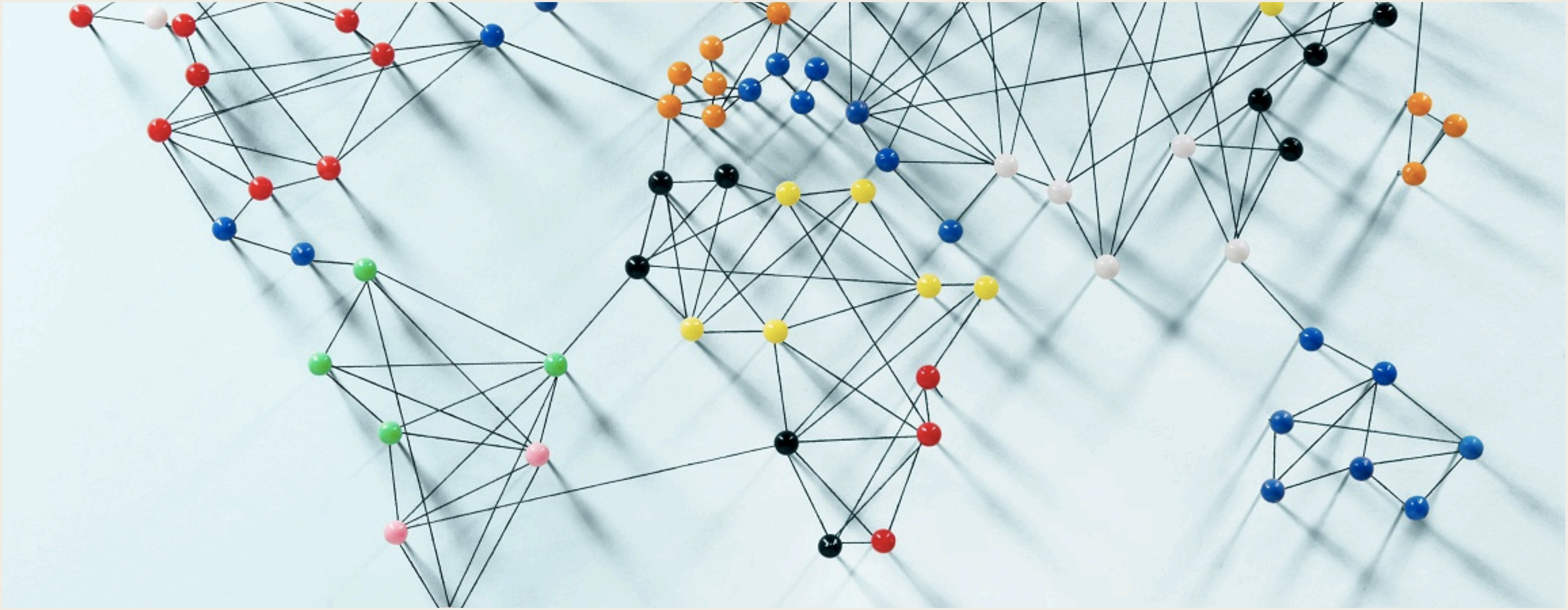


GRAPH PROBLEMS

AJAY KUMAR

Codeforces

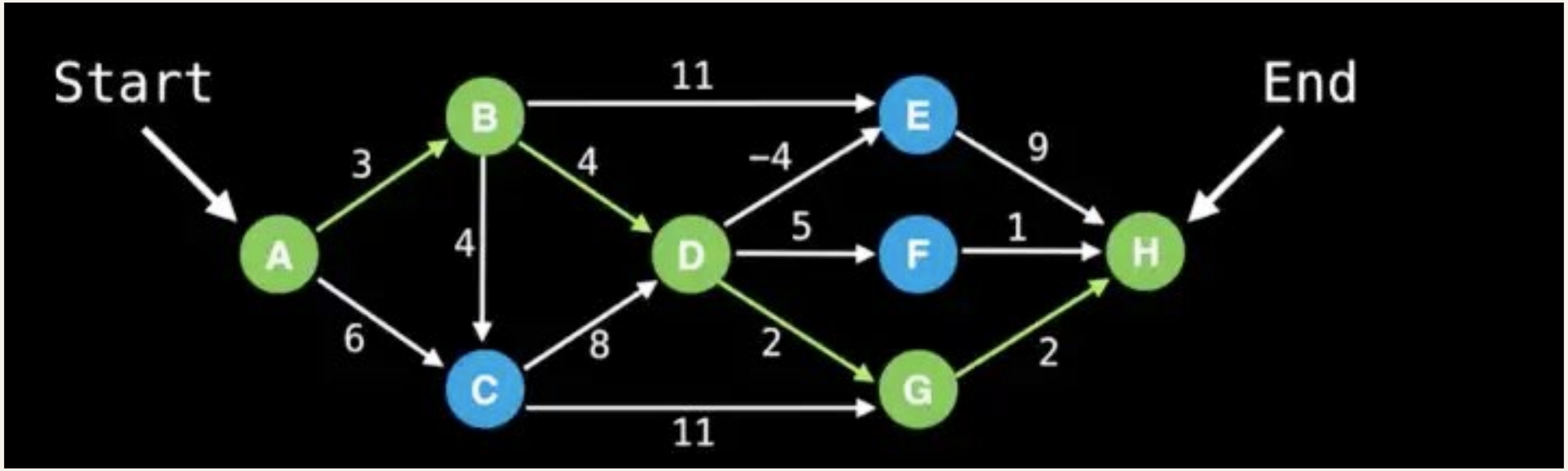
GRAPH THEORY

SHORTEST PATH PROBLEM	
CONNECTIVITY	
NEGATIVE CYCLES	
STRONGLY CONNECTED COMPONENTS	
BRIDGES AND MINIMUM SPANNING TREES	
TRAVELLING SALESMAN PROBLEM	

Shortest path problem

Given a weighted graph, find the shortest path of edges from node A to node B.

Algorithms – BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, etc.



BFS

Implementation of BFS traversal:

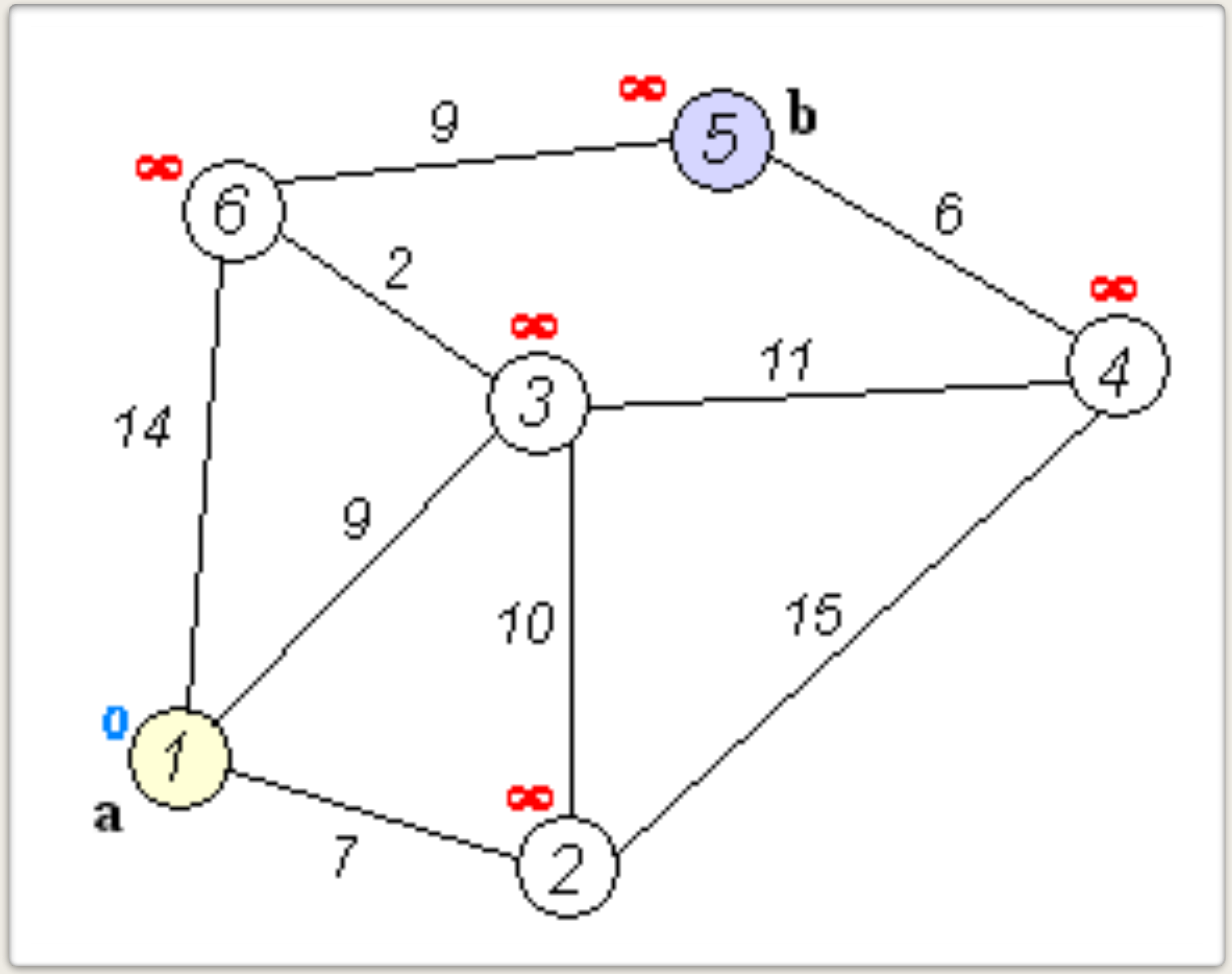
Follow the below method to implement BFS traversal.

- Declare a queue and insert the starting vertex.
- Initialize a visited array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
  - Remove the first vertex of the queue.
  - Mark that vertex as visited.
  - Insert all the unvisited neighbours of the vertex into the queue.

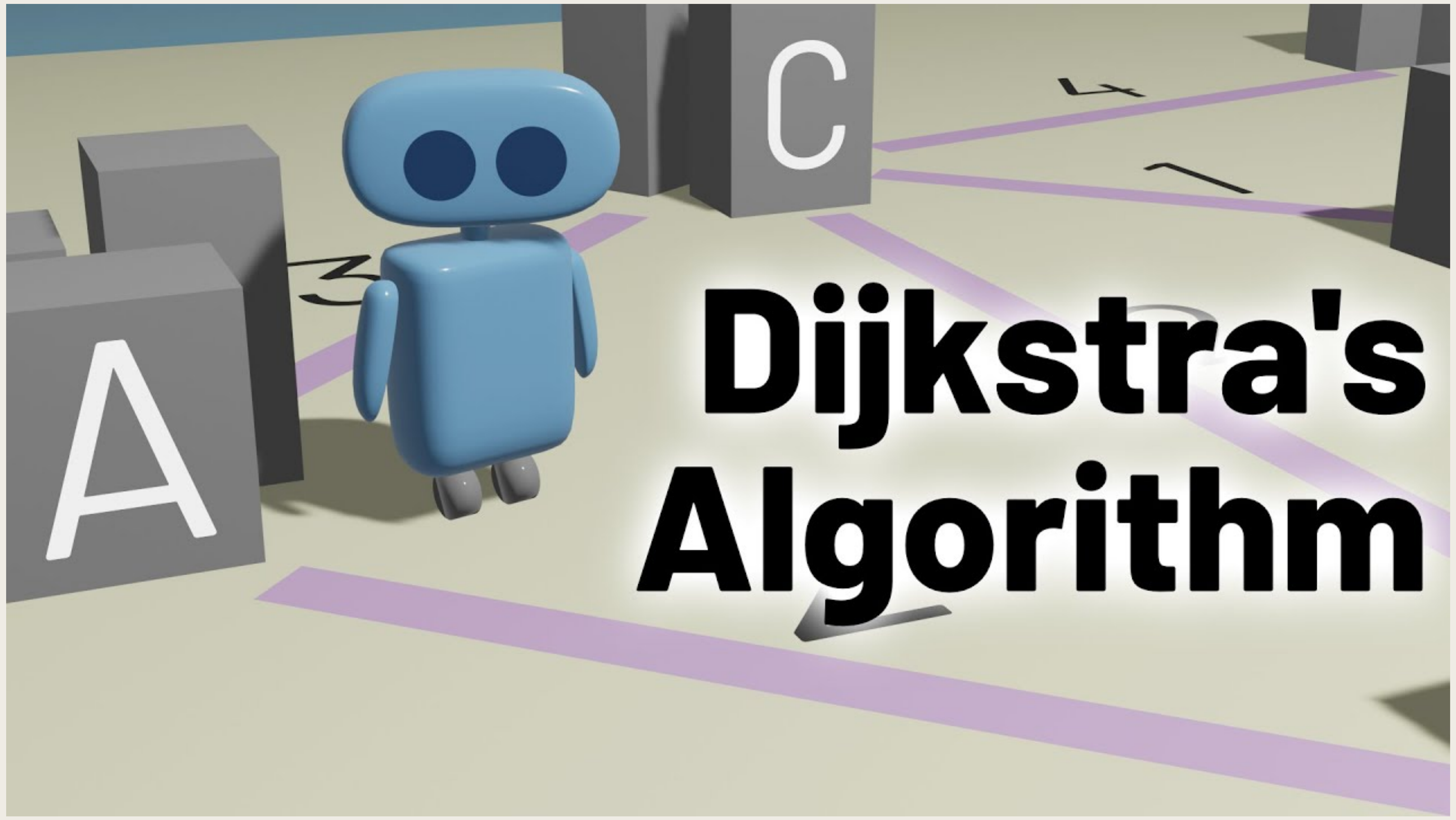
  - Important: Never do this thing What I have done in Labrynath problem at cses. Marking the visited outside the inner loop. At the time of pushing into the queue it should be marked visited

```
vector<int> bfsOfGraph(int V, vector<int> adj[]) {
    bool visited[V+1]
    vector<int>v;
    for(int i=1;i<=V;i++)
    visited[i]=false;
    queue<int>q;
    int s=0;
    visited[s]=true;
    q.push(s);
    while(!q.empty()){
        int node= q.front();
        q.pop();
        v.push_back(node);
        for(int i=0;i<adj[node].size();i++){
            if(!visited[adj[node][i]])
            {visited[adj[node][i]]=true;
            q.push(adj[node][i]);}
        }
     }
    return v;
}
// Time complexity: O(V+E)
```

Dijkstra's Alogorithm



Youtube video



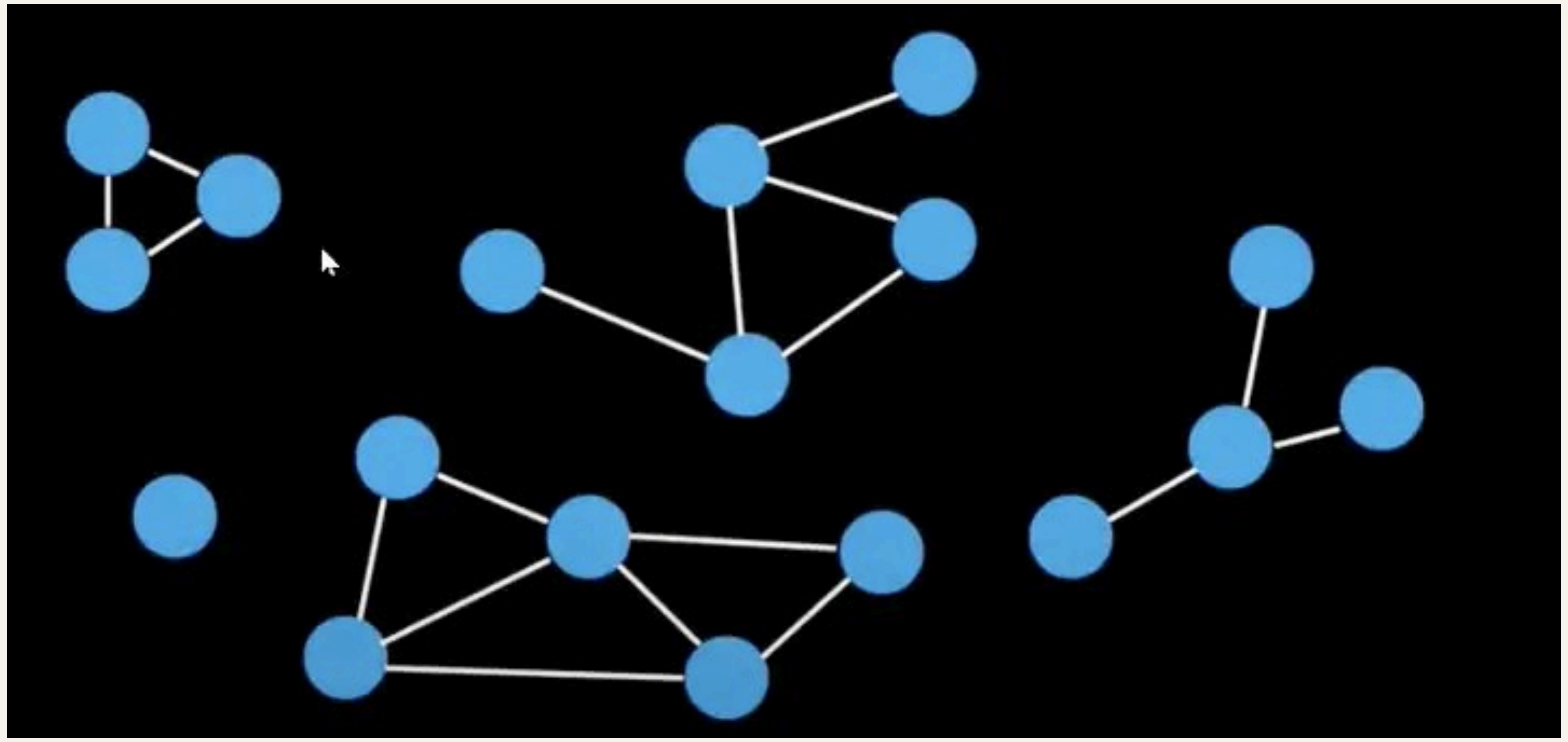
Keep hovering

```cpp
vector <int> dijkstra(int V, vector<vector<int>> adj[], int s){
        vector<int>dist(V,LLMAX);
        dist[s]=0;
        vector<bool>visited(V,false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,s});
        while(!pq.empty()){
            int u=pq.top().second;
            pq.pop();
            visited[u]=true;
            for(auto it: adj[u]){
                int v=it[0];
                int w=it[1];
                if(visited[v]==false&&dist[v]>dist[u]+w){
                    dist[v]=dist[u]+w;
                    pq.push({dist[v],v});
                 }
            }
        }
        return dist;
}
//practice it
```

Connectivity

Connectivity –

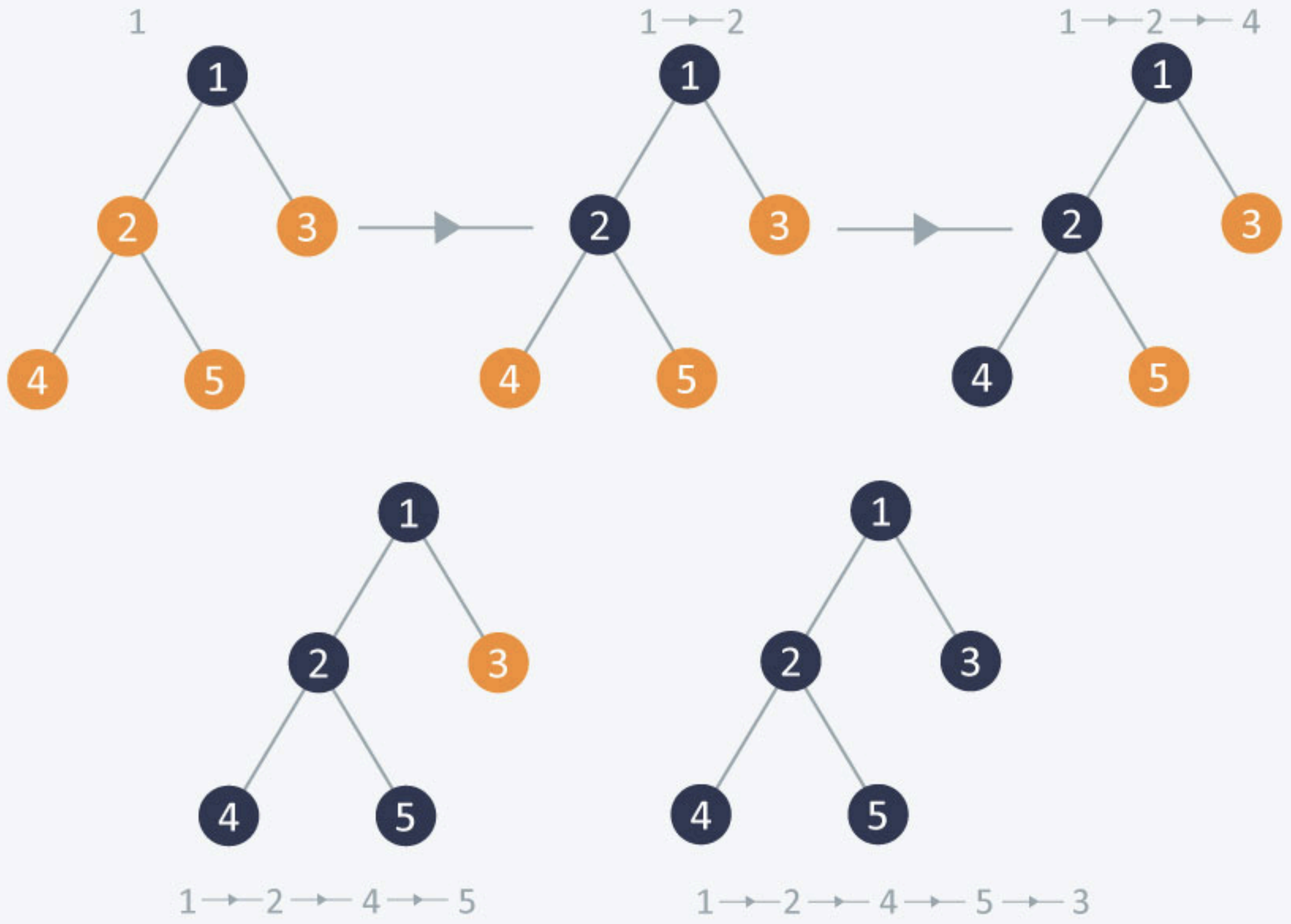
- Does there exist a path between node A and node B?
- **Algorithms** – Use **union find** data structure or any search algorithm (e.g DFS, BFS).



DFS

- The DFS algorithm is a recursive algorithm that uses the idea of backtracking.
- It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
- Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.
- All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.
- This recursive nature of DFS can be implemented using stacks.

DFS



```
 void traverse(vector<int>adj[],int node,vector<int>&sequence,bool visited[]){
        visited[node]=true;
        sequence.push_back(node);
        for(int u:adj[node]){
            if(visited[u]==false){
                traverse(adj,u,sequence,visited);       //don’t use return here
            }
        }
  }
  vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        vector<int>sequence;
        bool visited[V];
        for(int I=0;i<V;i++){
            visited[i]=false;
        }
        int start=0;
        traverse(adj,start,sequence,visited);
        return sequence;
  }

//practice it
```

Directed Graph

In this chapter, we focus on two classes of directed graphs:

Acyclic graphs:

- There are no cycles in the graph, so there is no path from any node to itself1 .

Successor graphs:

- The outdegree of each node is 1, so each node has a unique successor.
- It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

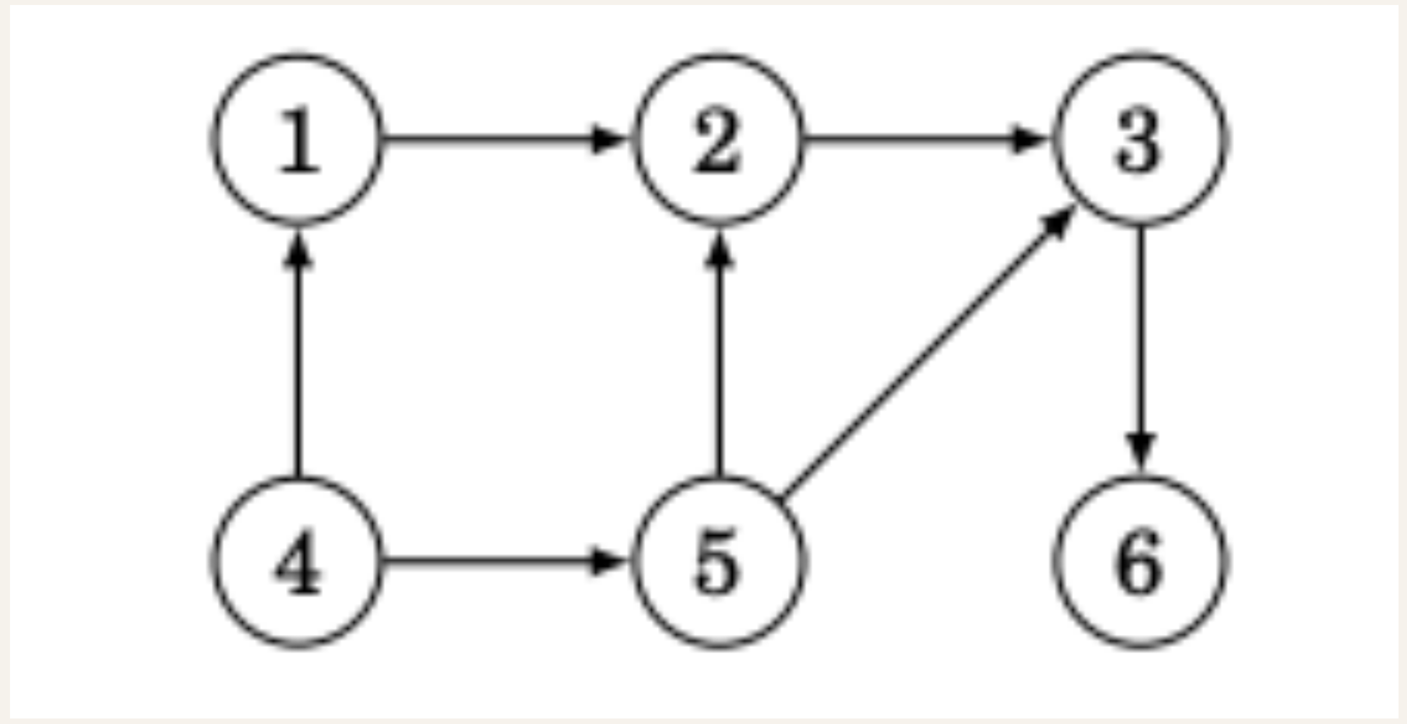
Topological Sort

A topological sort is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. For example, for the graph 1 2

DFS ALGORITHMS

To review, a **directed** graph consists of edges that can only be traversed in one direction.
Additionally, an **acyclic** graph defines a graph which does not contain cycles, meaning you are unable to traverse across one or more edges and return to the node you started on.
Putting these definitions together, a **directed acyclic** graph, sometimes abbreviated as DAG, is a graph which has edges which can only be traversed in one direction and does not contain cycles.

Simply visit this tutorial by CSA



An acyclic graph always has a topological sort.

However, if the graph contains a cycle, it is not possible to form a topological sort, because no node of the cycle can appear before the other nodes of the cycle in the ordering.

It turns out that depth-first search can be used to both check if a directed graph contains a cycle and, if it does not contain a cycle, to construct a topological sort.

At any point during the DFS traversal of a graph we can divide the nodes in 3 groups:

Nodes that we finished visiting, i.e. popped from the stack.

Nodes that are currently on the stack.

Nodes that are yet to be discovered.

Let's suppose during a DFS we are in a node A that has an outgoing edge to another node B. If the graph is a DAG we can be certain that B falls in categories 1 or 3. If B would be on the stack, than the arc $\rightarrow_{A\rightarrow B}$ would close a cycle, and this is impossible on a DAG.

This property actually tells us that any node is popped from the stack *after* all of its outgoing neighbours are popped. So in order to get a topological sorting we should just keep track of a list with the popped nodes in reverse order.