**Experiment No 6**

**AIM:** Study of Dijkstra's algorithm for minimum spanning tree.

**PROBLEM STATEMENT:** Optimal Route Planning Application: Create a route planning application that uses graphs to represent road networks and Dijkstra's algorithm for finding the shortest path between two locations. Allow users to input starting and destination points to get the optimal route.

**REQUIREMENT:**Turbo C/ GCC Compiler

**OPERATING SYSTEM:** Windows/Linux/Unix.

**THEORY:**

**Fundamentals of Dijkstra's Algorithm**

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm

A **graph** and **source vertex** are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

**Each Vertex in this Algorithm will have two properties defined for it:**

1. Visited Property
2. Path Property

Let us understand these properties in brief.

Visited Property:

1. The 'visited' property signifies whether or not the node has been visited.
2. We are using this property so that we do not revisit any node.
3. A node is marked visited only when the shortest path has been found.

In the mathematical sense, relaxation can be exemplified as:

**p[n] = minimum(p[n], p[m] + w)**

**The following is the step that we will follow to implement Dijkstra's Algorithm:**

**Step 1:** First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

**Step 2:** We will then set the unvisited node with the smallest current distance as the current node, suppose X.

**Step 3:** For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

**Step 4:** We will then mark the current node X as visited.

**CONCLUSION:** Program for Dijkstra's algorithm is implemented successfully.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

// Define an edge as a pair (neighbor, weight)

typedef pair<int, int> Edge;

void dijkstra(vector<vector<Edge>>& graph, int source) {

int V = graph.size();

vector<int> dist(V, numeric\_limits<int>::max()); // Distance array initialized to infinity

priority\_queue<Edge, vector<Edge>, greater<Edge>> pq; // Min-heap

dist[source] = 0;

pq.push({0, source}); // (distance, vertex)

while (!pq.empty()) {

int currDist = pq.top().first;

int u = pq.top().second;

pq.pop();

if (currDist > dist[u]) continue; // Ignore outdated distances

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

int v = neighbor.first;

int weight = neighbor.second;

if (dist[u] + weight < dist[v]) {

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

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

}

}

}

// Output the shortest distances

cout << "Vertex\tDistance from Source" << endl;

for (int i = 0; i < V; i++) {

cout << i << "\t" << dist[i] << endl;

}

}

int main() {

int V = 5; // Number of vertices

vector<vector<Edge>> graph(V);

// Add edges (u, v, weight)

graph[0].push\_back({1, 10});

graph[0].push\_back({3, 5});

graph[1].push\_back({2, 1});

graph[1].push\_back({3, 2});

graph[2].push\_back({4, 4});

graph[3].push\_back({1, 3});

graph[3].push\_back({2, 9});

graph[3].push\_back({4, 2});

graph[4].push\_back({2, 6});

int source = 0;

dijkstra(graph, source);

return 0;

}