Day 15

Assignment: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

A)

Introduction:

Dijkstra's algorithm finds the shortest path between nodes in a graph with non-negative edge weights. It starts from a source node and gradually explores the graph, updating the shortest distances to other nodes. By the end, it determines the shortest path from the source to every other node. It's widely used in navigation systems and network routing.

Java code:

```
package Day15;

import java.util.*;

public class DijkstraShortestPath {
  // Class to represent a node in the graph
  static class Node implements Comparable<Node> {
    int vertex;
    int distance;

    public Node(int vertex, int distance) {
      this.vertex = vertex;
      this.distance = distance;
    }

    @Override
    public int compareTo(Node other) {
      return Integer.compare(this.distance, other.distance);
    }
  }

  // Class to represent the graph
  static class Graph {
    private int numVertices;
    private List<List<Node>> adjList;

    public Graph(int numVertices) {
      this.numVertices = numVertices;
      adjList = new ArrayList<>();
      for (int i = 0; i < numVertices; i++) {
```

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

    public void addEdge(int u, int v, int weight) {
        adjList.get(u).add(new Node(v, weight));
        adjList.get(v).add(new Node(u, weight)); // If the graph is undirected
    }

    public void dijkstra(int start) {
        PriorityQueue<Node> pq = new PriorityQueue<>();
        int[] distances = new int[numVertices];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[start] = 0;
        pq.add(new Node(start, 0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
            int u = current.vertex;

            for (Node neighbor : adjList.get(u)) {
                int v = neighbor.vertex;
                int weight = neighbor.distance;
                int newDist = distances[u] + weight;

                if (newDist < distances[v]) {
                    distances[v] = newDist;
                    pq.add(new Node(v, newDist));
                }
            }
        }

        // Print the shortest distances
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < numVertices; i++) {
            System.out.println(i + " \t\t " + distances[i]);
        }
    }
}

public static void main(String[] args) {
    int numVertices = 9;
    Graph graph = new Graph(numVertices);

    graph.addEdge(0, 1, 4);
    graph.addEdge(0, 7, 8);
    graph.addEdge(1, 2, 8);
    graph.addEdge(1, 7, 11);
```

```
        graph.addEdge(2, 3, 7);
        graph.addEdge(2, 8, 2);
        graph.addEdge(2, 5, 4);
        graph.addEdge(3, 4, 9);
        graph.addEdge(3, 5, 14);
        graph.addEdge(4, 5, 10);
        graph.addEdge(5, 6, 2);
        graph.addEdge(6, 7, 1);
        graph.addEdge(6, 8, 6);
        graph.addEdge(7, 8, 7);

        int startVertex = 0;
        graph.dijkstra(startVertex);
    }
}
```

Explanation:

1.Graph Representation:
- A graph is represented using an adjacency list where each node (vertex) has a list of its neighboring nodes (vertices) and the corresponding edge weights.
- For undirected graphs, each edge is bidirectional, so the adjacency list reflects this by adding the edge in both directions.

2. Node Class:
- Each node in the graph has a vertex identifier and a distance value.
- The distance value represents the cost (or weight) to travel from the source vertex to this node.
- Nodes are compared based on their distance values, which helps in prioritizing nodes with the shortest known distance when using a priority queue.

3. Priority Queue:
- The algorithm uses a priority queue to efficiently select the next vertex with the smallest known distance.
- This helps in exploring the most promising paths first, ensuring the shortest paths are found.

4. Distance Array:
- An array is maintained to keep track of the shortest known distance from the source vertex to each vertex in the graph.
- Initially, all distances are set to infinity (or a very large number), except for the source vertex, which is set to zero.

5. Algorithm Steps:
- Start by initializing the distance array and adding the source vertex to the priority queue.
- While the priority queue is not empty:
- Extract the vertex with the smallest distance.
- For each neighboring vertex, calculate the potential new distance through the current vertex.
- If the new distance is smaller than the known distance, update the distance array and add the neighbor to the priority queue.

- *Repeat until all reachable vertices have been processed*.