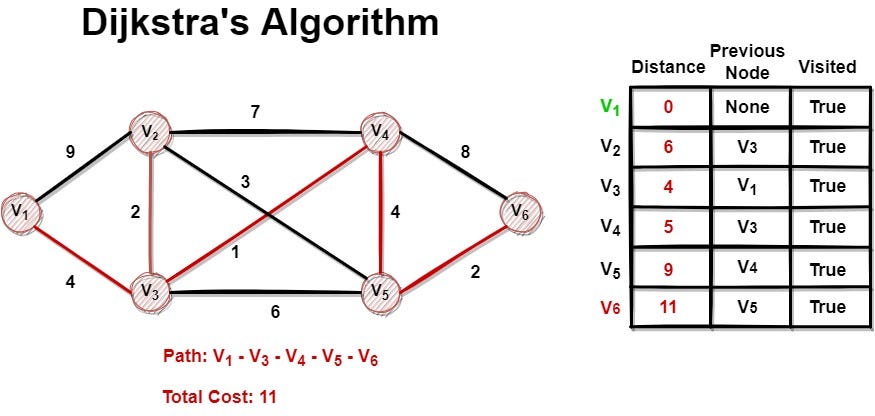
**AA LAB ASSIGNMENT | DIJKSTRA ALGORITHM**

**TITLE:**  Analysis, Proof of Analysis and Implementation of **Dijkstra Algortithm**

**MECHANISM**

The algorithm maintains a set of unvisited vertices and a set of visited vertices. It starts at the source vertex and iteratively selects the vertex with the smallest known distance from the source that has not yet been visited. It then explores all its neighboring vertices, updating their distances from the source if a shorter path is found. This process continues until all vertices have been visited or the target vertex (if specified) is reached.



**ALGORITHM / PSEUDOCODE**

**Input**: Graph represented as a set of vertices V and edges E

**Output:** Tree T

***Function Prim*** *(G= (V, E)):*

*T = tree with a single vertex, arbitrarily chosen from G*

*vis = {T}*

*pq = priority queue of edges adjacent to the initial vertex*

***while*** *pq* ***is******not*** *empty:*

*u, v = cheapest edge in pq*

*if v* ***not******in*** *vis:*

*add (u, v) to T*

*add v to vis*

*add all edges adjacent to v to pq*

***return*** *T*

**IMPLEMENTATION**

*#include* <iostream>

*#include* <vector>

*#include* <queue>

*#include* <limits>

using namespace std;

const int INF = numeric\_limits<int>::max(); *// define infinity as maximum value of int*

void dijkstra(int *start*, vector<vector<pair<int, int>>>& *graph*, vector<int>& *dist*) {

    int n = *graph*.size();

*// initialize distance array to infinity for all nodes except start node*

*dist*.assign(n, INF);

*dist*[*start*] = 0;

*// priority queue to store nodes to be visited, ordered by distance*

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

*// insert start node into priority queue*

    pq.push(make\_pair(0, *start*));

*// loop until all nodes have been visited*

*while* (!pq.empty()) {

        int u = pq.top().second; *// get node with smallest distance from priority queue*

        pq.pop();

*// loop through all neighbors of node u*

*for* (auto& neighbor : *graph*[u]) {

            int v = neighbor.first;

            int weight = neighbor.second;

*// update distance to neighbor if it is shorter than current distance*

*if* (*dist*[v] > *dist*[u] + weight) {

*dist*[v] = *dist*[u] + weight;

                pq.push(make\_pair(*dist*[v], v)); *// insert updated distance into priority queue*

            }

        }

    }

}

int main() {

    int n, m, start;

    cin >> n >> m >> start;

    vector<vector<pair<int, int>>> graph(n);

*for* (int i = 0; i < m; i++) {

        int u, v, w;

        cin >> u >> v >> w;

        graph[u].push\_back(make\_pair(v, w));

        graph[v].push\_back(make\_pair(u, w)); *// add edge in both directions for undirected graph*

    }

    vector<int> dist;

    dijkstra(start, graph, dist);

*for* (int i = 0; i < n; i++) {

*if* (dist[i] == INF) {

            cout << "No path from " << start << " to " << i << endl;

        } *else* {

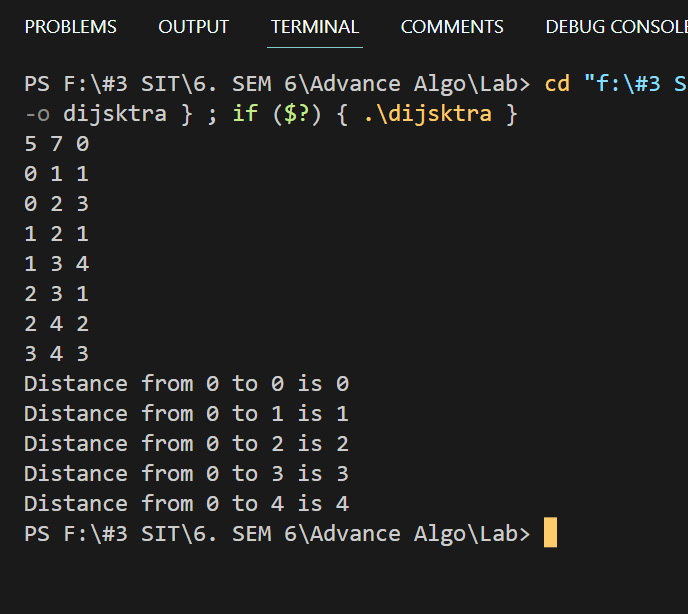
            cout << "Distance from " << start << " to " << i << " is " << dist[i] << endl;

        }

    }

*return* 0;

}



**T(n) ANALYSIS WITH PROOF**

Time Complexity - O(|V| + |E| log |V|)

**ADVANTAGES / DISADVANTAGES**

|  |
| --- |
|  |
| **Advantages** | **Disadvantages** |
| Dijkstra's algorithm guarantees finding the shortest path in terms of the sum of edge weights, from a single source vertex to all other vertices in a weighted graph, as long as all edge weights are non-negative. It provides an optimal solution to the shortest path problem. | Dijkstra's algorithm with a naive implementation using an adjacency matrix has a time complexity of O(|V|^2), which can be inefficient for large graphs with many vertices. The time complexity can be improved to O(|V| + |E| log |V|) with a priority queue, but it may still be slow for very large graphs. |
| Dijkstra's algorithm can be used in various applications, such as routing in computer networks, finding shortest routes in transportation systems, and solving optimization problems. It is a widely used algorithm with many real-world applications. | Dijkstra's algorithm assumes that all edge weights are non-negative. If there are negative edge weights in the graph, the algorithm may not work correctly, as it does not handle negative weight cycles. In such cases, other algorithms such as Bellman-Ford or Floyd-Warshall may be more appropriate. |
| Dijkstra's algorithm can be efficient for small graphs with non-negative edge weights, especially when implemented with a priority queue (e.g., min-heap), which allows for efficient selection of the vertex with the smallest distance in each iteration. | Dijkstra's algorithm finds the shortest path from a single source vertex to all other vertices in the graph. If multiple source vertices or multiple pairs of source and target vertices need to be considered, multiple invocations of the algorithm may be required, which can be inefficient. |

**REAL LIFE APPLICATIONS:**

Network Routing: Finding the shortest path between two points in a network, such as determining the fastest route for data to travel between two points on the internet.

GPS Navigation: Finding the shortest path between two locations on a map, such as when planning a driving route.

Transportation: Finding the shortest path between two points in a transportation system, such as a subway or train system, to optimize travel time.