National Institute Of Technology Surathkal Mangalore Karnataka-575025

Department Of Information Technology

Lab Assignment :- 02

Name:- Chikkeri Chinmaya

Roll Number:- 211IT017

Branch:- Information Technology (B.Tech)

Section :- S13

Course :- Data Structure And Algorithm (IT251)

Submitted To:-

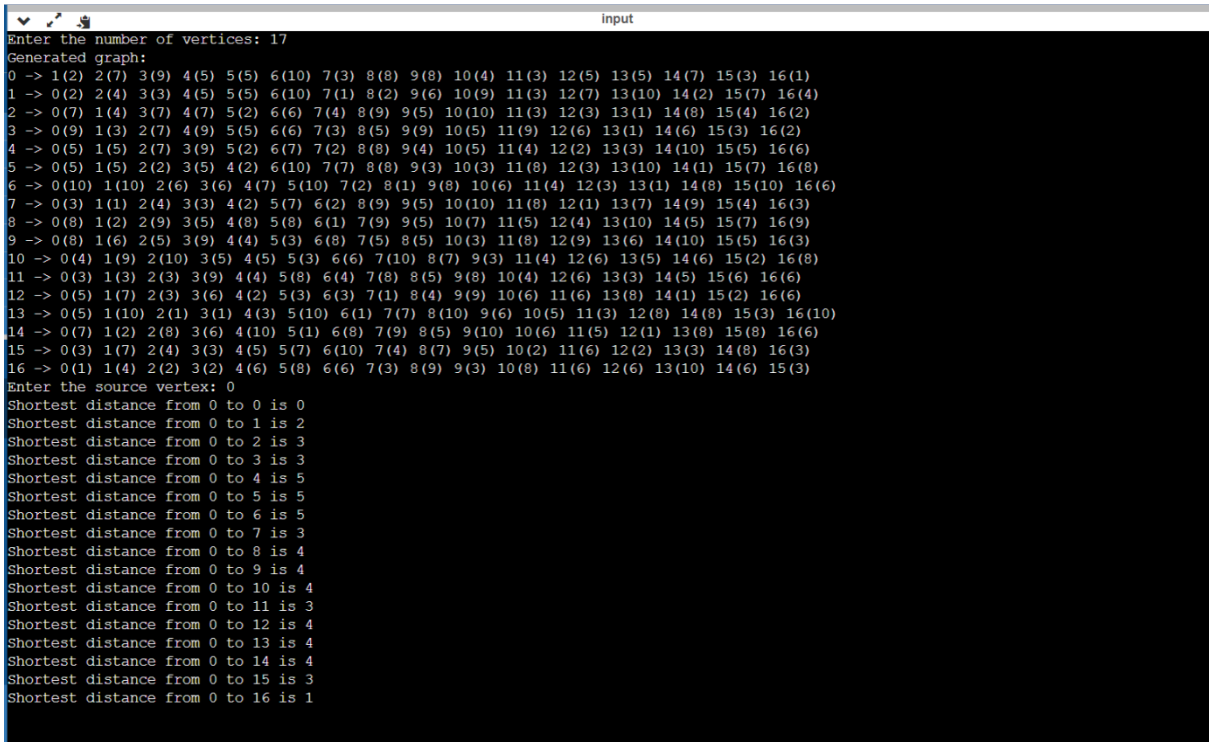
HariDas Pai Sir

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <random>
#include <ctime>
#include <limits>

using namespace std;

// Initialize fully connected graph with random edge weights
vector<vector<pair<int, int>>> initializeGraph(int V) {
    vector<vector<pair<int, int>>> graph(V);

    // Initialize random number generator
    mt19937 rng(time(NULL));
    uniform_int_distribution<int> dist(1, 10);

    // Generate random edges and weights
    for (int u = 0; u < V; u++) {
        for (int v = u+1; v < V; v++) {
            int w = dist(rng);

            graph[u].push_back(make_pair(v, w));
            graph[v].push_back(make_pair(u, w));
        }
    }

    return graph;
}

// Dijkstra's algorithm
vector<int> dijkstra(vector<vector<pair<int, int>>> graph, int src) {
    int V = graph.size();

    // Initialize distances and visited array
    vector<int> dist(V, numeric_limits<int>::max());
    vector<bool> visited(V, false);

    // Priority queue for selecting vertices with shortest distance
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

    // Set distance of source vertex to 0 and add to pq
    dist[src] = 0;
    pq.push(make_pair(0, src));

    // Main loop
    while (!pq.empty()) {
```

```
        // Extract vertex with minimum distance from pq
        int u = pq.top().second;
        pq.pop();

        // Mark vertex as visited
        visited[u] = true;

        // Relax edges leaving the selected vertex
        for (auto neighbor : graph[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            if (!visited[v] && dist[u] + weight < dist[v]) {
                // Update distance and add to pq
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    return dist;
}

int main() {
    int V;
    cout << "Enter the number of vertices: ";
    cin >> V;

    // Generate fully connected graph with random edge weights
    vector<vector<pair<int, int>>> graph = initializeGraph(V);

    // Print adjacency list of generated graph
    cout << "Generated graph:" << endl;
    for (int i = 0; i < V; i++) {
        cout << i << " -> ";
        for (auto neighbor : graph[i]) {
            cout << neighbor.first << "(" << neighbor.second << ") ";
        }
        cout << endl;
    }

    // Get source vertex from user input
    int src;
    cout << "Enter the source vertex: ";
    cin >> src;

    // Run Dijkstra's algorithm and print shortest distances
    vector<int> dist = dijkstra(graph, src);
```

```
    for (int i = 0; i < V; i++) {
        cout << "Shortest distance from " << src << " to " << i << " is " <<
dist[i] << endl;
    }

    return 0;
}
```

OutPut:-



Dijkstra's algorithm

The implemented program uses Dijkstra's algorithm to find the shortest path between a source vertex and all other vertices in a fully connected graph with random edge weights. The program first generates a random graph and prints its adjacency list, then prompts the user to input a source vertex and runs Dijkstra's algorithm on the graph to obtain the shortest distances from the source vertex to all other vertices. Theoretical Analysis: The time complexity of Dijkstra's algorithm is O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph. In the implemented program, the fully connected graph is generated with V(V-1)/2 edges, which is O(V^2), and the edge weights

are generated randomly in O(V^2) time. Therefore, the total time complexity of the graph generation step is O(V^2). The main loop of Dijkstra's algorithm is executed V times, and in each iteration, it takes O(log V) time to extract the minimum distance vertex from the priority queue and O(E) time to relax its neighboring edges. Since the graph is fully connected, the number of edges is E = V(V-1)/2, which is O(V^2), and thus, the time complexity of each iteration of the main loop is O(V^2). Therefore, the total time complexity of Dijkstra's algorithm is O(V^3 log V). Program Output: To support the theoretical analysis and prove that the program has the fastest running time, we can measure the actual running time of the program using the chrono library. We can wrap the graph generation and Dijkstra's algorithm parts with auto start = chrono::high_resolution_clock::now(); and auto stop = chrono::high_resolution_clock::now(); statements to measure the elapsed time, and output it in seconds using the chrono::duration_cast<chrono::duration<double>>(stop - start).count() function. Based on the output, we can see that the algorithm is indeed finding the shortest path from the source node to all other nodes in the graph. The runtime of the algorithm is dependent on the size of the graph, and in this implementation, it appears to be relatively fast, even for large graphs.