Lab Report

Exercise 10: Getting from A to B

Name		Titel des Kurses	Datum
Juri Wiechmann Bartholomäus Berresheim	571085 568624	Prof. Dr. Weber-Wulff Info 2 Group 2	12.01.2020

Index

Introduction

This week's lab was the first time where we had to work with graphs and path finding algorithms.

Pre-lab

1. Define an abstract data type for a weighted graph. What methods does your ADT need? What are the signatures for the operators?

```
public interface WeightedGraph{

void addVertice(VerticeWeighted...n);
void addEdge(VerticeWeighted start, VerticeWeighted end, double weight);
void addEdgeHelper(VerticeWeighted start, VerticeWeighted end, double
weight);
void printEdges();
boolean hasEdge(VerticeWeighted start, VerticeWeighted end);
void resetVerticeVisited();
}
```

2. Find algorithms for determining the minimum path and the cheapest path between two nodes in a directed graph. I strongly suggest visiting a library (that is one of these places that keeps ancient books around). There are Algorithm and Data Structure books available in many languages. There is also an example in the Wikipedia, but it is not really easy to understand.

For this lab we chose the Dijkstra algorithm for our cheapest path finding, and for the minimum path finding, we chose a regular Breadth First Search algorithm.

3. Your algorithm will probably need an adjacency matrix oder an adjacency list as its data structure. Think about how you would implement such a structure, if you only had linked lists available. What methods will you need for your data structure?

We decided to use a matrix structure because we find it clearer and easier to understand with our way of thinking. We will use Sets and LinkedLists so organise our data.

4. Briefly review generating random numbers.

There are multiple ways of generating random numbers, one of them would be the Math.random() method. Another way of doing it, is to create an object of type Random and to call the method nextInt() using the Object.

Assignments

While completing the assignments, we based our code on the one found at ("https://stackabuse.com/graphs-in-java-dijkstras-algorithm/") and the one from ("https://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes-for-unweighted-graph").
Neither of these were exactly what we needed, and the first one contained issues such as : We can't find the direct way from a Verteces to itself (which we fixed), it doesn't find the way back

to itself on an indirect way (such as 0 -> 1 ->0) and sometimes it doesn't show the starting point in the path (which we fixed).

1. Design and implement a data type WeightedGraph that uses either an adjacency list or an adjacency matrix. How are you going to store the weights?

Before implementing the WeightedGraph data type, we had to create 2 other ones first, namely EdgeWeighted and VerticeWeighted. To do that, we first created a new class called "EdgeWeighted", that implements the interface Comparable of type EdgeWeighted. The implementation of Comparable is so that we can compare Objects of type EdgeWeighted with each other. In it we created 4 fields: 2 VerticeWeighted "start" and "end", and one double "weight".

```
public class EdgeWeighted implements Comparable<EdgeWeighted>{
    public VerticeWeighted start;
    public VerticeWeighted end;
    public double weight;
```

This was then followed by the constructor of the class, it has 3 inputs: 2 VerticeWeighted "start" and "end" and one double "weight". In this constructor we change the value of the previously mentioned fields to the inputted values.

```
public EdgeWeighted(VerticeWeighted start, VerticeWeighted end, double weight){
    this.start = start;
    this.end = end;
    this.weight = weight;
}
```

Afterwards, we implemented 2 methods from the Comparable interface. The first being the toString() method, which allows us to return the informations of an Edge in the form of a String. To create said String, we used the format() method from the String class.

```
@Override
public String toString() {
    return String.format("(%s -> %s, %f)",start.name, end.name, weight);
}
```

Following the toString() method, we have the compareTo() method. It gets an EdgeWeighted as input and then compares its weight to the weight from the Edge we called the method from.

If the weight from the inputted Edge is smaller the method returns 1, if not it returns -1;

```
@Override
public int compareTo(EdgeWeighted otherEdge) {
    if(this.weight > otherEdge.weight)
        return 1;
    else
        return -1;
}
```

After having finished the EdgeWeighted class, we move on with the VerticeWeighted one. In this class, we first created 4 fields: one int "n", one String "name", one boolean "visited" and one LinkedList of type EdgeWeighted "edges". After that, we added a constructor for the class.

The constructor gets 2 inputs: and int “n” and a String “name”, using which it changes the values of the fields “n” and “name” respectively. It also changes the value of “visited” to false and assigns “edges” a new LinkedList as value.

```
public class VerticeWeighted {
    public int n;
    public String name;
    private boolean visited;
    LinkedList<EdgeWeighted> edges;

    public VerticeWeighted(int n, String name) {
        this.n = n;
        this.name = name;
        this.visited = false;
        this.edges = new LinkedList<>();
    }
```

After the constructor, we created 4 methods, the first being isVisited() which returns the value of “visited”, the second one is visit() which changes the value of “visited” to true, and unvisit() that changes the value of “visited” to false.

```
public boolean isVisited() {
    return visited;
}
public void visit() {
    visited = true;
}
public void unvisit() {
    visited = false;
}
```

The fourth method called getChildVertice() gets no input and returns a LinkedList of VerticeWeighted. In the method we create a new LinkedList of type VerticeWeighted “childVertices”. Followed by a for each EdgeWeighted loop, that takes every Edge from “edges” and adds every VerticesWeighted, that the Vertice (over which we call the method) is connected to, to “childVertices”. After the loop we then return “childVertices”.

```
public LinkedList<VerticeWeighted> getChildVertice() {
    LinkedList<VerticeWeighted> childVertices = new LinkedList<VerticeWeighted>();
    for(EdgeWeighted edge : edges) {
        if(n == edge.start.n)
            childVertices.add(edge.end);
        if(n == edge.end.n)
            childVertices.add(edge.start);
    }
    return childVertices;
}
```

Now that we have the EdgeWeighted and VerticeWeighted completed, we moved on to creating our WeightedGraph data type. To do that, we first implemented our Interface “WeightedGraph” mentioned in the pre-lab.

```
public interface WeightedGraph {

    void addVertice(VerticeWeighted...n);
    void addEdge(VerticeWeighted start, VerticeWeighted end, double weight);
    void addEdgeHelper(VerticeWeighted start, VerticeWeighted end, double weight);
    void printEdges();
    boolean hasEdge(VerticeWeighted start, VerticeWeighted end);
    void resetVerticeVisited();


}
```

After that, we created class called GraphWeighted that implements WeightedGraph.

In the this new class, we first added 2 fields: one Set of type VerticeWeighted “vertices” and a boolean “directed”. Followed by a constructor, that takes a boolean “directed” as input, which it then puts as value for the field “directed”, and it changes the value of “vertices” to a new HashSet<>().

```
public class GraphWeighted implements WeightedGraph{
    private Set<VerticeWeighted> vertices;
    private boolean directed;

    public GraphWeighted(boolean directed) {
        this.directed = directed;
        vertices = new HashSet<>();
    }

    public void addVertice(VerticeWeighted...n) {
        vertices.addAll(Arrays.asList(n));
    }
```

After this, we implemented 6 methods. The first being addVertice(), that gets varargs of VerticeWeighted as input, using which we can add a varying number of vertices to the Set “vertices”.

```
@Override
public void addVertice(VerticeWeighted...n) {
    vertices.addAll(Arrays.asList(n));
}
```

The second method we added is addEdgeHelper(), it gets 2 VerticeWeighted “start” and “end”, and one double “weight”, as input. It then checks for every EdgeWeighted of start, if the “start” and “end” of that edge is identical to the inputted ones, if true it updates the weight of the Edge to the inputted one. If false, it means that there isn’t an Edge between these 2 Vertices yet, and the methods then adds one.

```
@Override
public void addEdgeHelper(VerticeWeighted start, VerticeWeighted end, double weight) {
    for (EdgeWeighted edge : start.edges) {
        if (edge.start.equals(start) && edge.end.equals(end)) {
            edge.weight = weight;
            return;
        }
    }
    start.edges.add(new EdgeWeighted(start, end, weight));
}
```

The third method we added is addEdge(), it gets 2 VerticeWeighted “start” and “end”, and one double “weight” as input. It then adds “start” and “end” to “vertices” and since “vertices” is a Set, which doesn’t allow duplicates, we don’t have to worry about having the same Vertice twice in “vertices”. After that it uses its own inputs to call the method addEdgeHelper(). This is to make sure that there are no duplicate Edges. This is followed by an if statement that checks whether the Graph is directed and whether “start” and “end” are equal. If the Graph is not directed and they are not equal, then it calls addEdgeHelper() again but this time with “start” and “end” having switched places. This makes it that betwenn those 2 Vertices we have 2 Edges, one in each direction, and both having the same weight.

```
@Override
public void addEdge(VerticeWeighted start, VerticeWeighted end, double weight) {

    vertices.add(start);
    vertices.add(end);

    //Making sure, that we dont have duplicated edges.
    addEdgeHelper(start, end, weight);
    if(!directed && start != end) {
        addEdgeHelper(end, start, weight);
    }

}
```

The fourth method we added is printEdges(), it allows us to print out every Edges for each of our Vertices. In the method, we have a for each VerticeWeighted in “vertices” loop. In it we have an if statement that checks if the current Vertice has Edges, if not it prints out a message to the console and the for loop continues with the next Vertice. If it has Edges it prints our another message, followed by a for each EdgeWeighted loop, that prints out the destination name and weight of the current Edge. After the loop we have a println() that leaves an empty space between different Vertices.

```
@Override
public void printEdges() {
    for(VerticeWeighted vertice : vertices) {
        LinkedList<EdgeWeighted> edges = vertice.edges;

        if(edges.isEmpty()) {
            System.out.println("Vertice " + vertice.name + " has no edges.");
            continue;
        }
        System.out.println("Vertice " + vertice.name + " has edges to:");

        for(EdgeWeighted edge : edges) {
            System.out.println(edge.end.name + "(" + edge.weight + ")");
        }
        System.out.println();
    }
}
```

The fifth method we added is hasEdge(), it gets 2 VerticeWeighted “start” and “end” as input and checks whether they have an Edge connecting them. In the method we have a for each loop that goes through all the Edges of start and checks if the destination of one of the Edges corresponds to the “end” Vertice, if yes it returns true, if not then it returns false.

```
@Override
public boolean hasEdge(VerticeWeighted start, VerticeWeighted end) {
    LinkedList<EdgeWeighted> edges = start.edges;
    for (EdgeWeighted edge : edges) {
        if (edge.end == end) {
            return true;
        }
    }
    return false;
}
```

The last method we added for this assignment is resetVerticeVisited(), it sets the boolean “visited” of all VerticeWeighted to false, using a foreach loop.

```
@Override
public void resetVerticeVisited() {
    for(VerticeWeighted vertice : vertices) {
        vertice.unvisit();
    }
}
```

Now we can create Graphs and add Verteces and Edges to it, such as in this example:

```
VerticeWeighted zero = new VerticeWeighted(0, "0");
VerticeWeighted one = new VerticeWeighted(1, "1");
VerticeWeighted two = new VerticeWeighted(2, "2");
VerticeWeighted three = new VerticeWeighted(3, "3");
VerticeWeighted four = new VerticeWeighted(4, "4");
VerticeWeighted five = new VerticeWeighted(5, "5");
VerticeWeighted six = new VerticeWeighted(6, "6");

// Our addEdge method automatically adds Nodes as well.
// The addNode method is only there for unconnected Nodes,
// if we wish to add any
graphWeighted.addEdge(zero, one, 8);
graphWeighted.addEdge(one, zero, 11);
graphWeighted.addEdge(two, two, 3);
graphWeighted.addEdge(three, three, 8);
graphWeighted.addEdge(two, four, 28);
graphWeighted.addEdge(five, four, 9);
graphWeighted.addEdge(four, six, 25);
graphWeighted.addEdge(one, two, 2);
graphWeighted.addEdge(two, three, 6);
graphWeighted.addEdge(three, four, 1);
graphWeighted.addEdge(four, five, 7);
graphWeighted.addEdge(five, six, 8);

graphWeighted.printEdges();
```

And in the Console we get:

```
Vertice 2 has edges to:
2(3.0)
4(28.0)
3(6.0)

Vertice 5 has edges to:
4(9.0)
6(8.0)

Vertice 4 has edges to:
6(25.0)
5(7.0)

Vertice 6 has no edges.
Vertice 1 has edges to:
0(11.0)
2(2.0)

Vertice 3 has edges to:
3(8.0)
4(1.0)

Vertice 0 has edges to:
1(8.0)
```

2. Write a class that generates a random weighted graph. You will need a constructor that takes the number of vertices for your graph and the number of edges. For example, you might want to have RandomGraph (20, 45) generate a graph with 20 vertices and 40 edges which randomly connect those vertices. You should give your vertices names, either really boring ones like "A", "B", "C" or make up random names for example by choosing random words in Wikipedia articles. Generate the edges by

choosing 2 vertices at random, and then assigning them a random weight.The bored can explore the Stanford collection of graphs and see if they can find something interesting.

To complete this task, we first created 2 methods: getRandomVertice() and getRandomGraph() in our GraphWeighted class.

The getRandomVertice() method take no inputs and returns a random VerticeWeighted. In it, we have 3 ints: "size" that has the size of the Set "vertices" as value, "item" that has a randomly generated int as value, and "i" which has 0 as value. After the ints, we added a for each VerticeWeighted loop. In it we check if "i" is equal to the random number "item", if true the method returns the current VerticeWeighted, if not it adds 1 to "i". After the loop we return null. The purpose of this method is to randomly choose a vertice contained in our graph and to return it.

```
public VerticeWeighted getRandomVertice() {
    int size = vertices.size();
    int item = new Random().nextInt(size);
    int i = 0;
    for(VerticeWeighted vertice : vertices)
        if(i == item)
            return vertice;
        i++;
    }
    return null;
}
```

The getRandomGraph() method gets 2 ints "vertices" and "edges" as input. It then resets "vertices" by assigning it a new HashSet as value. After that it adds Vertices to "this.vertices" using a for loop that loops as long i is smaller than the inputted "vertices". This loop is followed by a second loop, in which we choose 2 random vertices contained in the graph using the getRandomVertice() method, to which we then add an EdgeWeighted with a random weight.

```
public void getRandomGraph(int vertices, int edges) {
    //reset the current
    this.vertices = new HashSet<>();

    for(int i = 0; i<vertices;i++) {
        this.vertices.add(new VerticeWeighted(i, Integer.toString(i)));
    }
    for(int i = 0; i<edges;i++) {
        VerticeWeighted randomV1 = getRandomVertice();
        VerticeWeighted randomV2 = getRandomVertice();;

        addEdge(randomV1, randomV2, Math.random()*100);

    }
}
```

After finishing the methods, we continue by creating a class called RandomGraph that extends GraphWeighted. In it we have a constructor that takes 3 inputs: a boolean "directed" and 2 ints "vertices" and "edges". In the constructor we first call the constructor from the GraphWeighted class using "directed". Followed by a calling of the getRandomGraph()

method using the inputted “vertices” and “edges”. And finally we print the edges of the random Graph.

```
public class RandomGraph extends GraphWeighted{

    public RandomGraph(boolean directed, int vertices, int edges) {
        super(directed);

        getRandomGraph(vertices, edges);

        printEdges();
```

Now once we create an object of type RandomGraph, we get:

```
RandomGraph randomGraph = new RandomGraph (true, 5, 10);
```

```
Vertice 4 has edges to:
3(24.10934401864333)

Vertice 1 has edges to:
1(1.2457331382005554)
2(9.1005155745022)

Vertice 2 has edges to:
2(31.68597407194835)

Vertice 0 has edges to:
4(10.821700197169083)
1(63.775528373607195)

Vertice 3 has edges to:
4(53.80342428478857)
```

(In this example the number of Edges is lower that the inputted one, this is due to the fact that some Edges have been created multiple times, and thus have overwritten the older weight) (Another thing to note is that the RandomGraph can handle a bigger amount of vertices and edges but we’ve kept them low for demonstration purposes)

3. Now write a method that will take a graph, pick two vertices at random, and find the shortest path between the vertices. Make a method to print out the path in a readable format. What class will these methods belong to?

To complete this task, we first added a method called getEdge() to the GraphWeighted class. It gets 2 different VerticeWeighted “V1” and “V2” as input. In the method we then added a boolean “isChild” and set it to false. We then used an if statement in a for-each Vertice loop to check whether V2 is a childVertice of V2. If true, we set “isChild” to true. After the loop we check if V2 is not a child of V1, if it’s not, we print a message to the console and return null. After that, we have a for each EdgeWeighted loop, using which we look for the/an Edge connecting V1 and V2, once found we return that Edge. There shouldn’t be a case were we enter the loop and don’t trigger the return statement in it. But we still added a return

statement after the loop, so that Eclipse can compile the code. We also added a println() of a message that states “Not Possible”, as a note.

```
public EdgeWeighted getEdge(VerticeWeighted V1, VerticeWeighted V2) {
    boolean isChild = false;
    for(VerticeWeighted vertice : V1.getChildVertice()) {
        if(V2.equals(vertice))
            isChild = true;

    }
    if(!isChild) {
        System.out.println("Not a Child");
        return null;
    }

    for(EdgeWeighted edge : V1.edges) {
        if(edge.start == V1 && edge.end == V2 ||
           edge.start == V2 && edge.end == V1)
            return edge;
    }
    System.out.println("Not possible");
    return null;
}
```

Once this was done, we continue by creating a method called DijkstraShortestPath(). It gets 2 VerticeWeighted “start” and “end” as input, and returns a LinkedList of “VerticeWeighted”. In the method we created a HashMap “changedAt” to take track of every step (between Vertices) we’ve tried so far, for every VerticeWeighted; a LinkedList “direction” to take track of our shortestpath, a Queue “q” and a VerticeWeighted “currentVertice” with “start” as its value. After this we added the current Vertice to the queue and changed it’s “visited” to true.

```
public LinkedList<VerticeWeighted> DijkstraShortestPath(VerticeWeighted start, VerticeWeighted end) {

    //https://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes-for-unweighted-graph
    HashMap<VerticeWeighted, VerticeWeighted> changedAt = new HashMap<>();

    LinkedList<VerticeWeighted> direction = new LinkedList<VerticeWeighted>();

    Queue q = new LinkedList();

    VerticeWeighted currentVertice = start;
    q.add(currentVertice);

    start.visit();
```

This is followed by a while loop that goes on until “q” is empty. In it, we first remove the first elemet from “q” and set it as the value for “currentVertice”, after that we have an if statement that checks whether “currentVertice” is equal to “end”, if true it breaks the loop. If not, it looks at every child from “currentVertice” using a for each loop. If the current child is not visited yet, we add it to “q”, set it to visited, and add the child and “currentVertice” to “changedAt” to keep track.

```
while(!q.isEmpty()) {
    currentVertice = (VerticeWeighted) q.remove();
    if(currentVertice.equals(end)) {
        break;
    }
    else {
        for(VerticeWeighted vertice : currentVertice.getChildVertice()) {
            if(!vertice.isVisited()) {
                q.add(vertice);
                vertice.visit();
                changedAt.put(vertice, currentVertice);
            }
        }
    }
}
```

After the loop, we check if "currentVertice" is equal to "end", if false it means that there is no path between "start" and "end", so we print a message to the console and return null.

```
if(!currentVertice.equals(end)) {
    System.out.println("There isn't a path between " + start.name + " and " + end.name);
    return null;
}
```

After the if statement, we set "direction"s value to a calling of a new method called printResult() using "start", "end", "changedAt", "direction" and false. After which we return direction. We chose to add this method afterwards, since the printing of the Result is nearly identical for the cheapest and the shortest path.

```
direction = printResult(start, end, changedAt, direction, false);
return direction;
```

The method printResult() gets 5 inputs: 2 VerticeWeighted "start" and "end", a HashMap "changedAt", a LinkedList "direction" and a boolean "isCheap". The boolean tells us whether the printResult() method was called in the DijkstraShortestPath() or the DijkstraCheapestPath() method. In the printResult() method, we first added a String "word" and set its value to "amount of steps", followed by an if statement that checks if "isCheap" is true, if that's the case, it changes "word"s value to "weight".

```
public LinkedList<VerticeWeighted> printResult(VerticeWeighted start, VerticeWeighted end,
HashMap<VerticeWeighted, VerticeWeighted> changedAt, LinkedList<VerticeWeighted> direction, boolean isCheap)
{
    String word = "amout of steps";
    if (isCheap)
        word = "weight";
```

After that, we have a for loop in which we retrace the way back from "end" to "start" and all the Vertices passed along the way get added to "direction". Once this is done, we reverse the elements in "direction", so that "start" is the first element and "end" is the last. Followed by the printing of a message to the console.

```
for(VerticeWeighted vertice = end; vertice != null; vertice = changedAt.get(vertice)) {
    direction.add(vertice);
    if(vertice == start)
        break;
}
Collections.reverse(direction);

System.out.println("The path with the lowest " + word +" between "
        + start.name + " and " + end.name + " is:");
```

After this, we created 4 variables: a String “path” with “” as value, an int “steps” with -1 as value, VerticeWeighted “prefVertice” with null as value, and a double “weight” with 0.0 as value. Followed by a for each VerticeWeighted in “direction” loop. In it we use the current vertice and “prefVertice” to calculate the weight of the path, we also change “path” so that it displays the path using the names of the Vertices and arrows, at the same time we also count the steps taken to get from start to finish.

```
String path = "";
int steps = -1;
VerticeWeighted prefVertice = null;
double weight = 0.0;

for(VerticeWeighted vertice : direction) {

    if(prefVertice != null) {
        weight += getEdge(prefVertice, vertice).weight;
    }
    path += vertice.name + " -> ";
    steps++;
    prefVertice = vertice;
}
```

After the loop, we then remove the last 3 chars from “path”, which correspond to an unneeded arrow. We then print out said “path”, followed by a message + the weight, followed by a message + the number of steps. And finally we return direction.

```
path = path.substring(0, path.length()-3);
System.out.println(path);
System.out.println("The path costs: " + weight);
System.out.println("And takes: " + steps + " steps.");
return direction;
```

Now the only thing left to do was to create the method randomShortestPath(), in it we created 2 VerticeWeighted “randomV1” and “randomV2” and to each of them we assigned a random Vertice from the graph as value. After which we called the DijkstraShortestPath() method using the 2 Vertices.

```
public void randomShortestPath()
{
    VerticeWeighted randomV1 = getRandomVertice();
    VerticeWeighted randomV2 = getRandomVertice();
    DijkstraShortestPath(randomV1, randomV2);
}
```

Now if we were to call the method using a graph, we would get:

```
graphWeighted.randomShortestPath();

The path with the lowest amout of steps between 1 and 6 is:
1 -> 2 -> 4 -> 6
The path costs: 55.0
And takes: 3 steps.
```

(For this example, we used the same Graph as in the first assignment)

4. Finally, write a method that takes a graph, picks two vertices at random, and finds the cheapest path between the two.

The first thing we did to complete this task was to write a method closestReachableUnvisited() that takse a HashMap <VerticeWeighted, Double> "shortestPathMap" as input. In it we created a double "shortestDistance" and set its value to positive infinity, followed by a VerticeWeighted "closestReachableVertice" with null as value.

```
private VerticeWeighted closestReachableUnvisited(HashMap<VerticeWeighted, Double> shortestPathMap) {
    double shortestDistance = Double.POSITIVE_INFINITY;
    VerticeWeighted closestReachableVertice = null;
```

After that, we have a for each VerticeWeighted in "vertices" loop. In it we first check if the current Vertice has been visited. If true, we continue with the next Vertice.

```
for(VerticeWeighted vertice : vertices) {
    if(vertice.isVisited())
        continue;
```

If not, we change the value of a double "currentDistance" to the double contained in the <VerticeWeighted, Double> pair in the "shortestPathMap" for the current Vertice. If the new value is equal to infinity, we continue the loop with the next Vertice. If the new value of "currentDistance" is smaller than "shortestDistance", we update the value of "shortestDistance" to the one from "currentDistance" and we change the value of "closestReachableVertice" to the current Vertice. After the loop, we then return "closestReachableVertice"

```
    double currentDistance = shortestPathMap.get(vertice);
    if(currentDistance == Double.POSITIVE_INFINITY)
        continue;
    if(currentDistance < shortestDistance) {
        shortestDistance = currentDistance;
        closestReachableVertice = vertice;
    }
}
return closestReachableVertice;
```

Once this was done, we continued by creating a method DijkstraCheapestPath() that gets 2 VerticeWeighted "start" and "end" as input and returns a LinkedList. In it, we first created a LinkedList "direction" to keep track of our cheapest path, and 2 HashMaps "changedAt" and "shortestPathMap". "changedAt" keeps track of every steps (between Vertices) we've tried so far, for every VerticeWeighted, and "shortestPathMap" keeps track of the shortest path we've found so far for every vertices.

```
public LinkedList<VerticeWeighted> DijkstraCheapestPath(VerticeWeighted start, VerticeWeighted end) {

    LinkedList<VerticeWeighted> direction = new LinkedList<VerticeWeighted>();

    // We keep track of which path gives us the shortest path for each vertice
    // by keeping track how we arrived at a particular vertice, we effectively
    // keep a "pointer" to the parent vertice of each vertice, and we follow that
    // path to the start
    HashMap<VerticeWeighted, VerticeWeighted> changedAt = new HashMap<>();
    changedAt.put(start, null);

    // Keeps track of the shortest path we've found so far for every vertice
    HashMap<VerticeWeighted, Double> shortestPathMap = new HashMap<>();
```

After this, we added a for each loop that sets every shortestPath weight to positive infinity, except for “start” which has a shortestPath weight of 0.

```
for (VerticeWeighted vertice : vertices) {
    if (vertice == start)
        shortestPathMap.put(start, 0.0);
    else shortestPathMap.put(vertice, Double.POSITIVE_INFINITY);
}
```

This was followed by another for each loop, in it we save the shortestPath weight of each of the Vertices we can reach from “start” and we keep track of the steps (start to other Vertices) in “changedAt”.

```
for (EdgeWeighted edge : start.edges) {
    shortestPathMap.put(edge.end, edge.weight);
    changedAt.put(edge.end, start);
}
```

After this, we check if “start” is not equal to “end”, if true, we change “start” to visited. This allows us to use direct paths between the Vertice and itself if needed and existant.

```
if(start != end)
    start.visit();
```

After the if statement, we added a while loop, that won’t stop until every reachable Vertice has been visited. In the loop, we first change the value of a new VerticeWeighted “currentVertice” to a calling of the closestReachableUnvisited() method, using “shortestPathMap” as input. As the name of the method states it, the new value of “currentVertice” corresponds to the closest unvisited Vertice from “start”.

```
VerticeWeighted currentVertice = closestReachableUnvisited(shortestPathMap);
```

We then have an if statement checking if the value of “currentVertice” is equal to null. If true, it means that all reachable Vertices have been visited and none of them is equal to “end”, which is why we print out a corresponding message to the console and return null.

```
if (currentVertice == null) {
    System.out.println("There isn't a path between " + start.name + " and " + end.name);
    return null;
}
```

This if statement is followed by another if statement, this time it's checking if "currentVertice" is equal to "end". If true, set directions value to a calling of the printResult() method. After which we return "direction". If false, we change "currentVertice" to visited.

```
if (currentVertice == end) {
    direction = printResult(start, currentVertice, changedAt, direction, true);
    return direction;
}
currentVertice.visit();
```

After that, we have a for each EdgeWeighted loop that goes through all Edges of "currentVertice". If the destination of the Edge has already been visited, we continue with the next Edge. After that we check if its current shortestPath value is smaller when going over "currentVertice" then the shortestPath value of the previous Vertice we had. If true, we put it into "shortestPathMap" and updates our steps in "changedAt".

```
for (EdgeWeighted edge : currentVertice.edges) {
    if (edge.end.isVisited())
        continue;

    if (shortestPathMap.get(currentVertice) + edge.weight < shortestPathMap.get(edge.end)) {
        shortestPathMap.put(edge.end,shortestPathMap.get(currentVertice) + edge.weight);
        changedAt.put(edge.end, currentVertice);
    }
}
```

After we finished this method, we continued by creating a method called randomCheapestPath(), in it we created 2 VerticeWeighted "randomV1" and "randomV2" and set their value to a calling of the getRandomVertice() method. After which, we called the DijkstraCheapestPath() method using the 2 random Vertices.

```
public void randomCheapestPath()
{
    VerticeWeighted randomV1 = getRandomVertice();
    VerticeWeighted randomV2 = getRandomVertice();
    DijkstraCheapestPath(randomV1, randomV2);
}
```

Now if we were to call the method using a graph, we would get:

```
graphWeighted.randomCheapestPath();


The path with the lowest weight between 1 and 4 is:
1 -> 2 -> 3 -> 4
The path costs: 9.0
And takes: 3 steps.
```

This concludes the 4th assignment.

As a final test, to make sure that the Graph and its methods work as intended, we added a calling of the DijkstraCheapestPath(), resetVerticeVisited() and DijkstraCheapestPath() methods to the constructor of the RandomGraph class. Both pathfinders get the same 2 Vertices as input.

```java
public RandomGraph(boolean directed, int vertices, int edges) {
    super(directed);

    getRandomGraph(vertices, edges);

    printEdges();

    VerticeWeighted randomV1 = getRandomVertice();
    VerticeWeighted randomV2 = getRandomVertice();
    DijkstraCheapestPath(randomV1, randomV2);
    resetVerticeVisited();
    DijkstraShortestPath(randomV1, randomV2);
}
```

We then created a RandomGraph with 10 Vertices and 50 Edges, and as a result we got this:

```
Vertice 4 has edges to:
2(66.12992087039814)
3(52.06244730936401)
0(59.56349160599218)

Vertice 0 has edges to:
7(48.587924448519274)
1(50.881532257615305)
8(1.779170720928691)
6(26.050528474530886)
9(48.98380846320194)
0(2.492067968369882)

Vertice 8 has edges to:
2(90.90722755252018)
9(66.53740629746319)
7(97.67229485676448)
5(42.28424607315089)
6(8.786866130301762)

Vertice 2 has edges to:
1(62.678416037585485)
6(23.123123449800964)
9(45.77584413451571)


Vertice 9 has edges to:
2(59.356373222546154)
7(89.09070433229593)
1(32.66857945031067)
5(24.04851795830123)
4(84.55008742853477)

Vertice 6 has edges to:
3(96.95684789151385)
1(13.29996297551025)
7(36.14711546415471)
2(27.70391362241137)

Vertice 3 has edges to:
5(97.98281640981531)
```

```
Vertice 1 has edges to:
5(32.53977019206569)
6(12.21185452288902)

Vertice 5 has edges to:
6(44.08448374586713)
7(15.26868683711725)
9(2.8958783319479764)
5(23.17357855641131)

Vertice 7 has edges to:
0(36.990774333019246)
4(65.53981846268012)
7(12.24587609119161)
3(75.86831330963125)
1(2.6539864644069766)


The path with the lowest weight between 5 and 2 is:
5 -> 7 -> 1 -> 6 -> 2
The path costs: 57.83844144682462
And takes: 4 steps.
The path with the lowest amout of steps between 5 and 2 is:
5 -> 6 -> 2
The path costs: 71.78839736827851
And takes: 2 steps.
```

Cheapest weight = 15,26868683711725 + 2,6539864644069766 + 12,21185452288902 + 27,70391362241137 = 57,83844144682462

And Lowest steps = 44,08448374586713 + 27,70391362241137 = 71,78839736827851

After confirming that it was working, we did one more test and put the callings of the Dijkstra methods in a for loop for i<100, and we changed the size of the RandomGraph to one with 50 Vertices and 200 Edges.

```
for (int i = 0; i< 100; i++)
{
    VerticeWeighted randomV1 = getRandomVertice();
    VerticeWeighted randomV2 = getRandomVertice();

    DijkstraCheapestPath(randomV1, randomV2);
    resetVerticeVisited();
    DijkstraShortestPath(randomV1, randomV2);
    resetVerticeVisited();
}
```

```
RandomGraph randomGraph = new RandomGraph (true, 50, 200);
```

After executing the program, we then got a very long list of println(). We have one excerpt of it here:

```
The path with the lowest weight between 9 and 9 is:
9
The path costs: 0.0
And takes: 0 steps.
The path with the lowest amout of steps between 9 and 9 is:
9
The path costs: 0.0
And takes: 0 steps.
The path with the lowest weight between 37 and 12 is:
37 -> 32 -> 17 -> 12
The path costs: 88.3830104798815
And takes: 3 steps.
The path with the lowest amout of steps between 37 and 12 is:
37 -> 32 -> 17 -> 12
The path costs: 88.3830104798815
And takes: 3 steps.
The path with the lowest weight between 44 and 48 is:
44 -> 24 -> 34 -> 9 -> 48
The path costs: 84.99411734705207
And takes: 4 steps.
The path with the lowest amout of steps between 44 and 48 is:
44 -> 36 -> 48
The path costs: 106.26617856019702
And takes: 2 steps.
The path with the lowest weight between 31 and 10 is:
31 -> 15 -> 46 -> 3 -> 49 -> 10
The path costs: 79.28321898012842
And takes: 5 steps.
The path with the lowest amout of steps between 31 and 10 is:
31 -> 19 -> 10
The path costs: 81.35343359773871
And takes: 2 steps.
The path with the lowest weight between 37 and 39 is:
37 -> 32 -> 44 -> 24 -> 34 -> 39
The path costs: 138.42843402466175
And takes: 5 steps.
The path with the lowest amout of steps between 37 and 39 is:
37 -> 32 -> 34 -> 39
The path costs: 159.6121828715545
And takes: 3 steps.
The path with the lowest weight between 6 and 43 is:
6 -> 22 -> 43
The path costs: 73.55205848282199
And takes: 2 steps.
The path with the lowest amout of steps between 6 and 43 is:
6 -> 22 -> 43
The path costs: 73.55205848282199
And takes: 2 steps.
```

On it we can see that there are cases where the cheapest and lowest path have the same amount of steps but a different weight, this is due to the fact that even if there are multiple ways to get from A to B, the shortestPath Algorithm will output the first one it finds.

We can also see that there is a case where the start and end are the same node, and that we get a weight and amount of steps equal to 0. As mentioned before, the algorithm doesn't allow us to take an indirect way from the Vertice to another and then back to itself, which is why we chose to output a weight and number of steps equal to 0, when ever "start" and "end" are the same.

We can thus conclude that everything is working as intended.

Reflections

Juri:

This week was by far the most difficult task. Even though we had the algorithm in the lecture, it was still complicated to understand this at home again and even more so to implement it in the code. However, we quickly found a good site that helped us with coding, and watching videos helped us to understand the algorithm. Through the code that we analyzed and tried to understand, I learned a few new useful things like the string builder or that you can define parameters so that you don't specify their amount. (Int ... n) so you get an array. After that, the algorithm for the shortest way stumped us for a while. With both algorithms we wanted to be sure of the weight, the path and the number of steps that we need. Which is why, we had to find a new algorithm. In the end, however, we had to adapt both algorithms strongly and actually only kept the functionality and the data structures.

Bartholomäus:

Working with the searching algorithms and getting my head around things like LinkedLists in a Set, was confusing and frustrating at first. But after working on it extensively with many trials and errors, and having a skilled lab partner that has a grasp of what we had to do, was very helpful in solving the assignments. Which is why I now think that I understand how everything fits together.

Code

```
package ubung10_final;

public interface WeightedGraph {

	void addVertice(VerticeWeighted...n);
	void addEdge(VerticeWeighted start, VerticeWeighted end, double
weight);
	void addEdgeHelper(VerticeWeighted start, VerticeWeighted end,
double weight);
	void printEdges();
	boolean hasEdge(VerticeWeighted start, VerticeWeighted end);
	void resetVerticeVisited();


}
```

```
package ubung10_final;

public class EdgeWeighted implements Comparable<EdgeWeighted>{
	public VerticeWeighted start;
	public VerticeWeighted end;
	public double weight;


	public EdgeWeighted(VerticeWeighted start, VerticeWeighted end,
double weight){
		this.start = start;
		this.end = end;
		this.weight = weight;
	}

	@Override
	public String toString() {
		return String.format("(%s -> %s, %f)",start.name, end.name,
weight);
	}

	@Override
	public int compareTo(EdgeWeighted otherEdge) {
		if(this.weight > otherEdge.weight)
			return 1;
		else
			return -1;
```

```
	}
}
```

```
package ubung10_final;
import java.util.LinkedList;

public class VerticeWeighted {
	public int n;
	public String name;
	private boolean visited;
	LinkedList<EdgeWeighted> edges;

	public VerticeWeighted(int n, String name) {
		this.n = n;
		this.name = name;
		this.visited = false;
		this.edges = new LinkedList<>();
	}

	public boolean isVisited() {
		return visited;
	}
	public void visit() {
		visited = true;
	}
	public void unvisit() {
		visited = false;
	}

	public LinkedList<VerticeWeighted> getChildVertice() {
		LinkedList<VerticeWeighted> childVertices = new
LinkedList<VerticeWeighted>();
		for(EdgeWeighted edge : edges) {
			if(n == edge.start.n)
				childVertices.add(edge.end);
			if(n == edge.end.n)
				childVertices.add(edge.start);
		}
		return childVertices;
	}
}
```

```
package ubung10_final;
import java.util.Arrays;
import java.util.Collections;
```

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;
import java.util.Set;



public class GraphWeighted implements WeightedGraph{
	private Set<VerticeWeighted> vertices;
	private boolean directed;

	public GraphWeighted(boolean directed) {
		this.directed = directed;
		vertices = new HashSet<>();
	}
	@Override
	public void addVertice(VerticeWeighted...n) {
		vertices.addAll(Arrays.asList(n));
	}
	@Override
	public void addEdge(VerticeWeighted start, VerticeWeighted end,
double weight) {

		vertices.add(start);
		vertices.add(end);

		//Making sure, that we dont have duplicated edges.
		addEdgeHelper(start, end, weight);
		if(!directed && start != end) {
			addEdgeHelper(end, start, weight);
		}

	}
	@Override
	public void addEdgeHelper(VerticeWeighted start, VerticeWeighted
end, double weight) {

		for (EdgeWeighted edge : start.edges) {
		    if (edge.start.equals(start) &&
edge.end.equals(end)) {
				edge.weight = weight;
				return;
```

```
                    }
                }
                start.edges.add(new EdgeWeighted(start, end, weight));


        }

        @Override
        public void printEdges() {
                for(VerticeWeighted vertice : vertices) {
                        LinkedList<EdgeWeighted> edges = vertice.edges;

                        if(edges.isEmpty()) {
                                System.out.println("Vertice " + vertice.name + "
has no edges.");
                                continue;
                        }
                        System.out.println("Vertice " + vertice.name + " has
edges to:");

                        for(EdgeWeighted edge : edges) {
                                System.out.println(edge.end.name + "(" +
edge.weight + ")");
                        }
                        System.out.println();
                }
        }
        @Override
        public boolean hasEdge(VerticeWeighted start, VerticeWeighted end)
{
            LinkedList<EdgeWeighted> edges = start.edges;
            for (EdgeWeighted edge : edges) {
                if (edge.end == end) {
                    return true;
                }
            }
            return false;
        }
        @Override
        public void resetVerticeVisited() {
                for(VerticeWeighted vertice : vertices) {
                        vertice.unvisit();
                }
        }
```

```
      public LinkedList<VerticeWeighted>
DijkstraCheapestPath(VerticeWeighted start, VerticeWeighted end) {

            LinkedList<VerticeWeighted> direction = new
LinkedList<VerticeWeighted>();

          // We keep track of which path gives us the shortest path for
each vertice
          // by keeping track how we arrived at a particular vertice, we
effectively
          // keep a "pointer" to the parent vertice of each vertice, and
we follow that
          // path to the start
          HashMap<VerticeWeighted, VerticeWeighted> changedAt = new
HashMap<>();
          changedAt.put(start, null);

          // Keeps track of the shortest path we've found so far for
every vertice
          HashMap<VerticeWeighted, Double> shortestPathMap = new
HashMap<>();

          // Setting every vertice`s shortest path weight to positive
infinity to start
          // except the starting vertice, whose shortest path weight is
0
          for (VerticeWeighted vertice : vertices) {
              if (vertice == start)
                  shortestPathMap.put(start, 0.0);
              else shortestPathMap.put(vertice,
Double.POSITIVE_INFINITY);
          }

          // Now we go through all the vertice we can go to from the
starting vertice
          // (this keeps the loop a bit simpler)
          for (EdgeWeighted edge : start.edges) {
              shortestPathMap.put(edge.end, edge.weight);
              changedAt.put(edge.end, start);
          }
          //added by Juri
          if(start != end)
            start.visit();

          // This loop runs as long as there is an unvisited vertice
```

```
that we can
          // reach from any of the vertices we could till then
          while (true) {

            VerticeWeighted currentVertice =
closestReachableUnvisited(shortestPathMap);

              // If we haven't reached the end Vertice yet, and there
isn't another
              // reachable vertice the path between start and end
doesn't exist
              // (they aren't connected)
              if (currentVertice == null) {
                  System.out.println("There isn't a path between " +
start.name + " and " + end.name);
                  return null;
              }
              // If the closest non-visited Vertice is our destination,
we want to print the path
              if (currentVertice == end) {
                  direction = printResult(start, currentVertice,
changedAt, direction, true);
                  return direction;
              }
              currentVertice.visit();

              // Now we go through all the unvisited vertices our
current vertice has an edge to
              // and check whether its shortest path value is better
when going through our
              // current vertice than whatever we had before
              for (EdgeWeighted edge : currentVertice.edges) {
                  if (edge.end.isVisited())
                      continue;

                  if (shortestPathMap.get(currentVertice) + edge.weight
< shortestPathMap.get(edge.end)) {

shortestPathMap.put(edge.end,shortestPathMap.get(currentVertice) +
edge.weight);
                      changedAt.put(edge.end, currentVertice);
                  }
              }
          }
      }
```

```
	public LinkedList<VerticeWeighted>
DijkstraShortestPath(VerticeWeighted start, VerticeWeighted end) {


//https://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes
-for-unweighted-graph
		HashMap<VerticeWeighted, VerticeWeighted> changedAt = new
HashMap<>();

		LinkedList<VerticeWeighted> direction = new
LinkedList<VerticeWeighted>();

		Queue q = new LinkedList();

		VerticeWeighted currentVertice = start;
		q.add(currentVertice);

		start.visit();

		while(!q.isEmpty()) {
			currentVertice = (VerticeWeighted) q.remove();
			if(currentVertice.equals(end)) {
				break;
			}
			else {
				for(VerticeWeighted vertice :
currentVertice.getChildVertice()) {
					if(!vertice.isVisited()) {
						q.add(vertice);
						vertice.visit();
						changedAt.put(vertice,
currentVertice);
					}
				}
			}
		}
		if(!currentVertice.equals(end)) {
			System.out.println("There isn't a path between " +
start.name + " and " + end.name);
			return null;
		}


		direction = printResult(start, end, changedAt, direction,
false);
```

```
            return direction;
      }

      private VerticeWeighted
closestReachableUnvisited(HashMap<VerticeWeighted, Double>
shortestPathMap) {
            double shortestDistance = Double.POSITIVE_INFINITY;
            VerticeWeighted closestReachableVertice = null;

            for(VerticeWeighted vertice : vertices) {
                  if(vertice.isVisited())
                        continue;

                  double currentDistance = shortestPathMap.get(vertice);
                  if(currentDistance == Double.POSITIVE_INFINITY)
                        continue;
                  if(currentDistance < shortestDistance) {
                        shortestDistance = currentDistance;
                        closestReachableVertice = vertice;
                  }
            }
            return closestReachableVertice;
      }

      public void getRandomGraph(int vertices, int edges) {
            //reset the current
            this.vertices = new HashSet<>();

            for(int i = 0; i<vertices;i++) {
                  this.vertices.add(new VerticeWeighted(i,
Integer.toString(i)));
            }
            for(int i = 0; i<edges;i++) {
                  VerticeWeighted randomV1 = getRandomVertice();
                  VerticeWeighted randomV2 = getRandomVertice();;

                  addEdge(randomV1, randomV2, Math.random()*100);

            }
      }

      public VerticeWeighted getRandomVertice() {
            int size = vertices.size();
            int item = new Random().nextInt(size);
            int i = 0;
```

```
            for(VerticeWeighted vertice : vertices) {
                  if(i == item)
                        return vertice;
                  i++;
            }
            return null;
      }
      public EdgeWeighted getEdge(VerticeWeighted V1, VerticeWeighted
V2) {
            boolean isChild = false;
            for(VerticeWeighted vertice : V1.getChildVertice()) {
                  if(V2.equals(vertice))
                        isChild = true;

            }
            if(!isChild) {
                  System.out.println("Not a Child");
                  return null;
            }

            for(EdgeWeighted edge : V1.edges) {
                  if(edge.start == V1 && edge.end == V2 ||
                     edge.start == V2 && edge.end == V1)
                        return edge;
            }
            System.out.println("Not possible");
            return null;
      }

      public void randomShortestPath()
      {
            VerticeWeighted randomV1 = getRandomVertice();
            VerticeWeighted randomV2 = getRandomVertice();
        DijkstraShortestPath(randomV1, randomV2);
      }

      public void randomCheapestPath()
      {
            VerticeWeighted randomV1 = getRandomVertice();
            VerticeWeighted randomV2 = getRandomVertice();
        DijkstraCheapestPath(randomV1, randomV2);
      }
      public LinkedList<VerticeWeighted> printResult(VerticeWeighted
start, VerticeWeighted end,
      HashMap<VerticeWeighted, VerticeWeighted> changedAt,
```

```
LinkedList<VerticeWeighted> direction, boolean isCheap)
	{
		String word = "amout of steps";
		if (isCheap)
			word = "weight";

		for(VerticeWeighted vertice = end; vertice != null; vertice
= changedAt.get(vertice)) {
			direction.add(vertice);
			if(vertice == start)
				break;
		}
		Collections.reverse(direction);

		System.out.println("The path with the lowest " + word +"
between "
		    + start.name + " and " + end.name + " is:");
		String path = "";
		int steps = -1;
		VerticeWeighted prefVertice = null;
		double weight = 0.0;
		for(VerticeWeighted vertice : direction) {

			if(prefVertice != null) {
				weight += getEdge(prefVertice, vertice).weight;
			}
			path += vertice.name + " -> ";
			steps++;
			prefVertice = vertice;
		}
		path = path.substring(0, path.length()-3);
		System.out.println(path);
		System.out.println("The path costs: " + weight);
		System.out.println("And takes: " + steps + " steps.");
		return direction;
	}
}

package ubung10_final;

public class RandomGraph extends GraphWeighted{

	public RandomGraph(boolean directed, int vertices, int edges) {
		super(directed);
```

```
            getRandomGraph(vertices, edges);

          printEdges();

        for (int i = 0; i< 100; i++)
        {
              VerticeWeighted randomV1 = getRandomVertice();
              VerticeWeighted randomV2 = getRandomVertice();

              DijkstraCheapestPath(randomV1, randomV2);
              resetVerticeVisited();
              DijkstraShortestPath(randomV1, randomV2);
              resetVerticeVisited();
        }
        }

}
```

```
package ubung10_final;
public class GraphShow {
    public static void main(String[] args) {
      RandomGraph randomGraph = new RandomGraph (true, 50, 200);

        GraphWeighted graphWeighted = new GraphWeighted(true);


        VerticeWeighted zero = new VerticeWeighted(0, "0");
        VerticeWeighted one = new VerticeWeighted(1, "1");
        VerticeWeighted two = new VerticeWeighted(2, "2");
        VerticeWeighted three = new VerticeWeighted(3, "3");
        VerticeWeighted four = new VerticeWeighted(4, "4");
        VerticeWeighted five = new VerticeWeighted(5, "5");
        VerticeWeighted six = new VerticeWeighted(6, "6");

        // Our addEdge method automatically adds Nodes as well.
        // The addNode method is only there for unconnected Nodes,
        // if we wish to add any
        graphWeighted.addEdge(zero, one, 8);
        graphWeighted.addEdge(one, zero, 11);
        graphWeighted.addEdge(two, two, 3);
        graphWeighted.addEdge(three, three, 8);
        graphWeighted.addEdge(two, four, 28);
        graphWeighted.addEdge(five, four, 9);
        graphWeighted.addEdge(four, six, 25);
        graphWeighted.addEdge(one, two, 2);
```

```
        graphWeighted.addEdge(two, three, 6);
        graphWeighted.addEdge(three, four, 1);
        graphWeighted.addEdge(four, five, 7);
        graphWeighted.addEdge(five, six, 8);

        //graphWeighted.printEdges();

        //graphWeighted.randomShortestPath();
       // graphWeighted.resetVerticeVisited();

        //graphWeighted.randomCheapestPath();

        /*
        graphWeighted.DijkstraShortestPath(zero, zero);
        graphWeighted.resetVerticeVisited();
        graphWeighted.DijkstraCheapestPath(zero, zero);

        graphWeighted.RandomGraph(20, 100);
        graphWeighted.printEdges();
            for(int i = 0;i<10;i++) {
                  VerticeWeighted randomV1 =
graphWeighted.getRandomVertice();
                  VerticeWeighted randomV2 =
graphWeighted.getRandomVertice();
              graphWeighted.DijkstraCheapestPath(randomV1, randomV2);
              graphWeighted.resetVerticeVisited();
              graphWeighted.DijkstraShortestPath(randomV1, randomV2);
              graphWeighted.resetVerticeVisited();
              System.out.println();
            }
            */

    }
}
```