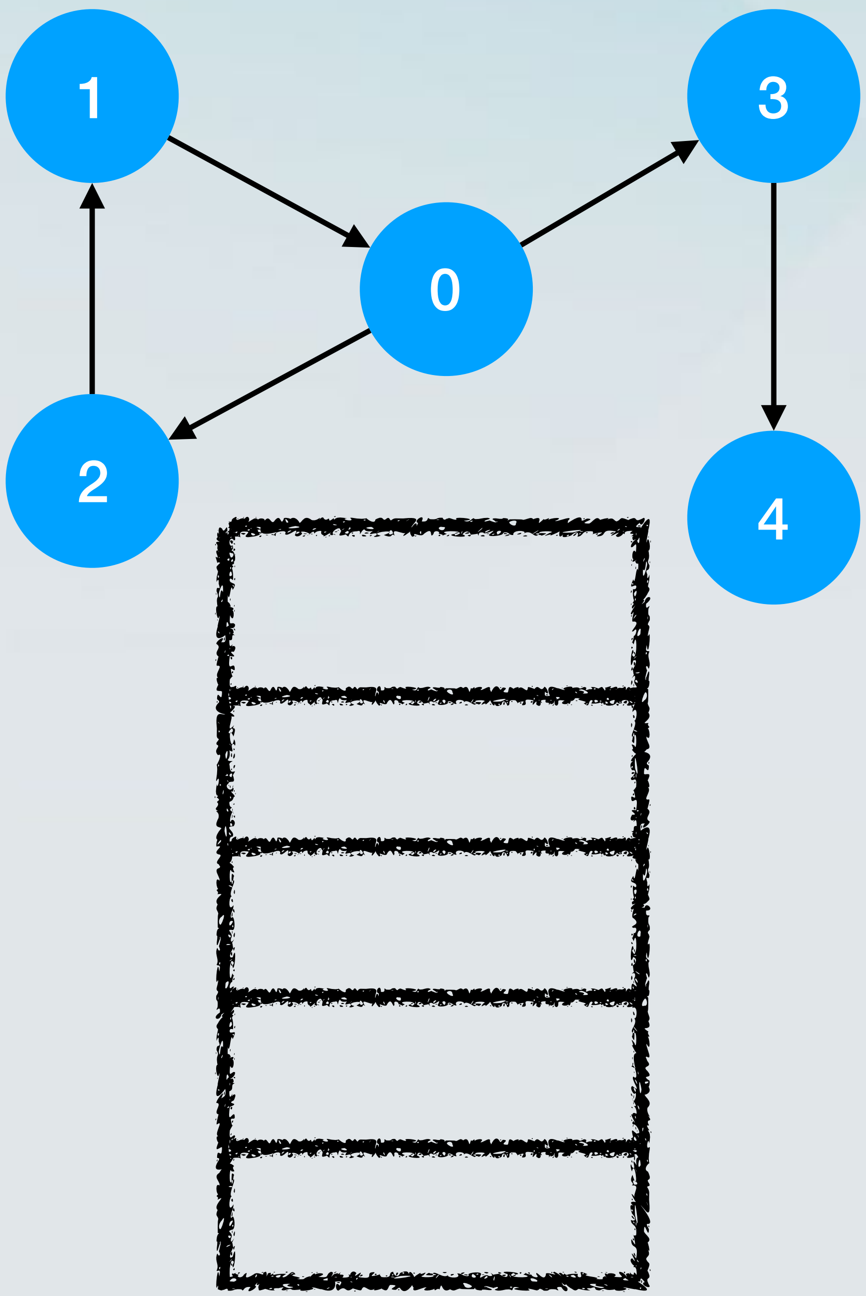
Kosajaru’s algorithm

Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.

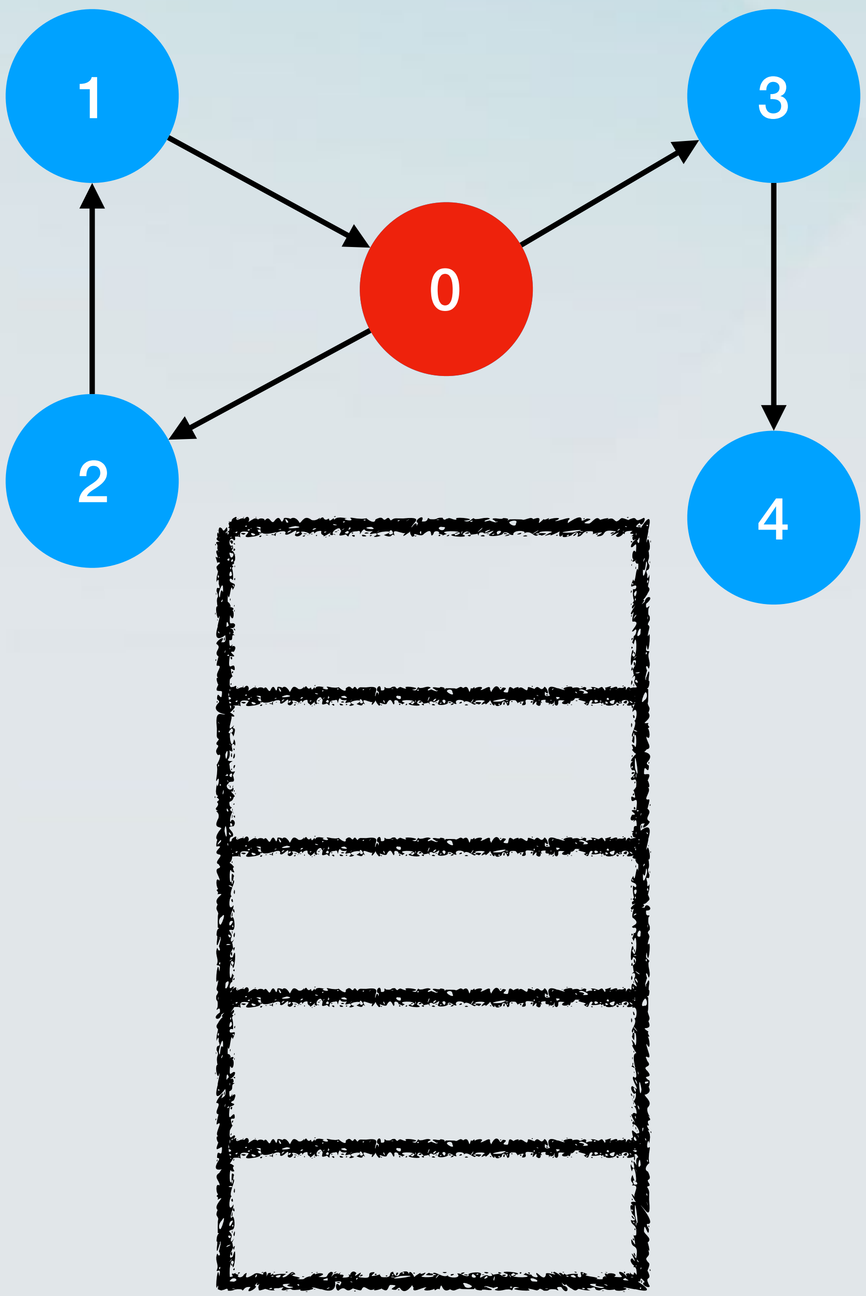
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



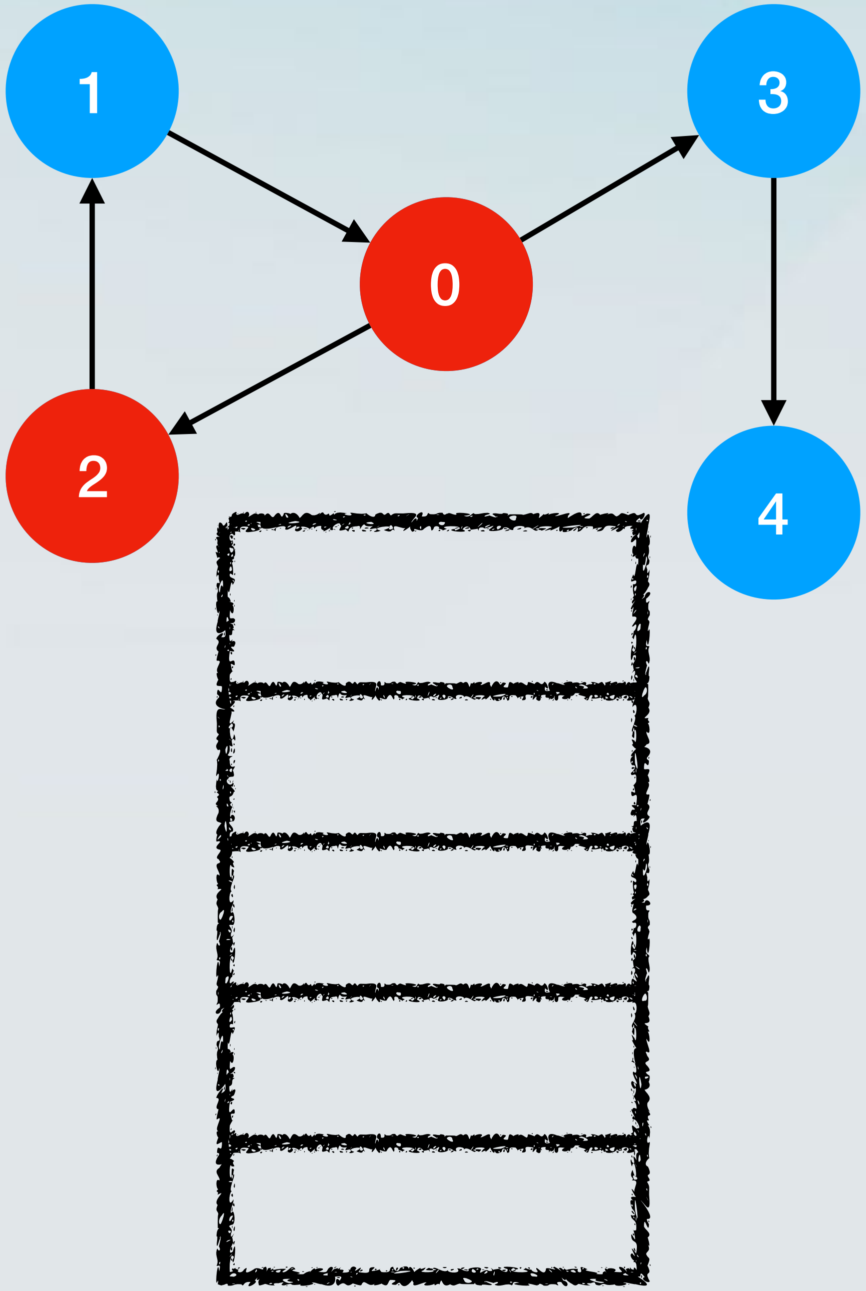
Kosajaru’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



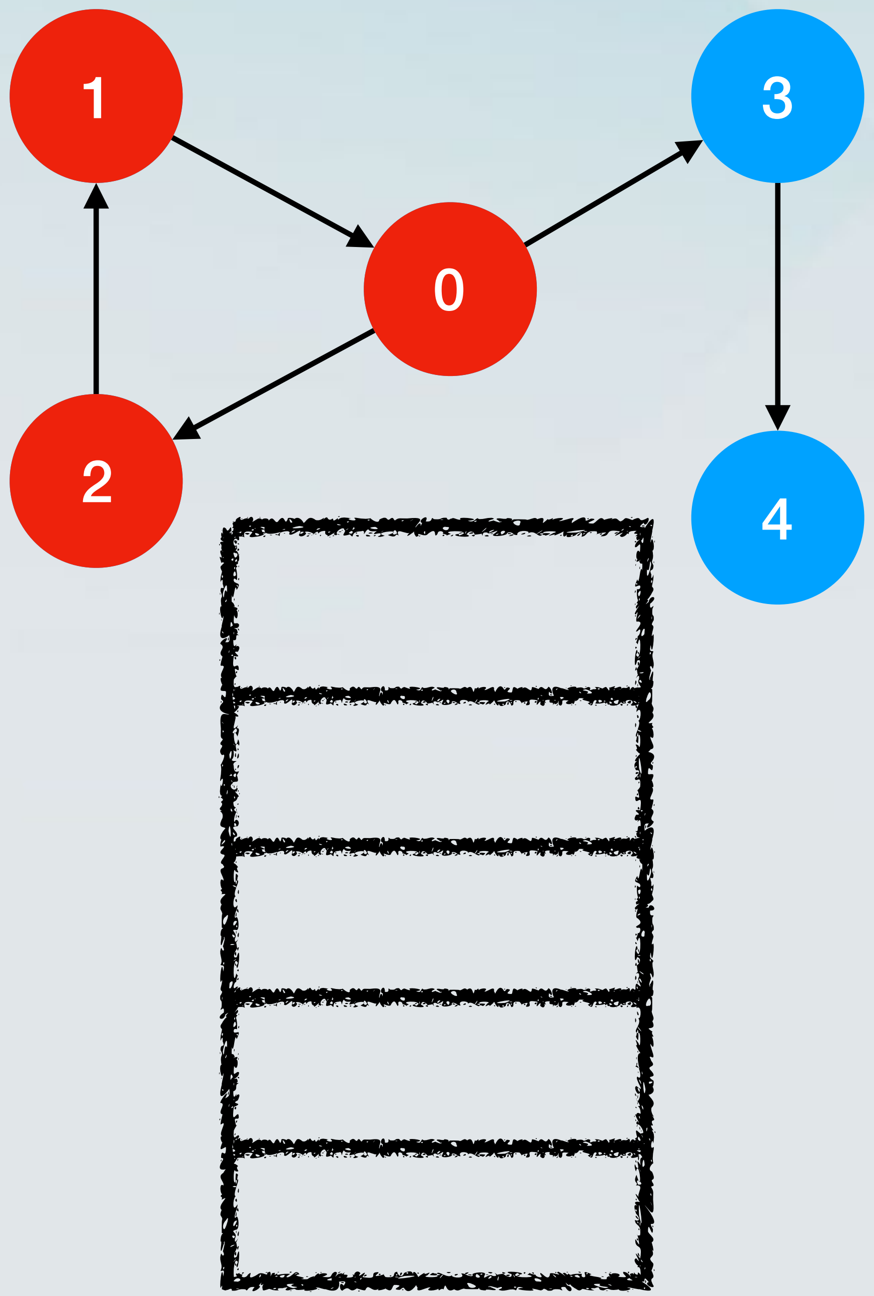
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



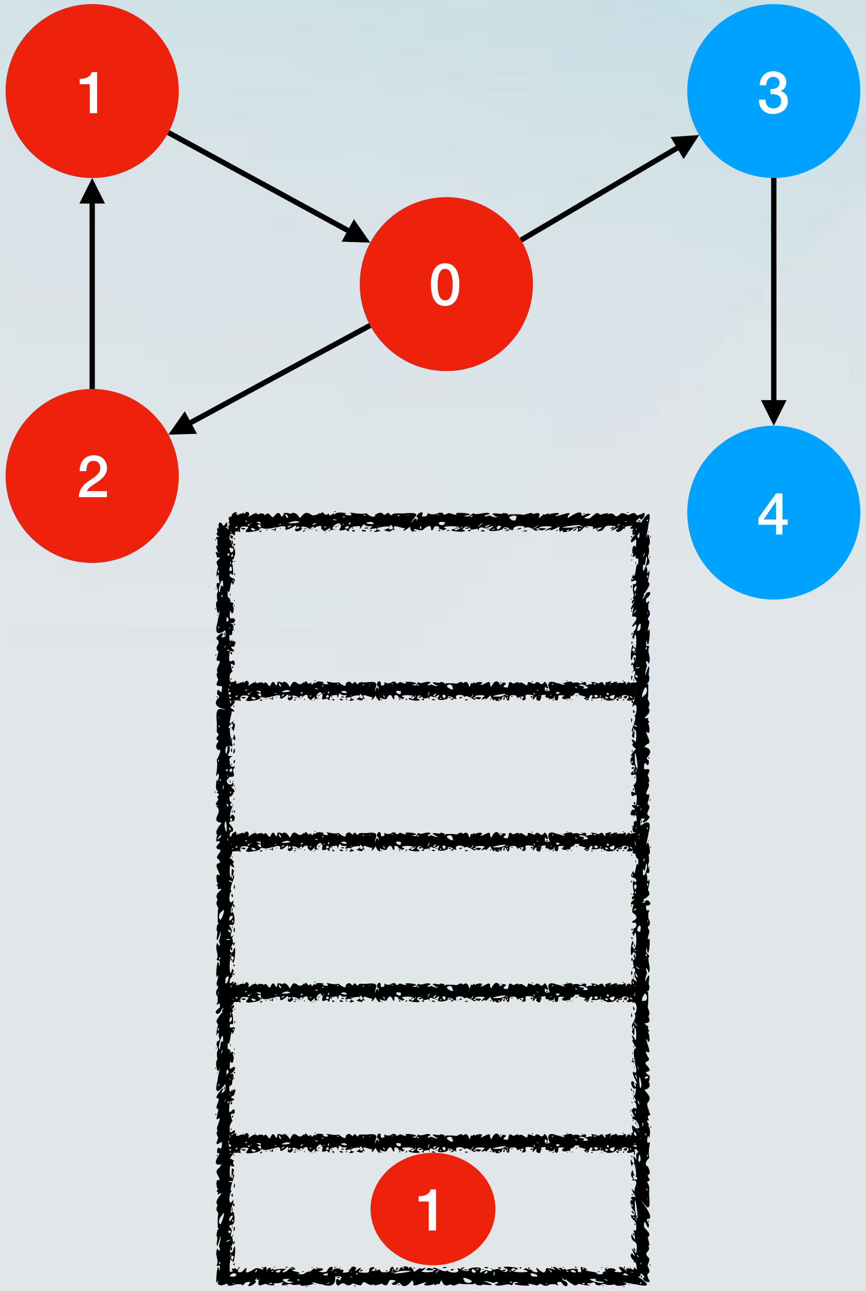
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



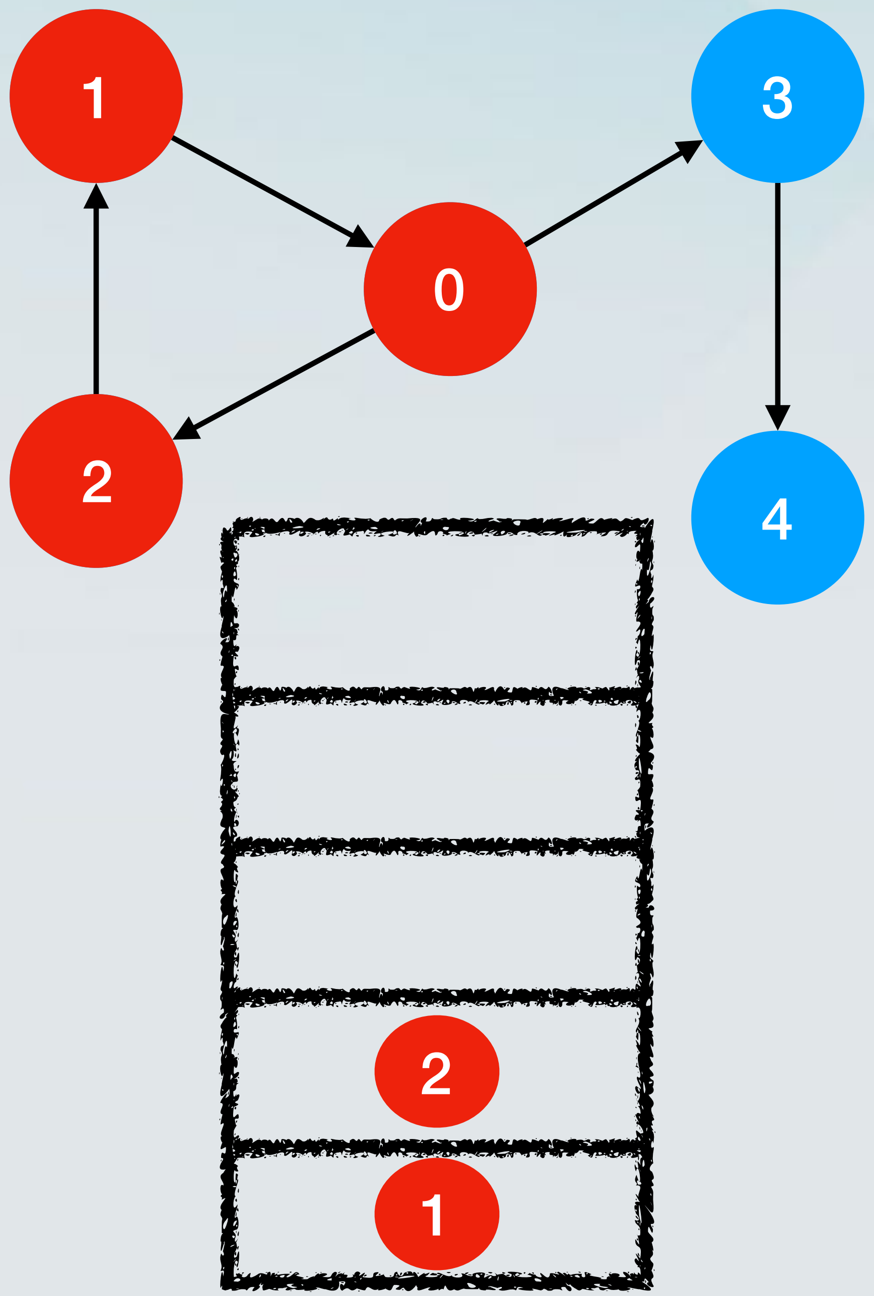
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



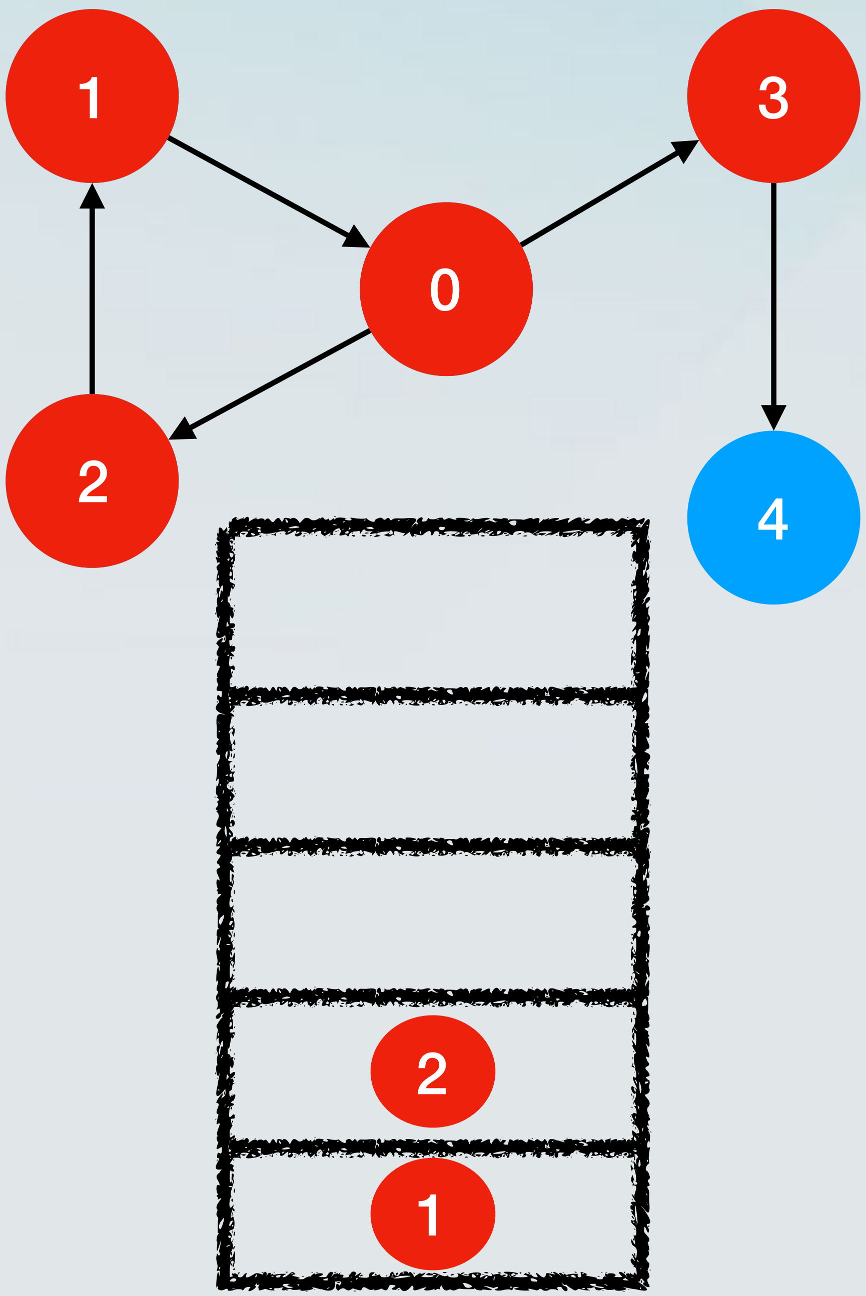
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



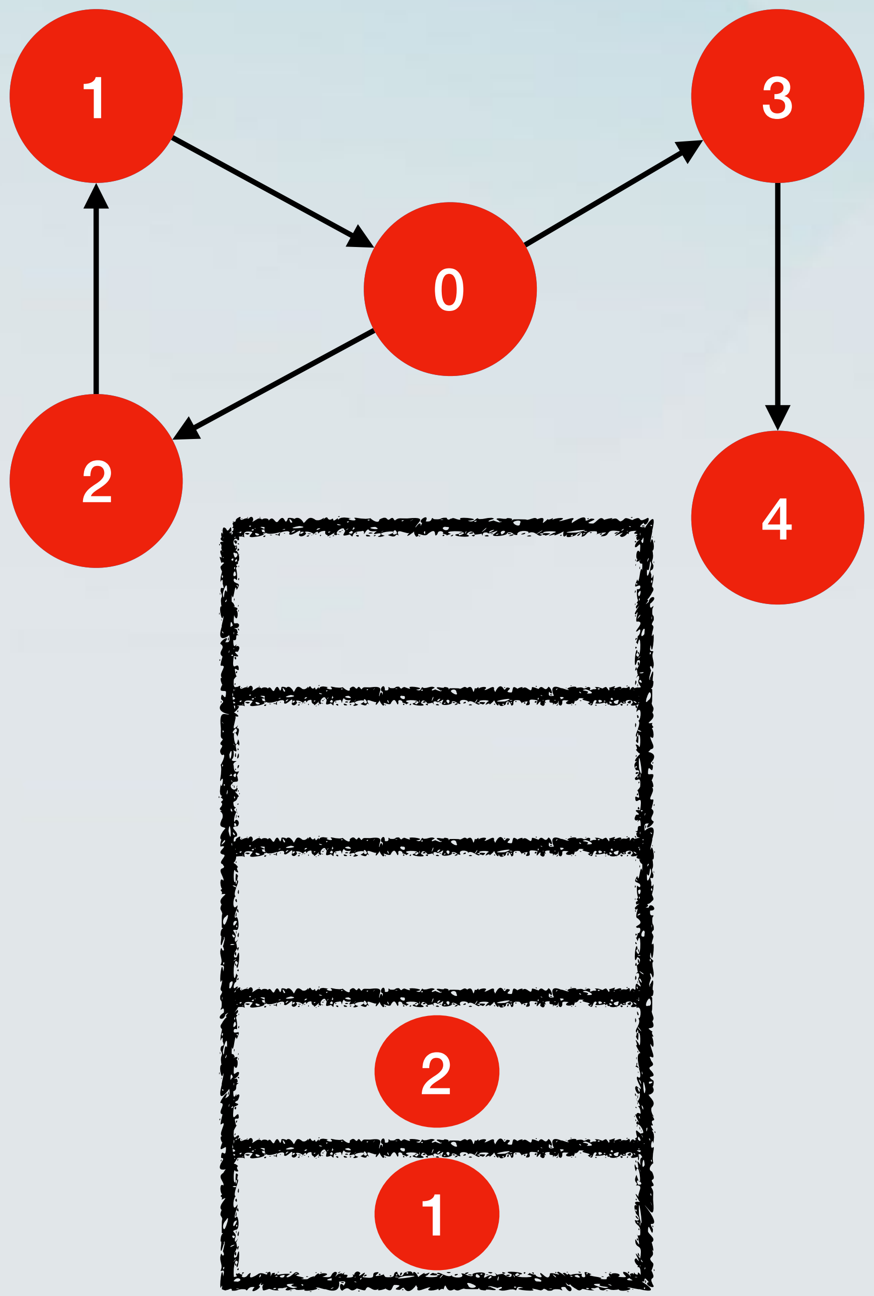
Kosajaru’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



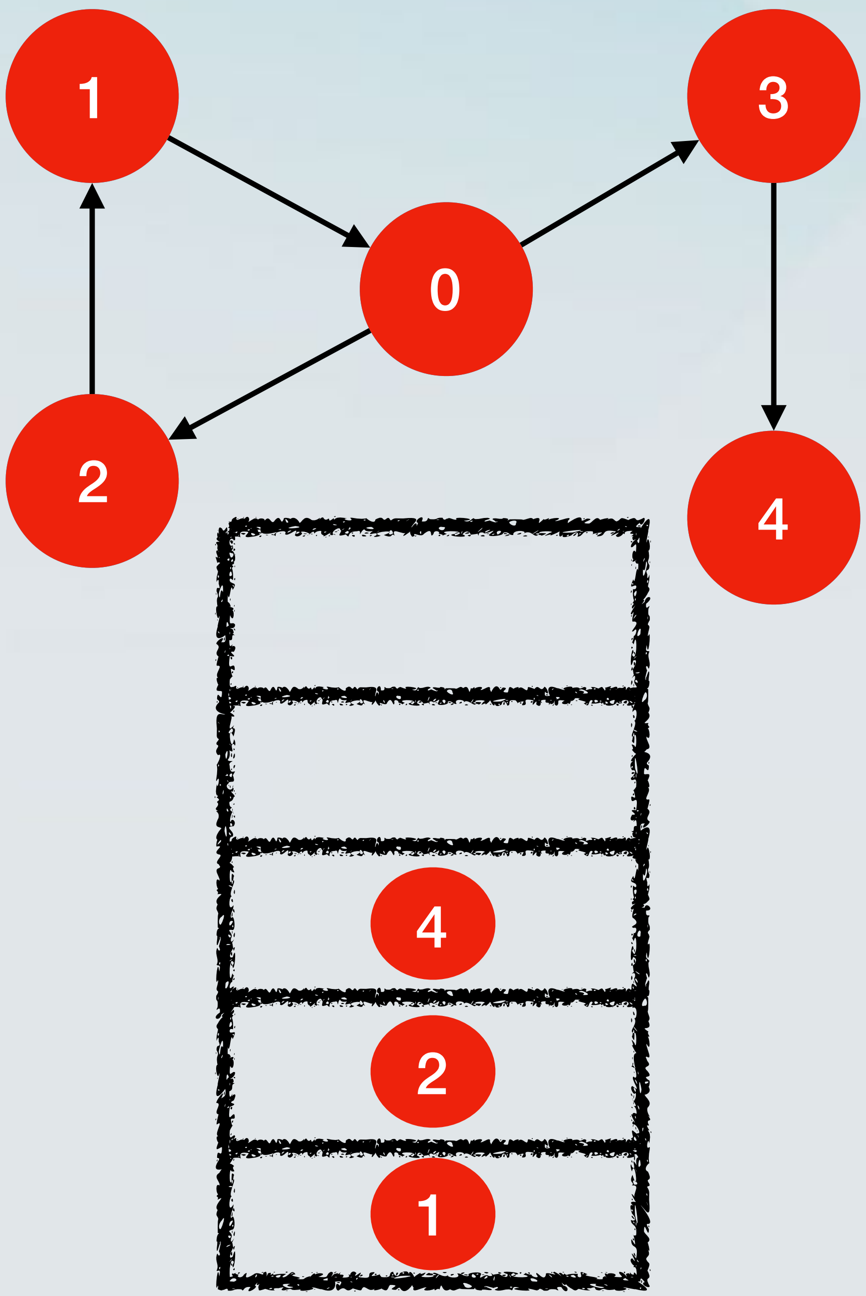
Kosajaru’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



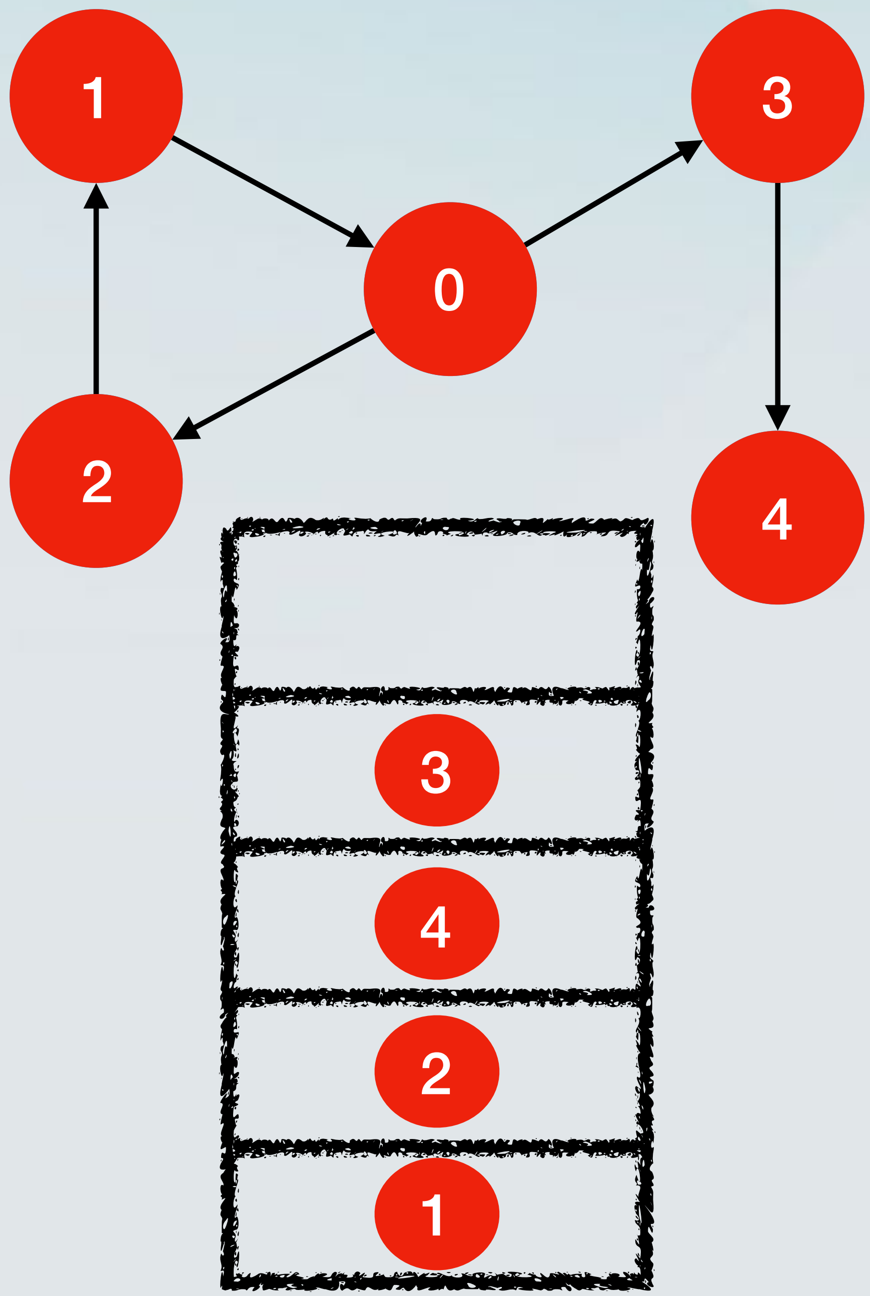
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



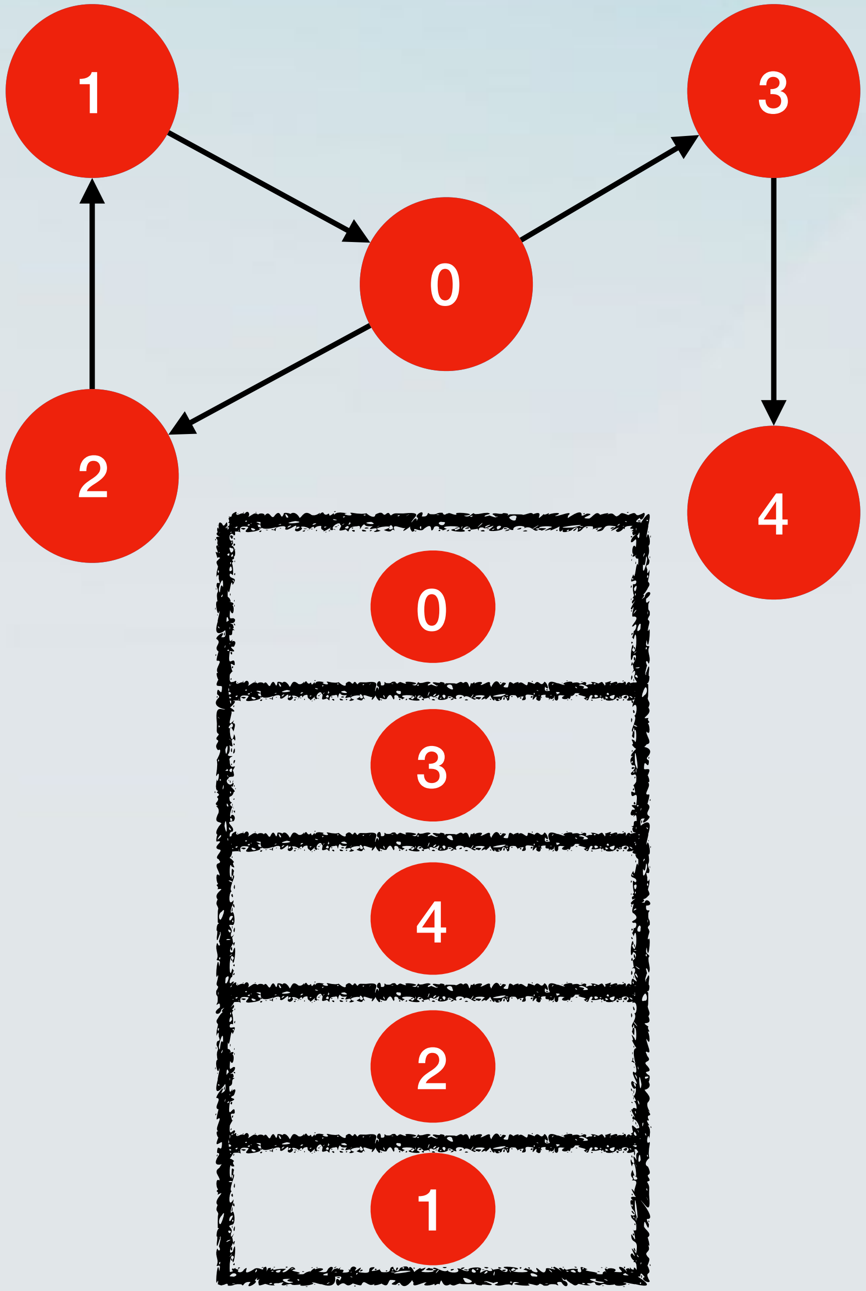
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



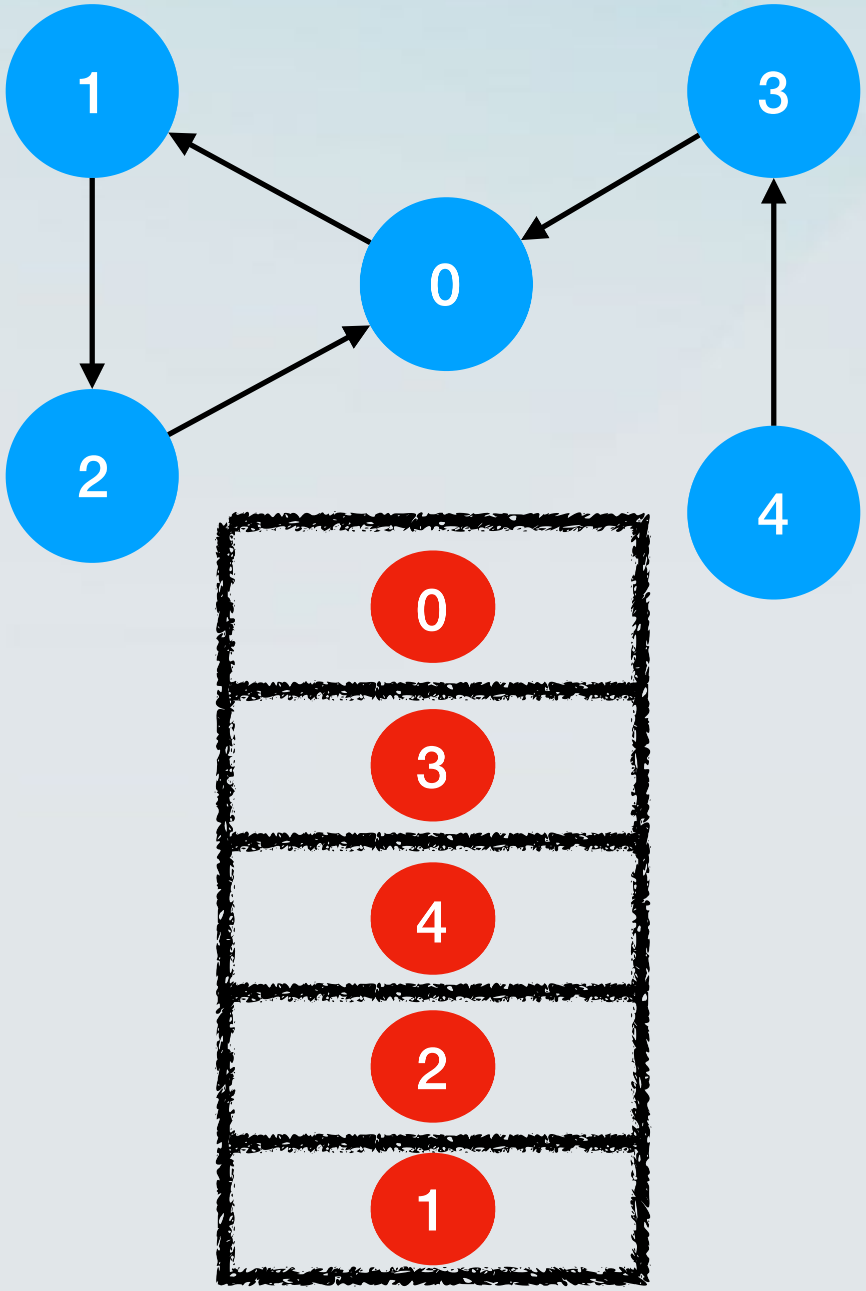
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



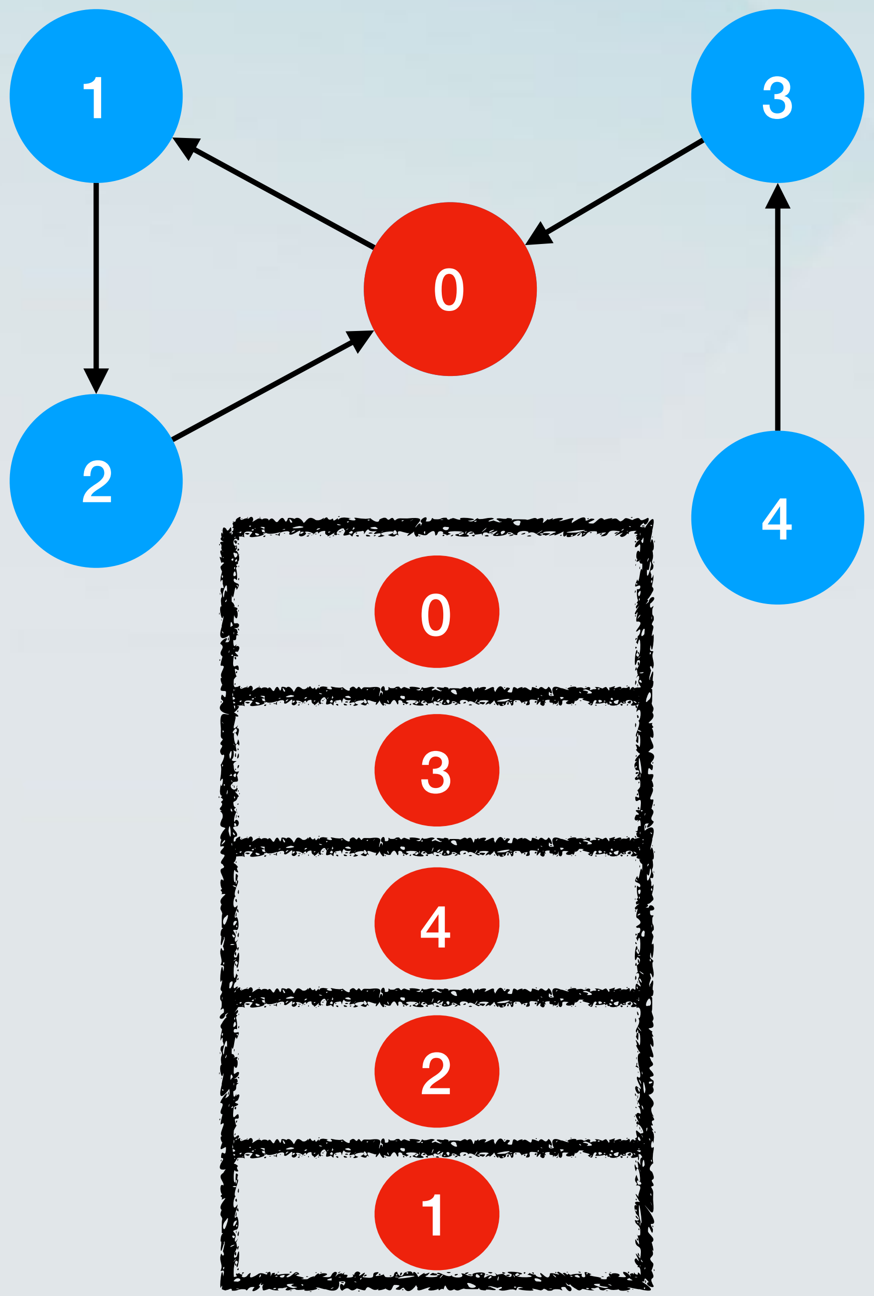
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



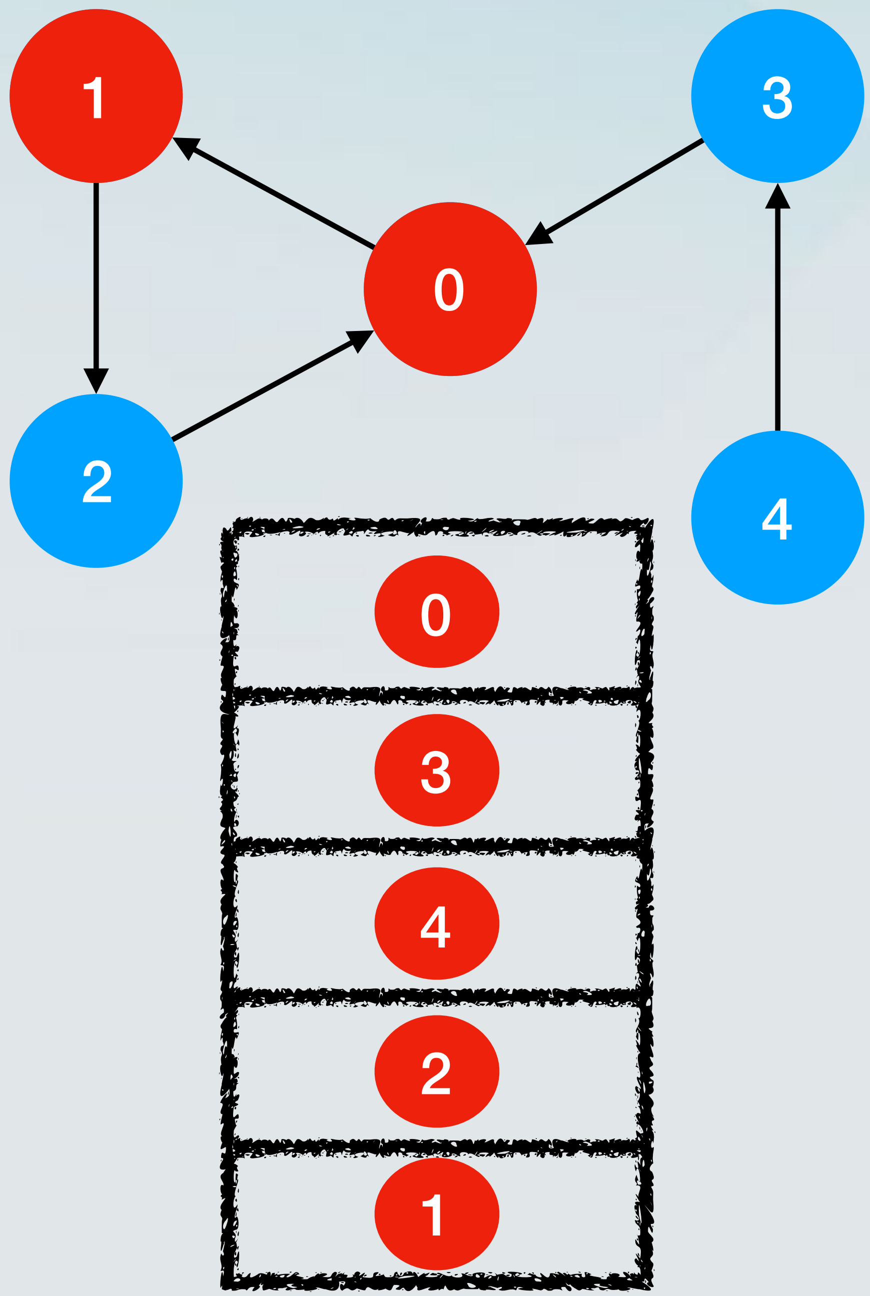
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



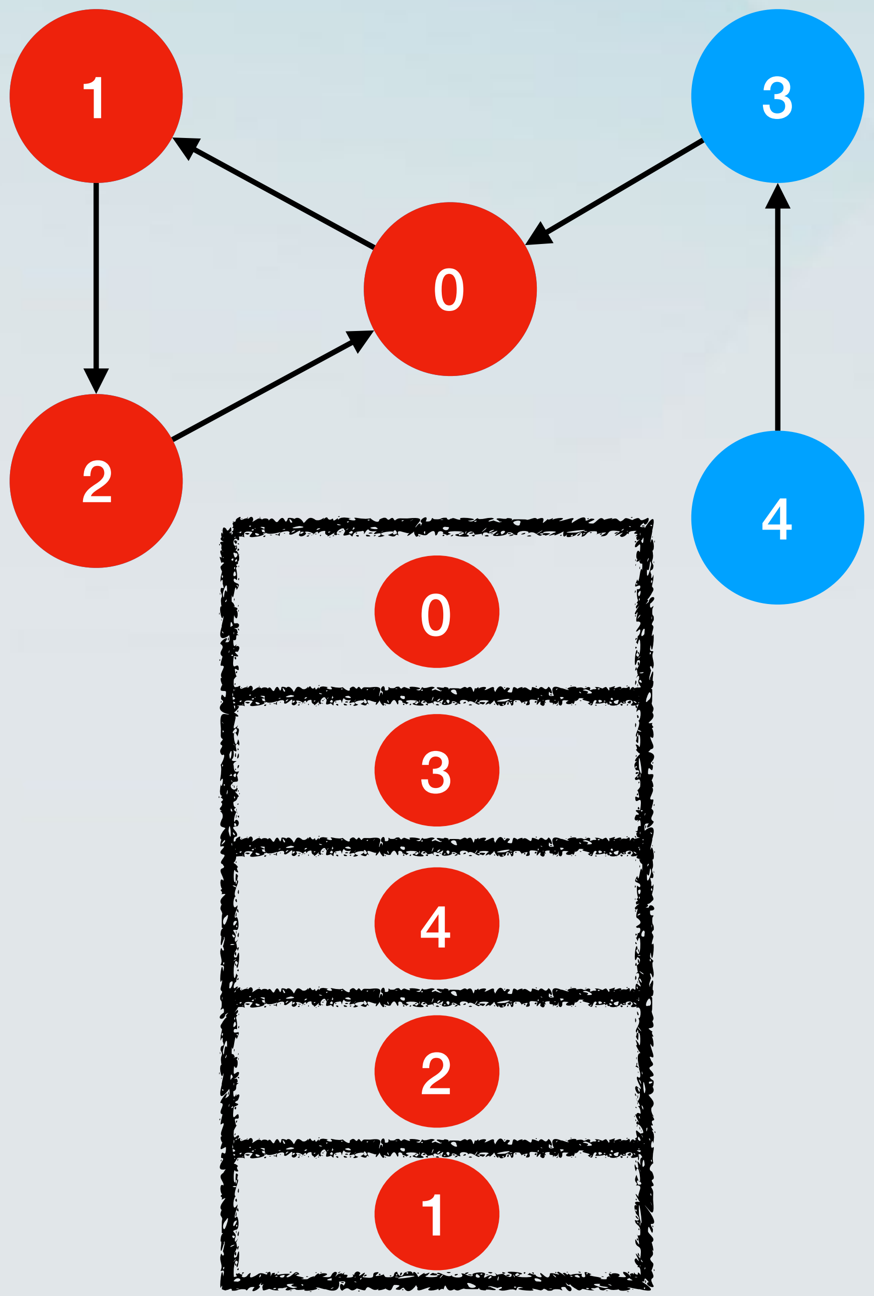
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



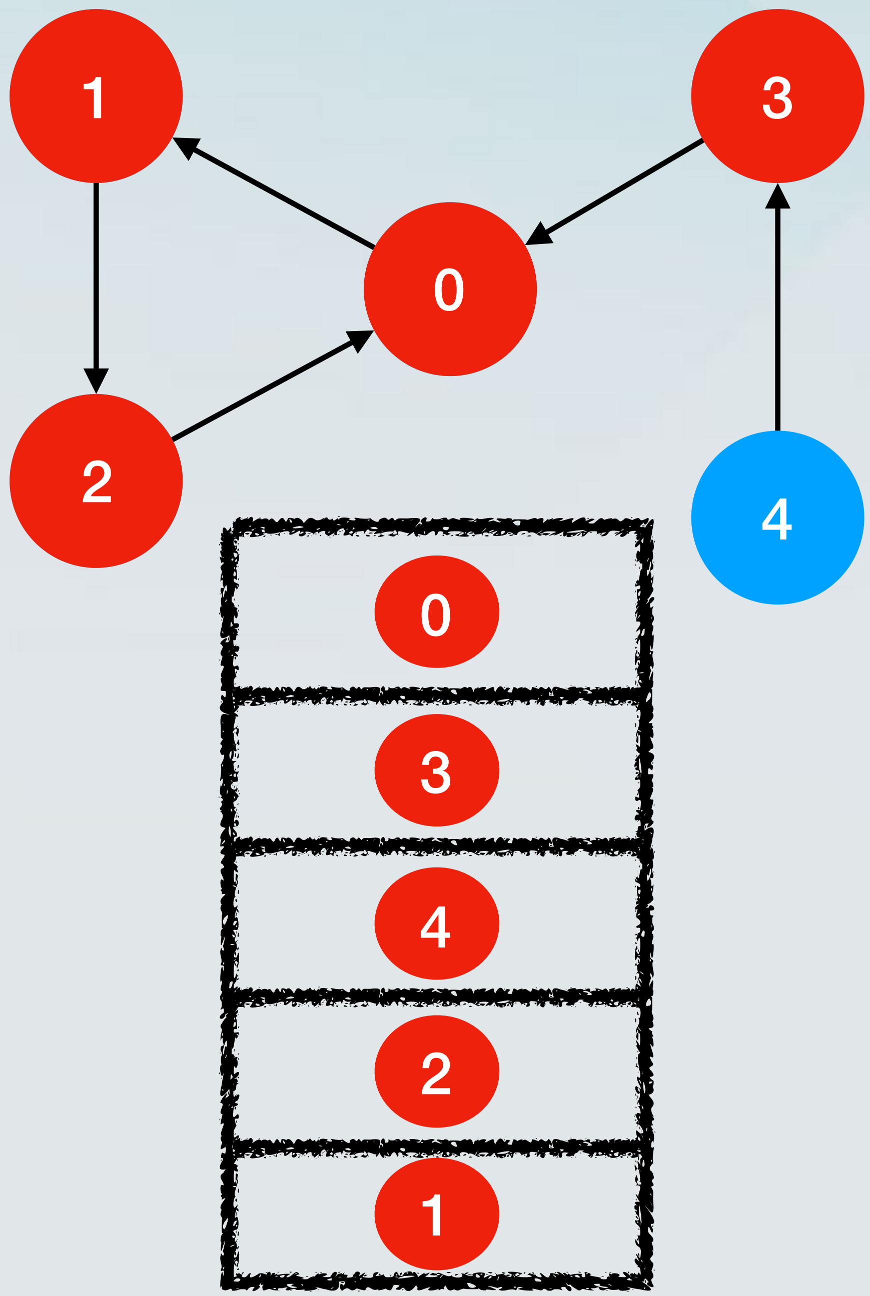
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



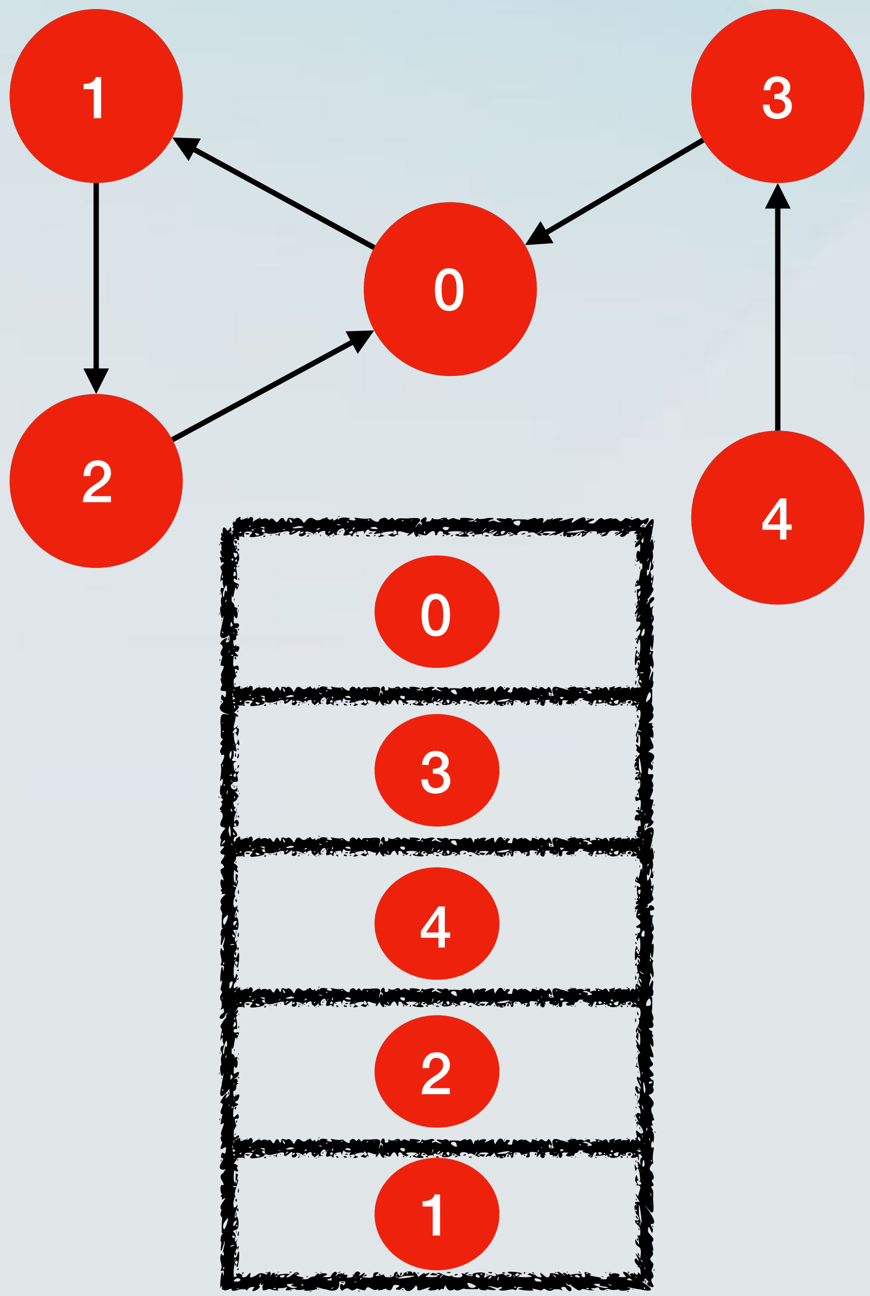
Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



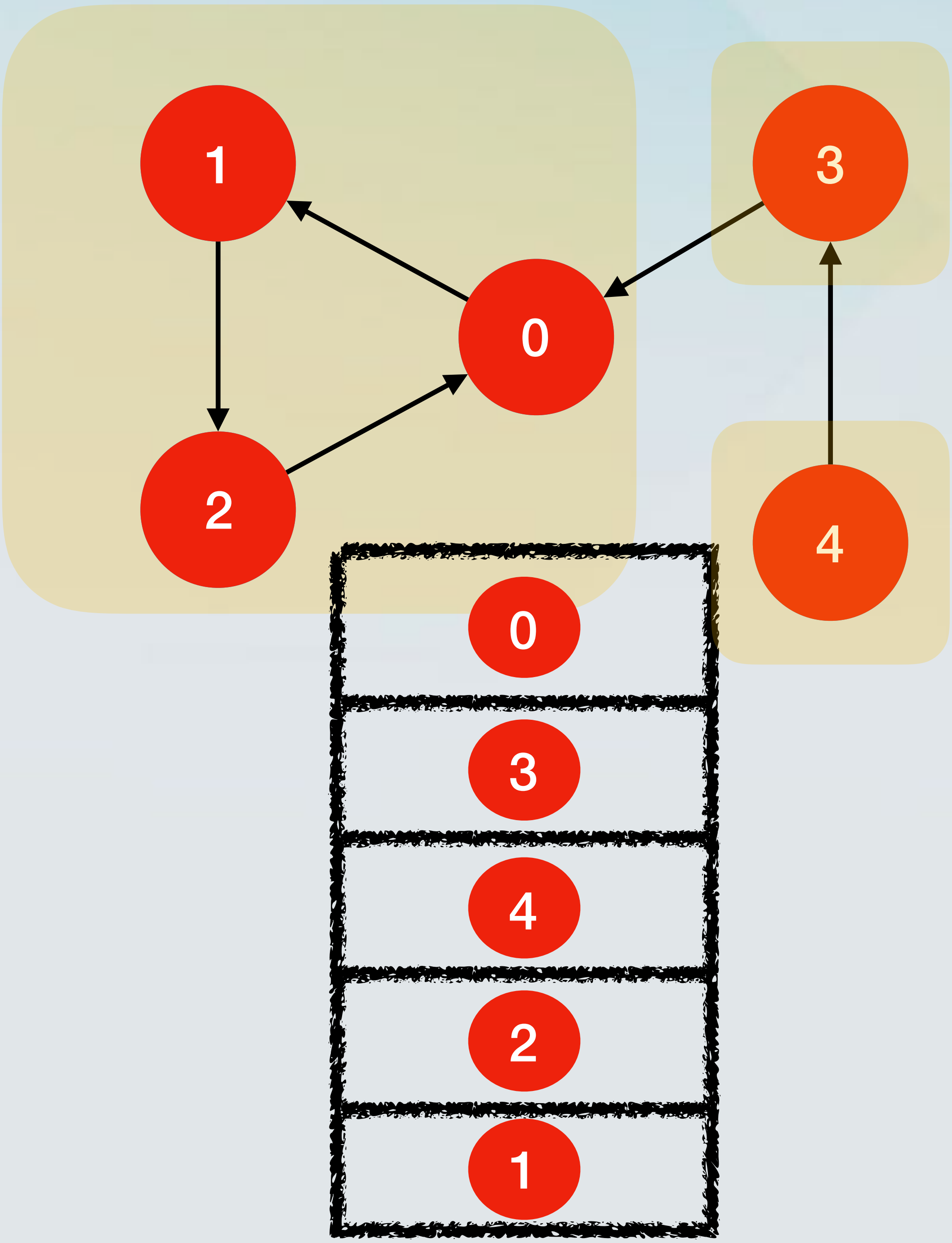
Kosajaru’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



Kosajaru's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



Running time

Running time

- We perform DFS twice.
- The running time is **O(m+n)**.

Correctness

- Next lecture.