Name: Vaishali Ramesh Kale

Email id : kalevaishalir16@gmail.com

Day 9 and 10:

Task 1: 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.

Solution::::

Explanation

1. Vertex Class:
- Represents a node in the graph with a label.
- Implements Comparable to allow priority queue operations based on minDistance.
- Includes methods for equality checks and proper hashing.
2. Edge Class:
- Represents an edge between two vertices with a specified weight.
3. Graph Class:
- Manages the vertices and edges using an adjacency list.
- addVertex: Adds a vertex to the graph.
- addEdge: Adds an edge between two vertices.
- dijkstra: Implements Dijkstra's algorithm to find the shortest path from the start vertex to all other vertices.
- Uses a priority queue to efficiently get the vertex with the smallest known distance.
- Updates distances of adjacent vertices and adds them to the priority queue if a shorter path is found.
- printShortestPaths: Prints the shortest path distances from the start vertex to all other vertices.
4. main Method:
    - Tests the Dijkstra's algorithm implementation by creating a graph, adding vertices and edges, and finding shortest paths from a starting vertex.

Explaination:::

Vertices: A, B, C, D, E

Edges with weights:

A- B: 4

A- C: 2

B- C: 5

B- D: 10

C- D: 3

D- E: 4

C- E: 8

Execution of Dijkstra's Algorithm from Vertex A:

1. Initialization:

- Start from A, set minDistance of A to 0.
- Set minDistance of all other vertices to infinity (which is represented by Integer.MAX_VALUE in the code).

2. Step-by-Step Calculation:

- Vertex A:
- Distance to A is 0.
- Update distance to B: 0 + 4 = 4 (A to B)
- Update distance to C: 0 + 2 = 2 (A to C)
- Vertex C (next closest vertex with current minDistance 2):
- Distance to D through C: 2 + 3 = 5 (C to D)
- Distance to E through C: 2 + 8 = 10 (C to E)
- Vertex B (next closest vertex with current minDistance 4):
- No update to D since current known shortest distance to D is 5.
- Vertex D (next closest vertex with current minDistance 5):
- Update distance to E through D: 5 + 4 = 9 (D to E)
- Vertex E (next closest vertex with current minDistance 9):
- All shortest paths to E are already determined.

3. Final Shortest Path Distances:

- A: 0
- B: 4
- C: 2
- D: 5
- E: 9

CODE::::

```
import java.util.*;

// Class to represent a vertex in the graph

class Vertex implements Comparable<Vertex> {

    String label;

    int minDistance = Integer.MAX_VALUE; // Initially set to infinity
```

```
    Vertex(String label) {
        this.label = label;

    // Override equals method to compare vertices based on their labels
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Vertex vertex = (Vertex) obj;
        return Objects.equals(label, vertex.label);
    }
    // Override hashCode method to ensure vertices can be used in collections like HashMap
and HashSet
    @Override
    public int hashCode() {
        return Objects.hash(label);
    }
    // Override toString method to return the label of the vertex
    @Override
    public String toString() {
        return this.label;
    }

    // Implement compareTo method to compare vertices based on their minDistance for
priority queue
    @Override
    public int compareTo(Vertex other) {
        return Integer.compare(this.minDistance, other.minDistance);
    }
}
```

```
// Class to represent an edge in the graph
class Edge {
    Vertex source;
    Vertex destination;
    int weight;
    Edge(Vertex source, Vertex destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }
}
// Class to represent the graph
class Graph {
    private final Map<String, Vertex> vertices = new HashMap<>();
    private final Map<Vertex, List<Edge>> adjList = new HashMap<>();

    // Method to add a vertex to the graph
    public void addVertex(String label) {
        Vertex v = new Vertex(label);
        vertices.put(label, v);
        adjList.putIfAbsent(v, new ArrayList<>());
    }
    // Method to add an edge between two vertices
    public void addEdge(String label1, String label2, int weight) {
        Vertex v1 = vertices.get(label1);
        Vertex v2 = vertices.get(label2);
        if (v1 != null && v2 != null) {
            adjList.get(v1).add(new Edge(v1, v2, weight));
            adjList.get(v2).add(new Edge(v2, v1, weight)); // For an undirected graph
```

```
    }
}

// Method to perform Dijkstra's algorithm
public void dijkstra(String startLabel) {
    Vertex source = vertices.get(startLabel);
    if (source == null) {
        System.out.println("Source vertex not found.");
        return;
    }
    PriorityQueue<Vertex> pq = new PriorityQueue<>();
    source.minDistance = 0;
    pq.add(source);

    while (!pq.isEmpty()) {
        Vertex u = pq.poll();

        for (Edge edge : adjList.get(u)) {
            Vertex v = edge.destination;
            int weight = edge.weight;
            int distanceThroughU = u.minDistance + weight;

            if (distanceThroughU < v.minDistance) {
                pq.remove(v); // Remove v from the queue if it exists
                v.minDistance = distanceThroughU;
                pq.add(v);
            }
        }
    }
```

```java
    printShortestPaths();
}

// Method to print the shortest paths from the source to all other vertices
private void printShortestPaths() {
    for (Vertex v : vertices.values()) {
        System.out.println("Vertex: " + v + ", Min Distance: " + v.minDistance);
    }
}

// Main method to test the Dijkstra's algorithm implementation
public static void main(String[] args) {
    Graph graph = new Graph();

    // Add vertices
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");
    graph.addVertex("E");

    // Add edges
    graph.addEdge("A", "B", 4);
    graph.addEdge("A", "C", 2);
    graph.addEdge("B", "C", 5);
    graph.addEdge("B", "D", 10);
    graph.addEdge("C", "D", 3);
    graph.addEdge("D", "E", 4);
```

```
        graph.addEdge("C", "E", 8);

        // Find shortest paths from vertex "A"
        System.out.println("Dijkstra's shortest path starting from vertex A:");
        graph.dijkstra("A");
    }
}
```

OUTPUT::::

```
Dijkstra's shortest path starting from vertex A:
Vertex: A, Min Distance: 0
Vertex: B, Min Distance: 4
Vertex: C, Min Distance: 2
Vertex: D, Min Distance: 5
Vertex: E, Min Distance: 9


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

****
****

Task 2: Kruskal’s Algorithm for MST Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

Solution::::

To implement Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a given connected, undirected graph with non-negative edge weights, we need to follow these steps:

1. Sort all edges in non-decreasing order of their weight.
2. Use a Union-Find data structure to keep track of the connected components.
3. Iterate through the sorted edges and add them to the MST if they don't form a cycle.

Explanation:

1. Edge Class:
- Represents an edge in the graph.
- Implements the Comparable interface to allow sorting of edges based on weight.
2. UnionFind Class:

- Implements the Union-Find data structure with path compression and union by rank.

3. KruskalMST Class:

- Contains the graph represented by its edges.
- The addEdge method adds an edge to the graph.
- The findMST method implements Kruskal's algorithm to find and return the edges in the MST.
- The main method demonstrates how to use the KruskalMST class to find the MST of a sample graph.

In the findMST method, we sort all edges, iterate through them, and use the Union-Find data structure to add edges to the MST while ensuring no cycles are formed. This approach ensures the MST is found efficiently.

CODE::::

```
import java.util.*;


class Edge implements Comparable<Edge> {
    int src, dest, weight;


    // Constructor
    public Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }


    // Compare two edges based on their weight
    public int compareTo(Edge compareEdge) {
        return this.weight- compareEdge.weight;
    }
}
```

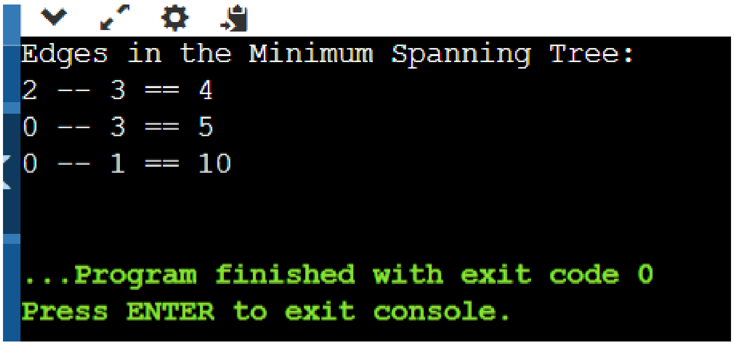
```
class UnionFind {
    private int[] parent, rank;

    // Constructor
    public UnionFind(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    // Find the root of the element with path compression
    public int find(int p) {
        if (parent[p] != p) {
            parent[p] = find(parent[p]); // Path compression
        }
        return parent[p];
    }

    // Union two subsets by rank
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);

        if (rootP != rootQ) {
            if (rank[rootP] > rank[rootQ]) {
                parent[rootQ] = rootP;
```

```
            } else if (rank[rootP] < rank[rootQ]) {
                parent[rootP] = rootQ;
            } else {
                parent[rootQ] = rootP;
                rank[rootP]++;
            }
        }
    }
}

public class KruskalMST {
    private int vertices; // Number of vertices in the graph
    private LinkedList<Edge> edges; // List of all edges

    // Constructor
    public KruskalMST(int vertices) {
        this.vertices = vertices;
        edges = new LinkedList<>();
    }

    // Function to add an edge to the graph
    public void addEdge(int src, int dest, int weight) {
        edges.add(new Edge(src, dest, weight));
    }

    // Function to find the Minimum Spanning Tree using Kruskal's algorithm
    public LinkedList<Edge> findMST() {
        LinkedList<Edge> mst = new LinkedList<>(); // To store the resultant MST
        Collections.sort(edges); // Sort all the edges based on their weight
```

```
        // Create a Union-Find to keep track of connected components
        UnionFind uf = new UnionFind(vertices);

        // Iterate through all sorted edges
        for (Edge edge : edges) {
            int rootSrc = uf.find(edge.src);
            int rootDest = uf.find(edge.dest);

            // If including this edge doesn't form a cycle
            if (rootSrc != rootDest) {
                mst.add(edge); // Include this edge in the MST
                uf.union(rootSrc, rootDest); // Union the sets
            }
        }

        return mst;
    }

    public static void main(String[] args) {
        int vertices = 4; // Number of vertices in the graph
        KruskalMST graph = new KruskalMST(vertices);

        // Adding edges to the graph
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);
```

```
        // Find the MST
        LinkedList<Edge> mst = graph.findMST();

        // Print the MST
        System.out.println("Edges in the Minimum Spanning Tree:");
        for (Edge edge : mst) {
            System.out.println(edge.src + "-- " + edge.dest + " == " + edge.weight);
        }
    }
}
```

OUTPUT:::



****
****

Task 3: Union-Find for Cycle Detection Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

Solution:::

Explanation:

1. UnionFind Class:

- Constructor: Initializes the parent and rank arrays. Each node is its own parent initially.
- find Method: Uses path compression to flatten the structure, making future operations faster.
- union Method: Implements union by rank, attaching the smaller tree to the root of the deeper tree.

- isConnected Method: Checks if two elements are in the same subset by comparing their roots.

2. CycleDetection Class:
   - detectCycle Method: Takes a list of edges and the number of vertices. For each edge, it checks if the vertices are already connected. If they are, a cycle is detected. Otherwise, it unites the subsets.
3. main Method:
   - Provides an example usage of the cycle detection method.
   - This implementation efficiently detects cycles in an undirected graph using the Union-Find data structure with path compression and union by rank.

```
class UnionFind {
    private int[] parent;
    private int[] rank;

    // Constructor to initialize the Union-Find structure
    public UnionFind(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i; // Each node is initially its own parent
            rank[i] = 1;   // Initialize the rank of each node to 1
        }
    }

    // Find the root of the element with path compression
    public int find(int p) {
        if (parent[p] != p) {
            parent[p] = find(parent[p]); // Path compression
        }
        return parent[p];
```

```
    }

    // Union two subsets by rank
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP != rootQ) {
            // Union by rank
            if (rank[rootP] > rank[rootQ]) {
                parent[rootQ] = rootP;
            } else if (rank[rootP] < rank[rootQ]) {
                parent[rootP] = rootQ;
            } else {
                parent[rootQ] = rootP;
                rank[rootP]++;
            }
        }
    }

    // Check if two elements are in the same subset
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
}
public class CycleDetection {
    // Function to detect a cycle in an undirected graph
    public static boolean detectCycle(int[][] edges, int numVertices) {
        UnionFind uf = new UnionFind(numVertices);
        for (int[] edge : edges) {
```

```
            int u = edge[0];
            int v = edge[1];
            if (uf.isConnected(u, v)) {
                return true; // Cycle detected
            }
            uf.union(u, v);
        }
        return false; // No cycle detected
    }
    public static void main(String[] args) {
        // Example usage:
        int[][] edges = {
            {0, 1},
            {1, 2},
            {2, 3},
            {3, 4},
            {4, 1}  // This edge creates a cycle
        };
        int numVertices = 5;

        if (detectCycle(edges, numVertices)) {
            System.out.println("Cycle detected");
        } else {
            System.out.println("No cycle detected");
        }
    }
}
```

OUTPUT::::

```
Cycle detected


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