Prajwalhonashetti143@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.

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights. This means it finds the shortest paths from a starting node (source) to all other nodes in the graph

Algorithm Steps

Initialization:

- Set the distance to the start node to 0 and to all other nodes to infinity.
- Insert the start node into the priority queue with a distance of 0.

Processing:

- Extract the node with the smallest distance from the priority queue.
- For each neighboring node, calculate the potential new path distance.
- If this new path is shorter than the currently known path to the neighbor, update the shortest distance and insert the neighbor into the priority queue with the updated distance.

Termination:

- The algorithm continues until the priority queue is empty, which means all reachable
- nodes have been processed and their shortest paths determined.

Time Complexity

The time complexity of Dijkstra's algorithm is O((V+E)logV), where V is the number of vertices and E is the number of edges. This is achieved by using a priority queue to efficiently fetch and update the shortest paths.

```
import java.util.*;

public class Dijkstra {

  public static class Node implements Comparable<Node> {

    int vertex;

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

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

public static Map<Integer, Integer> dijkstra(Map<Integer, List<Node>> graph, int start) {
    // Number of nodes in the graph
    int n = graph.size();

    // Dictionary to store the shortest distance to each node
    Map<Integer, Integer> distances = new HashMap<>();
    for (int vertex : graph.keySet()) {
        distances.put(vertex, Integer.MAX_VALUE);
    }
    distances.put(start, 0);

    // Priority queue to select the node with the smallest distance
    PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
    priorityQueue.add(new Node(start, 0));

    while (!priorityQueue.isEmpty()) {
        Node currentNode = priorityQueue.poll();
        int currentDistance = currentNode.weight;
        int currentVertex = currentNode.vertex;

        // Skip processing if we find a larger distance (as it won't be optimal)
        if (currentDistance > distances.get(currentVertex)) {
            continue;
        }
        // Check neighbors of the current node
        for (Node neighbor : graph.get(currentVertex)) {
            int distance = currentDistance + neighbor.weight;

            // If a shorter path to the neighbor is found
            if (distance < distances.get(neighbor.vertex)) {
                distances.put(neighbor.vertex, distance);
                priorityQueue.add(new Node(neighbor.vertex, distance));
            }
        }
    }

    return distances;
}

public static void main(String[] args) {
    // Example graph represented as an adjacency list
    Map<Integer, List<Node>> graph = new HashMap<>();
    graph.put(0, Arrays.asList(new Node(1, 4), new Node(2, 1)));
    graph.put(1, Arrays.asList(new Node(3, 1)));
    graph.put(2, Arrays.asList(new Node(1, 2), new Node(3, 5)));
    graph.put(3, new ArrayList<>());
```

```
        int startNode = 0;

        Map<Integer, Integer> distances = dijkstra(graph, startNode);

        // Print shortest distances from startNode to all other nodes

        for (Map.Entry<Integer, Integer> entry : distances.entrySet()) {

          System.out.println("Shortest distance from node " + startNode + " to node " +
entry.getKey() + " is " + entry.getValue());

        }

    }

}
```

Output:

```
Shortest distance from node 0 to node 0 is 0
Shortest distance from node 0 to node 1 is 3
Shortest distance from node 0 to node 2 is 1
Shortest distance from node 0 to node 3 is 4
```

Explanation of the Output:

- Shortest distance from node 0 to node 0 is 0: The distance from the start node to itself is always 0.
- Shortest distance from node 0 to node 1 is 3: The shortest path from node 0 to node 1 is via node 2, with a total weight of 1 (0 -> 2) + 2 (2 -> 1) = 3.
- Shortest distance from node 0 to node 2 is 1: Direct edge from node 0 to node 2 with weight 1.
- Shortest distance from node 0 to node 3 is 4: The shortest path from node 0 to node 3 is via node 1, with a total weight of 1 (0 -> 2) + 2 (2 -> 1) + 1 (1 -> 3) = 4.

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.

Steps of Kruskal's Algorithm

- Sort all the edges in the graph in non-decreasing order of their weights.
- Initialize a forest (a set of trees), where each vertex is a separate tree.
- Iterate through the sorted edges and for each edge, if the edge connects two different trees, add it to the MST and merge the two trees.
- Stop when there are V−1 edges in the MST, where V is the number of vertices.

```java
import java.util.*;

class Edge implements Comparable<Edge> {
    int source, destination, weight;

    public Edge(int source, int destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }

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

class Subset {
    int parent, rank;
}

public class Kruskal {
    int vertices;
    List<Edge> edges = new ArrayList<>();

    public Kruskal(int vertices) {
        this.vertices = vertices;
    }
```

```
public void addEdge(int source, int destination, int weight) {
    edges.add(new Edge(source, destination, weight));
}

public int find(Subset[] subsets, int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

public void union(Subset[] subsets, int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

public void kruskalMST() {
    List<Edge> result = new ArrayList<>();
    Collections.sort(edges);
```

```
        Subset[] subsets = new Subset[vertices];
        for (int v = 0; v < vertices; ++v) {
            subsets[v] = new Subset();
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }

        int e = 0;
        int i = 0;

        while (e < vertices - 1) {
            Edge nextEdge = edges.get(i++);
            int x = find(subsets, nextEdge.source);
            int y = find(subsets, nextEdge.destination);

            if (x != y) {
                result.add(nextEdge);
                union(subsets, x, y);
                e++;
            }
        }

        System.out.println("Edges in the Minimum Spanning Tree:");
        for (Edge edge : result) {
            System.out.println(edge.source + " -- " + edge.destination + " == " + edge.weight);
        }
    }

    public static void main(String[] args) {
        int vertices = 4;
        Kruskal graph = new Kruskal(vertices);

        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);

        graph.kruskalMST();
    }
}
```

Explanation

- **Edge Class**: Represents an edge in the graph. It implements Comparable<Edge> to sort edges by weight.
- **Subset Class:** Used in the Union-Find structure to keep track of the parent and rank of each vertex.

Kruskal Class:

- **Constructor**: Initializes the number of vertices and the list of edges.
- **addEdge Method**: Adds an edge to the graph.
- **find Method**: Finds the root of the set that contains a given vertex (with path compression).
- **union Method:** Unites two subsets into a single subset (by rank).
- **kruskalMST Method:** Implements Kruskal's algorithm to find the MST. It sorts the edges, processes each edge to determine if it forms a cycle, and adds it to the result if it doesn't.

Main Method: Sets up a graph, adds edges, and runs the Kruskal's algorithm to print the edges in the MST.

```
   0
  /|\
 10 | 6
```

```
/ | \
1  5  2
|  |  |
15 |  |
|  4  |
3-------3
```

Edge 2 -- 3 == 4: This is the smallest weight edge.

Edge 0 -- 3 == 5: The next smallest weight edge that doesn't form a cycle.

Edge 0 -- 1 == 10: The next smallest weight edge that doesn't form a cycle.

These edges collectively connect all the vertices with the minimum total weight of 4+5+10=19

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.

Union-Find Data Structure

The Union-Find data structure is useful for managing disjoint sets and supports two main operations efficiently:

- **Find**: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- **Union**: Join two subsets into a single subset.

With path compression, the find operation is optimized to flatten the structure of the tree whenever find is called, making the trees shallower and the operations faster over time.

Cycle Detection in an Undirected Graph

To detect a cycle in an undirected graph, we can use the Union-Find data structure:

- For each edge in the graph, use find to determine the roots of the sets for the two vertices of the edge.
- If the roots are the same, a cycle is detected.
- If the roots are different, perform a union to merge the sets.

```
import java.util.*;

public class UnionFind {

  private int[] parent;
private int[] rank;

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;
    }
}

public int find(int i) {
    if (parent[i] != i) {
        parent[i] = find(parent[i]); // Path compression
    }
    return parent[i];
}

public void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX != rootY) {
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

public static boolean hasCycle(List<int[]> edges, int numVertices) {
    UnionFind unionFind = new UnionFind(numVertices);

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];

        int rootU = unionFind.find(u);
        int rootV = unionFind.find(v);

        if (rootU == rootV) {
            return true; // Cycle detected
        }

        unionFind.union(u, v);
    }

    return false; // No cycle found
}

public static void main(String[] args) {
    int numVertices = 4;
    List<int[]> edges = Arrays.asList(
            new int[]{0, 1},
            new int[]{1, 2},
            new int[]{2, 3},
            new int[]{3, 0}
        );

        boolean hasCycle = UnionFind.hasCycle(edges, numVertices);
        System.out.println("Graph contains cycle: " + hasCycle);
    }
}
```

Explanation

Union-Find Class:

- parent array: Tracks the parent of each element.
- rank array: Helps in keeping the tree shallow by attaching the smaller tree under the root of the deeper tree.
- find(int i): Uses path compression to find the root of the element i.
- union(int x, int y): Uses union by rank to attach the smaller tree under the root of the deeper tree.

Cycle Detection Method:

- The hasCycle method takes a list of edges and the number of vertices.
- For each edge, it finds the roots of the vertices.
- If both vertices have the same root, a cycle is detected.
- If not, it unites the two sets.

Main Method:

- Creates an example graph with a cycle.
- Calls the hasCycle method to check if the graph contains a cycle.
- Prints the result.

Output:

```
Graph contains cycle: true
```