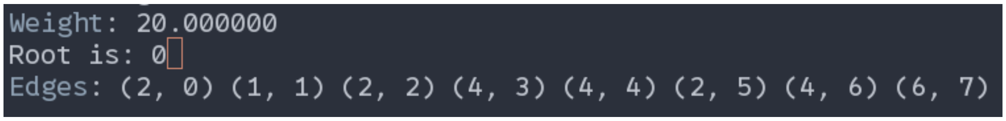
Assignment 11: Weighted Graphs
Part 2: Minimum Spanning Tree with Kruskal's Algorithm
CS3305/W01 Data Structures

Casey Hampson

November 20, 2024

Program Output



```
Weight: 20.000000
Root is: 0
Edges: (2, 0) (1, 1) (2, 2) (4, 3) (4, 4) (2, 5) (4, 6) (6, 7)
```

Source Code

```
// Name: Casey
// Class: CS 3305/W01
// Term: Fall 2024
// Instructor:  Sharon Perry
// Assignment:  11-Part-2-Kruskals

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;


class WeightedGraphAdjacencyMatrix<V> extends WeightedGraph<V> {
        WeightedGraphAdjacencyMatrix(int[][] edges, int num_vertices) {
                super(edges, num_vertices);
        }


        @Override
        public MST getMinimumSpanningTree(int starting_vertex) {
                // first grab the sorted edges
                List<WeightedEdge> sorted_edges = this.sortEdges();

                // define the parent/rank sets
                int size = this.getSize();
                int[] parent = new int[size];
                int[] rank = new int[size];
                for (int i=0; i<size; i++) {
                        parent[i] = i;
                        rank[i] = 0;
                }

                // create the vertex list and initial total weight
                List<Integer> T = new ArrayList<>();
                double total_weight = 0;

                while ((T.size() < size) && !sorted_edges.isEmpty()) {
                        // grab an edge
                        WeightedEdge proposed_edge = sorted_edges.remove(0);


                        // check if it leads to a cycle
                        int p1 = find(parent, proposed_edge.u);
                        int p2 = find(parent, proposed_edge.v);

                        if (p1 != p2) {
                                // if not, handle updating of parent and ranks
                                union(parent, rank, proposed_edge.u, proposed_edge.v);

                                // add the edge's vertices to the vertex list
                                // if they are not already there
                                if (!T.contains(proposed_edge.u)) T.add(proposed_edge.u);
                                if (!T.contains(proposed_edge.v)) T.add(proposed_edge.v);
```

```java
                    }

                    // also increment the weight
                    total_weight += proposed_edge.weight;
            }

            // we now need to construct the parent array
            // and the list of vertices

            return new MST(starting_vertex, parent, T, total_weight);
    }


    // these two functions are from the geeksforgeeks site

    // finds the root/parent of a node
private int find(int[] parent, int u) {
            if (parent[u] == u) return u;
            parent[u] = find(parent, parent[u]);
            return parent[u];
    }

    // handles finding parent of a set
    private void union(int[] parent, int[] rank, int u, int v) {
            int p1 = find(parent, u);
            int p2 = find(parent, v);

            if (p1 != p2) {
                    if (rank[u] > rank[v]) parent[v] = u;
                    else if (rank[u] < rank[v]) parent[u] = v;
                    else {parent[v] = u; rank[u] += 1;}
            }
    }


    // this creates one single list of all the edges, sorted
    // from smallest to largest
    private List<WeightedEdge> sortEdges() {
            // we first just grab all the unique edges in the graph
            // and stick them all in one list
            // i.e. we don't include both (1,2) and (2,1)
            List<List<Edge>> edge_lists = this.neighbors;
            List<WeightedEdge> edge_list = new ArrayList<>();

            for (int i=0; i<edge_lists.size(); i++) {
                    for (Edge e : edge_lists.get(i)) {
                            // if the "from" vertex is greater,
                            // then we have already done the edge
                            if (e.u >= e.v) continue;
                            edge_list.add((WeightedEdge)e);
                    }
            }

            // weightededge supports comparing,
                // so we can just sort and return
                Collections.sort(edge_list);
                return edge_list;
        }
}
```

```
public class P2 {
        public static void main(String[] args) {
                // set up the graph given in the example
                final int num_vertices = 8;
                int[][] edges = {
                        {0, 2, 4}, {0, 5, 7},
                        {1, 4, 9}, {1, 7, 3},
                        {2, 0, 4}, {2, 3, 3}, {2, 5, 2}, {2, 6, 9},
                        {3, 2, 3}, {3, 4, 3}, {3, 6, 7},
                        {4, 1, 9}, {4, 3, 3}, {4, 6, 2}, {4, 7, 7},
                        {5, 0, 7}, {5, 2, 2}, {5, 6, 8},
                        {6, 2, 9}, {6, 3, 7}, {6, 4, 2}, {6, 5, 8}, {6, 7, 3},
                        {7, 1, 3}, {7, 4, 7}, {7, 6, 3}
                };

                WeightedGraphAdjacencyMatrix<Integer> weighted_graph =
                        new WeightedGraphAdjacencyMatrix<>(edges, num_vertices);

                WeightedGraph.MST mst = weighted_graph.getMinimumSpanningTree(0);
                System.out.printf("Weight: %f\n", mst.getTotalWeight());
                mst.printTree();
        }
}
```