Project Report

Submitted by	Submitted to
Nafiz Khan Tasnul ID: 2020-2-60-175	Redwan Ahmed Rizvee Lecturer, Department of Computer Science and Engineering, East West University
Md Sabit Hossen ID: 2020-1-60-046	
Tanvir Hasan ID: 2020-3-60-067	

Semester: Fall 2022; **Course title**: Algorithms;
Course code: CSE246; **Sec**: 05
Group: 7;
Submission Date: 27 December, 2022

Project Title

Problem ID: 11

<u>Multiple Shortest Path Printing</u>

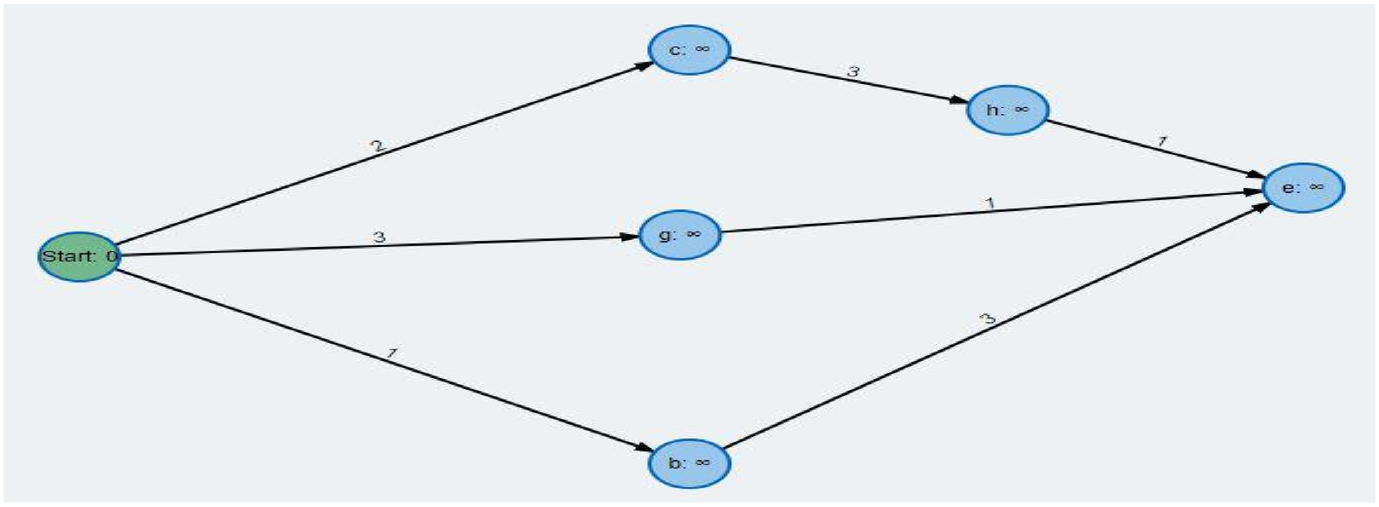
Problem Statement

In this problem, we will have to apply Dijktstra's algorithm(modified) on a given directed weighted graph containing no negative cycles. We will input a source node S from which we can calculate the distance to another destination node V which will also be used as an input. The edge weight denotes the "to be covered" distance between two nodes by traversing through that edge.
We also need to find the path that will help us to reach the destination. If there are multiple paths, to reach a destination, we have to print the number of shortest paths available.

Algorithm Discussion

In this problem, our algorithm's base is Dijkstra's single source shortest path algorithm. In Dijkstra's algorithm, from a given source and destination it returns the minimum cost to reach the destination.
Our algorithm is almost the same, it too returns the minimum cost to reach the destination except if there are multiple paths to the destination, it is printed as well.

Let us simulate a graph that has multiple paths of same cost to destination(e),



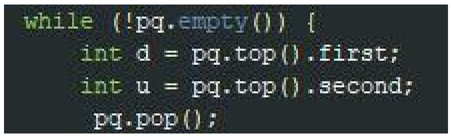
Priority queue:

a

In the graph above, it has 6 vertices and 7 edges. Our destination is the vertex ‘e’ and start is given. Here,

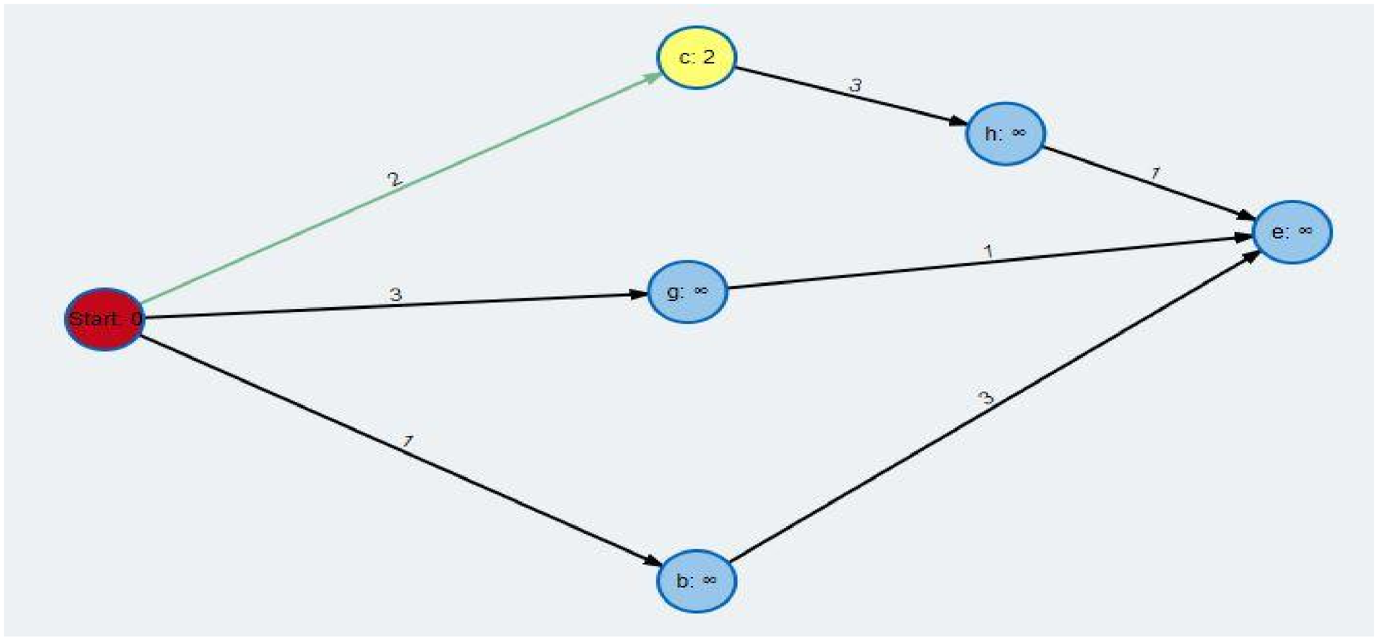
a = node 1	b = 4
c = 2	h = 5
g = 3	e = 6

At first all the vertices' distance from the source is set to infinity. Starting node has been added to the queue.

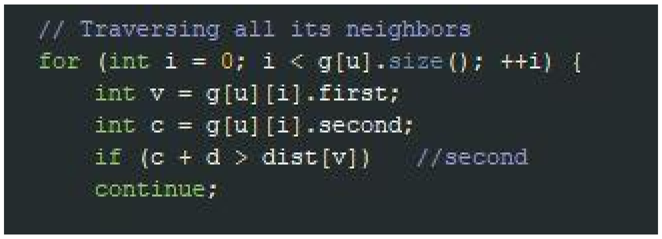


```
while (!pq.empty()) {
    int d = pq.top().first;
    int u = pq.top().second;
     pq.pop();
```

We perform this action on the first element of the queue.

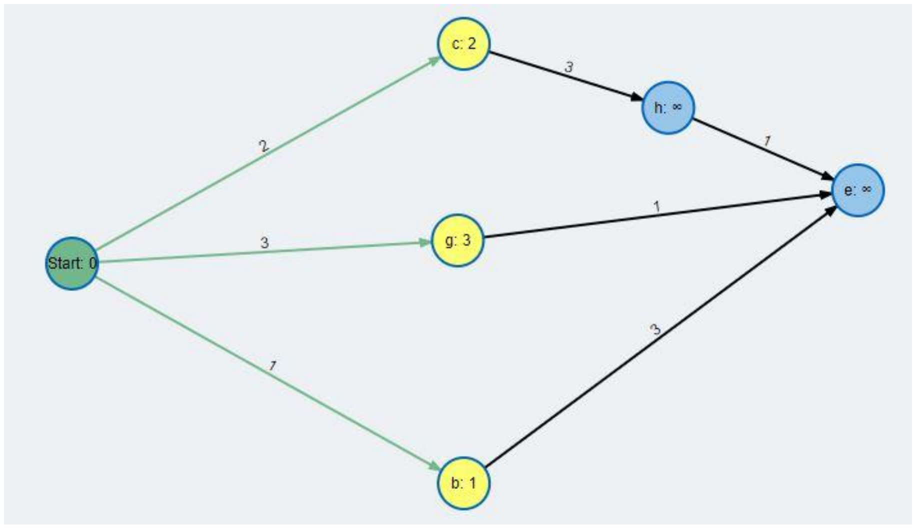


We are traversing the neighbors of the node we popped from the queue earlier and each time the newly discovered neighbor is added to the queue



```
// Traversing all its neighbors
for (int i = 0; i < g[u].size(); ++i) {
    int v = g[u][i].first;
    int c = g[u][i].second;
    if (c + d > dist[v])   //second
    continue;
```

The picture above shows the part carrying out traversal.

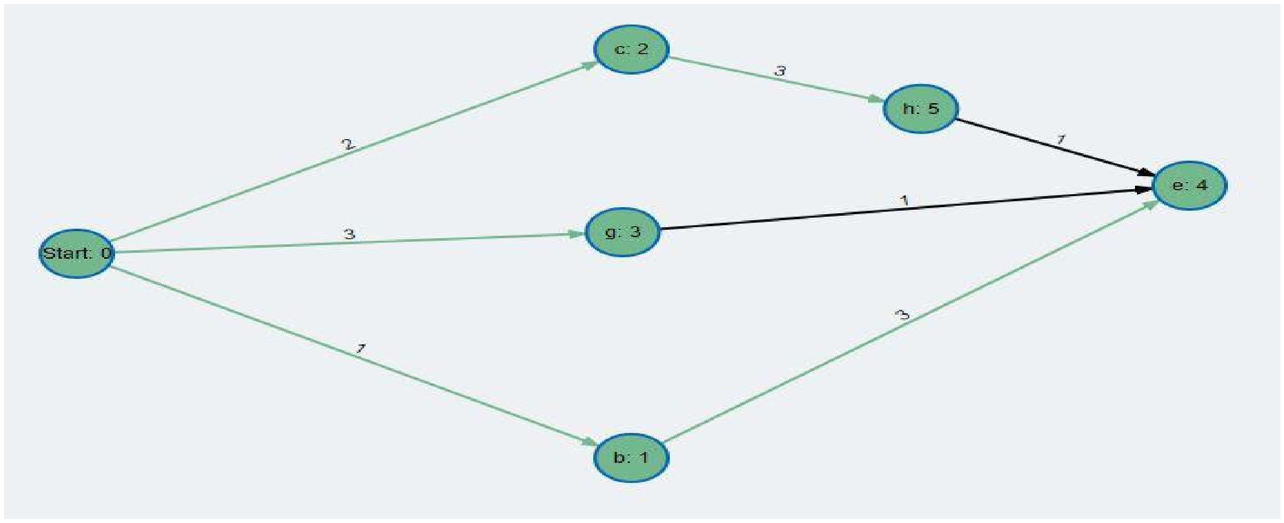


Yellow refers to nodes that are waiting in the queue.

Priority queue:

b	c	g

We are going to perform the operations as before and finally the graph we get is ,



We can see the lowest cost to reach the destination is 4.

Complexity Analysis

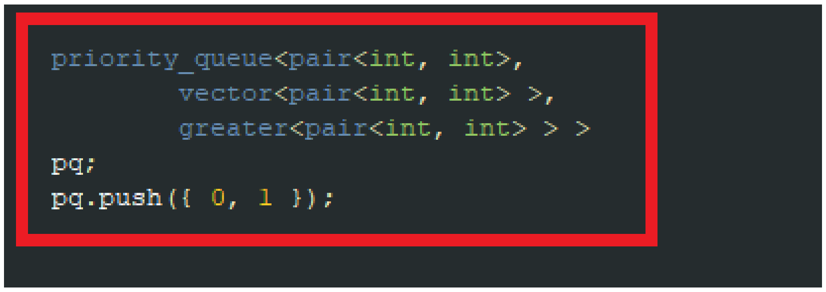
The space complexity is determined by the amount of memory used by the algorithm as it executes. The space complexity of Dijkstra's algorithm is typically O(n), where n is the number of nodes in the graph.

As for the time complexity, the overall running time of this modified implementation of Dijkstra algorithm is **O((V + E) log V).**

The graph is stored using an adjacency list, which has a time complexity of **O(E)** to set up.

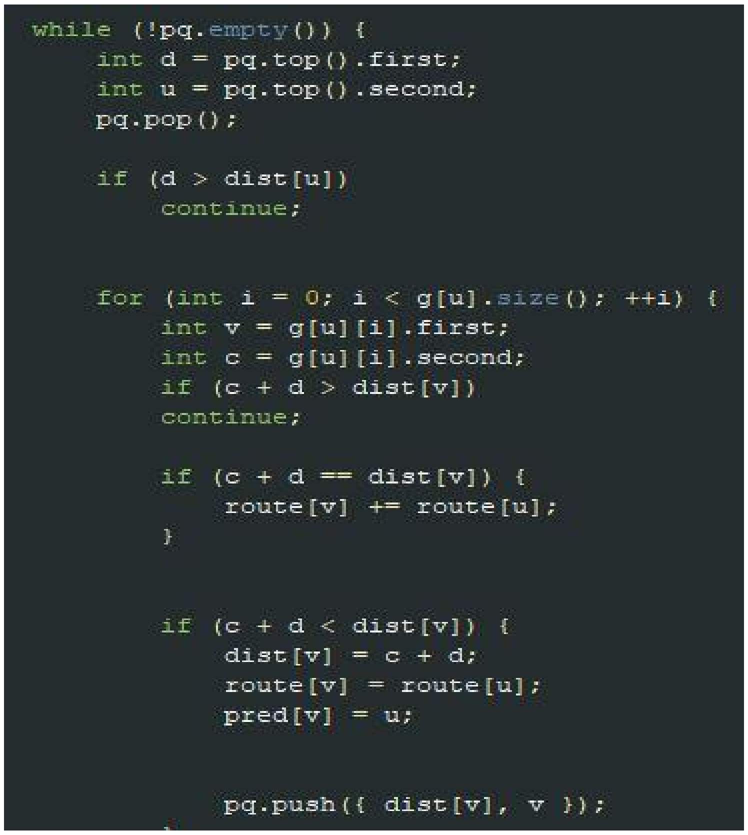
We are using priority queue which to store the nodes that needs to be visited, the complexity of this part **O(logV)**

Selecting a node and then traversing all its neighbors takes up **O(V+E)** time complexity.



```
priority_queue<pair<int, int>,
        vector<pair<int, int> >,
        greater<pair<int, int> > >
pq;
pq.push({ 0, 1 });
```

This part is where we are initializing our priority queue. The time complexity of adding an element to the priority queue is **O(logV)**, here V is the number of nodes in the graph.



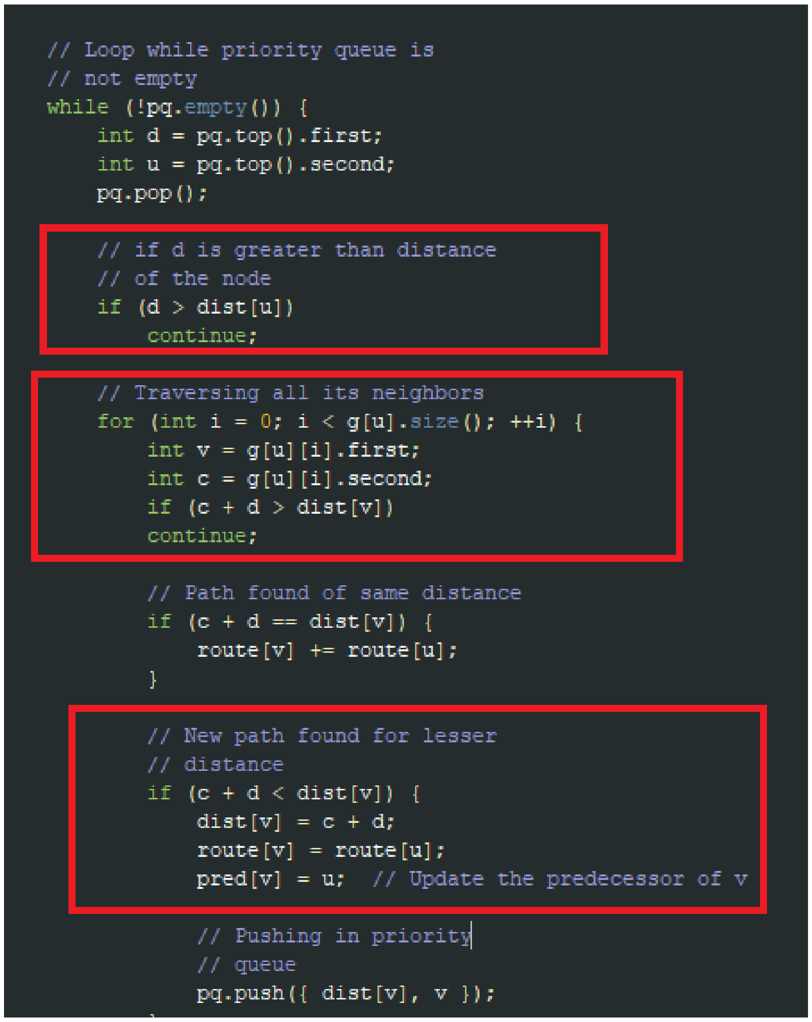
```
while (!pq.empty()) {
    int d = pq.top().first;
    int u = pq.top().second;
    pq.pop();

    if (d > dist[u])
        continue;


    for (int i = 0; i < g[u].size(); ++i) {
        int v = g[u][i].first;
        int c = g[u][i].second;
        if (c + d > dist[v])
        continue;

        if (c + d == dist[v]) {
            route[v] += route[u];
        }


        if (c + d < dist[v]) {
            dist[v] = c + d;
            route[v] = route[u];
            pred[v] = u;


            pq.push({ dist[v], v });
```

The complexity of this part is **O(V + E)**.
The while loop that processes each node and its neighbors. This loop runs once for each node in the graph, so the time complexity is O(V). For each time when the loop iterates, the current node's neighbors are visited, which takes time complexity of O(E). Therefore, the overall time complexity of this part is **O(V + E)**.

The implementation uses adjacency list to store the graph therefore the overall complexity remains **O((V + E) log V).**

Implementation

- **Calculating the number of shortest paths existence:**



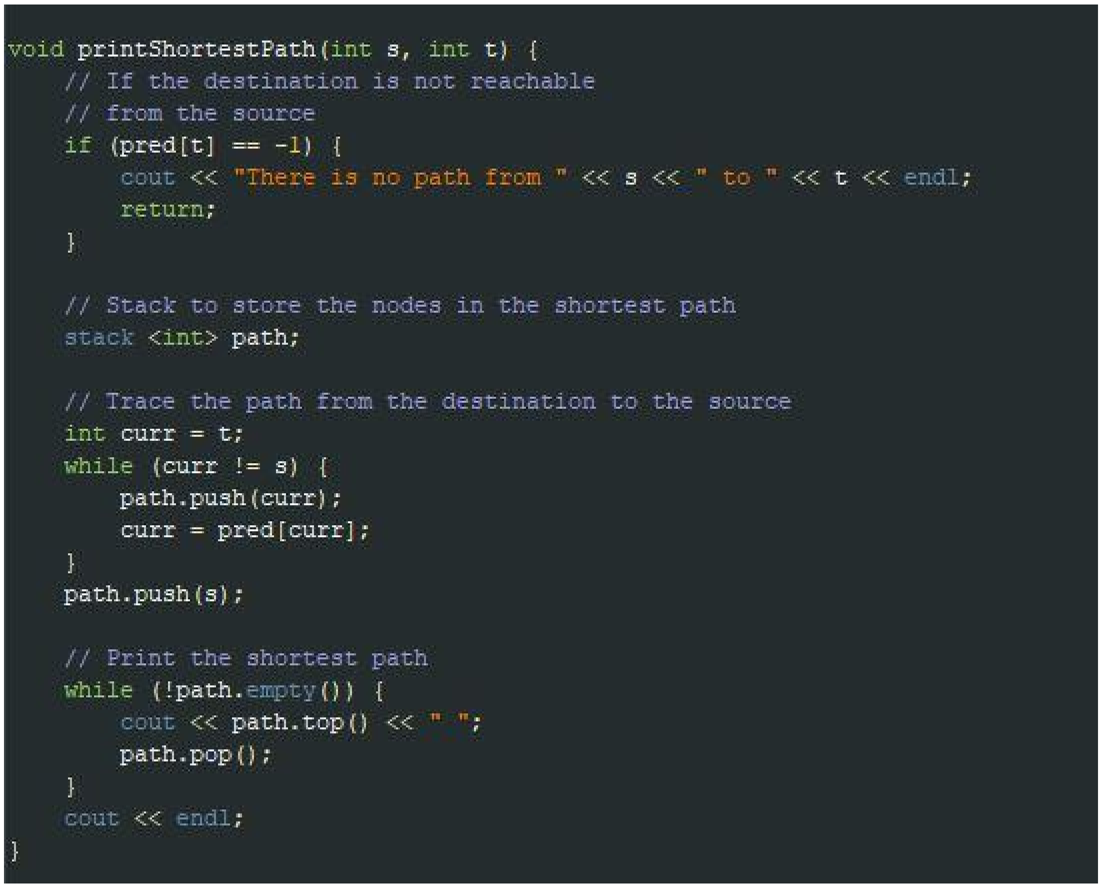
```
// Loop while priority queue is
// not empty
while (!pq.empty()) {
    int d = pq.top().first;
    int u = pq.top().second;
    pq.pop();

    // if d is greater than distance
    // of the node
    if (d > dist[u])
        continue;

    // Traversing all its neighbors
    for (int i = 0; i < g[u].size(); ++i) {
        int v = g[u][i].first;
        int c = g[u][i].second;
        if (c + d > dist[v])
            continue;

        // Path found of same distance
        if (c + d == dist[v]) {
            route[v] += route[u];
        }

        // New path found for lesser
        // distance
        if (c + d < dist[v]) {
            dist[v] = c + d;
            route[v] = route[u];
            pred[v] = u;  // Update the predecessor of v

            // Pushing in priority
            // queue
            pq.push({ dist[v], v });
```

Here, the **first red block** (d>dist[u]) is a check to see if the distance d of a node 'u' is greater than the current known shortest distance to that node. This part also avoids revisiting a node and from updating to a larger value.

The **second red block**, the neighbors of node 'u' is being traversed with the help of a loop.
The expression (c + d > dist[v]) is a check to see if the distance to a node v can be updated. Here c denotes the weight of the processing edge.

And finally the **third red block**, the vector pred is used to track the nodes used to reach the destination from the source node. Which can also be used to print the route from source to destination node. This is also the part where relaxation is being performed, which makes it the most important part of the function.

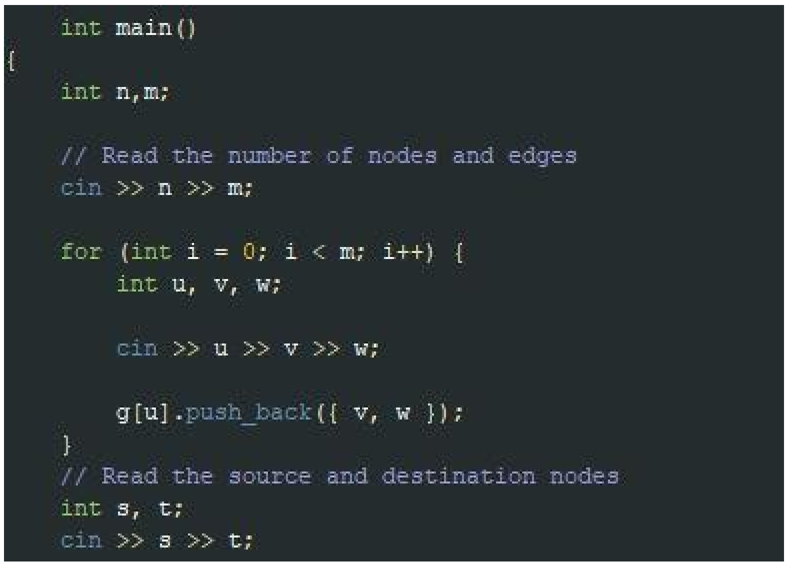
Printing the Path :



```
void printShortestPath(int s, int t) {
    // If the destination is not reachable
    // from the source
    if (pred[t] == -1) {
        cout << "There is no path from " << s << " to " << t << endl;
        return;
    }

    // Stack to store the nodes in the shortest path
    stack <int> path;

    // Trace the path from the destination to the source
    int curr = t;
    while (curr != s) {
        path.push(curr);
        curr = pred[curr];
    }
    path.push(s);

    // Print the shortest path
    while (!path.empty()) {
        cout << path.top() << " ";
        path.pop();
    }
    cout << endl;
}
```

This function takes source and destination as perimeter and prints the path/route used to reach the destination from the given source.

Storing the graph :



```
    int main()
{
    int n,m;

    // Read the number of nodes and edges
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int u, v, w;

        cin >> u >> v >> w;

        g[u].push_back({ v, w });
    }
    // Read the source and destination nodes
    int s, t;
    cin >> s >> t;
```

Sample Input	Output

4 5
1 4 5
1 2 4
2 4 5
1 3 2
3 4 3
1 4
	2
	1 4
	The cost of the shortest path is: 5

6 7
1 2 2
2 5 3
5 6 1
1 4 1
1 3 1
3 6 3
4 6 3
1 6
	2
	1 3 6
	The cost of the shortest path is: 4

- The first line of input is the number of vertices and edges
- The last line of the input is the source and destination
- The lines in between first and last contain the information of the edges.

- The first line of the output prints the number of shortest paths that exist from source to destination of the same cost.
- The second line prints the path (Incomplete)
- And the last line prints the cost to reach the destination.

Applications

1. **Route Calculation**: If we consider,
(i) Path=Edge
(ii) Source/Destination=Node
(iii) Time/Distance=Weight
Then the Shortest path algorithm can be used to find out the minimum time/distance for traversing the best path in map, vehicle navigation and airplane navigation.

2. **Map**: There may be many paths on the map to go from source to destination. Randomly selecting any path may take us more time and distance to cover. In this case a convenient route can be found with the help of weighted shortest path on google or any other maps.

3. **Vehicle Navigation**: Car Navigation Systems are sometimes offered as a special feature on new cars. These systems are capable of performing some of the tasks traditionally performed by drivers, such as determining the best route to the destination. This process of finding the shortest path from one point to another, is called routing.

4. **Airplane Navigation**: In airplane navigation, the Shortest path algorithm is used for multiple plane traffic control and for short distance and short time travel.

5. **In a telephone network**: Telephone Network uses Circuit Switching. With the advancement of technology, there comes a feature to carry data in addition to voice.

Today’s network is both analog and digital. Digital way uses the shortest path algorithm for efficient data communication.