University of Southampton

COMP1201
Tutorial 3
Trees and Graphs

Krasimir Marinov
Email: km1u17@soton.ac.uk
Student ID: 29545528
May 4, 2018

1 Binary Search Tree

1.1 Write a test program to test add, remove, contains and the iterator

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;
import java.util.Iterator;
/**
 * TesterBST Class
 * Tests the add, remove, contain method and the iterator of a BST
 */
public class TesterBST {
    BinarySearchTree<String> binarySearchTree;
    /**
     * Setting the BinarySearchTree
     */
    @Before
    public void setUp(){
        binarySearchTree=new BinarySearchTree<>();
    }
    /**
     * Testing the add method()
     * Adding String to an Empty Tree
     */
    @Test
    public void addOnEmptyTree(){
        binarySearchTree.add("Element");
        assertEquals("String : \"Element\" is not
             found",true,binarySearchTree.contains("Element"));
    }
    /**
     * Testing the add method()
     * Adding same element to the Tree
     * It should not add duplicate elements
     */
    @Test
    public void addSameElement(){
        BinarySearchTree<String> secondBST=new BinarySearchTree<>();
        secondBST.add("A");
        assertEquals("BST cannot add duplicate elements",false,secondBST.add("A"));
        secondBST=null;
    }
    /**
     * Testing the add method()
     * Adding many elements to Tree
     * Checking if the BST has them
     */
    @Test
    public void addManyElements(){

        String[] listStrings={"John","Declan","Woodhouse","Whittaker","King","Savage"};
        for(String testString:listStrings){
```

```
        assertEquals("Successfully added String
            \""+testString+"\"",true,binarySearchTree.add(testString));
        System.out.println("Successfuly added String \""+testString+"\"");
    }
}
/**
 * Testing the remove method()
 * Removing elements that are already in the tree
 */
@Test
public void removeActiveElement(){
    binarySearchTree.add(new String("Whittaker"));
    assertEquals("Binary search tree has not removed the element
        properly",true,binarySearchTree.remove("Whittaker"));
    assertEquals("Binary search tree has not removed the element
        properly",false,binarySearchTree.contains("Whittaker"));
}
/**
 * Testing the removing method()
 * Removing element that is not in the tree
 * @throws Exception
 */
@Test
public void removeInactiveElement() throws Exception{
    assertEquals("Binary search tree removes inactive
        nodes",false,binarySearchTree.remove("adsa"));
    assertEquals("Binary search tree contains inactive
        nodes",false,binarySearchTree.contains("adsa"));
}
/**
 * Testing the contain method()
 * Adding string to the BST and checking if the contain method contains the added string
 */
@Test
public void containOneElement(){
    binarySearchTree.add("John");
    assertEquals("Binary search tree does not contain the string:
        \"John\"",true,binarySearchTree.contains("John"));
}
/**
 * Testing the contain method()
 * Adding array of strings to the BST and checking if it contains them all
 */
@Test
public void cointainMoreElements(){
    String[] listStrings={"John","Declan","Woodhouse","Whittaker","King","Savage"};
    for(String testString:listStrings){
        assertEquals("Unsuccessfully added String
            \""+testString+"\"",true,binarySearchTree.add(testString));
        assertEquals("BST does not contain added
            strings",true,binarySearchTree.contains(testString));
        System.out.println("BST contains String \""+testString+"\"");
    }
}
/**
 * Testing the iterator
 * Testing if it returns the one and only element
```

```
 */
@Test
public void testIteratorForOneElement(){
    binarySearchTree.add("A");
    Iterator it=binarySearchTree.iterator();
    assertEquals("The iterator does not show that we have an
         element",true,it.hasNext());
    assertEquals("The iterator does not show the expected element","A",it.next());
}
/**
 * Testing the iterator
 * Testing if it returns the many elements
 */
@Test
public void testIteratorForManyElements(){
    // Added strings
    String[] listStrings={"A","B","C","D"};
    for(String testString:listStrings){
        binarySearchTree.add(testString);
    }
    // Iterator
    Iterator it=binarySearchTree.iterator();
    for(String testString:listStrings){
        assertEquals("The iterator does not show that we have an
             element",true,it.hasNext());
        assertEquals("The iterator does not show the expected
             element",testString,it.next());
    }
    // New BST
    binarySearchTree=new BinarySearchTree<>();
    // Different strings
    String[] newListStrings={"C","A","D","F"};
    for(String testString:newListStrings){
        binarySearchTree.add(testString);
    }
    it=binarySearchTree.iterator();
    // The expected array of strings that the iterator should iterate through
    newListStrings=new String[]{"A","C","D","F"};
    // Asserting that that is the correct order
    for(String testString:newListStrings){
        assertEquals("The iterator does not show that we have an
             element",true,it.hasNext());
        assertEquals("The iterator does not show the expected
             element",testString,it.next());
    }
}
/**
 * Testing the iterator
 * Removing nothing from the iterator
 */
@Test(expected = IllegalStateException.class)
public void testIteratorRemoveNothing(){
    binarySearchTree.add("A");
    Iterator it=binarySearchTree.iterator();
    it.remove();
}
/**
```

```
     * Testing the iterator
     * Removing one element
     */
    @Test
    public void testIteratorRemoveOneElement(){
        binarySearchTree.add("A");
        Iterator it=binarySearchTree.iterator();
        it.next();
        it.remove();
        assertEquals("The iterator has not removed
             anything",false,binarySearchTree.contains("A"));
    }
    /**
     * Testing the iterator
     * Removing more elements
     */
    @Test
    public void testIteratorRemoveMoreElements(){
        // Adding the strings
        String[] listStrings={"John","Josh","Cara","Anite"};
        for(String testString:listStrings){
            binarySearchTree.add(testString);
        }
        // Getting the iterator
        Iterator it=binarySearchTree.iterator();
        // Removing the elements
        while(it.hasNext()){
            it.next();
            it.remove();
        }
        for(String testString:listStrings){
           assertEquals("Iterator has not removed any
                objects",false,binarySearchTree.contains(testString));
        }
    }
}
```

1.2 Average Depth of a Tree

Write a method (using recursion) to count the average depth of a tree (you are allowed to modify the class). I have added the method **findLevel()** , which takes the root, the root level and the overallDistance and returns the sum of the level of each node in the tree. Then I have made a **findAverageDepth()** method that prints the average depth of the tree.

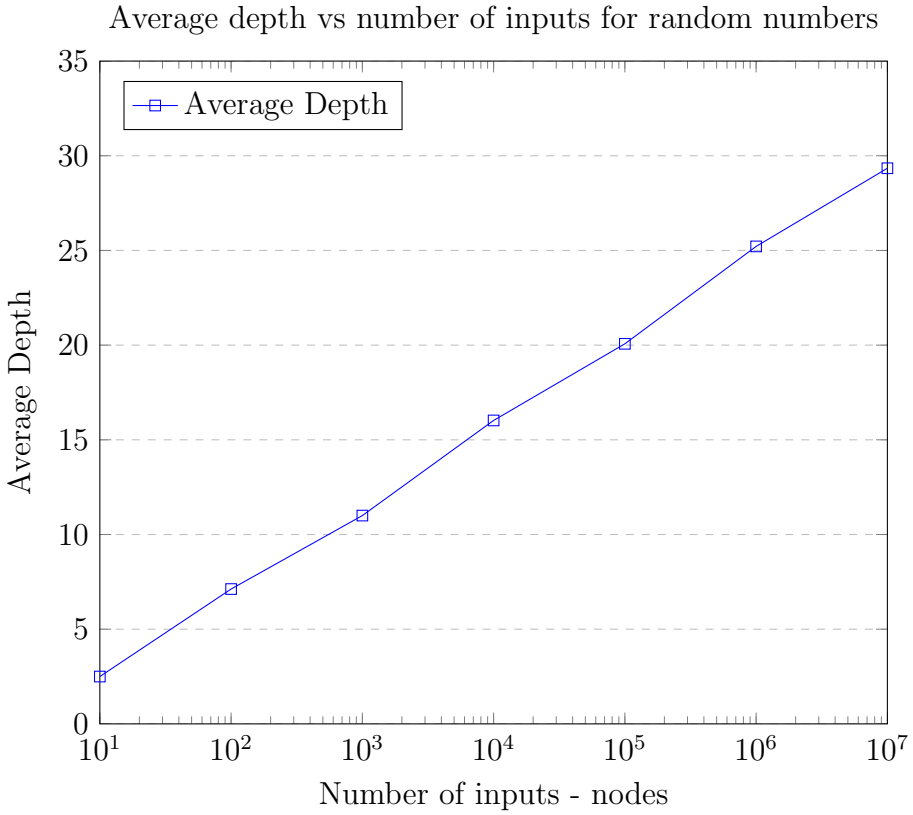
```
    /**
     * Finding level of every node and adding it to the overallDistance
     * @param node
     * @param level
     * @param overallDistance
     */
    public int findLevel(Node<T> node,int level,int overallDistance){
        /**
```

```
     * If the left node is not null we find its level
     * We add it to the overall Distance
     */
    if(node.left!=null) {
        overallDistance=findLevel(node.left, level+1,overallDistance);
    }
    /**
     * If the right node is not null we find its level
     * We add it to the overall Distance
     */
    if(node.right!=null){
        overallDistance=findLevel(node.right,level+1,overallDistance);
    }
    /** Return overallDistance*/
    return overallDistance+=level;
}
/** Wrapper method find Average Depth */
public void findAverageDepth(){
    /** Find the averageLevel of every node from the root */
    double averageLevel=findLevel(root,0,0);
    /** Divide it by the size of the BST */
    double averageDepth=averageLevel/size;
    /** Printing it out */
    System.out.println("The average depth "+averageDepth);
}
```

1.3 Average Depth vs Number of inputs

Write a program to compute the average depth versus the number of inputs for randomly generated numbers. Plot this as a graph.

```
/**
 * Method that computes AverageDepth of BST
 * For different samples ofrandom numbers
 */
public static void computeRandomAverageDepth(){
    /** The number of nodes */
    Integer[]
         numberNodes={1,10,50,100,500,1000,5000,10000,50000,100000,1000000,5000000,10000000};
    BinarySearchTree<Integer> binarySearchTree=new BinarySearchTree<>();
    for(Integer currentNumberNodes:numberNodes){
        Random random=new Random();
        while(binarySearchTree.size()<currentNumberNodes){
            /** Adding the random int */
            binarySearchTree.add(random.nextInt());
        }
        /** Printing the size and the average depth */
        System.out.println("The binary search tree has "+currentNumberNodes+" and size
             of"+binarySearchTree.size());
        binarySearchTree.findAverageDepth();
        binarySearchTree.clear();
    }
}
```



2 Graph Algorithms

2.1 Implementation Graph class

Write a Java Graph class which stores a set of vertices and their 2-D positions. It should also store a set of edges together with weights on the edges.

```
import java.lang.reflect.Array;
import java.util.*;

public class Graph {
    /** List of edges */
    private List<Edge> edges;
    /** List of edges */
    private List<Vertex> vertices;
    /** List of neighbours for every vertex */
    private Map<Vertex, List<Edge>> neighbours;

    /** Constructor for graph */
    public Graph() {
```

```
    this.edges = new ArrayList<>();
    this.vertices = new ArrayList<>();
    this.neighbours = new HashMap<>();
}
/** Add Vertex Method */
public boolean addVertex(Vertex vertex) {
    if (vertices.contains(vertex)) return false;
    else {
        vertices.add(vertex);
        return true;
    }
}
/** Add Edge Method */
public boolean addEdge(Edge edgeToAdd) {
    if(edges.contains(edgeToAdd)) return false;

    edges.add(edgeToAdd);

    Vertex vertex1 = edgeToAdd.getVertex1();
    Vertex vertex2 = edgeToAdd.getVertex2();

    neighbours.putIfAbsent(vertex1, new ArrayList<>());
    neighbours.putIfAbsent(vertex2, new ArrayList<>());

    neighbours.get(vertex1).add(edgeToAdd);
    neighbours.get(vertex2).add(edgeToAdd);

    return true;
}
/** Remove Vertex Method */
public boolean removeVertex(Vertex vertex) {
    if (!vertices.contains(vertex)) return false;
    else {
        vertices.remove(vertex);
        for (Edge edge : edges) {
            if (edge.getVertex1() == vertex) {
                edges.remove(edge);
                neighbours.get(edge.getVertex2()).remove(edge);
            }
            if (edge.getVertex2() == vertex) {
                edges.remove(edge);
                neighbours.get(edge.getVertex1()).remove(edge);
            }
            neighbours.remove(vertex);
        }
        return true;
    }
}
/**
 * RemoveEdge Method
 * @param edgeToRemove
 */
public boolean removeEdge(Edge edgeToRemove) {
    if(!edges.contains(edgeToRemove)) return false;
    edges.remove(edgeToRemove);
    Vertex firstVertex = edgeToRemove.getVertex1();
    Vertex secondVertex = edgeToRemove.getVertex2();
```

```
        neighbours.get(firstVertex).remove(edgeToRemove);
        neighbours.get(secondVertex).remove(edgeToRemove);
        return true;
    }
    /** Position method that gets the position of Vertex */
    public Position position(Vertex vertex) throws Exception {
        if (!vertices.contains(vertex)) throw new Exception("Vertex not found");
        int indexOfVertex = vertices.indexOf(vertex);
        Position vertexPosition = vertices.get(indexOfVertex).getPosition();
        return vertexPosition;
    }
    /** Method that checks if edge exists */
    public boolean edgeExist(Vertex vertex1, Vertex vertex2, int weight) {
        Edge toSearch = new Edge(vertex1, vertex2, weight);
        if (edges.contains(toSearch)) return true;
        else return false;
    }
    /**
     * Method that returns the weight between two vertices
     * @throws Exception
     */
    public int weight(Vertex vertex1, Vertex vertex2) throws Exception {
        if (neighbours.containsKey(vertex1)) throw new Exception("Edge not found");
        if (neighbours.containsKey(vertex2)) throw new Exception("Edge not found");
        List<Edge> edges = neighbours.get(vertex1);
        for (Edge currentEdge : edges) {
            if (currentEdge.getVertex2() == vertex2) return currentEdge.getWeight();
        }
        return 0;
    }
    /** Get vertices method returns list with vertices */
    public List<Vertex> getVertices() {
        return vertices;
    }
    /** Get edge method returns list with edges */
    public List<Edge> getEdges() {
        return edges;
    }
    /** Neighbours of every vertex */
    public Map<Vertex, List<Edge>> getNeighbours() {
        return neighbours;
    }
    /** Method that prints the vertices id */
    public void getVerticesID() {
        for (Vertex currentVertex : vertices) {
            System.out.println(currentVertex.getId());
        }
    }
}
/** Class for Position */
class Position {

    /** X and Y for Position */
    private int x;
    private int y;
    /** Constuctor */
```

```
    public Position() {
        this.x = 0;
        this.y = 0;
    }
    /** Constuctor */
    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
    /** Getters and Setters */
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
/**
 * Class Vertex
 * @param <T>
 */
class Vertex<T> {

    /** Member variables of the Vertex */
    private int id;
    private Position position;
    private T value;

    /** Constructor for Vertex */
    public Vertex(int id) {
        this.id = id;
        this.position = new Position();
        this.value = null;
    }
    /** Constructor for Vertex */
    public Vertex(int id, Position position) {
        this.id = id;
        this.position = position;
        this.value = null;
    }

    /** Constructor for Vertex */
    public Vertex(int id, Position position, T value) {
        this.id = id;
        this.position = position;
        this.value = value;
    }
    public int getId() {
        return id;
    }
    public Position getPosition() {
        return position;
    }
}
/**
 * Edge class
 */
```

```
class Edge {
    /** Member variables */
    private Vertex vertex1;
    private Vertex vertex2;
    private int weight;
    /** Constructor for Edge */
    public Edge(Vertex vertex1, Vertex vertex2, int weight) {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.weight = weight;
    }
    /** Getters and Setters */
    public Vertex getVertex1() {
        return vertex1;
    }
    public Vertex getVertex2() {
        return vertex2;
    }
    public int getWeight() {
        return weight;
    }
    public void setVertex1(Vertex vertex1) {
        this.vertex1 = vertex1;
    }
    public void setVertex2(Vertex vertex2) {
        this.vertex2 = vertex2;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
}
```

2.2 Random Graph

Write a method **randomGraph(n,p)** which generates a graph with n vertices placed at random in the unit square and where the probability of an edge is p. The weight of the edges are the distances between the nodes.Visualised using the GraphDisplay Class

```
import java.util.Random;
/** Display Random Graph Class */
public class DisplayRandomGraph {

    public static void main(String[] args) throws Exception{
        /** We create a random graph and display it */
        Graph graph=randomGraph(10,0.10);
        drawGraph(graph);
    }
    /**
     * Method that Generated random Graph
     * Given nodes - n
     * Given probability - p
     */
```

```
public static Graph randomGraph(int n, double p) throws Exception {
    Random random = new Random();
    /** Initialising the vertices and the Graph */
    Vertex[] vertices = new Vertex[n];
    Graph randomGraph = new Graph();
    /**
     * Initialising n vertices with random posX and posY
     * posX and posY<=400
     */
    for (int i = 0; i < n; i++) {
        int x = random.nextInt(400);
        int y = random.nextInt(400);

        vertices[i] = new Vertex(i, new Position(x, y));
        /** Adding the vertices to the graph */
        randomGraph.addVertex(vertices[i]);
    }
    randomGraph.getVerticesID();
    /**
     * For every pair of vertices we have a random probability
     * If random probability <= p then we have a edge
     */
    for (int i = n - 1; i > 1; i--) {
        for (int j = i - 1; j >= 0; j--) {

            System.out.println(i + "--->" + j);
            Double randomProbability = random.nextDouble();
            System.out.println(randomProbability + " " + p);
            if (randomProbability <= p) {
                int weight = pitagor(vertices[i].getPosition(),
                        vertices[j].getPosition());

                Edge edgeToAdd = new Edge(vertices[j], vertices[i], weight);
                // We add the edge to the random graph
                randomGraph.addEdge(edgeToAdd);
                System.out.println("Edge added between " + vertices[i] + " and " +
                        vertices[j] + " has the weight of " + weight);
            }
        }
        System.out.println();
    }
    return randomGraph;
}
/** Method used to calculate the distance between two positions */
public static int pitagor(Position one, Position two) {
    int c = (int) Math.sqrt((one.getX() - two.getX()) * (one.getX() - two.getX()) +
         (one.getY() - two.getY()) * (one.getY() - two.getY()));
    return c;
}
/**
 * Method that draws a given Graph
 */
public static void drawGraph(Graph g) throws Exception {
    GraphDisplay graphDisplay = new GraphDisplay();
    graphDisplay.showInWindow(400, 400, "A Random Graph");
    /** For every vertex we add it to the graphDisplay */
    for (Vertex vertexToAdd : g.getVertices())
```

```
            graphDisplay.addNode(vertexToAdd, vertexToAdd.getPosition().getX(),
                 vertexToAdd.getPosition().getY());
        /** For every edge we add it to the graphDisplay */
        for (Edge edgeToAdd : g.getEdges()) {
            Vertex firstVertex = edgeToAdd.getVertex1();
            Vertex secondVertex = edgeToAdd.getVertex2();
            graphDisplay.addEdge(firstVertex, secondVertex);
            Thread.sleep(50);
        }
    }
}
```

2.3 Prim's algorithm

Write a method that implements Prim's algorithm to compute the minimum spanning tree. Prim's algorithm is illsutrated in here. Your program should use your Graph class and the Java's collection class priority queue. You should visualise the answer using **GraphDisplay**.

```
    /**
     * Prim's Algorithm to compute
     * Minimum spanning tree
     * @param g
     */
    public static Graph Prim(Graph g) {
        /** Create a MST Graph */
        Graph MST = new Graph();

        /** Size of give graph g */
        int n = g.getVertices().size();

        /** Array used to store the distance */
        int[] d = new int[n];
        Arrays.fill(d, Integer.MAX_VALUE);

        /** Priority Queue that will hold the edgs */
        Queue<Edge> weightedEdges = new PriorityQueue<>(1, new Comparator<Edge>() {
            @Override
            public int compare(Edge o1, Edge o2) {
                // System.out.println(o1.getWeight()+" "+o2.getWeight());
                return o1.getWeight() - o2.getWeight();
            }
        });

        /** Neighbours of all vertices */
        Map<Vertex, List<Edge>> neighbours = g.getNeighbours();
        /** Adding them to the MST */
        for (Vertex vertex : g.getVertices()) MST.addVertex(vertex);
        /** Get the first node */
        Vertex node = g.getVertices().get(0);
        /** Setting the inital distance */
        d[0] = 0;
```

```
        /** Until we have every node - 1 */
        for (int i = 0; i < n - 1; i++) {

            /** We have visited the node */
            d[node.getId()] = 0;
            /**
             * For every edge we get the neighbours
             */
            for (Edge edge : neighbours.get(node)) {
                Vertex neighbourVertex = null;
                /** We get the node that is not visited */
                if (edge.getVertex2().getId() == node.getId()) neighbourVertex =
                     edge.getVertex1();
                else neighbourVertex = edge.getVertex2();

                /** Check for the weight */
                if (edge.getWeight() < d[neighbourVertex.getId()])
                     d[neighbourVertex.getId()] = edge.getWeight();
                weightedEdges.add(edge);
            }
            Vertex nextNode = node;
            Edge minEdge = null;

            /** While we do not have a not visited node */
            while (d[nextNode.getId()] <= 0) {

                /** We remove the top edge and check if we have not visited the nextNode*/
                minEdge = weightedEdges.remove();
                if (d[minEdge.getVertex2().getId()] > 0) {
                    nextNode = minEdge.getVertex2();
                    break;
                    //minEdge=weightedEdges.remove();

                } else if (d[minEdge.getVertex1().getId()] > 0) {
                    nextNode = minEdge.getVertex1();
                    break;
                    //minEdge=weightedEdges.remove();
                }
                System.out.println(nextNode.getId() + " d is " + d[nextNode.getId()]);
            }
            /** To the MST add the minimum Edge */
            MST.addEdge(minEdge);
            node = nextNode;
        }
        /** Return MST */
        return MST;
    }
}
```

After that we could visualise it using the drawGraph() method **2.2**.

```
    public static void main(String[] args) throws Exception {
      // Graph with 20 nodes and 100% for edge
      Graph graph = randomGraph(20, 1);
      Graph mst = Prim(graph);
      drawGraph(mst);
    }
```

2.4 Kruskal's Algorithm

Write a method that implements Kruskal's algorithm to compute the minimum spanning tree.

```
/**
 * Kruskal's algorithm to compute the MST
 * I use the DisjointSets class
 */
public static Graph Kruskal(Graph g) {
    /** MST graph */
    Graph MST = new Graph();

    /** The priority queue we used to store every edge */
    Queue<Edge> weightedEdges = new PriorityQueue<>(1, new Comparator<Edge>() {
        @Override
        public int compare(Edge o1, Edge o2) {
            // System.out.println(o1.getWeight()+" "+o2.getWeight());
            return o1.getWeight() - o2.getWeight();
        }
    });
    /** For every edge we add it to the priority queue */
    for (Edge edge : g.getEdges()) {
        weightedEdges.add(edge);
        System.out.println(" Edge x:" + edge.getVertex1().getId() + " y:" +
             edge.getVertex2().getId());
    }
    /** Edges accepted = 0 and n is equal to the size of the BST */
    int edgesAccepted = 0;
    int n = g.getVertices().size();

    /** For every vertex we add it to the MST */
    for (Vertex vertex : g.getVertices()) MST.addVertex(vertex);

    /** Initialise disjointSets dj*/
    DisjointSets dj = new DisjointSets(n);

    /** While the acceptedEdges < n- 1 */
    while (edgesAccepted < n - 1) {
        /** We remove the top edge */
        Edge currentEdge = weightedEdges.remove();

        int finda = dj.find(currentEdge.getVertex1().getId());
        int findb = dj.find(currentEdge.getVertex2().getId());

        /** If the current edge forms an acyclic graph with the set of the edges */
        if (finda != findb) {
            /** We union it and increment the edges Acceoted */
            dj.union(finda, findb);
            edgesAccepted++;
            /** Add it to the MST */
            MST.addEdge(currentEdge);
        }
    }
    /** Return the MST */
    return MST;
}
```

It uses the given Disjoint Sets Class

```
  public class DisjointSets {
    public DisjointSets(int numElements) {
        s = new int[numElements];
        for (int i = 0; i < s.length; i++)
            s[i] = -1;
    }

    public void union(int root1, int root2) {
        if (s[root2] < s[root1]) {
            s[root1] = root2;
        } else {
            if (s[root1] == s[root2])
                s[root1]--;
            s[root2] = root1;
        }
    }

    public int find(int x) {
        if (s[x] < 0)
            return x;
        else
            return s[x] = find(s[x]);
    }

    private int[] s;
}
```

After that we could visualise it using the drawGraph() method **2.2**.

```
     public static void main(String[] args) throws Exception {
       // Graph with 20 nodes and 100% for edge
       Graph graph = randomGraph(20, 1);
       Graph mst =Kruskal(graph);
       drawGraph(mst);
    }
```