**Assignment No. 05**

Name:- Anirudha Shripad Kale.

Roll No. :- 223029

PRN No :- 22210288

Batch: C1

**Problem Statement:** A customer wants to travel from source A to destination B. He books a cab from source A to reach destination B. Calculate a shortest path by avoiding real time traffic to reach destination B.

**Objectives:**

1. To understand the algorithm to find the shortest path from the source to destination.
2. To implement the Dijkstra's algorithm.
3. Analyse it’s space and time complexity.

**Theory:**

Dijkstra's algorithm is a greedy algorithm used to find the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It efficiently solves the single-source shortest path problem, providing the shortest path tree from the source vertex to all other vertices.

Time Complexity:

The time complexity of Dijkstra's algorithm depends on the implementation and the chosen data structure for the priority queue.

Using a binary heap (priority queue) results in a time complexity of O((V + E) log V).

Using Fibonacci heap for priority queue operations can reduce the time complexity to O(V log V + E).

Dijkstra's algorithm produces the shortest path tree from the source vertex to all other vertices.

The algorithm works correctly only for graphs with non-negative edge weights.

If there are negative edge weights, Dijkstra's algorithm may produce incorrect results.

**Algorithm:**

* **Initialization:**
  + Set the distance of the source vertex to 0 and initialize the distance of all other vertices to infinity.
  + Add the source vertex to the priority queue.
* **Greedy Step:**
  + Repeat until all vertices have been visited:
    - Remove the vertex with the smallest tentative distance from the priority queue.
    - For each neighbour of the current vertex:
      * Calculate the distance to the neighbour as the sum of the distance to the current vertex and the weight of the edge between them.
      * If this distance is smaller than the current tentative distance to the neighbour, update the tentative distance.
      * Add the neighbour to the priority queue if it has not been visited.
* **Termination:**
  + After all vertices have been visited, the distance array contains the shortest distances from the source vertex to all other vertices.

**Code:**

import java.util.\*;

class Edge {

    int destination;

    int weight;

    public Edge(int destination, int weight) {

        this.destination = destination;

        this.weight = weight;

    }

}

class Graph {

    int V;

    List<List<Edge>> adjList;

    public Graph(int V) {

        this.V = V;

        adjList = new ArrayList<>(V);

        for (int i = 0; i < V; i++) {

            adjList.add(new ArrayList<>());

        }

    }

    public void addEdge(int source, int destination, int weight) {

        adjList.get(source).add(new Edge(destination, weight));

        // For undirected graph, uncomment the line below

        // adjList.get(destination).add(new Edge(source, weight));

    }

    public List<Integer> dijkstraShortestPath(int source, int destination) {

        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

        pq.offer(new int[] { source, 0 });

        int[] distances = new int[V];

        Arrays.fill(distances, Integer.MAX\_VALUE);

        distances[source] = 0;

        int[] parent = new int[V];

        Arrays.fill(parent, -1);

        while (!pq.isEmpty()) {

            int[] curr = pq.poll();

            int u = curr[0];

            int dist = curr[1];

            if (u == destination) {

                break; // Shortest path to destination found

            }

            if (dist > distances[u]) {

                continue; // Skip outdated distances

            }

            for (Edge edge : adjList.get(u)) {

                int v = edge.destination;

                int weight = edge.weight;

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

                    distances[v] = dist + weight;

                    parent[v] = u;

                    pq.offer(new int[] { v, distances[v] });

                }

            }

        }

        List<Integer> shortestPath = new ArrayList<>();

        int current = destination;

        while (current != -1) {

            shortestPath.add(current);

            current = parent[current];

        }

        Collections.reverse(shortestPath);

        return shortestPath;

    }

}

public class Assignment5 {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        Graph graph = null;

        while (true) {

            System.out.println("\nMenu:");

            System.out.println("1. Create graph");

            System.out.println("2. Find shortest path");

            System.out.println("3. Exit");

            System.out.print("Enter your choice (1-3): ");

            int choice = scanner.nextInt();

            switch (choice) {

                case 1:

                    System.out.print("Enter the number of vertices in the graph: ");

                    int V = scanner.nextInt();

                    graph = new Graph(V);

                    System.out.print("Enter the number of edges in the graph: ");

                    int E = scanner.nextInt();

                    System.out.println("Enter the details of each edge (source, destination, weight):");

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

                        int src = scanner.nextInt();

                        int dest = scanner.nextInt();

                        int weight = scanner.nextInt();

                        graph.addEdge(src, dest, weight);

                    }

                    break;

                case 2:

                    if (graph == null) {

                        System.out.println("Create a graph first!");

                        break;

                    }

                    int source, destination;

                    System.out.print("Enter the source node: ");

                    source = scanner.nextInt();

                    System.out.print("Enter the destination node: ");

                    destination = scanner.nextInt();

                    List<Integer> shortestPath = graph.dijkstraShortestPath(source, destination);

                    System.out.println("Shortest path from source " + source + " to destination " + destination + ":");

                    for (int node : shortestPath) {

                        System.out.print(node + " ");

                    }

                    break;

                case 3:

                    System.out.println("Exiting program. Goodbye!");

                    scanner.close();

                    System.exit(0);

                    break;

                default:

                    System.out.println("Invalid choice. Please enter a number between 1 and 3.");

            }

        }

    }

}

**Output:**

Menu:

1. Create graph

2. Find shortest path

3. Exit

Enter your choice (1-3): 1

Enter the number of vertices in the graph: 5

Enter the number of edges in the graph: 6

Enter the details of each edge (source, destination, weight):

0 1 2

1 2 3

0 3 6

1 4 5

2 4 7

1 3 8

Menu:

1. Create graph

2. Find shortest path

3. Exit

Enter your choice (1-3): 2

Enter the source node: 0

Enter the destination node: 4

Shortest path from source 0 to destination 4:

0 1 4

Menu:

1. Create graph

2. Find shortest path

3. Exit

Enter your choice (1-3): 3

Exiting program. Goodbye!