G-35 : Print Shortest Path – Dijkstra's Algorithm

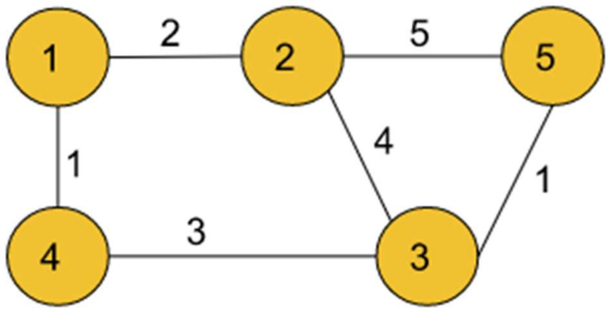
Problem Statement:

You are given a weighted undirected graph having **n+1** vertices numbered from 0 to n and **m** edges describing there are edges between a to b with some weight, find the shortest path between the vertex 1 and the vertex n, and if the path does not exist then return a list consisting of only **-1**.

Note: Please read the G-32 and the G-33 article before reading this article to get a clear understanding of Dijkstra's Algorithm will form the base for this particular problem.

Examples:

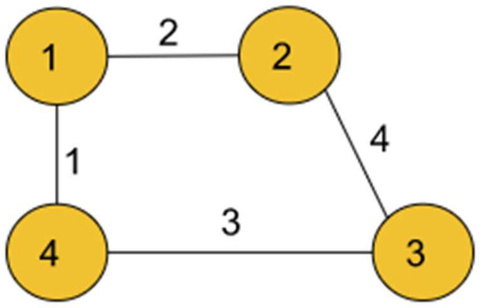
Example 1:



```
Input:

n = 5, m= 6

edges = [[1,2,2], [2,5,5], [2,3,4], [1,4,1],[4,3,3],[3,5,1]]

Output:

1 4 3 5

Explanation:

The source vertex is 1. Hence, the shortest distance path

of node 5 from the source will be 1->4->3->5 as this is
```

```
the path with a minimum sum of edge weights from source

to destination.
```

Example 2:



```
Input:

V = 4, E = 4

edges = [[1,2,2], [2,3,4], [1,4,1],[4,3,3]]

Output: 1 4

Explanation:

The source vertex is 1. Hence, the shortest distance

path of node 4 from the source will be 1->4 as this is

the path with the minimum sum of edge weights from

source to destination.
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Problem Link

Approach:

We’ll be using Dijkstra’s Algorithm with a slight modification for solving this particular problem.

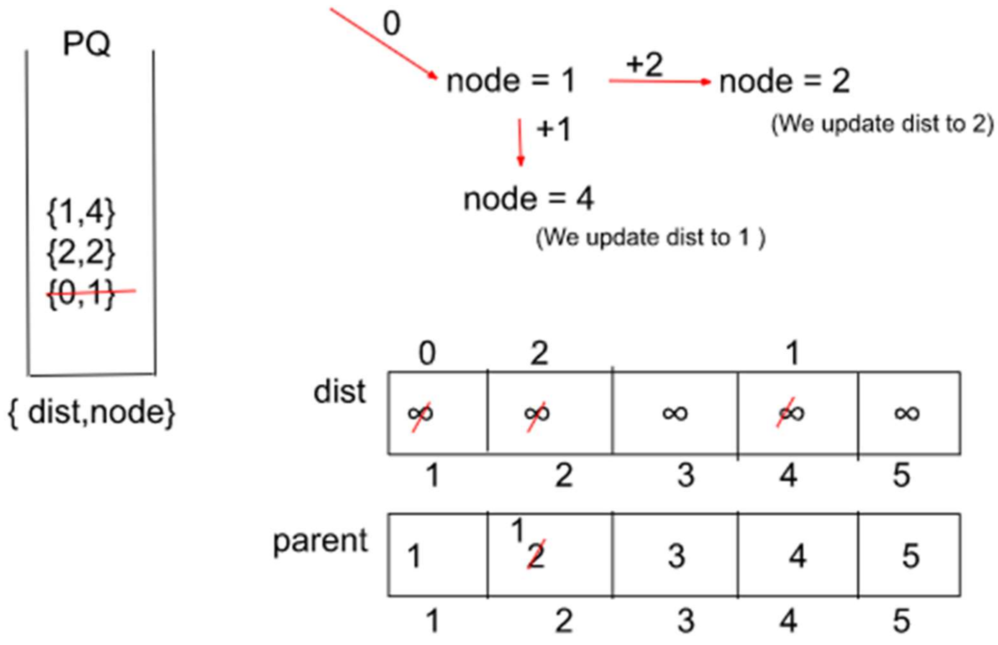
Initial configuration:

- **Source Node:** Before starting off with the Algorithm, we need to define a source node from which the shortest distances to every other node would be calculated.
- **Priority Queue:** Define a Priority Queue which would contain pairs of the type {dist, node}, where ‘dist’ indicates the currently updated value of the shortest distance from the source to the ‘node’.
- **Dist Array:** Define a dist array initialized with a large integer number at the start indicating that the nodes are unvisited initially. This array stores the shortest distances to all the nodes from the source node.
- **Parent Array:** This array’s sole purpose is to store the parent of a particular node i.e., the node from where that node came while traversing through the graph by Dijkstra’s Algorithm.

The Algorithm consists of the following steps :

- We start by initializing an adjacency list which will store all the adjacent nodes for a particular node along with the weights associated with them.
- Then, as a part of the initial configuration, we define a dist array to store the updated shortest distances for each node, a priority queue for storing the distance-node pairs, and a source node.
- In addition to this, we also declare a ‘parent’ array which would store the parent node for each node and will update itself to a different parent if a shorter path from a node is found at some point in time.
- At the start, all nodes’ parents have been set to the nodes themselves to indicate that the traversal has not yet been started.
- For every node at the top of the queue, we pop the element out and look out for its adjacent nodes. If the current reachable distance is better than the previous distance (dis + edW < dist[adjNode]), indicated by the distance array, we update the distance and push it into the queue.
- A node with a lower distance would be at the top of the priority queue as opposed to a node with a higher distance because we are using a min-heap.

- In addition to the previous step, we will also update the parent array to the node from where the current node came while traversing.
- By following step 5 repeatedly until our queue becomes empty, we would get the minimum distance from the source node to all other nodes and also our parent array would be updated according to the shortest path.
- Now, we run a loop starting from the destination node storing the node's parent and then moving to the parent again (backtrack) till the parent[node] becomes equal to the node itself.
- At last, we reverse the array in which the path is being stored as the path is in reverse order. Finally, we return the 'path' array.
- Here's a quick demonstration of the algorithm :



In a similar way, we can perform the iterations for all the other nodes and hence return the path.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

The intuition behind the above problem is based on Dijkstra's Algorithm with a combination of a little bit of memoization in order to **print** the shortest possible path and not just calculate the shortest distance between the source and the destination node. In order to print the path we will try to remember the node from which we came while traversing each node by Dijkstra's Algorithm along with calculating the shortest distance.

An array called 'parent' can be used for this purpose which would store the parent node for each node and will update itself if a shorter path from a node is found at some point in time. This will help us to print the path easily at the end by backtracking through the parent array till we reach the source node.

Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    vector<int> shortestPath(int n, int m, vector<vector<int>> &edges)
    {
        // Create an adjacency list of pairs of the form node1 -> {node2, edge
weight}
        // where the edge weight is the weight of the edge from node1 to
node2.
        vector<pair<int, int>> adj[n + 1];
        for (auto it : edges)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }
        // Create a priority queue for storing the nodes along with distances
        // in the form of a pair { dist, node }.
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int,int>>> pq;

        // Create a dist array for storing the updated distances and a parent
array
```

```cpp
        //for storing the nodes from where the current nodes represented by
indices of
        // the parent array came from.
        vector<int> dist(n + 1, 1e9), parent(n + 1);
        for (int i = 1; i <= n; i++)
            parent[i] = i;

        dist[1] = 0;

        // Push the source node to the queue.
        pq.push({0, 1});
        while (!pq.empty())
        {
            // Topmost element of the priority queue is with minimum distance
value.
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int dis = it.first;

            // Iterate through the adjacent nodes of the current popped node.
            for (auto it : adj[node])
            {
                int adjNode = it.first;
                int edW = it.second;

                // Check if the previously stored distance value is
                // greater than the current computed value or not,
                // if yes then update the distance value.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});

                    // Update the parent of the adjNode to the recent
                    // node where it came from.
                    parent[adjNode] = node;
                }
            }
        }

        // If distance to a node could not be found, return an array
containing -1.
        if (dist[n] == 1e9)
            return {-1};

        // Store the final path in the ‘path’ array.
        vector<int> path;
        int node = n;
        // Iterate backwards from destination to source through the parent
array.
        while (parent[node] != node)
        {
            path.push_back(node);
            node = parent[node];
        }
        path.push_back(1);

        // Since the path stored is in a reverse order, we reverse the array
        // to get the final answer and then return the array.
        reverse(path.begin(), path.end());
        return path;
    }
};

int main()
{
    // Driver Code

    int V = 5, E = 6;
    vector<vector<int>> edges = {{1, 2, 2}, {2, 5, 5}, {2, 3, 4}, {1, 4, 1},
{4, 3, 3},
    {3, 5, 1}};
    Solution obj;
    vector<int> path = obj.shortestPath(V, E, edges);

    for (int i = 0; i < path.size(); i++)
    {

        cout << path[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output :

```
1 4 3 5
```

Time Complexity: O(E log(V)) { for Dijkstra's Algorithm } + O(V) { for backtracking in order to find the parent for each node } Where E = Number of edges and V = Number of Nodes.

Space Complexity: O(|E| + |V|) { for priority queue and dist array } + O(|V|) { for storing the final path } Where E = Number of edges and V = Number of Nodes.