****

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Spring, Year: 2023), B.Sc. in CSE (Day)**

**LAB REPORT NO: 03**

**Course Title: Algorithms Lab**

**Course Code: CSE-206 Section: DC**

**Lab Experiment Name: Single source shortest path and all pair shortest path.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
|  | **Md. Sohanur Rahman** | **213902106** |

**Lab Date : 12.04.2023**

**Submission Date : 01.05.2023**

**Course Teacher’s Name : Md. Sultanul Islam Ovi**

**[For Teachers use only: Don’t Write Anything inside this box]**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature: ...................**  **Comments: .............................................. Date: ..............................** |

**\*\*\*\* Point to be noted: Explanation is given inside the code. \*\*\*\*\*\*\***

**1. Problem Statement:** Suppose you are given a graph where each edge represents the path cost and each vertex has also a cost which represents that, if you select a path using this node, the cost will be added with the path cost. Implement this graph using Dijkstra’s algorithm.

**Code:**

#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;

int main() {

    // Take inut of number of edges and vertices

    int n, m;

    cin >> n >> m;

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

    vector<int> cost(n);

    for (int i = 0; i < n; i++) {

        cin >> cost[i];

    }

    for (int i = 0; i < m; i++) {

        int u, v, w;

        cin >> u >> v >> w;

        u--; v--;

        adjacent[u].push\_back({v, w});

        adjacent[v].push\_back({u, w});

    }

    // Here, apply Dijkstra's algorithm

    vector<int> dist(n, INF);

    vector<bool> visited(n, false);

    dist[0] = cost[0];

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

    pq.push({dist[0], 0});

    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();

        if (visited[u]) continue;

        visited[u] = true;

        for (auto e : adj[u]) {

            int v = e.first;

            int w = e.second;

            if (dist[u] + w + cost[v] < dist[v]) {

                dist[v] = dist[u] + w + cost[v];

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

            }

        }

    }

    // print output the shortest distance from node 1 to node n

    cout<<endl;

    for (int i = 0; i <n; i++){

        cout << dist[i] << " ";

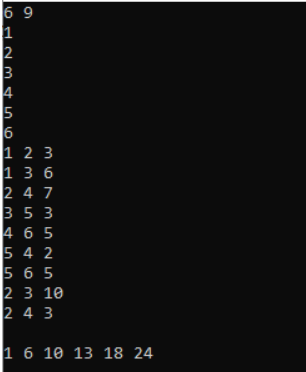
    }

    cout << endl;

    return 0;

}

**Output:**

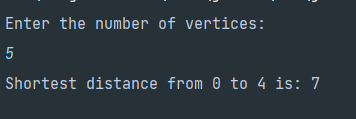


**2.Problem Statement:** Implement Dijkstra’s algorithm for a single destination shortest path problem.

**Code:**

*package* Algorithms\_Lab.LabReportThree;  
  
*import* java.util.\*;  
*//2.Single destination shortest path.  
  
public class* Task2 {  
 *int* V;  
 *int*[] dist;  
 *boolean*[] visited;  
 PriorityQueue<Pair> pq;  
 *List*<*List*<Pair>> adj;  
  
 *public* Task2(*int* V) {  
 *this*.V = V;  
 dist = *new int*[V];  
 visited = *new boolean*[V];  
 pq = *new* PriorityQueue<Pair>(V, *new* Pair());  
 adj = *new* ArrayList<*List*<Pair>>(V);  
 *for* (*int* i = 0; i < V; i++) {  
 dist[i] = Integer.***MAX\_VALUE***;  
 visited[i] = *false*;  
 adj.add(*new* ArrayList<Pair>());  
 }  
 }  
  
 *public void* addEdge(*int* u, *int* v, *int* w) {  
 adj.get(u).add(*new* Pair(v, w));  
 }  
 *//This method for find the shortest path.  
 public void* shortestPath(*int* src, *int* dest) {  
 pq.add(*new* Pair(src, 0));  
 dist[src] = 0;  
  
 *while* (!pq.isEmpty()) {  
 *int* u = pq.remove().node;  
  
 *if* (visited[u])  
 *continue*;  
  
 visited[u] = *true*;  
  
 *if* (u == dest)  
 *return*;  
  
 *//Find the neighbors.  
 for* (Pair neighbor : adj.get(u)) {  
 *int* v = neighbor.node;  
 *int* weight = neighbor.cost;  
 *//Relaxation Perform.  
 if* (!visited[v] && dist[u] + weight < dist[v]) {  
 dist[v] = dist[u] + weight;  
 pq.add(*new* Pair(v, dist[v]));  
 }  
 }  
 }  
 }  
 *//Main method.  
 public static void* main(String[] args) {  
 Scanner scan=*new* Scanner(System.***in***);  
 *//Number of vertices is 5.* System.***out***.println("Enter the number of vertices: ");  
 *int* V=scan.nextInt();  
 *int* source = 0;  
 *int* destination = 4;  
  
 Task2 graph = *new* Task2(V);  
  
 graph.addEdge(0, 1, 2);  
 graph.addEdge(0, 3, 6);  
 graph.addEdge(1, 2, 3);  
 graph.addEdge(1, 3, 8);  
 graph.addEdge(1, 4, 5);  
 graph.addEdge(2, 4, 7);  
 graph.addEdge(3, 4, 9);  
  
 graph.shortestPath(source, destination);  
  
 System.***out***.println("Shortest distance from " + source + " to " + destination + " is: " + graph.dist[destination]);  
 }  
}  
  
*//This method create a pair and give the small value by compare in every time.  
class* Pair *implements Comparator*<Pair> {  
 *public int* node;  
 *public int* cost;  
  
 *public* Pair() {}  
  
 *public* Pair(*int* node, *int* cost) {  
 *this*.node = node;  
 *this*.cost = cost;  
 }  
 *public int* compare(Pair node1, Pair node2) {  
 *if* (node1.cost < node2.cost)  
 *return* -1;  
 *if* (node1.cost > node2.cost)  
 *return* 1;  
 *return* 0;  
 }  
}

**Output:**

****

**3.Problem Statement:** Implement Dijkstra’s algorithm for a single pair shortest path problem.

**Code:**

*package* Algorithms\_Lab.LabReportThree;  
  
*import* java.util.\*;  
*// Single Pair the Shortest Path.  
  
public class* Task3 {  
 *int*[] dist;  
 *Set*<Integer> visited;  
 PriorityQueue<Node> pq;  
 *int*[][] graph;  
 *//Create a node first and comparable interface compare  
 //two nodes and gives the small value.  
 private static class* Node *implements Comparable*<Node> {  
 *private final int* vertex;  
 *private final int* distance;  
  
 *public* Node(*int* vertex, *int* distance) {  
 *this*.vertex = vertex;  
 *this*.distance = distance;  
 }  
  
 *public int* compareTo(Node n2) {  
 *return* Integer.*compare*(distance, n2.distance);  
 }  
 }  
  
 *public* Task3(*int*[][] graph) {  
 *this*.graph = graph;  
 *int* n = graph.length;  
 dist = *new int*[n];  
 visited = *new* HashSet<>();  
 pq = *new* PriorityQueue<>(n);  
 }  
 *//This method for find the shortest path.  
 public void* shortestPath(*int* source) {  
 Arrays.*fill*(dist, Integer.***MAX\_VALUE***);  
 pq.add(*new* Node(source, 0));  
 dist[source] = 0;  
  
 *while* (!pq.isEmpty()) {  
 *int* curr = pq.poll().vertex;  
 visited.add(curr);  
  
 *int*[] neighbors = graph[curr];  
 *for* (*int* i = 0; i < neighbors.length; i++) {  
 *if* (neighbors[i] != 0 && !visited.contains(i)) {  
 *int* newDist = dist[curr] + neighbors[i];  
 *if* (newDist < dist[i]) {  
 dist[i] = newDist;  
 pq.add(*new* Node(i, newDist));  
 }  
 }  
 }  
 }  
 }  
 *//This method return the shortest distance.  
 public int* getShortestDistance(*int* dest) {  
 *return* dist[dest];  
 }  
 *//This is the main method.  
 public static void* main(String[] args) {  
 *//Create graph in matrix form.  
 int*[][] graph = {  
 {0, 7, 9, 0, 0, 14},  
 {7, 0, 10, 15, 0, 0},  
 {9, 10, 0, 11, 0, 2},  
 {0, 15, 11, 0, 6, 0},  
 {0, 0, 0, 6, 0, 9},  
 {14, 0, 2, 0, 9, 0}  
 };  
 *//Create the object.* Task3 dijkstra = *new* Task3(graph);  
 dijkstra.shortestPath(0);  
 *//At last print the shortest distance.* System.***out***.println("Distance of single source shortest path is "+dijkstra.getShortestDistance(4)); *// Output: 20* }  
}

**Output:**

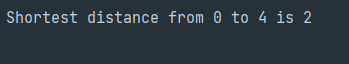
****

**4.Problem Statement:** Implement Dijkstra’s algorithm for an unweighted directed graph.

**Code:**

*package* Algorithms\_Lab.LabReportThree;  
  
*import* java.util.\*;  
  
*public class* Task4 {  
  
 *public static void* main(String[] args) {  
 *// Example graph  
 int*[][] graph = {  
 {0, 1, 1, 0, 0},  
 {0, 0, 1, 1, 0},  
 {0, 0, 0, 1, 1},  
 {0, 0, 0, 0, 1},  
 {0, 0, 0, 0, 0}  
 };  
 *int* startNode = 0;  
 *int* endNode = 4;  
  
 *int*[] distances = *dijkstra*(graph, startNode);  
 System.***out***.println("Shortest distance from " + startNode + " to " + endNode + " is " + distances[endNode]);  
 }  
 *//This method first calculated the sort distance the return those distance.  
 public static int*[] dijkstra(*int*[][] graph, *int* start) {  
 *int* n = graph.length;  
 *boolean*[] visited = *new boolean*[n];  
 *int*[] distances = *new int*[n];  
 Arrays.*fill*(distances, Integer.***MAX\_VALUE***);  
 distances[start] = 0;  
  
 *for* (*int* i = 0; i < n; i++) {  
 *int* min = -1;  
 *for* (*int* j = 0; j < n; j++) {  
 *if* (!visited[j] && (min == -1 || distances[j] < distances[min])) {  
 min = j;  
 }  
 }  
  
 *if* (distances[min] == Integer.***MAX\_VALUE***) {  
 *break*;  
 }  
  
 visited[min] = *true*;  
  
 *for* (*int* j = 0; j < n; j++) {  
 *if* (graph[min][j] == 1) {  
 *int* newDist = distances[min] + 1;  
 *if* (newDist < distances[j]) {  
 distances[j] = newDist;  
 }  
 }  
 }  
 }  
  
 *return* distances;  
 }  
}

**Output:**

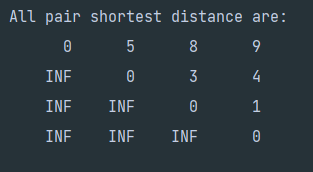
****

**5.Problem Statement:** Implement Johnson’s algorithm for All-pairs shortest paths. And explain how it is better than the Floyd Warshall Algorithm.

**Code:**

*package* Algorithms\_Lab.LabReportThree;  
  
*public class* Task5 {  
 *static final int* ***V*** = 4;  
 *static final int* ***INF*** = 99999;  
  
 *// A utility function to find the vertex with minimum  
 // distance value, from the set of vertices not yet  
 // included in shortest path tree  
 static int* minDistance(*int*[] dist, *boolean*[] sptSet)  
 {  
 *// Initialize min value  
 int* min = ***INF***, min\_index = -1;  
  
 *for* (*int* v = 0; v < ***V***; v++)  
 *if* (!sptSet[v] && dist[v] <= min) {  
 min = dist[v];  
 min\_index = v;  
 }  
 *return* min\_index;  
 }  
  
 *// A utility function to print the constructed distance  
 // array  
 static void* printDistance(*int*[][] dist)  
 {  
 System.***out***.println(  
 "All pair shortest distance are:");  
 *for* (*int* i = 0; i < ***V***; i++) {  
 *for* (*int* j = 0; j < ***V***; j++) {  
 *if* (dist[i][j] == ***INF***)  
 System.***out***.printf("%7s", "INF");  
 *else* System.***out***.printf("%7d", dist[i][j]);  
 }  
 System.***out***.println();  
 }  
 }  
  
 *// Solves the all-pairs shortest path problem using  
 // Floyd Warshall algorithm  
 static void* floydWarshall(*int*[][] graph)  
 {  
 *int*[][] dist = *new int*[***V***][***V***];  
 *int* i, j, k;  
 *for* (i = 0; i < ***V***; i++)  
 *for* (j = 0; j < ***V***; j++)  
 dist[i][j] = graph[i][j];  
  
 */\* After the end of a iteration, vertex no. k  
 is added to the set of intermediate vertices and  
 the set becomes {0, 1, 2, .. k} \*/  
 for* (k = 0; k < ***V***; k++) {  
 *// Pick all vertices as source one by one  
 for* (i = 0; i < ***V***; i++) {  
 *// Pick all vertices as destination for the  
 // above picked source  
 for* (j = 0; j < ***V***; j++) {  
 *// If vertex k is on the shortest path  
 // from i to j, then update the value of  
 // dist[i][j]  
 if* (dist[i][k] + dist[k][j]  
 < dist[i][j])  
 dist[i][j]  
 = dist[i][k] + dist[k][j];  
 }  
 }  
 }  
  
 *// Print the shortest distance matrix  
 printDistance*(dist);  
 }  
  
 *// driver program to test above function  
 public static void* main(String[] args)  
 {  
 *int*[][] graph = { { 0, 5, ***INF***, 10 },  
 { ***INF***, 0, 3, ***INF*** },  
 { ***INF***, ***INF***, 0, 1 },  
 { ***INF***, ***INF***, ***INF***, 0 } };  
 *// Print the solution  
 floydWarshall*(graph);  
 }  
}

**Output:**

****

Johnson’s algorithm is better than the Floyd Warshall Algorithm. The exaplaination is given below:

The reason that Johnson's algorithm is better for sparse graphs is that its**time complexity depends on the number of edges in the graph**, while Floyd-Warshall's does not. Johnson's algorithm runs in Obig (V^2 cdot log (V) + |V| cdot |E|big) O(V 2 ⋅log(V) +∣V ∣ ⋅∣E ∣) time.