

Day 9 and 10: Manisha Assignment

Task 1: Dijkstra's Shortest Path Finder

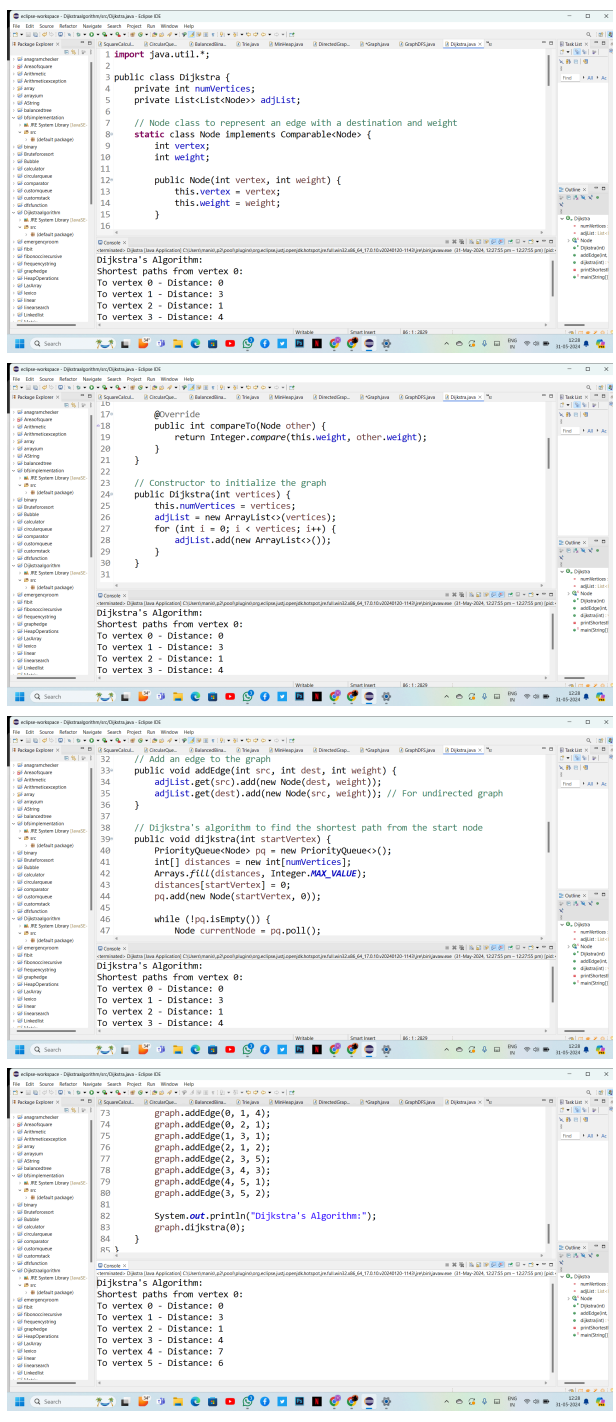
Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

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.



```
1 import java.util.*;
2
3 public class Dijkstra {
4     private int numVertices;
5     private List<ListNode> adjList;
6
7     // Node class to represent an edge with a destination and weight
8     static class Node implements Comparable<Node> {
9         int vertex;
10        int weight;
11
12        public Node(int vertex, int weight) {
13            this.vertex = vertex;
14            this.weight = weight;
15        }
16
17        @Override
18        public int compareTo(Node other) {
19            return Integer.compare(this.weight, other.weight);
20        }
21    }
22
23    // Constructor to initialize the graph
24    public Dijkstra(int vertices) {
25        this.numVertices = vertices;
26        adjList = new ArrayList<ListNode>(vertices);
27        for (int i = 0; i < vertices; i++) {
28            adjList.add(new ArrayList<Node>());
29        }
30    }
31
32    // Add an edge to the graph
33    public void addEdge(int src, int dest, int weight) {
34        adjList.get(src).add(new Node(dest, weight));
35        adjList.get(dest).add(new Node(src, weight)); // For undirected graph
36    }
37
38    // Dijkstra's algorithm to find the shortest path from the start node
39    public void dijkstra(int startVertex) {
40        PriorityQueue<Node> pq = new PriorityQueue<>();
41        int[] distances = new int[numVertices];
42        Arrays.fill(distances, Integer.MAX_VALUE);
43        distances[startVertex] = 0;
44        pq.add(new Node(startVertex, 0));
45
46        while (!pq.isEmpty()) {
47            Node currentNode = pq.poll();
48
49            // Dijkstra's Algorithm:
50            Shortest paths from vertex 0:
51            To vertex 0 - Distance: 0
52            To vertex 1 - Distance: 3
53            To vertex 2 - Distance: 1
54            To vertex 3 - Distance: 4
55
56            graph.addEdge(0, 1, 4);
57            graph.addEdge(0, 2, 1);
58            graph.addEdge(1, 3, 1);
59            graph.addEdge(2, 1, 2);
60            graph.addEdge(2, 3, 5);
61            graph.addEdge(3, 4, 3);
62            graph.addEdge(4, 5, 1);
63            graph.addEdge(3, 5, 2);
64
65            System.out.println("Dijkstra's Algorithm:");
66            graph.dijkstra(0);
67        }
68    }
69
70    // Print shortest paths from the start vertex to all other vertices
71    public void printShortestPaths(int startVertex) {
72        dijkstra(startVertex);
73    }
74
75    // Main method to test the Dijkstra's algorithm
76    public static void main(String[] args) {
77        Dijkstra graph = new Dijkstra(6);
78        graph.addEdge(0, 1, 4);
79        graph.addEdge(0, 2, 1);
80        graph.addEdge(1, 3, 1);
81        graph.addEdge(2, 1, 2);
82        graph.addEdge(2, 3, 5);
83        graph.addEdge(3, 4, 3);
84        graph.addEdge(4, 5, 1);
85        graph.addEdge(3, 5, 2);
86        graph.printShortestPaths(0);
87    }
88}
```

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 $\text{numVertices} - 1$ 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.

```
25 List<Edge> edges;
26
27 // Constructor to initialize the graph
28 public kruskal(int vertices) {
29     this.numVertices = vertices;
30     edges = new ArrayList<>();
31 }
32
33 // Add an edge to the graph
34 public void addEdge(int src, int dest, int weight) {
35     edges.add(new Edge(src, dest, weight));
36 }
37
38 // Find the root of the set in which element i is present
39 int find(Subset[] subsets, int i) {
40     if (subsets[i].parent != i) {
41         subsets[i].parent = find(subsets, subsets[i].parent);
42     }
43     return subsets[i].parent;
44 }
45
46 // Union of two sets x and y (uses union by rank)
47 void union(Subset[] subsets, int x, int y) {
48     int rootX = find(subsets, x);
49     int rootY = find(subsets, y);
50
51     if (subsets[rootX].rank < subsets[rootY].rank) {
52         subsets[rootX].parent = rootY;
53     } else if (subsets[rootX].rank > subsets[rootY].rank) {
54         subsets[rootY].parent = rootX;
55     } else {
56         subsets[rootY].parent = rootX;
57         subsets[rootX].rank++;
58     }
59 }
60
61 // Kruskal's algorithm to find the Minimum Spanning Tree
62 public void kruskalMST() {
63     List<Edge> result = new ArrayList<>();
64     Collections.sort(edges);
65
66     Subset[] subsets = new Subset[numVertices];
67
68     for (int i = 0; i < numVertices; i++) {
69         subsets[i] = new Subset(i);
70     }
71
72     for (Edge edge : edges) {
73         int src = edge.src;
74         int dest = edge.dest;
75         int weight = edge.weight;
76
77         int rootX = find(subsets, src);
78         int rootY = find(subsets, dest);
79
80         if (rootX != rootY) {
81             union(subsets, rootX, rootY);
82             result.add(edge);
83         }
84     }
85
86     printMST(result);
87 }
88
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 }
```

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

```
1 import java.util.*;
2
3 public class UnionFind {
4     int[] parent;
5     int[] rank;
6
7     // Constructor to initialize the Union-Find data structure
8     public UnionFind(int size) {
9         parent = new int[size];
10        rank = new int[size];
11        for (int i = 0; i < size; i++) {
12            parent[i] = i;
13            rank[i] = 0;
14        }
15    }
16
17    // Find function with path compression
18    public int find(int i) {
19        if (parent[i] != i) {
20            parent[i] = find(parent[i]);
21        }
22        return parent[i];
23    }
24
25    // Union function with union by rank
26    public void union(int x, int y) {
27        int rootx = find(x);
28        int rooty = find(y);
29
30        if (rootx != rooty) {
31            if (rank[rootx] < rank[rooty]) {
32                parent[rootx] = rooty;
33            } else if (rank[rootx] > rank[rooty]) {
34                parent[rooty] = rootx;
35            } else {
36                parent[rooty] = rootx;
37                rank[rootx]++;
38            }
39        }
40    }
41
42    // Function to detect a cycle in the graph
43    public boolean iscycle(List<Edge> edges, int numVertices) {
44        UnionFind uf = new UnionFind(numVertices);
45
46        for (Edge edge : edges) {
47            int rootSrc = uf.find(edge.src);
48            int rootDest = uf.find(edge.dest);
49
50            if (rootSrc == rootDest) {
51                return true; // Cycle detected
52            }
53
54            uf.union(rootSrc, rootDest);
55        }
56
57        return false; // No cycle detected
58    }
59
60    public static void main(String[] args) {
61        int numVertices = 5;
62        List<Edge> edges = new ArrayList<>();
63
64        // Adding edges to form a cycle: 0-1, 1-2, 2-3, 3-4, 4-0
65        edges.add(new Edge(0, 1));
66        edges.add(new Edge(1, 2));
67        edges.add(new Edge(2, 3));
68        edges.add(new Edge(3, 4));
69        edges.add(new Edge(4, 0));
70
71        UnionFind graph = new UnionFind(numVertices);
72
73        System.out.println("Cycle detection using Union-Find:");
74        if (graph.iscycle(edges, numVertices)) {
75            System.out.println("Graph contains a cycle");
76        } else {
77            System.out.println("Graph does not contain a cycle");
78        }
79    }
80}
```

Cycle detection using Union-Find:
Graph contains a cycle