Day 9 and 10: Manisha Assignment
Task 1: Dijkstra’s Shortest Path Finder

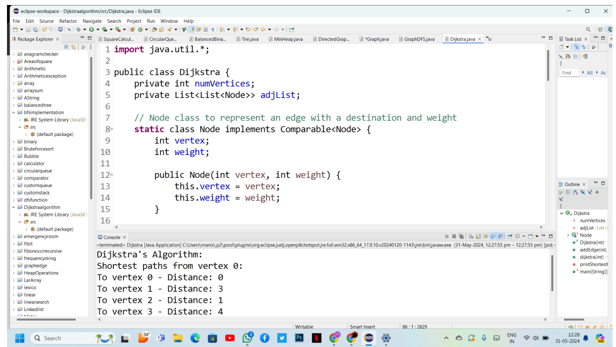
Code Dijkstra’s algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

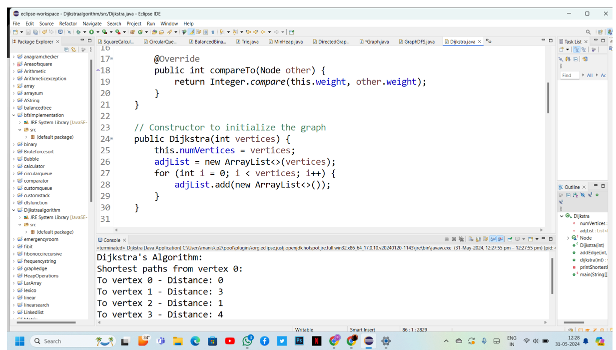
Explanation

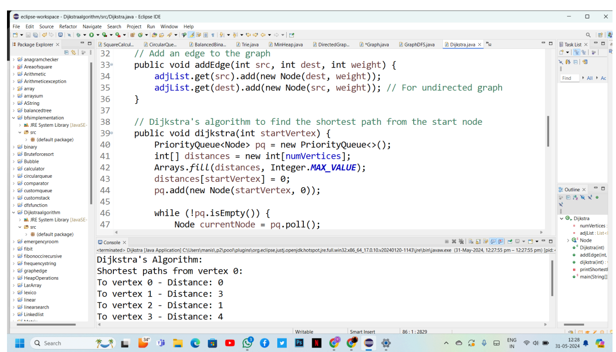
- Graph Initialization: The graph is initialized with 6 vertices.
- Adding Edges: Edges with weights are added between the vertices to form the graph.
- Dijkstra’s Algorithm:
- Initialize the distance array with infinity (except the start vertex which is set to 0).
- Use a priority queue to process vertices in order of their current known shortest distance.
- For each vertex processed, update the distances to its neighbors if a shorter path is found.
- Print Shortest Paths: The `printShortestPaths` method prints the shortest distances from the start vertex to all other vertices.

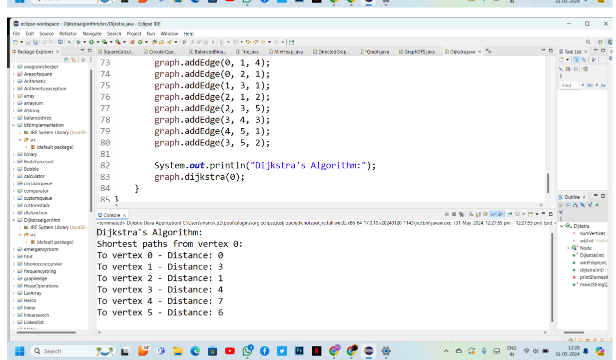
Running the Program in Eclipse

1. Open Eclipse: Create a new Java project and a new Java class named `Dijkstra`.
2. Copy the Code: Copy the provided code into the `Dijkstra.java` file.
3. Run the Program: Right-click on the file and select `Run As -> Java Application`.
4. View Output: The output will be displayed in the Eclipse console, showing the shortest paths from vertex 0.









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.

Step-by-Step Explanation

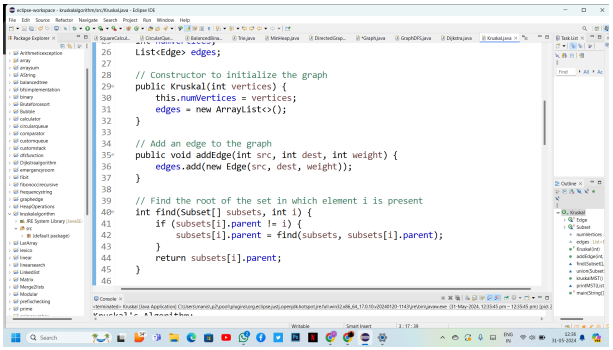
1. Graph Representation: Use an edge list to represent the graph.
2. Kruskal's Algorithm:
   - Sort all edges by their weight.
   - Add edges to the MST, ensuring no cycles using the Union-Find data structure.
3. Disjoint Set (Union-Find): Helps in detecting cycles efficiently.

Explanation

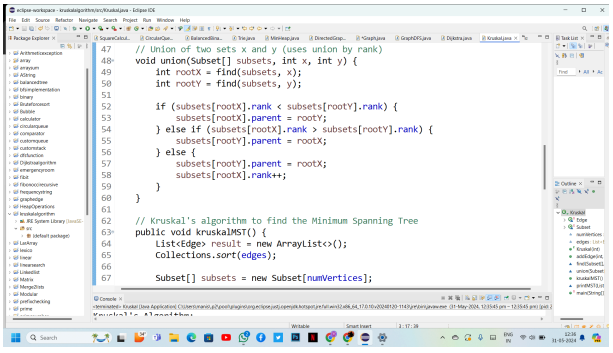
- Graph Initialization: The graph is initialized with 4 vertices.
- Adding Edges: Edges with weights are added between the vertices to form the graph.
- Kruskal's Algorithm:
  - Sort all edges by their weight.
  - Use a union-find data structure to add edges to the MST without forming a cycle.
  - Continue until the MST includes `numVertices - 1` edges.
- Print MST: The `printMST` method prints the edges included in the Minimum Spanning Tree.

Running the Program in Eclipse

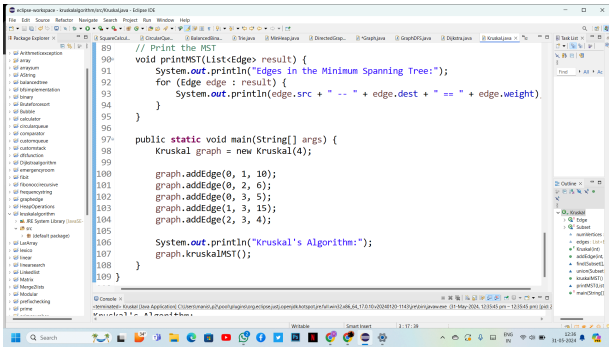
1. **Open Eclipse:** Create a new Java project and a new Java class named `Kruskal`.
2. **Copy the Code:** Copy the provided code into the `Kruskal.java` file.
3. **Run the Program:** Right-click on the file and select `Run As -> Java Application`.
4. **View Output:** The output will be displayed in the Eclipse console, showing the edges included in the MST.



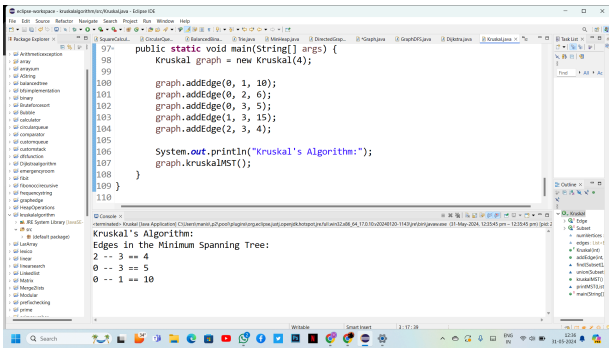
```
26      List<Edge> edges;
27
28      // Constructor to initialize the graph
29      public Kruskal(int vertices) {
30          this.numVertices = vertices;
31          edges = new ArrayList<>();
32      }
33
34      // Add an edge to the graph
35      public void addEdge(int src, int dest, int weight) {
36          edges.add(new Edge(src, dest, weight));
37      }
38
39      // Find the root of the set in which element i is present
40      int find(Subset[] subsets, int i) {
41          if (subsets[i].parent != i) {
42              subsets[i].parent = find(subsets, subsets[i].parent);
43          }
44          return subsets[i].parent;
45      }
46
```



```
47      // Union of two sets x and y (uses union by rank)
48      void union(Subset[] subsets, int x, int y) {
49          int rootX = find(subsets, x);
50          int rootY = find(subsets, y);
51
52          if (subsets[rootX].rank < subsets[rootY].rank) {
53              subsets[rootX].parent = rootY;
54          } else if (subsets[rootX].rank > subsets[rootY].rank) {
55              subsets[rootY].parent = rootX;
56          } else {
57              subsets[rootY].parent = rootX;
58              subsets[rootX].rank++;
59          }
60      }
61
62      // Kruskal's algorithm to find the Minimum Spanning Tree
63      public void kruskalMST() {
64          List<Edge> result = new ArrayList<>();
65          Collections.sort(edges);
66
67          Subset[] subsets = new Subset[numVertices];
```



```
89      // Print the MST
90      void printMST(List<Edge> result) {
91          System.out.println("Edges in the Minimum Spanning Tree:");
92          for (Edge edge : result) {
93              System.out.println(edge.src + " -- " + edge.dest + " == " + edge.weight);
94          }
95      }
96
97      public static void main(String[] args) {
98          Kruskal graph = new Kruskal(4);
99
100         graph.addEdge(0, 1, 10);
101         graph.addEdge(0, 2, 6);
102         graph.addEdge(0, 3, 5);
103         graph.addEdge(1, 3, 15);
104         graph.addEdge(2, 3, 4);
105
106         System.out.println("Kruskal's Algorithm:");
107         graph.kruskalMST();
108     }
109 }
```



```
97      public static void main(String[] args) {
98          Kruskal graph = new Kruskal(4);
99
100         graph.addEdge(0, 1, 10);
101         graph.addEdge(0, 2, 6);
102         graph.addEdge(0, 3, 5);
103         graph.addEdge(1, 3, 15);
104         graph.addEdge(2, 3, 4);
105
106         System.out.println("Kruskal's Algorithm:");
107         graph.kruskalMST();
108     }
109 }
110

Kruskal's Algorithm:
Edges in the Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

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.

Step-by-Step Explanation

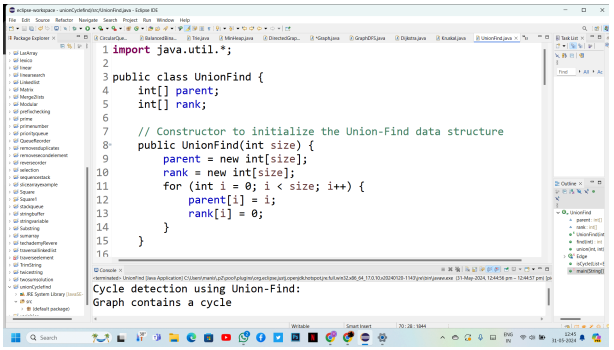
1. **Union-Find Data Structure**: Implement the Union-Find data structure with path compression.
2. **Cycle Detection**: Use the Union-Find data structure to detect cycles in an undirected graph.
3. **Graph Representation:** Use an edge list to represent the graph.

Explanation

- **Union-Find Initialization:** The Union-Find data structure is initialized with each vertex being its own parent and having a rank of 0.
- **Adding Edges**: Edges are added between the vertices to form the graph.
- Cycle Detection:
  - For each edge, find the root of both vertices.
  - If the roots are the same, a cycle is detected.
  - Otherwise, unite the sets containing the two vertices.
- **Path Compression:** During the `find` operation, path compression is used to flatten the structure, ensuring that future operations are faster.

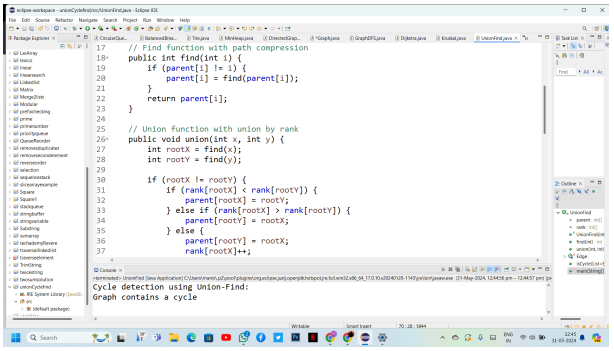
Running the Program in Eclipse

1. **Open Eclipse:** Create a new Java project and a new Java class named `UnionFind`.
2. **Copy the Code**: Copy the provided code into the `UnionFind.java` file.
3. **Run the Program**: Right-click on the file and select `Run As -> Java Application`.
4. **View Output**: The output will be displayed in the Eclipse console, indicating whether the graph contains a cycle.



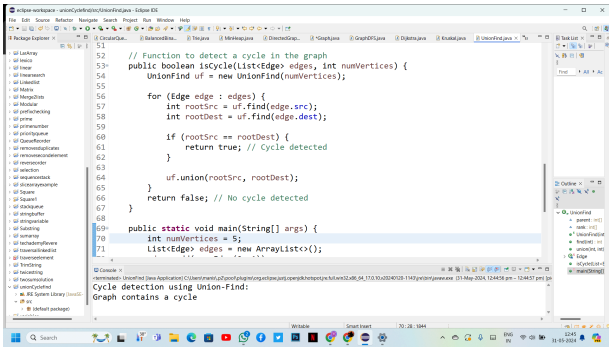
```
import java.util.*;

public class UnionFind {
    int[] parent;
    int[] rank;

    // Constructor to initialize the Union-Find data structure
    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;
        }
    }
```

```
Cycle detection using Union-Find:
Graph contains a cycle
```



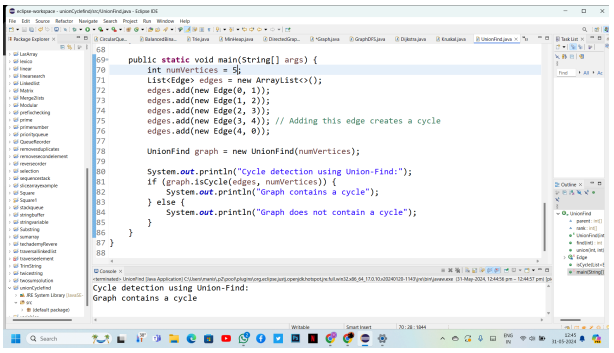
```
    // Find function with path compression
    public int find(int i) {
        if (parent[i] != i) {
            parent[i] = find(parent[i]);
        }
        return parent[i];
    }

    // Union function with union by rank
    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]++;
```

```
Cycle detection using Union-Find:
Graph contains a cycle
```



```
    // Function to detect a cycle in the graph
    public boolean isCycle(List<Edge> edges, int numVertices) {
        UnionFind uf = new UnionFind(numVertices);

        for (Edge edge : edges) {
            int rootSrc = uf.find(edge.src);
            int rootDest = uf.find(edge.dest);

            if (rootSrc == rootDest) {
                return true; // Cycle detected
            }

            uf.union(rootSrc, rootDest);
        }
        return false; // No cycle detected
    }

    public static void main(String[] args) {
        int numVertices = 5;
        List<Edge> edges = new ArrayList<>();
```

```
Cycle detection using Union-Find:
Graph contains a cycle
```



```
    public static void main(String[] args) {
        int numVertices = 5;
        List<Edge> edges = new ArrayList<>();
        edges.add(new Edge(0, 1));
        edges.add(new Edge(1, 2));
        edges.add(new Edge(2, 3));
        edges.add(new Edge(3, 4)); // Adding this edge creates a cycle
        edges.add(new Edge(4, 0));

        UnionFind graph = new UnionFind(numVertices);

        System.out.println("Cycle detection using Union-Find:");
        if (graph.isCycle(edges, numVertices)) {
            System.out.println("Graph contains a cycle");
        } else {
            System.out.println("Graph does not contain a cycle");
        }
    }
}
```

```
Cycle detection using Union-Find:
Graph contains a cycle
```