# DSA LAB ASSIGNMENT 11

**Name : Mahesh Jagtap**

**Reg. No. 24MCS1017**

Implementing Dijkstra's Algorithm to Find the Shortest Path in a Weighted Graph as given below:

A --(4)--> B
A --(1)--> C
B --(1)--> C
B --(2)--> D
C --(5)--> D
D --(3)--> E

A) Implement the algorithm and run it on the above graph, with a starting node, say A.

B) Print the shortest distances from node A to all other nodes.

C) Discuss the time complexity of the algorithm, specifically O((V+E)logV), where V is the number of vertices and E is the number of edges.

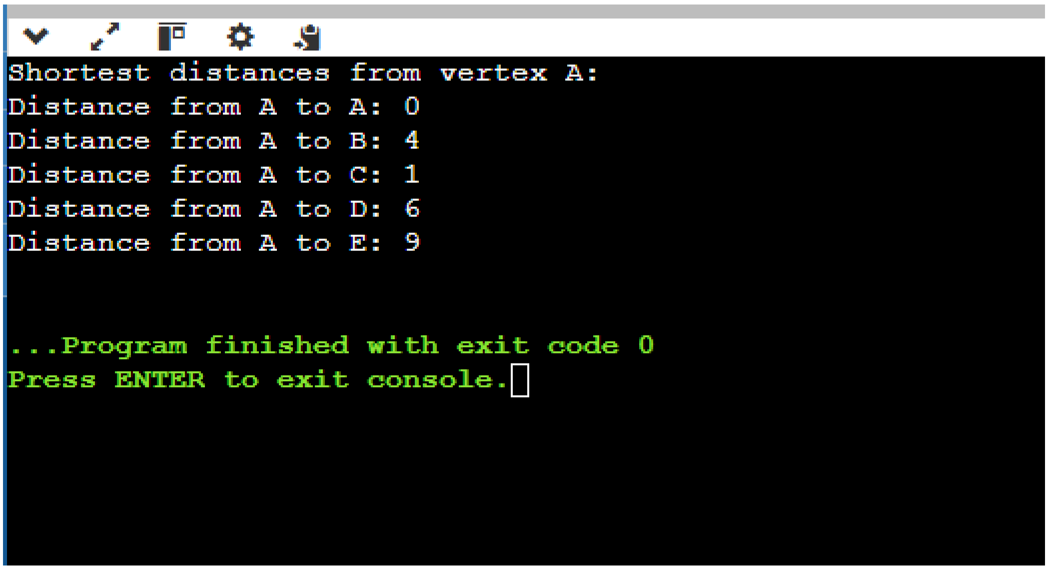
CODE:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>


using namespace std;


// Function to perform Dijkstra's Algorithm
void dijkstra(int graph[5][5], int start, int vertices) {
   vector<int> distances(vertices, INT_MAX); // Initialize distances
   distances[start] = 0; // Distance to the start vertex is 0
   priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; //
Min-heap priority queue
```

```
    pq.push(make_pair(0, start)); // Push the start vertex to the priority queue

    while (!pq.empty()) {
        int currentVertex = pq.top().second; // Get the vertex with the smallest distance
        pq.pop();

        // Explore neighbors of the currentVertex
        for (int i = 0; i < vertices; i++) {
            // If there is an edge
            if (graph[currentVertex][i] != 0) {
                int weight = graph[currentVertex][i];
                // If the new distance is smaller
                if (distances[currentVertex] + weight < distances[i]) {
                    distances[i] = distances[currentVertex] + weight; // Update the distance
                    pq.push(make_pair(distances[i], i)); // Push the neighbor to the priority
queue
                }
            }
        }
    }

    // Print the shortest distances from the start vertex
    cout << "Shortest distances from vertex A:" << endl;
    for (int i = 0; i < vertices; i++) {
        char vertex = 'A' + i;
        cout << "Distance from A to " << vertex << ": " << distances[i] << endl;
    }
}

int main() {
    // Define the graph as an adjacency matrix
    // 5 vertices: A, B, C, D, E
    int graph[5][5] = {
```

```
    {0, 4, 1, 0, 0}, // A
    {0, 0, 1, 2, 0}, // B
    {0, 0, 0, 5, 0}, // C
    {0, 0, 0, 0, 3}, // D
    {0, 0, 0, 0, 0}  // E
  };

  // Call Dijkstra's algorithm from vertex A (index 0)
  dijkstra(graph, 0, 5);

  return 0;
}
```

**OUTPUT:**



```
Shortest distances from vertex A:
Distance from A to A: 0
Distance from A to B: 4
Distance from A to C: 1
Distance from A to D: 6
Distance from A to E: 9


...Program finished with exit code 0
Press ENTER to exit console.
```

**Time Complexity:**

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edges. Its time complexity of O((V + E) log V) ) arises due to:

1. **Initialization**: Setting up distances for all ( V ) vertices, which is ( O(V) ).
2. **Priority Queue Operations**: Using a min-heap, each vertex insertion and each distance update (decrease-key operation) takes ( O(log V) ).
3. **Edge Relaxation**: For each edge ( (u, v) ), the algorithm might update the shortest path to ( v ), which triggers a priority queue operation.

Combining these factors, we get ( O(V log V + E log V) = O((V + E) log V) ), making it efficient for graphs where ( E ) is roughly proportional to ( V ), such as sparse graphs.