SMPL programming assignment 1

Pleun Scholten (s4822250) & Anouk Prins (s4858956)

March 2018

1 Our implementation

Our implementation was written in Java. The main reason for this is that an object oriented language, which made it easy to create the necessary objects.

We had three different objects in our program: Graph, Vertex and Edge. The Graph consists of two ArrayLists of Vertices and Edges. A Vertex consists of a name, key and a parent Vertex. The Edge consists of a `firstVertex`, a `secondVertex` and a weight.

Our main loop is in the `MST_Prim` class. It contains a PriorityQueue, called frontier, which sorts the elements in the graph that are not in the MST by their weight. The comparator for this PriorityQueue is implemented in the class `WeightComparator`.

The method `run()` actually runs Prim's algorithm as specified in pseudocode in Figure 1 below.

```
MST-PRIM(G, w, r)
  for each u ∈ G.V        // including root r
    u.key = ∞
    u.parent = NILL
  r.key = 0
  Q = G.V
  while Q ≠ ∅
    u = EXTRACT-MIN(Q)    // in 1st iteration, u←r
    for each v ∈ G.Adjacent[u]
      if v ∈ Q and w(u, v) < v.key
        v.parent = u      // but v still loose (in Q) for now
        v.key = w(u, v)
```

Figure 1: The pseudocode of Prim's algorithm

We construct our MST as a new Graph. Initially, the frontier contains the full graph, while the MST is still empty. While running the main loop, vertices from

the frontier are added to the MST. Since we initialize the root node "a" with key 0, this will be the first vertex chosen to be added to the MST. Furthermore, we find the corresponding edge and we add this as well. After this we update the key of the vertices with the last added vertex and we start again at the beginning of the while-loop. This continues until the frontier is empty and no other vertices are left to be added to the MST.

For updating the key, the function `updateKey()` is used. We make an ArrayList of all the edges in the graph and we run through this ArrayList. The input to the method is the vertex u. We check to which other vertex u is connected. This vertex v is then checked whether it is not null, the key is bigger than the weight of the edge and if v does not occur in the MST. If all of this is true, then v is removed from the frontier, the key is updated with the weight from the current edge considered and the parent of v is set. Then v is added to the frontier again. We remove it and add it again to the frontier because if we do not do this, the PriorityQueue does not update.

To see to which other vertex some vertex u is connected to, we have to look at the edge. Since the edge keeps track which to vertices it connects, we just have to search for the other vertex the edge is connected to. The function `isConnected()` checks this with two if-statements and returns the found vertex.

The full code is given at the end of the document.

2 How we measure runtime

We wanted to measure runtime using a counter. This is because using a timer is dependent on the hardware and the state of the hardware. For example, during one experiment the computer could be running another process in the background, slowing everything down and skewing the data. That, as well as the easy implementation of a counter, is why we decided to measure runtime.

We decided to put the counter inside the if-statement in the `updateKey` function, because otherwise the counter would always be some relation between the number of vertices and number of edges. Then, every edge and vertex combination would be considered, and this would not tell us much interesting information. This also makes sense because the actual updating of the keys and the updating of the priority queue is done inside this if-statement. If we would only count the number of added vertices to the MST, this would actually only count the size of the MST and this is not what we want to count.

3 Experiments

For all of our experiments, we used a GraphMaker class to create random graphs with certain given constraints. The code for this can also be seen in Section 5: Attachments.

In the table below, we plotted the outcome of running Prim's algorithm on different graphs with random edge-weights between 0 and 10. The number of edges was always twice the number of vertices. Since there needed to be some sort of consistency in the number of edges, we chose to make that a very obvious, linear relationship.

Vertices	Count
5	8
10	14
15	25
20	33
25	53
30	53
35	61
40	65
45	74
50	80

Table 1: Runtime as increasing numbers of vertices

The data from Table 1 is plotted in the Figure 2. As shown, the data is clearly linear, having a linear trendline ($R^2 = 0.9924$).

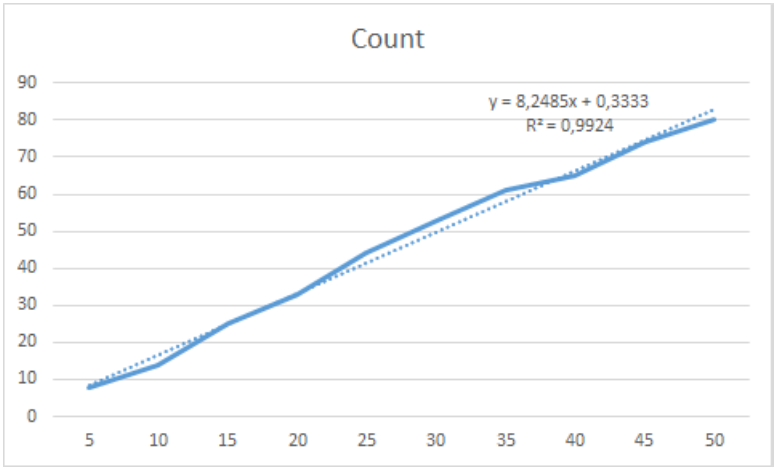

Figure 2: Number of vertices against number of counts (runtime)

Since the data from experiment 2 and onward is too much to be put into a table and shown here, we'll provide our spreadsheet as well, for the raw data.

The data from experiment 2 is shown below in Figure 3.

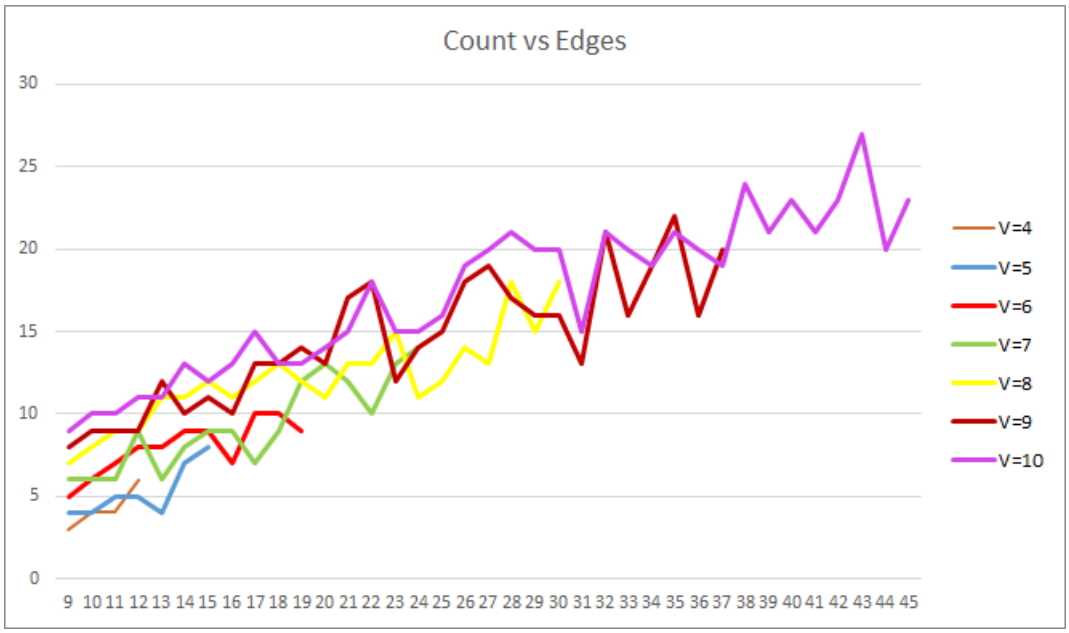

Figure 3: Count against edges for different numbers of vertices

The data from experiment 3 is shown below in Figure 4.

In experiment 3, the amount of vertices was set to 50 and the amount of edges to 100, for consistency. Here we measure the range of weights.

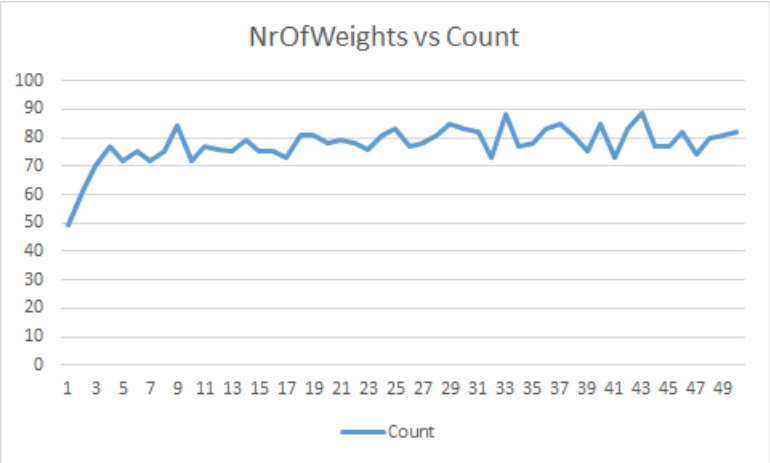

Figure 4: Possible number of weights against counts

The data from experiment 4 is shown below in Figure 5.

In experiment 4, the amount of vertices was set to 50 and the amount of edges to 100, for consistency. The range of weights was always set to 10, for consistency in the data. In this experiment we measure the magnitude of the weights.

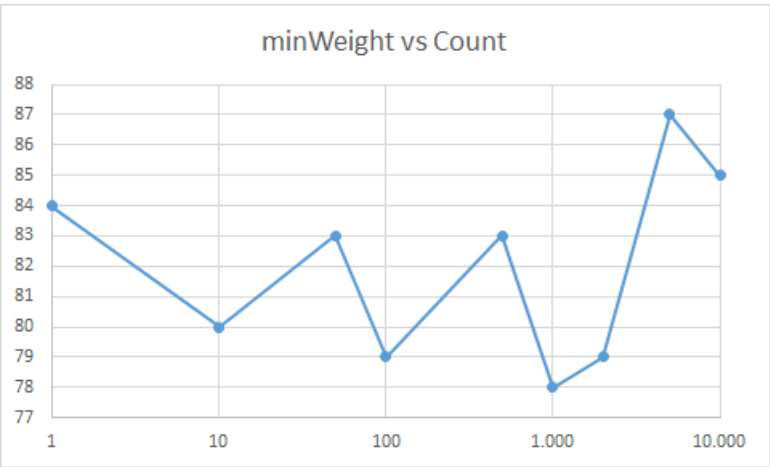

Figure 5: Minimum weights against counts

4 Conclusions

If we look at Table 1, we clearly expect a linear relationship between the amount of vertices and the runtime. Then considering Figure 2 we see a linear trendline

with $R^2 = 0.99$. This means that with increasing the amount of vertices, the runtime increases linearly with $R^2 \approx 1$.

For experiment 2, there seems to be a rough linear relation between the number of vertices and the runtime, expressed in counts. However, the data still varies quite a lot, and thus it is hard to make definitive conclusions about the effect of different amounts of edges on the runtime. The reason for the variation is probably the randomness of the edge-weights. Because these are random, sometimes many edges have to be evaluated, and sometimes only very few edges have to be evaluated. A rough conclusion is that for all considered vertices, there is a linear increase in runtime if the amount of edges increases. Calculating the trendlines we come to the following R^2 for the different vertices:

Vertices	R^2
5	0.8526
6	0.669
7	0.6769
8	0.7758
9	0.7396
10	0.8512

Table 2: The calculated trendlines from Figure 3

The average trendline is then $R^2 = 0.76085$.

In figure 4 for experiment 3, one can clearly see a linear trendline that does not increase. The runtime seems to variate between 70 and 90, except for the first part. In the range 1-5 for the number of weights, the time increases, but after that it becomes linear. So for very small numbers of weights, the runtime increases, after that, the runtime roughly stays the same.

In the data of experiment 4, the range of the values is quite narrow, between 87 and 78, which is likely within the range of variation our data naturally fluctuates in. Thus, we can conclude that the magnitude of the weights does not influence the runtime.

The conclusion from our experiments is that there seem to be some clear linear relationships, both for the amount of vertices ($R^2 \approx 1$) and the number of edges ($R^2 \approx 0.76$). Futhermore, the range of the weights and the magnitude of the weights seem to be factors that do not influence the runtime.

5 Attachments

5.1 Graph

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package spmlassignment1;

import java.util.ArrayList;

/**
 *
 * @author Anouk & Pleun
 */
public class Graph {
    private ArrayList<Edge> edges;
    private ArrayList<Vertex> vertices;

    public Graph(ArrayList<Edge> edges, ArrayList<Vertex> vertices) {
        this.edges = edges;
        this.vertices = vertices;
    }

    public ArrayList<Vertex> getVertices() {
        return vertices;
    }

    public ArrayList<Edge> getEdges() {
        return edges;
    }

    public void addVertex(Vertex v) {
        vertices.add(v);
    }

    public void addEdge(Edge e) {
        edges.add(e);
    }

    public boolean contains (Vertex vertex) {
        for (Vertex v : vertices)
            if (v.equals(vertex))
                return true;
```

```
            return false;
        }

        @Override
        public String toString() {
            String s = "";
            for(Edge e : edges)
                s+= e+"\n";
            return s;
        }
}
```

5.2 Vertex

```
package spmlassignment1;

/**
 *
 * @author Anouk & Plean
 */
public class Vertex {
    private double key;
    private String name;
    private Vertex parent;

    /**
     * Constructor function for Vertex.
     * @param key
     */
    public Vertex(double key, String name, Vertex parent){
        this.key = key;
        this.name = name;
        this.parent = parent;
    }

    public boolean isConnected(Vertex vertex, Edge[] edges) {
        for (Edge edge : edges) {
            //If the nodes are connected. (this refers to this Vertex object.)
            if((edge.getFirst() == this && edge.getSecond() == vertex))// || (ed
                return true;
        }
        return false;
    }

    /**
     * Setter function for getKey.
     * @param key
     */
    public void setKey(double key) {
        this.key = key;
    }

    /**
     * Getter function for getKey
     * @return getKey
     */
    public double getKey() {
        return key;
```

```
    }

    /**
     *
     * @param parent
     */
    public void setParent(Vertex parent) {
        this.parent = parent;
    }

    public Vertex getParent() {
        return parent;
    }

    @Override
    public String toString() {
        return name;
//          return String.format("%s : %f",name,key);
    }
}
```

5.3 Edge

```
package spmlassignment1;

/**
 *
 * @author Anouk and also Plean plz
 */
public class Edge {
    private Vertex firstNode;
    private Vertex secondNode;
    private double weight; //want dan kunnen we inf gebruiken

    /**
     * Constructor function for Edge
     * @param firstNode
     * @param secondNode
     * @param weight
     */
    public Edge(Vertex firstNode, Vertex secondNode, double weight){
        this.firstNode = firstNode;
        this.secondNode = secondNode;
        this.weight = weight;
    }

    public Vertex isConnected(Vertex v) {
        if(v.equals(firstNode))
            return secondNode;
        else if (v.equals(secondNode))
            return firstNode;
        else return null;
    }

    /**
     * Getter function for weight.
     * @return weight
     */
    public double getWeight() {
        return this.weight;
    }

    public Vertex getFirst() {
        return this.firstNode;
    }

    public Vertex getSecond() {
```

```
        return this.secondNode;
    }

    @Override
    public String toString() {
        return String.format("%s <--%f--> %s",firstNode,weight,secondNode);
    }
}
```

5.4 MST_Prim

```
package spmlassignment1;

import java.util.ArrayList;
import java.util.PriorityQueue;

/**
 *
 * @author Pleun & noukie
 */
public class MST_Prim {

    Graph graph;
    Graph mst;
    PriorityQueue<Vertex> frontier;
    public int count=0; //We know, public variables, but it's just a counter who

    public MST_Prim(Graph graph) {
        this.graph = graph;
        //Initialize MST as an empty graph.
        mst = new Graph(new ArrayList<Edge>(), new ArrayList<Vertex>());
        //Priority Queue with comparator
        WeightComparator comparator = new WeightComparator();
        frontier = new PriorityQueue(comparator);
    }

    /**
     * Initializes the frontier
     */
    public void initialize() {
        //Add every vertex into the frontier.
        ArrayList<Vertex> vertices = graph.getVertices();
        for (Vertex v : vertices)
            frontier.add(v);

    }

    /**
     * Runs Prim's algorithm on the graph.
     */
    public void run() {
        while (!frontier.isEmpty()) {
            Vertex u = frontier.poll();
//              System.out.printf("U: %s",u);
//              System.out.printf(" Parent: %s",u.getParent());
```

```
                Edge e = new Edge(u.getParent(),u,u.getKey());
                mst.addVertex(u);
                mst.addEdge(e);
                updateKey(u);
//                System.out.println(frontier);
            }
        }

        /**
         * @return the MST
         */
        public Graph getMST() {
            return mst;
        }

        /**
         * Updates the weights within the frontier.
         *
         * @param graph
         * @param u Vertex just added to the MST.
         * @param frontier (expanded nodes =/= INF.)
         */
        private void updateKey(Vertex u) {
            ArrayList<Edge> edges = graph.getEdges();
            for (Edge e : edges) {
                Vertex v = e.isConnected(u);
                if (v != null && e.getWeight() < v.getKey() && !mst.contains(v)) {
                    count++;
                    frontier.remove(v);
                    v.setKey(e.getWeight());
                    v.setParent(u);
                    frontier.add(v);
                }
            }
        }

    }
```

5.5 WeightComparator

```
package spmlassignment1;

import java.util.Comparator;

/**
 *
 * @author Pleun
 */
public class WeightComparator implements Comparator<Vertex>{
    @Override
    public int compare(Vertex o1, Vertex o2) {
        if (o1.getKey() > o2.getKey())
            return 1;
        else if (o1.getKey() < o2.getKey())
            return -1;
        else return 0;
    }
}
```

5.6 Main

```
package spmlassignment1;

import java.util.ArrayList;
import java.util.PriorityQueue;

/**
 * Main class of the assignment. Runs a bunch of tests in the main.
 *
 * @author Anouk & Pleun
 */
public class SPMLAssignment1 {

    private static final double INF = Double.POSITIVE_INFINITY;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Graph graph = createGraph();
        MST_Prim prim = new MST_Prim(graph);
        prim.initialize();
        prim.run();
        System.out.printf("Final mst: %s",prim.getMST());
    }

    /**
     * @return a graph as drawn in the slides.
     */
    private static Graph createGraph() {
        ArrayList<Vertex> vertices = new ArrayList();

        vertices.add(new Vertex(0, "a", null));  //a 0
        vertices.add(new Vertex(INF, "b", null));//b 1
        vertices.add(new Vertex(INF, "c", null));//c 2
        vertices.add(new Vertex(INF, "d", null));//d 3
        vertices.add(new Vertex(INF, "e", null));//e 4
        vertices.add(new Vertex(INF, "f", null));//f 5
        vertices.add(new Vertex(INF, "g", null));//g 6
        vertices.add(new Vertex(INF, "h", null));//h 7
        vertices.add(new Vertex(INF, "i", null));//i 8*/

        ArrayList<Edge> edges = new ArrayList();
        edges.add(new Edge(vertices.get(0), vertices.get(1), 4)); //ab
        edges.add(new Edge(vertices.get(0), vertices.get(7), 80)); //ah
```

```
        edges.add(new Edge(vertices.get(1), vertices.get(7), 100)); //bh
        edges.add(new Edge(vertices.get(1), vertices.get(2), 8000)); //bc
        edges.add(new Edge(vertices.get(2), vertices.get(8), 2000)); //ci
        edges.add(new Edge(vertices.get(2), vertices.get(5), 30)); //cf
        edges.add(new Edge(vertices.get(2), vertices.get(3), 7000)); //cd
        edges.add(new Edge(vertices.get(3), vertices.get(4), 9000)); //de
        edges.add(new Edge(vertices.get(5), vertices.get(3), 400000000)); //fd
        edges.add(new Edge(vertices.get(5), vertices.get(4), 1)); //fe
        edges.add(new Edge(vertices.get(6), vertices.get(8), 6000)); //gi
        edges.add(new Edge(vertices.get(6), vertices.get(5), 200)); //gf
        edges.add(new Edge(vertices.get(7), vertices.get(6), 100)); //hg
        edges.add(new Edge(vertices.get(8), vertices.get(7), 7000)); //ih

        return new Graph(edges, vertices);
    }
}
```

5.7 GraphMaker

```
package spmlassignment1;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;

/**
 * There is probably a whole bunch of stuff that we could improve no doubt but
 * we just needed to get it done quick and dirty sooo there you go.
 *
 * @author Pleun & Noukie
 */
public class GraphMaker {

    private int nrOfVertices;
    private int nrOfEdges;
    private double minWeight;
    private double maxWeight;
    private ArrayList<Double> categories;
    private ArrayList<Double> catCopies;

    /**
     * @param nrOfVertices
     * @param nrOfEdges
     * @param minWeight
     * @param maxWeight
     * @return a graph with the nr of vertices and edges given. Throws error if
     * the graph is impossible.
     */
    public Graph makeGraph(int nrOfVertices, int nrOfEdges, double minWeight, do
        //Minimum nr of edges gives a tree, maximum gives a fully connected grap
        if (nrOfEdges < nrOfVertices - 1 || nrOfEdges > nrOfVertices * (nrOfVert
            throw new IllegalArgumentException("Illegal nr of edges/vertices.");

        //For allowed trees:
        this.nrOfEdges = nrOfEdges;
        this.nrOfVertices = nrOfVertices;
        this.minWeight = minWeight;
        this.maxWeight = maxWeight;

        ArrayList<Vertex> vertices = makeVertices();
        vertices.get(0).setKey(0); // Root node. Since the vertices are shuffled
        ArrayList<Edge> edges = makeEdges(vertices);
        return new Graph(edges, vertices);
```

```
}

/**
 * @param nrOfVertices
 * @param nrOfEdges
 * @param categories
 * @return a graph with edge weights from the categories arraylist.
 */
public Graph makeCatGraph(int nrOfVertices, int nrOfEdges, ArrayList<Double>
    //It's a slightly weird system but it works and it's something ^^
    this.categories = new ArrayList();
    this.catCopies = new ArrayList();
    for (Double d : categories) {
        this.categories.add(d);
        this.catCopies.add(d);
    }
    return makeGraph(nrOfVertices, nrOfEdges, Collections.min(this.categorie
}

/**
 * @param nrOfVertices
 * @param minWeight edge
 * @param maxWeight
 *
 * @return a fully connected graph with the nr of vertices given.
 */
public Graph makeFullyConnectedGraph(int nrOfVertices, double minWeight, dou
    int n = nrOfVertices * (nrOfVertices - 1) / 2;
    return makeGraph(nrOfVertices, n, minWeight, maxWeight);
}

/**
 * @return a list of vertices with random keys and a name 0 to nrOfVertices
 */
private ArrayList<Vertex> makeVertices() {
    ArrayList<Vertex> vertices = new ArrayList();
    for (int i = 0; i < nrOfVertices; i++)
        vertices.add(new Vertex(Double.POSITIVE_INFINITY, Integer.toString(i
    return vertices;
}

/**
 * @return a random double between minEWeight and maxEWeight.
 */
private double getRand() {
    Random rnd = new Random();
```

```
    return minWeight + (maxWeight - minWeight) * rnd.nextDouble();
}

/**
 * @param categories ArrayList of categories
 * @return a unique random double from a category of doubles.
 */
private double getCatRand() {
    double rnd = getRand();
    double category;
    //The difference is that the catcopies removes, so every weight is used
    // but if we need more edges than there are categories, we don't take ra
    if (catCopies.isEmpty()) {
        double dist = Math.abs(categories.get(0) - rnd);
        category = categories.get(0);
        for (int i = 1; i < categories.size(); i++) {
            double newDist = Math.abs(categories.get(i) - rnd);
            if (newDist < dist) {
                dist = newDist;
                category = categories.get(i);
            }
        }
    } else {
        double dist = Math.abs(catCopies.get(0) - rnd);
        category = catCopies.get(0);
        for (int i = 1; i < catCopies.size(); i++) {
            double newDist = Math.abs(catCopies.get(i) - rnd);
            if (newDist < dist) {
                dist = newDist;
                category = catCopies.get(i);
            }
        }
        catCopies.remove(category);
    }
    return category;
}

private ArrayList<Edge> makeEdges(ArrayList<Vertex> vertices) {
    ArrayList<Edge> edges = new ArrayList();
    Collections.shuffle(vertices); //Shuffle the vertices
    //Connect at least all vertices once, by connecting them all randomly (s
    for (int i = 0; i < vertices.size() - 1; i++)
        if (categories == null)
            edges.add(new Edge(vertices.get(i), vertices.get(i + 1), getRand
        else
            edges.add(new Edge(vertices.get(i), vertices.get(i + 1), getCatR
```

```
        //Then if we still need more edges, make them randomly.
        while (edges.size() < nrOfEdges) {
            Collections.shuffle(vertices);
            Edge rndEdge;
            if (categories == null)
                rndEdge = new Edge(vertices.get(0), vertices.get(1), getRand());
            else
                rndEdge = new Edge(vertices.get(0), vertices.get(1), getCatRand(
            if (!hasEdge(edges, rndEdge))
                edges.add(rndEdge);
        }
        return edges;
    }

    /**
     * @param edges
     * @param rndEdge
     * @return returns true if any version of rndEdge is contained in edges.
     */
    private boolean hasEdge(ArrayList<Edge> edges, Edge rndEdge) {
        for (int i = 0; i < edges.size(); i++) {
            Vertex v11 = edges.get(i).getFirst();
            Vertex v12 = edges.get(i).getSecond();
            Vertex v21 = rndEdge.getFirst();
            Vertex v22 = rndEdge.getSecond();
            if (v11.equals(v21) && v12.equals(v22) || v11.equals(v22) && v12.equ
                return true;
        }
        return false;
    }
}
```