Navigation System

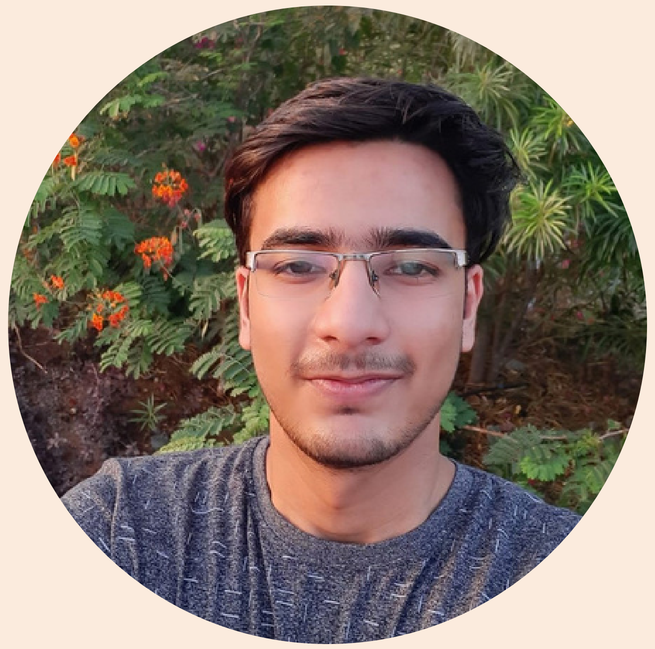
CS263 PROJECT

Introduction

Our idea is to make a navigation system.

This navigation system also takes into account the fuel/charge in the car. It tells the driver if the trip is possible in the given amount of fuel/charge in the car, and, if or where he needs to stop to refuel the car.

Team Members


Akshat Khandelwal
(202051016)

Aradhya Mishra
(202051034)

Anubhav Kushwaha
(202052306)

Harsh Jitesh Dawda
(202051081)

Chitranshi Srivastva
(202051055)

Archit Agrawal
(202051213)

Aaryan Ajay Sharma
(202051001)

Ishan Pandey
(202051089)

Aman Gangwar
(202051020)

Amanshu Jaiswal
(202051023)

Algorithm 1

Problem Statement

Given two nodes, we have to find if there exists any path with which we can reach the other node.

Algorithm Used

Depth First Search

Explanation

The following is how the DFS algorithm will work:

1. Place the source node on top of a stack to begin.
2. Add the top item in the stack to the list of nodes you've visited.
3. Make a list of the nodes that are adjacent to that vertex. Place the nodes that aren't on the visited list at the top of the stack.
4. Repeat steps 2 and 3 until the end node is found or the stack becomes empty.

Pseudocode

```
isPathDFS(Graph G, node source, node end):
    Stack S
    S.push( source )

    tag source as visited

    while(S is not empty):
        v = S.top()
        S.pop()

        for all neighbours of k of v in G:
            if k is end :
                return true

            if k is not visited :
                S.push( k )
                tag k as visited

    return false
```

Complexity Analysis

In worst case, the maximum number of times the function will be the sum of nodes and edges when maximum backtracking is involved. So, Time Complexity : O(V+E)

Since we store each vertex in the recursion stack, the maximum space that could be required will be the number of nodes. Hence, Space Complexity : O(V)

V=Number of vertices, E= Number of edges

Algorithm 2

Problem Statement -

For small areas where Our Application would be frequently used we would use this algorithm to pre-save the shortest paths between all significant places in that area as cache.

Algorithm Used -

Floyd-Warshall

Explanation

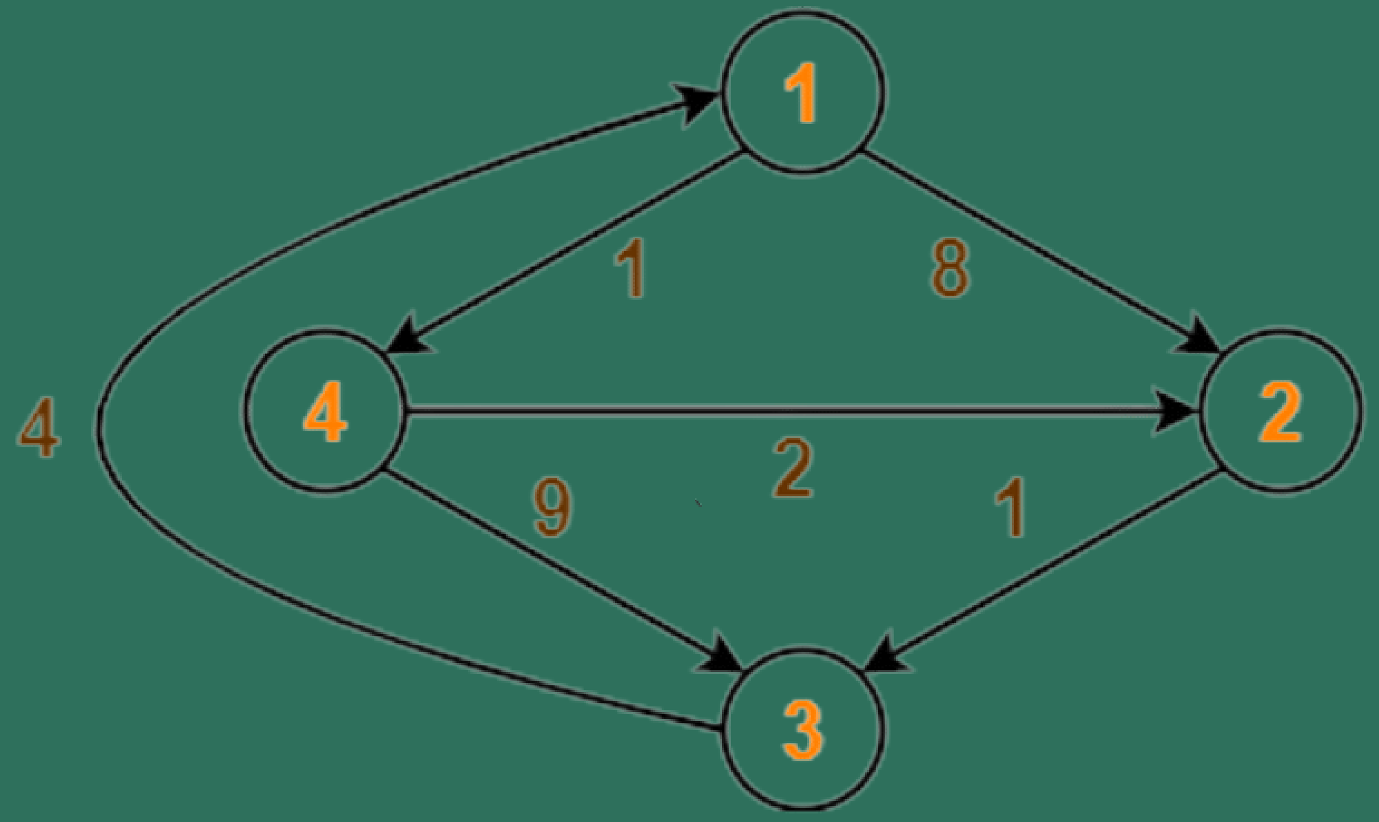
We used Floyd-Warshall cause it finds the lengths of shortest paths between all pairs of vertices in single execution, which we require as cache for areas with are very frequently visited.

n = Number of vertices

We initialize the distance matrix of n*n. Then we update the distance matrix by considering all vertices as intermediate vertex.

We update the distance matrix in incremental phases starting from k = 0 to k = n - 1. We pick all vertices one by one and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices less than k as intermediate vertices.

Sample Graph



$$= \begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & & & & \\ \hline 2 & & & & \\ \hline 3 & & & & \\ \hline 4 & & & & \\ \hline \end{array}$$

Pseudocode

```
D[n][n] # Distance Matrix

for k = 0 to n - 1
    for i = 0 to n - 1
        for j = 0 to n - 1
            D[i, j] = min(D[i, j], D[i, k] + D[k, j])

Where n is no of vertices
```

Complexity Analysis

Time Complexity : O(n * n * n) =O(n3)
As we have a 3 nested loop all runs for 0 to n - 1

Space Complexity : O(n*n) = O(n2)
As we are storing the Distance matrix which is of n * n

Algorithm 3

Problem Statement

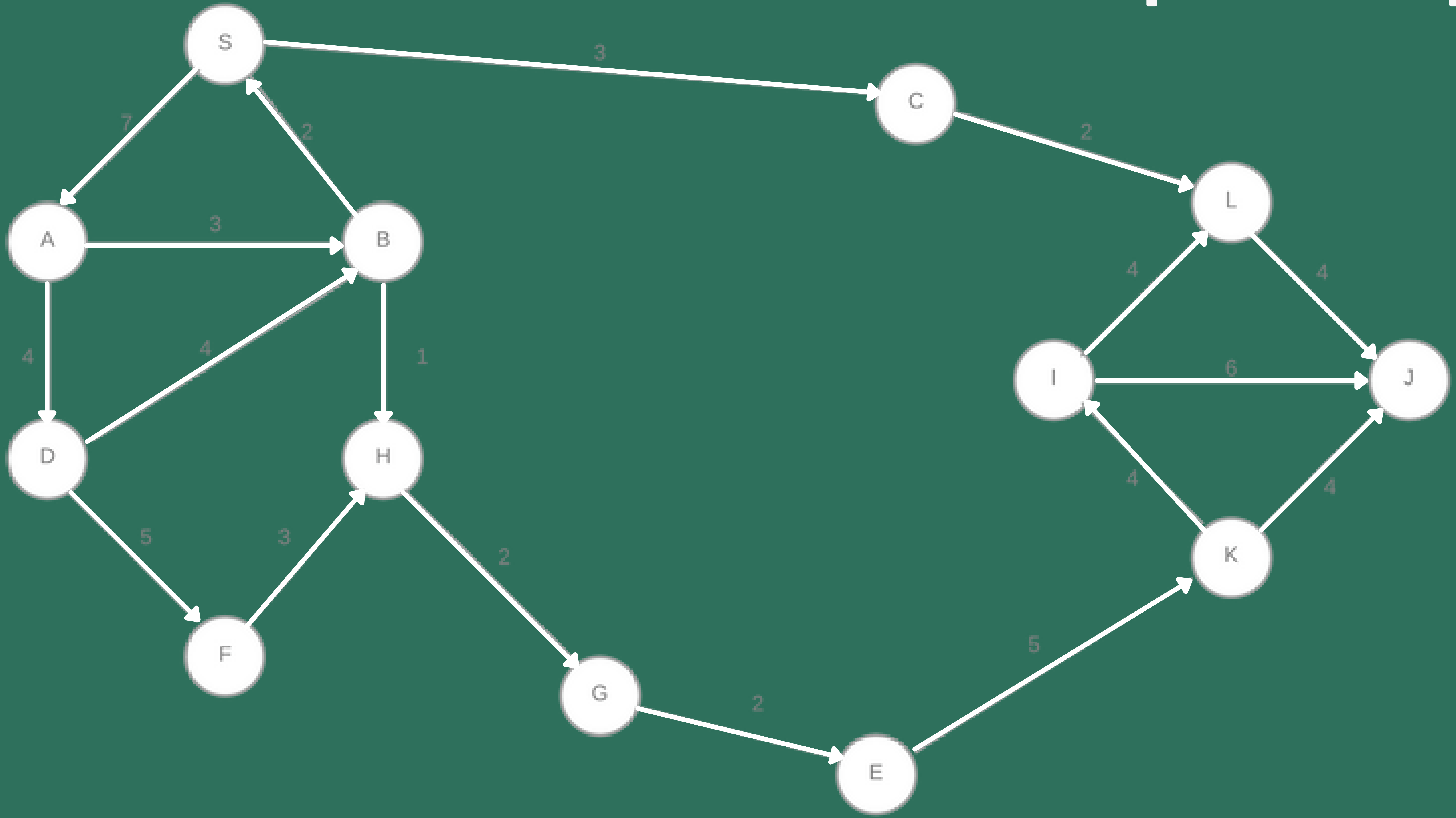
Given a starting node, we have to find if there exists any path with which we can reach to where we started. In short, we have to check if there exists any cycle in the given graph.

Algorithm Used : Cycle Detection (Depth First Search)

Explanation

1. Creating a recursive function that initializes the current vertex/index, the visited list and the recursion stack.
2. Mark the current vertex as visited and also mark the same in the recursion stack.
3. Find the vertices adjacent to the current vertex and not visited. Call the recursive function for those vertices and if the recursive function returns true, return true.
4. If any adjacent vertex is already marked true in the recursion stack then return true.
5. Create a helper/wrapper class which calls the recursive function for all vertices and if any function returns true, return true. Else if the recursive function for all vertices returns false, return false.

Sample Graph



Pseudocode

```
recur_fun{
       if(!visited[node]){
             visited[node] equals true
             recur_stack[node] equals true

             for each ‘i’ in adjacent vertices of ‘node’ :
                  if(!visited[i] and recur_fun(i, visited, recur_stack)) return true
                  else if(rec_stack[i]) return true
             }
       res_stack[node] = false
       return false
}

help_fun {
      for each vertex :
          if(recur_fun(vertex, visited, recur_stack) return true
return false
}
```

Complexity Analysis

We are basically using Depth-First-Search Algorithm to check for cycle by traversing through every vertex and edge hence,
Time Complexity : O(V+E)

And as we are storing every vertex in the recursion stack which takes the space mainly hence,
Space Complexity : O(V)

where, V = number of vertices , E = number of edges

Algorithm 4

Problem Statement -

To find the shortest path between source and destination..

Algorithm Used -

Dijkstra's algorithm

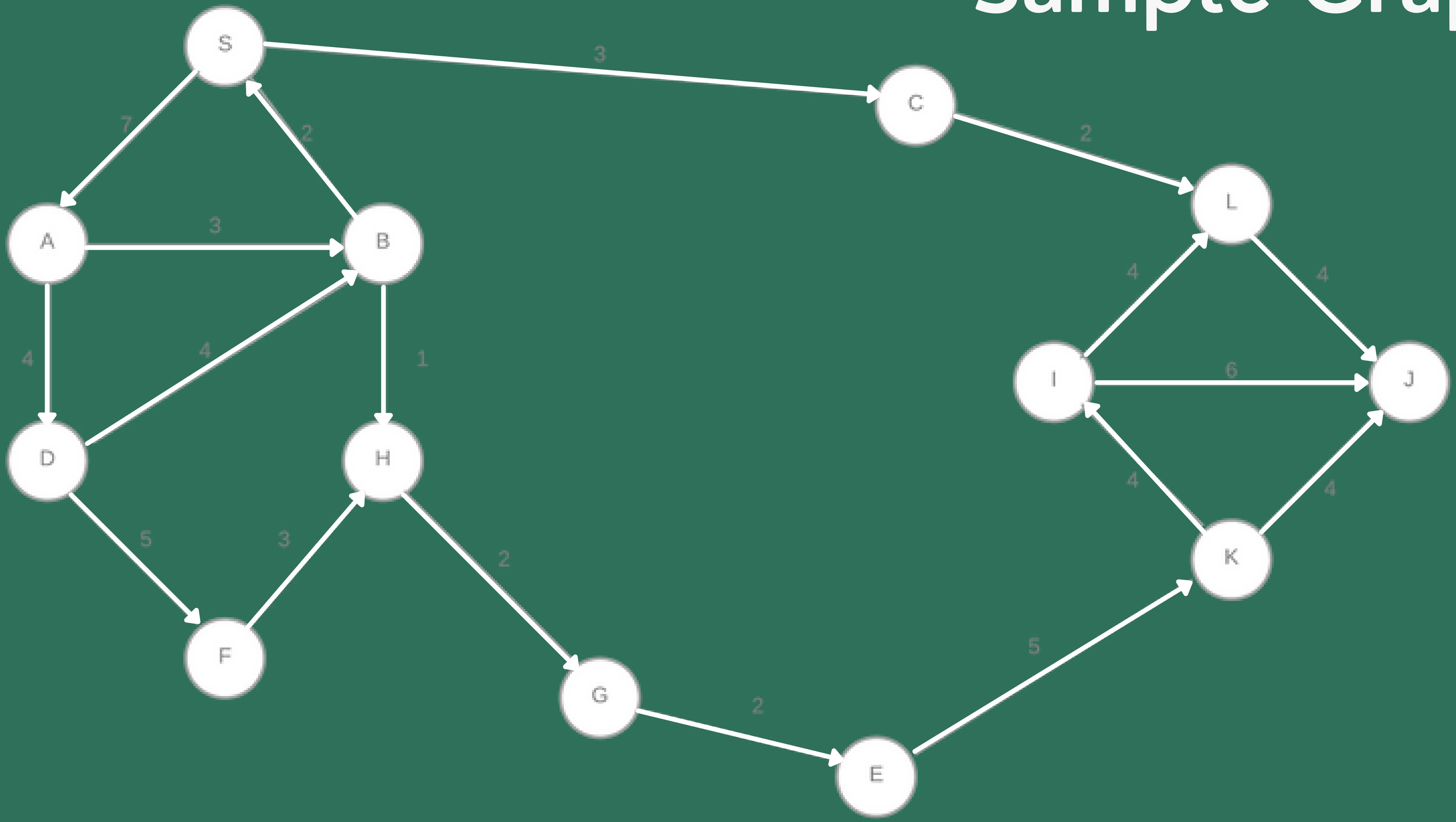
Pseudocode

```
Shortest path () {
·   Mark start
·   Mark destination
·   Result [] = { } // empty initially
·   Distance [start] = 0
·   Set distance of all other nodes to INFINITY initially
-------> while (Result != graph.size()) {
·   Pick the minimum distance vertex u from source which is not in Result []
·   Add u to Result [] and update its distance in distance array [].
·   For each adjacent vertex 'v' of 'u'
If (distance [u] + edge(u,v) < distance [v])
Distance [v] = distance [u] + edge(u,v)
   }
·   Search destination in Result [] and its corresponding distance that is distance
[destination]
```

Complexity Analysis

```
Time Complexity :O(v^2)
```

Sample Graph



Algorithm 5

Problem Statement

Given two locations (source and destination) on a map, we need to find the shortest path distance between the source and the destination . Also, we need to retrieve the path traversing which the distance is minimum.

Algorithm Used - Dijkstra's Algorithm with Path Printing

Pseudocode

```
void djikstra(int[][] adjacencyMatrix, int source, int destination){
        int n = adjacencyMatrix[0].length
        int[] distance = new int[n];
        boolean[] inserted = new boolean[n]
        for(vertexIndex = 0 to n-1){
            distance[vertexIndex] = Integer.MAX_VALUE
            inserted[vertexIndex] = false
        }
        distance[source] = 0
        int[] parent = new int[n]
        parent[source] = -1
        for(i = 1 to n-1) {
            int closestVertex = -1
            int minDist = Integer.MAX_VALUE
            for(vertexIndex = 0 to n-1){
                if(!inserted[vertexIndex] && minDist< distance[vertexIndex]{
                    closestVertex = vertexIndex
                    minDist = distance[vertexIndex]
                }
            }
            inserted[closestVertex] = true
            for(vertexIndex = 0 to n-1){
                    int edgeDist = adjacencyMatrix[closestVertex][vertexIndex]
                            if(edgeDist > 0 && (minDist + edgeDist) < distance[vertexIndex]){
                                        parent[vertexIndex] = closestVertex
                                        distance[vertexIndex] = minDist+edgeDist
                            }
            }
            printPath(int destination, int[] parent)
      }
}

//helper method to print the path
void printPath(int current, int[] parent){
           //base case- source node has been processed
           if(curr == -1) {return}
           printPath(parent[curr], parent)
           System.out. println(curr + " ");
}
```

Complexity Analysis

```
Let the number of vertices in the graph be ‘n’,
The outer loop runs for 1 to n-1,
For each iteration of outer loop , inner loops run
from n times.
Total number of iterations = O((n-1)*n)
     = O(n^2)

Time Complexity : O(V^2)
where V is the number of vertices in graph
```

Algorithm 6

Problem Statement

Given two locations (u and v) on a map, we need to find the minimum distance path between them. This should be done considering the fact that the electric vehicle is discharging at a constant rate and needs be charged again during the trip. The output should return such a path.

Algorithm Used - Dijkstra's Algorithm Path Printing and DFS for path exsistence

There are 2 algorithms on which the final algorithm is based. First algorithm is DFS which is employed in order to check if the path exists or not. The second algorithm is based on Dijkstra Algorithm whose implementation is done through min priority queue.
The basics of this algorithm is that we first find the shortest path in the graph , then we calculate if we can traverse from one vertex to other. If we can travel the whole distance without recharge , we directly travel (with 10% of the battery remaining as buffer). If we cannot travel the whole distance then we go to the furthermost vertex of the shortest path and back track from that vertex to source to check if we can find a nearest charging station using dijkstra and if it is feasible to reach there. If we find such a station, then we charge the battery and use dijkstra again towards destination. If the whole distance is backtracked , then we return output as null .

Pseudocode

```
bool possible =PathexistenceBool(source, destination);
if(possible==true){
   Q == Min Priority Queue
   S == set of vertices whose final shortest weigths from source s have already been
      determined
 Initially, S = φ
 //source and destination are vertices
       Q.enqueue(source);
        U = null;
        charge = initialCharge;
        ChargingVis[]
         while(Q.top()!=destination)
 {
                U = extractMinFrom(Q);
                S = S <union> {U};
```

Pseudocode

```
for each vertex v ∈ G.Adj[u]{
    //relax(u,v,w)
   if(distance[v]> distance [u] + edge(u,v)){
    Distance[v]= distance [u] + edge(u,v);
    Pathvia[v]= u;
   }
 Enqueue v (wrt priority to min distance)
 }
U.charge = Initalcharge- minDistanceofU*mileage;
  if (charge <10% of max) {
        u= Pathvia[u];
 //define a linked list for directions to nearest charging station
       List = thisMethod(Graph, U, chargingStation);
       ChargingVis[ChargingStation ]++;
       If ChargingVis[ChargingStation ]>2
       {break; }
       Else
       List = List+ thisMethod(Graph, chargingStation, destination);
     }
 }
```

Pseudocode

```
 node = destination;
 path == LinkedList<nodes>
 while (pathvia(node) != null) {
    path.add(node);
    node = pathvia[node];
    }
 Path.add(source);
  return path;
}
else return null;
```

Complexity Analysis

Worst Case: If the path through the Dijkstra algorithm contains all the vertex of the graph and for each vertex, we apply Dijkstra to find the charging station so V vertex all multiplied by the complexity of the Dijkstra algorithm

Time Complexity:

O(V*(V+ElogV))

Space Complexity

O(V^2)

Algorithm 7

Problem Statement

Given a source and a destination, we need to find 3 shortest paths to reach the destination.

Algorithm Used

Yen's k shortest paths algorithm

Explanation

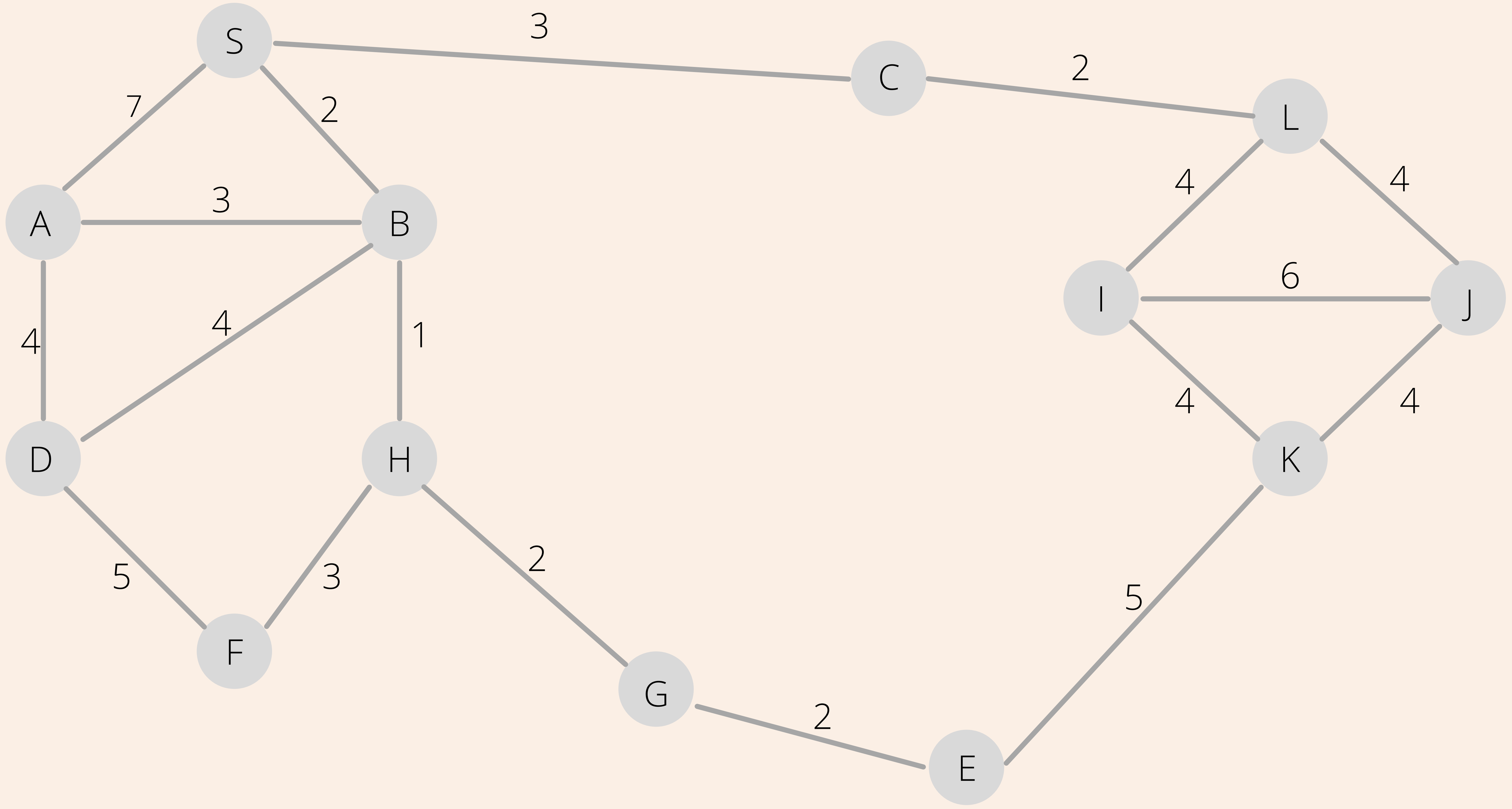
1. The algorithm finds the shortest path using Dijkstra's algorithm.
2. From first shortest path, we remove one edge, one by one, and then we find different possible paths.
3. We select the shortest path from the above possible paths, that is second shortest path.
4. We repeat step 2 and 3, but this time removing edges from second shortest path, and we get the third shortest path to reach the destination.

Pseudocode

```
function kShortestPaths(Graph, source, sink, K):
  A[0] = Dijkstra(Graph, source, sink);
  B = [];

  for k from 1 to K:
    for i from 0 to size(A[k − 1]) − 2:

      spurNode = A[k-1].node(i);
      rootPath = A[k-1].nodes(0, i);

      for each path p in A:
        if rootPath == p.nodes(0, i):
          remove p.edge(i,i + 1) from Graph;

      for each node rootPathNode in rootPath except spurNode:
        remove rootPathNode from Graph;

      spurPath = Dijkstra(Graph, spurNode, sink);
}
```

Pseudocode

```
totalPath = rootPath + spurPath;
      if (totalPath not in B):
        B.append(totalPath);

      restore edges to Graph;
      restore nodes in rootPath to Graph;

    if B is empty:
      break;
    B.sort();
    A[k] = B[0];
    B.pop();

  return A;
```


S
C
L
A
B
I
J
D
H
K
F
G
E
3
2
7
2
3
4
4
6
4
1
4
4
4
5
3
2
5
2

Complexity Analysis

The algorithm uses Dijkstra's Algorithm as base algorithm,
Time Complexity of Dijkstra's Algorithm => O(M + NlogN)

Yen's Algorithm makes K*l calls to Dijkstra's Algorithm,
and in a condensed graph l = O(N), hence,

Time Complexity => O(KN(M+NlogN))

Algorithm 8

Problem Statement

Given two locations (u and v) on a map, we need to find the minimum amount of toll amount need to be paid while travelling from u to v. Also, we need to retrieve the path on which the toll amount is minimum.

Algorithm Used - Bellman - Ford Algorithm

Graph with the toll amount of each edge will be given. Input will be the source and destination nodes. Two auxiliary arrays minAmount and parent are created to store the minimum toll and parent node at which toll is minimized.

Explanation

1. Firstly, update the minAmount of source to 0, and others to infinity.
2. Run a loop for V – 1 times, where V is number of vertices in the graph, and relax each edge in every iteration i.e.
   - => Run a loop on each edge and do the following:
     - =>For edge (u, v), if minAmount of u from source plus the toll amount of edge (u,v) is lesser than the minAmount of v from source, then update the minAmount of v from source with minAmount of u from source plus the toll amount of edge (u,v). Also, update the parent of node v to be node u as the minAmount of node v is reached via the path that has node u as node v's parent.
3. Return the minAmount of the destination node if it is not infinity, and print the minimum amount path using another function described below. If minAmount of destination node is infinity, then there is no path between source and destination.

Pseudocode

```
void bellmanford(source, destination){
        //this array stores minimum toll amount for each vertex
        minAmount[] = new int[V];
        //this array stores parent for each node
        parent[] = new Node[V];
        //V is number of nodes in graph
        for(i = 0 to V - 1){
                flag = false;
                for(each edge j between node u and v in graph){
                        if(minAmount[u] < infinity){
                                if(minAmount[v] > minAmount[u] + tollAmount(j)){
                                        minAmount[v] = minAmount[u] + tollAmount(j);
                                        parent[v] = u;
                                        flag = true;
                                }
                        }
                }
                if(!flag) break;
        }
}
```

Complexity Analysis

Since there are E edges and V vertices in the graph and each edge is relaxed (V - 1) times, hence the time complexity is simply O(V.E). Since, in a complete graph E <= V^2, hence the time complexity can also be written as O(V^3).
The algorithm uses two arrays each of size equal to number of vertices. Hence, space complexity is O(2V) = O(V)
Time Complexity : O(V^3)
Space Complexity: O(V)

Algorithm 9

Problem Statement -

The algorithm will be used in our application to locate the shortest path for the nearest emergency location like police station, hospital etc.

Algorithm Used -

Dijkstra's algorithm

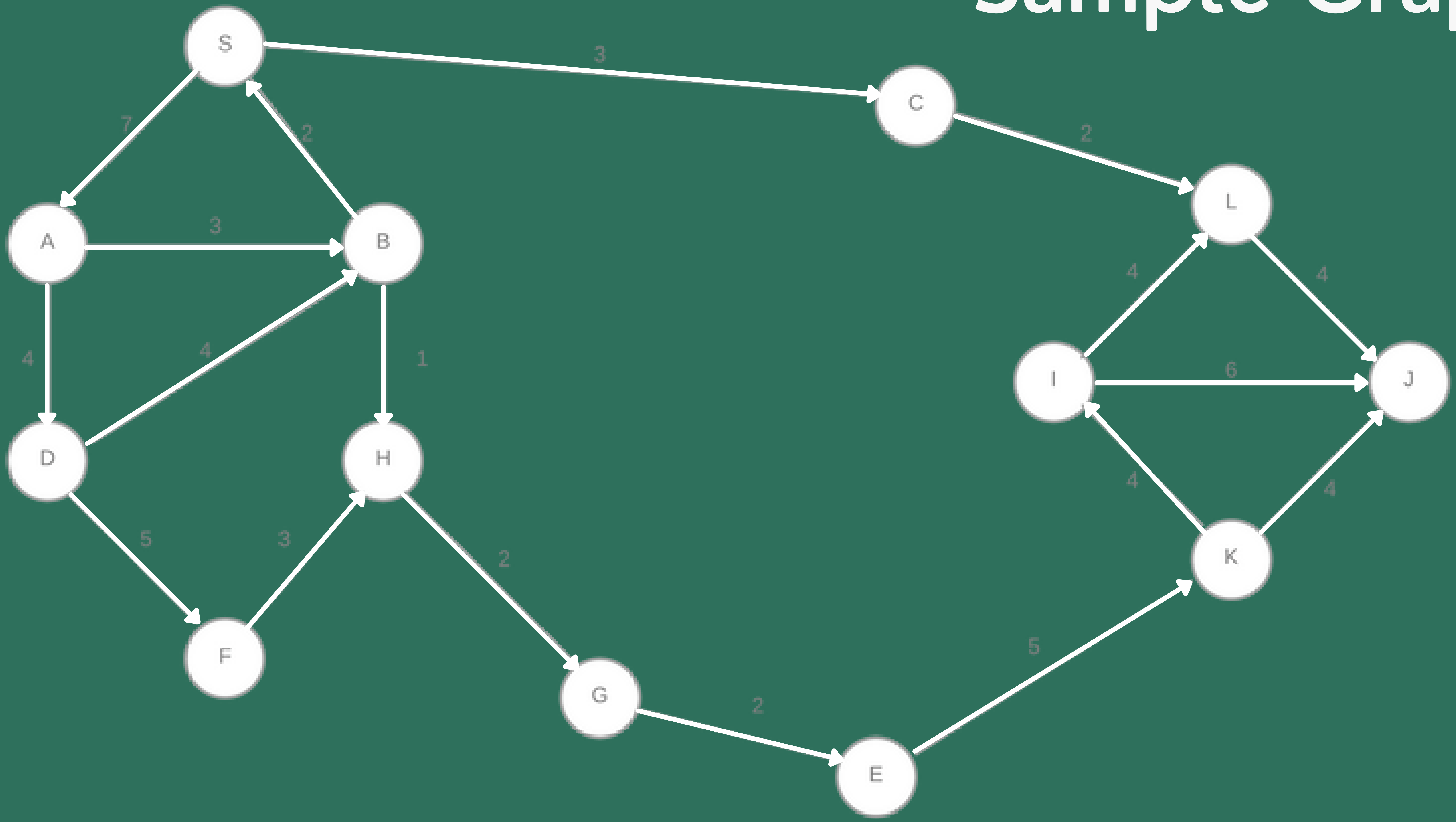
Pseudocode

```
v = no of vertex

   Dist[v] = [INF, INF, ...]
        dist[src] = 0
        sptSet[v] = [False, False, ...]

        for i = 0 to v:

          min = INF
  for u in v:
   if dist[u] < min and sptSet[u] = false and
    min = dist[u]
    x = u


          # Put the minimum distance vertex in the
          # shortest path treesptSet[x] = True
          # Update dist value of the adjacent vertices
          # of the picked vertex only if the current
          # distance is greater than new distance and
          # the vertex in not in the shortest path tree
          for y = 0 to V - 1:
           if graph[x][y] > 0 and sptSet[y] == False and \
                dist[y] > dist[x] + graph[x][y]:
                        dist[y] = dist[x] + self.graph[x][y]

  min_distance = INF
  nearest_emergency_stop = -1
  for node = 0 to v - 1:
   if min_distance > dist[min_distance]:
    min_distance = dist[min_distance]
    nearest_emergency_stop = node
  return nearest_emergency_stop
```

Complexity Analysis

```
Time Complexity : O(V^2)
```

Sample Graph



Algorithm 10

Problem Statement -

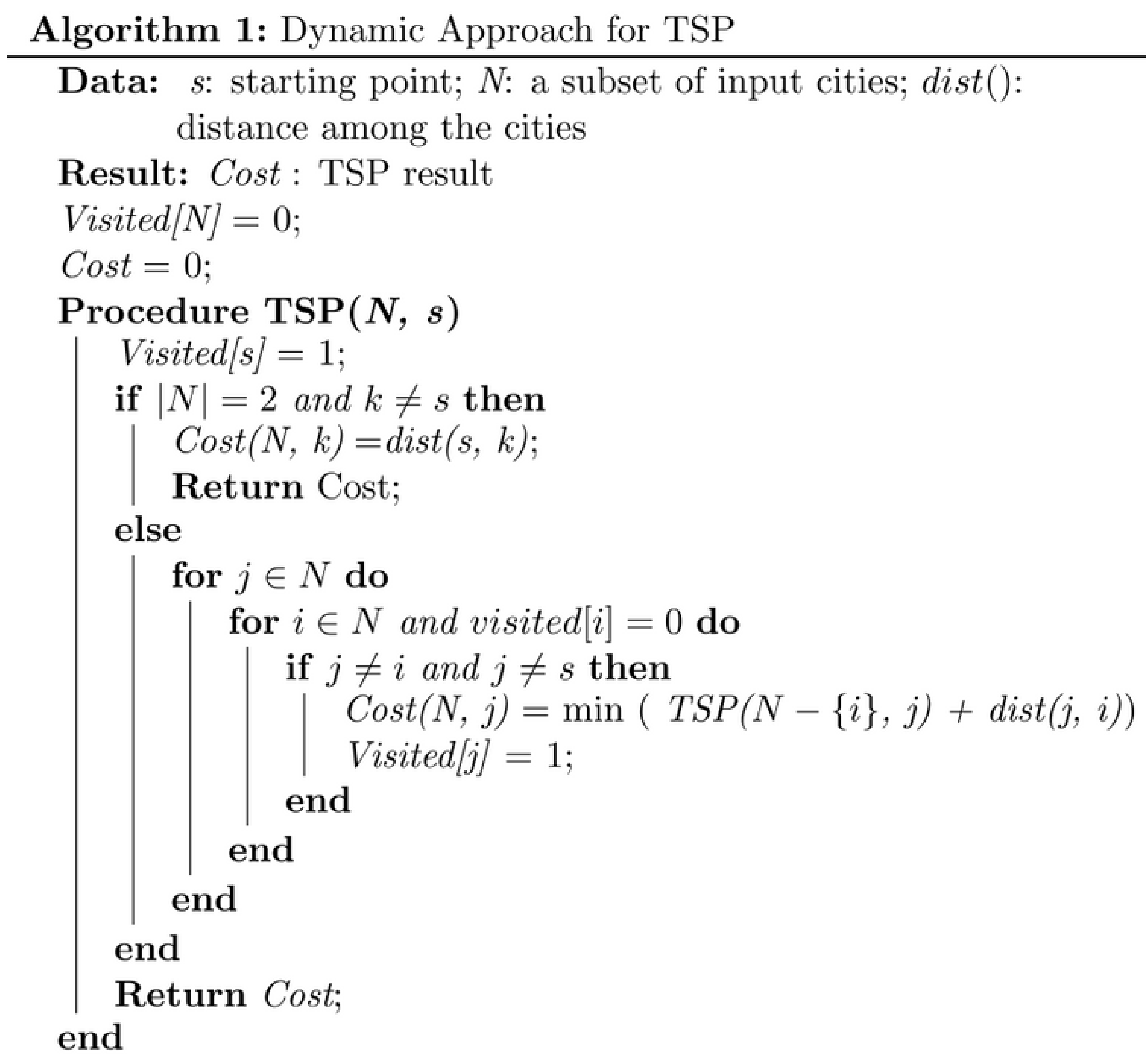
Suppose we have a salesperson who lives in City 1. He wakes up every day in the morning, and has (N - 1) cities to visit, named:
City 2, City 3, City 4 ... City (N - 1) & City N
to sell his product. At the end of the day, he also has to go back to his home i.e. to City 1.

Therefore, the problem is to find the Hamiltonian cycle that has the least cost associated with it.

Algorithm Used -

Dynamic Programming (Top down Approch)

Pseudocode



Algorithm 1: Dynamic Approach for TSP

```
Data: $s$: starting point; $N$: a subset of input cities; $dist()$:
      distance among the cities
Result: $Cost$ : TSP result
$Visited[N] = 0$;
$Cost = 0$;
Procedure TSP($N$, $s$)
    $Visited[s] = 1$;
    if $|N| = 2$ and $k \neq s$ then
        $Cost(N, k) = dist(s, k)$;
        Return Cost;
    else
        for $j \in N$ do
            for $i \in N$ and $visited[i] = 0$ do
                if $j \neq i$ and $j \neq s$ then
                    $Cost(N, j) = \min\ (\ TSP(N - \{i\}, j) + dist(j, i))$
                    $Visited[j] = 1$;
                end
            end
        end
    end
    Return $Cost$;
end
```

Complexity Analysis

We have the recursive equation:
T(i, S) = min((i, j) + T(j, S - {j})); S ≠ Ø ; j € S ;
T(i, S) = (i, 1); S=Ø (Base condition)

Here, T (i, S) means we are travelling from a vertex "i" and have to visit a set of non-visited vertices "S" and have to go back to vertex 1 (let us start our journey from vertex 1).
(i, j) here is the cost of the path from node i to node j.
After analysing the recursion tree of the equation, we find that we have a total of (n-1)·2^(n-2) unique sub-problems. ∴ to store all values, the **space complexity** required is **O(n·2^n)**

Solving each sub-problem might take O(n) operations in the worst case, ∴ the worst-case **time complexity** will be O(n·2^n)·O(n) = **O(n^2·2^n)**

Dry Run of algorithm

T (1, {2,3,4}) = minimum of

= (1,2) + T (2, {3,4})

= (1,3) + T (3, {2,4})

= (1,4) + T (4, {2,3})

T (2, {3,4}) = minimum of

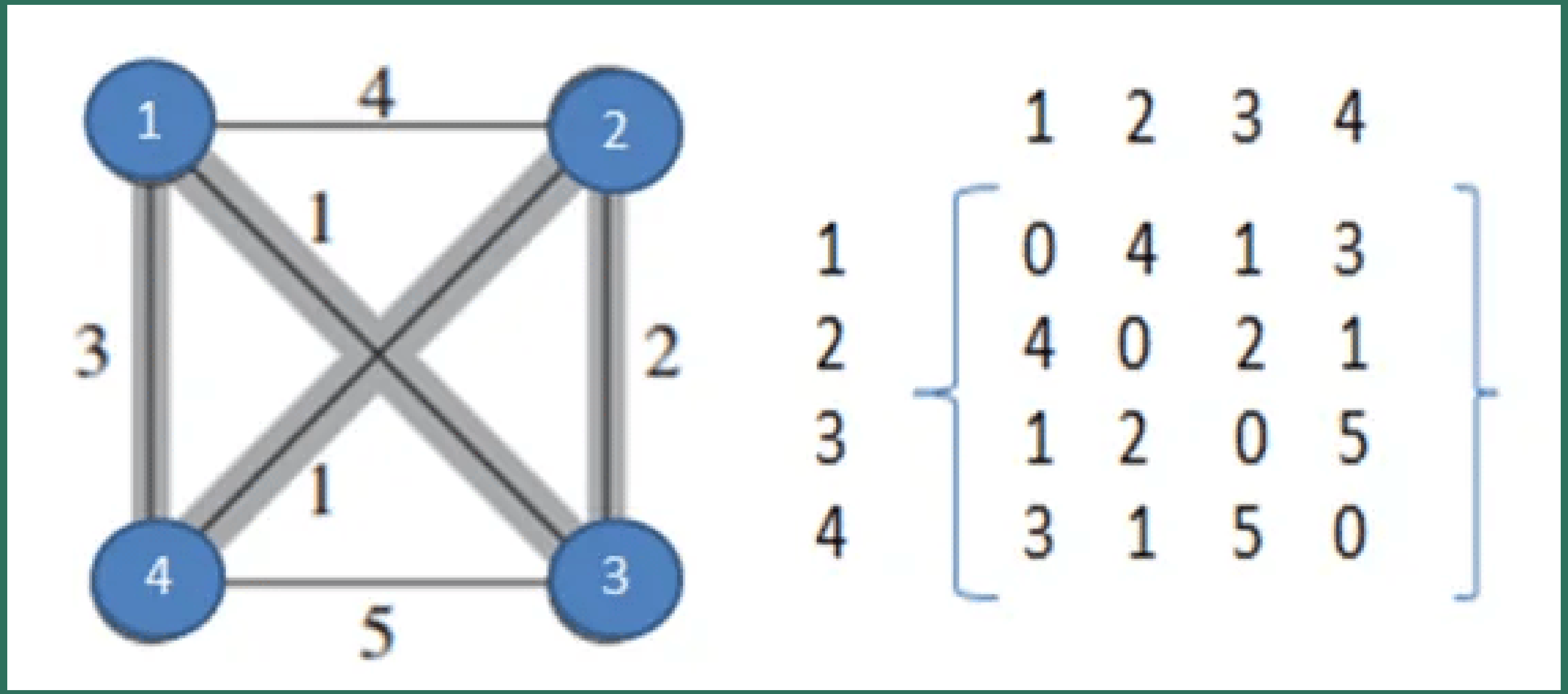
= (2,3) + T (3, {4})

= (2,4) + T (4, {3})

T (3, {4}) = (3,4) + T (4, {})

T (4, {}) = (4, 1) = 3

T(2, {3, 4}) = 7; T(3, {2, 4}) = 6;
T(4, {2, 3}) = 4



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 4 & 1 & 3 \\ 2 & 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 0 & 5 \\ 4 & 3 & 1 & 5 & 0 \end{array}$$

Like this, we continue until we exhaust storing the value of all unique sub-problems and finally find the answer to our original problem

Thank You