Info 2

Exercise 10: Getting from A to B

Prof. Weber-Wulff

2020.01.07

Pavel Tsvyatkov s0559632

(Pavel.Tsvyatkov@Student.HTW-Berlin.de)

Cansu Ilhan s0571394

(Cansu.Ilhan@Student.HTW-Berlin.de)

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?

We started by discussing the finger exercises. We looked at the resources we had found and talked about what we are potentially going to need for our data type. There was a lot of information available online, so we spent some time reading about abstract data types for weighted graphs.

Then we thought about whether we should use an adjacency list or an adjacency matrix. We had the idea that using a list would be easier for us to understand, but we decided to search for implementations online to get a better understanding of the whole process. We found several pages that were very helpful and reading through them was valuable. The second page that we found had some great explanations about representing graphs in code and Dijkstra's Algorithm.

https://algorithms.tutorialhorizon.com/weighted-graph-implementation-java/
https://stackabuse.com/graphs-in-java/

After taking our time reading the resources, we saw that we wouldn't all the methods we found in the first finger exercise to design our ADT for a WeightedGraph, so we included only what we thought was necessary for now. We also asked Juri's group for advice and they helped us with the implementation.

```java
public interface WeightedGraph {

    //https://stackoverflow.com/questions/3158730/java-3-dots-in-parameters
    void addVertex(Vertex...n);
    void addEdge(Vertex start, Vertex destination, double weight);
    void addEdgeHelper(Vertex start, Vertex destination, double weight);
    void printEdges();
    boolean hasEdge(Vertex start, Vertex destination);
    void resetVisitedVertices();

}
```

While reading about the implementation here: https://stackabuse.com/graphs-in-java-dijkstras-algorithm/ we noticed a method that had three dots in it.

We read about it and found out that this is a construct called varargs to pass an arbitrary number of values to a method.
https://stackoverflow.com/questions/3158730/java-3-dots-in-parameters
https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html#varargs

We started by creating a new class MyWeightedGraph that implements the WeightedGraph interface. We are using a Set for the vertices and we have a boolean variable called directed to choose whether we want the Graph to be directed or undirected.

```
13 public class MyWeightedGraph implements WeightedGraph{
14     // using a Set will not allow duplicated vertices
15     private Set<Vertex> vertices;
16     // is the Graph directed (digraph) or undirected?
17     private boolean directed;
18     Random r = new Random();
19 
20     // Constructor
21     public MyWeightedGraph(boolean directed) {
22         this.directed = directed;
23         vertices = new HashSet<>();
24     }
```

After that, to store the weights we created a class called EdgeWeighted which is going to represent an edge in the graph.
(as shown here: https://stackabuse.com/graphs-in-java-dijkstras-algorithm/)

```
1  public class EdgeWeighted implements Comparable<EdgeWeighted>{
2  
3      public Vertex start;
4      public Vertex end;
5      public double weight;
6  
7      public EdgeWeighted(Vertex start, Vertex end, double weight){
8          this.start = start;
9          this.end = end;
10         this.weight = weight;
11     }
```

We also implement Comparable and override the compareTo method to be able to deal with exercise 4 later on.

We created another class called Vertex. We use an adjacency list for the neighbors of a Vertex (LinkedList of edges) and we keep track if a certain Vertex has been visited in a boolean variable called visited.

```java
public class Vertex {
    public int n;
    public String name;
    private boolean visited; // keeping track if the Vertex has been visited
    LinkedList<EdgeWeighted> edges;

    public Vertex(int n, String name) {
        this.n = n;
        this.name = name;
        this.visited = false;
        this.edges = new LinkedList<>(); // a vertex has a linked list of edges
    }
```

We remembered that in the lab Prof. Weber-Wulff said we are definitely going to need a method that gets the neighbors of a given vertex. In the Vertex class we have the following method that returns the neighbors.

```java
    public LinkedList<Vertex> getAdjVertices() {
        LinkedList<Vertex> adjVertices = new LinkedList<Vertex>();
        for (EdgeWeighted edge : edges) {
            if (n == edge.start.n)
                adjVertices.add(edge.end);
            if (n == edge.end.n)
                adjVertices.add(edge.start);
        }
        return adjVertices;
    }
```

The addEdge method takes three parameters, a start and an end Vertex and the weight of the specific edge.

Since we are using a Set for the vertices, there won't be any duplicates. To ensure we also have no duplicate edges, we are using an addEdgeHelper method that goes through all the edges and checks if an edge has already been added.

```
31      public void addEdge(Vertex start, Vertex end, double weight) {
32
33          vertices.add(start);
34          vertices.add(end);
35
36          // no duplicated edges
37          addEdgeHelper(start, end, weight);
38          if(!directed && start != end) {
39              addEdgeHelper(end, start, weight);
40          }
41
42      }
43      @Override
44      public void addEdgeHelper(Vertex start, Vertex end, double weight) {
45
46              for (EdgeWeighted edge : start.edges) {
47                  if (edge.start.equals(start) && edge.end.equals(end)) {
48                      edge.weight = weight;
49                      return;
50                  }
51              }
52              start.edges.add(new EdgeWeighted(start, end, weight));
53
54
55      }
```

The following method prints out the edges. We use a for loop which goes over the Set of vertices and stores the edges in a LinkedList that we named edges. In case the LinkedList is empty we print out a proper message. Otherwise we print out the name of the Vertex, then loop through the LinkedList and print out the names of the edges that this vertex has an edge to and the weight(cost going to that edge) in parentheses.

```
58      public void printEdges() {
59          for(Vertex vertice : vertices) {
60              LinkedList<EdgeWeighted> edges = vertice.edges;
61
62              if(edges.isEmpty()) {
63                  System.out.println("Vertex " + vertice.name + " has no edges.");
64                  continue;
65              }
66              System.out.println("Vertex " + vertice.name + " has edges to:");
67
68              for(EdgeWeighted edge : edges) {
69                  System.out.println(edge.end.name + "(" + edge.weight + ")");
70              }
71              System.out.println();
72          }
73      }
```

We also have the following method that sets all visited notes to unvisited.

```
84    @Override
85    public void resetVisitedVertices() {
86        for(Vertex vertice : vertices) {
87            vertice.unvisit();
88        }
89    }
```

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.

To generate a random vertex we wrote the following method. We collaborated with Juri's group and got some help from them.

```
276    public Vertex getRandomVertex() {
277        int index = 0;
278        int size = vertices.size();
279        int counter = r.nextInt(size);
280
281        for(Vertex v : vertices) {
282            if(index == counter)
283                return v;
284            index++;
285        }
286        return null;
287    }
```

To create the edges randomly and assign a random weight, we use a for loop. Every time we loop, two random vertices called vert1 and vert2 are created and then we call the addEdge in which we pass vert1, vert2 and generate a random weight between 1 and 9.

```
267        for(int i = 0; i<edges;i++) {
268            Vertex vert1 = getRandomVertex();
269            Vertex vert2 = getRandomVertex();
270
271            addEdge(vert1, vert2, r.nextInt(9)+1);
272
273        }
274    }
```

We created a new class RandomGraph and created a new myWeightedGraph object. We wanted to have an undirected Graph so we set the condition to false. We set the amount of vertices to be generated to 20 and of edges to 45.

```
5       MyWeightedGraph myWeightedGraph = new MyWeightedGraph(false);
6       int vertices = 20;
7       int edges = 45;
```

To print our graph, we now had to call the printEdges() method.We have 20 vertices in total with names: (Vertex) 0, (Vertex) 1 … until (Vertex) 19.
The edges and weights were also generated randomly every time we ran the program. We show the **edge's weight in parentheses**.

```
Vertex 4 has edges to: 1(5.0) 6(6.0) 19(2.0) 11(7.0)
Vertex 5 has edges to: 17(5.0) 11(7.0) 5(1.0)
Vertex 3 has edges to: 17(9.0) 15(4.0)
Vertex 11 has edges to: 12(7.0) 7(6.0) 8(4.0) 5(7.0) 13(8.0) 9(9.0) 17(1.0) 4(7.0)
Vertex 17 has edges to: 3(9.0) 5(5.0) 10(4.0) 11(1.0) 7(5.0)
Vertex 13 has edges to: 11(8.0) 10(1.0) 19(5.0)
Vertex 19 has edges to: 2(7.0) 14(2.0) 4(2.0) 13(5.0)
Vertex 7 has edges to: 11(6.0) 12(3.0) 7(9.0) 1(9.0) 6(7.0) 17(5.0) 9(7.0)
Vertex 16 has edges to: 2(8.0) 12(4.0)
Vertex 14 has edges to: 9(3.0) 8(2.0) 19(2.0) 0(3.0) 12(9.0)
Vertex 2 has edges to: 19(7.0) 16(8.0) 0(8.0)
Vertex 9 has edges to: 14(3.0) 11(9.0) 7(7.0) 15(9.0)
Vertex 15 has edges to: 3(4.0) 9(9.0)
Vertex 1 has edges to: 4(5.0) 6(6.0) 7(9.0) 12(1.0)
Vertex 10 has edges to: 6(9.0) 17(4.0) 13(1.0) 18(9.0)
Vertex 12 has edges to: 11(7.0) 7(3.0) 16(4.0) 14(9.0) 8(7.0) 1(1.0)
Vertex 8 has edges to: 14(2.0) 11(4.0) 6(8.0) 12(7.0)
Vertex 0 has edges to: 0(5.0) 14(3.0) 18(4.0) 2(8.0)
Vertex 6 has edges to: 10(9.0) 4(6.0) 8(8.0) 1(6.0) 18(8.0) 7(7.0)
Vertex 18 has edges to: 6(8.0) 10(9.0) 0(4.0)
```

We ran the program some more times and saw that every time the edges were randomly generated and had different weights.

It was also possible that a Vertex has no edges.

```
Vertex 4 has edges to: 18(6.0) 16(7.0) 7(8.0) 13(6.0) 19(4.0)
Vertex 5 has no edges.
Vertex 3 has edges to: 7(5.0) 10(3.0) 6(6.0) 11(4.0)
Vertex 11 has edges to: 14(7.0) 2(5.0) 18(7.0) 3(4.0) 15(1.0)
Vertex 17 has edges to: 7(8.0) 10(7.0) 9(7.0) 19(6.0)
Vertex 13 has edges to: 6(5.0) 4(6.0)
Vertex 19 has edges to: 17(6.0) 14(4.0) 4(4.0)
Vertex 7 has edges to: 17(8.0) 4(8.0) 3(5.0) 18(8.0)
Vertex 16 has edges to: 14(7.0) 18(3.0) 4(7.0) 9(5.0) 0(9.0) 12(9.0)
Vertex 14 has edges to: 16(7.0) 11(7.0) 15(5.0) 9(3.0) 10(5.0) 19(4.0) 2(3.0)
Vertex 2 has edges to: 11(5.0) 12(5.0) 14(3.0)
Vertex 9 has edges to: 8(6.0) 17(7.0) 10(3.0) 16(5.0) 14(3.0) 15(9.0)
Vertex 15 has edges to: 14(5.0) 10(9.0) 9(9.0) 8(6.0) 6(2.0) 18(6.0) 11(1.0)
Vertex 1 has edges to: 8(5.0)
Vertex 10 has edges to: 17(7.0) 6(4.0) 15(9.0) 9(3.0) 3(3.0) 14(5.0)
Vertex 12 has edges to: 2(5.0) 16(9.0)
Vertex 8 has edges to: 9(6.0) 1(5.0) 8(9.0) 18(7.0) 15(6.0)
Vertex 0 has edges to: 16(9.0)
Vertex 6 has edges to: 13(5.0) 10(4.0) 15(2.0) 3(6.0)
Vertex 18 has edges to: 16(3.0) 4(6.0) 8(7.0) 7(8.0) 11(7.0) 15(6.0)
```

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 implement this and the following method we looked at different examples and explanations online again. We had to adapt it to our needs as well.

https://www.baeldung.com/java-dijkstra
https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm

https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex
  with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

We should also give credit to Sophia's and Juri's groups, because they collaborated with us and helped us with the implementation.

We created the method getShortestPath, which takes a start and an end vertex and returns a LinkedList. The main idea is to get from A to B, choosing the shortest length of the complete path, without looking at the weights. (https://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes-for-unweighted-graph)

```
184    public LinkedList<Vertex> getShortestPath(Vertex start, Vertex end) {
185
186        //
187        HashMap<Vertex, Vertex> changedAt = new HashMap<>();
188
189        LinkedList<Vertex> direction = new LinkedList<Vertex>();
190
191        Queue q = new LinkedList();
192        Vertex currentVertice = start;
193        q.add(currentVertice);
194        start.visit();
195        while(!q.isEmpty()) {
196            currentVertice = (Vertex) q.remove();
197            if(currentVertice.equals(end)) {
198                break;
199            }
200            else {
201                for(Vertex vertice : currentVertice.getAdjVertices()) {
202                    if(!vertice.isVisited()) {
203                        q.add(vertice);
204                        vertice.visit();
205                        changedAt.put(vertice, currentVertice);
206                    }
207                }
208            }
209        }
```

The idea was to keep track of which path gives us the shortest path for every vertex by keeping track how we arrived at the particular vertex. We also need to use a queue, in which we add the current vertex, and we need to mark vertices as visited. In the for loop, we go through the neighbors of a specific vertex by calling the getAdjVertices method. If the current vertex is unvisited, we add it to the queue, mark it as visited and update the HashMap.

In the end of the method we print the shortest path in a readable format.

```
222        System.out.println("The shortest path between "
223                + start.name + " and " + end.name + " is:");
224        String path = "";
225        int steps = -1;
226        Vertex prefVertice = null;
227        double weight = 0.0;
228        for(Vertex vertice : direction) {
229
230            if(prefVertice != null) {
231                weight += getEdge(prefVertice, vertice).weight;
232            }
233            path += vertice.name + " -> ";
234            steps++;
235            prefVertice = vertice;
236        }
237        path = path.substring(0, path.length()-3);
238        System.out.println(path);
239        System.out.println("The shortest path costs: " + weight);
240        System.out.println("The shortest path needs " + steps + " steps.");
241        return direction;
```

To check how the method works, first we created two random vertices. After that we call the getShortestPath method and pass the random vertices as parameters.

```
13          // choose two random vertices
14          Vertex v = myWeightedGraph.getRandomVertex();
15          Vertex v2 = myWeightedGraph.getRandomVertex();
16
17          // find the shortest path between the two random vertices
18          myWeightedGraph.getShortestPath(v, v2);
```

We got the following result. We made sure to check if the cost is getting calculated correctly.

```
Vertex 4 has edges to: 10(1.0) 8(4.0) 18(2.0)
Vertex 5 has edges to: 1(9.0) 15(1.0) 14(8.0)
Vertex 3 has edges to: 7(7.0) 2(8.0)
Vertex 11 has edges to: 17(3.0) 6(1.0) 13(1.0)
Vertex 17 has edges to: 2(6.0) 0(1.0) 8(6.0) 11(3.0) 10(2.0)
Vertex 13 has edges to: 16(3.0) 8(1.0) 2(8.0) 11(1.0) 6(8.0)
Vertex 19 has edges to: 14(8.0) 10(7.0)
Vertex 7 has edges to: 16(1.0) 14(5.0) 3(7.0) 2(4.0) 15(4.0)
Vertex 16 has edges to: 1(7.0) 13(3.0) 9(1.0) 7(1.0) 14(6.0) 8(2.0)
Vertex 14 has edges to: 19(8.0) 16(6.0) 7(5.0) 5(8.0)
Vertex 2 has edges to: 17(6.0) 1(1.0) 13(8.0) 3(8.0) 7(4.0)
Vertex 9 has edges to: 16(1.0) 1(7.0) 12(4.0) 15(3.0)
Vertex 15 has edges to: 8(2.0) 1(4.0) 9(3.0) 5(1.0) 7(4.0) 12(4.0)
Vertex 1 has edges to: 16(7.0) 2(1.0) 5(9.0) 6(4.0) 15(4.0) 9(7.0)
Vertex 10 has edges to: 4(1.0) 6(4.0) 17(2.0) 12(3.0) 19(7.0)
Vertex 12 has edges to: 9(4.0) 10(3.0) 15(4.0)
Vertex 8 has edges to: 15(2.0) 13(1.0) 16(2.0) 4(4.0) 17(6.0) 18(4.0)
Vertex 0 has edges to: 17(1.0)
Vertex 6 has edges to: 10(4.0) 1(4.0) 18(2.0) 11(1.0) 13(8.0)
Vertex 18 has edges to: 6(2.0) 4(2.0) 8(4.0)

The shortest path between 6 and 8 is:
6 -> 18 -> 8
The shortest path costs: 6.0
The shortest path needs 2 steps.
```

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

We started by creating a new method getCheapestPath, that takes a start and an end vertex and returns a LinkedList. This is similar to the shortest path, but the difference between the two is the value that they try to minimize. In the cheapest path case, we try to get from A to B and find the cheapest weighted path length.

To implement this method we used the following source:
(https://stackabuse.com/graphs-in-java-dijkstras-algorithm/)

```
91    public LinkedList<Vertex> getCheapestPath(Vertex start, Vertex end) {
92        LinkedList<Vertex> direction = new LinkedList<Vertex>();
93        // We keep track of which path gives us the shortest path for each vertice
94        // by keeping track how we arrived at a particular vertice, we effectively
95        // keep a "pointer" to the parent vertice of each vertice, and we follow that
96        // path to the start
97        HashMap<Vertex, Vertex> changedAt = new HashMap<>();
98        changedAt.put(start, null);
99
100       // Keeps track of the shortest path we've found so far for every vertice
101       HashMap<Vertex, Double> shortestPathMap = new HashMap<>();
102
103       // Setting every vertice`s shortest path weight to positive infinity to start
104       // except the starting vertice, whose shortest path weight is 0
105       for (Vertex vertice : vertices) {
106           if (vertice == start)
107               shortestPathMap.put(start, 0.0);
108           else shortestPathMap.put(vertice, Double.POSITIVE_INFINITY);
109       }
110
111       // Now we go through all the vertice we can go to from the starting vertice
112       // (this keeps the loop a bit simpler)
113       for (EdgeWeighted edge : start.edges) {
114           shortestPathMap.put(edge.end, edge.weight);
115           changedAt.put(edge.end, start);
116       }
```

In the following lines of code of the cheapest path method, we check if the weighted path is better than what we had before and update the HashMap.

```
166           // Now we go through all the unvisited vertices our current vertice has an edge to
167           // and check whether its shortest path value is better when going through our
168           // current vertice than whatever we had before
169           for (EdgeWeighted edge : currentVertice.edges) {
170               if (edge.end.isVisited())
171                   continue;
172
173               if (shortestPathMap.get(currentVertice) + edge.weight
174                  < shortestPathMap.get(edge.end)) {
175                   shortestPathMap.put(edge.end,
176                                       shortestPathMap.get(currentVertice) + edge.weight);
177                   changedAt.put(edge.end, currentVertice);
178               }
179           }
```

We also need a way to print out the cheapest path in a readable format.

```
143                 System.out.println("The cheapest path between "
144                         + start.name + " and " + end.name + " is:");
```

```
159             System.out.println(path);
160             System.out.println("The cheapest path costs: " + weight);
161             System.out.println("The cheapest path needs " + steps + " steps.");
```

For the getCheapestPath method to work, we also need the following method that evaluates which is the closest node that we can reach and haven't visited before.

```
243    private Vertex closestReachableUnvisited(HashMap<Vertex, Double> shortestPathMap) {
244        double shortestDistance = Double.POSITIVE_INFINITY;
245        Vertex closestReachableVertice = null;
246        for(Vertex vertice : vertices) {
247            if(vertice.isVisited())
248                continue;
249
250            double currentDistance = shortestPathMap.get(vertice);
251            if(currentDistance == Double.POSITIVE_INFINITY)
252                continue;
253            if(currentDistance < shortestDistance) {
254                shortestDistance = currentDistance;
255                closestReachableVertice = vertice;
256            }
257        }
258        return closestReachableVertice;
259    }
```

We tried to print out the cheapest path and got the following result.

```
The cheapest path between 16 and 1 is:
16 -> 9 -> 15 -> 18 -> 1
The cheapest path costs: 19.0
The cheapest path needs 4 steps.
```

Now we were able to print everything together. We went on to print the edges, shortest path and cheapest path altogether.

```
Vertex 4 has edges to: 8(5.0)
Vertex 5 has edges to: 17(2.0) 15(5.0)
Vertex 3 has edges to: 0(9.0) 15(4.0) 1(5.0)
Vertex 11 has edges to: 1(2.0) 14(2.0) 0(9.0) 13(9.0)
Vertex 17 has edges to: 16(6.0) 5(2.0) 15(7.0) 13(9.0) 9(3.0)
Vertex 13 has edges to: 0(4.0) 6(4.0) 11(9.0) 17(9.0) 16(2.0)
Vertex 19 has edges to: 2(5.0)
Vertex 7 has edges to: 0(1.0) 18(9.0) 9(6.0)
Vertex 16 has edges to: 8(4.0) 17(6.0) 10(8.0) 13(2.0)
Vertex 14 has edges to: 6(2.0) 8(3.0) 11(2.0) 9(3.0) 2(1.0)
Vertex 2 has edges to: 12(5.0) 18(9.0) 15(4.0) 19(5.0) 14(1.0)
Vertex 9 has edges to: 8(4.0) 14(3.0) 18(4.0) 7(6.0) 17(3.0)
Vertex 15 has edges to: 2(4.0) 6(9.0) 10(2.0) 17(7.0) 3(4.0) 5(5.0)
Vertex 1 has edges to: 11(2.0) 10(2.0) 3(5.0)
Vertex 10 has edges to: 1(2.0) 15(2.0) 16(8.0)
Vertex 12 has edges to: 2(5.0) 0(8.0)
Vertex 8 has edges to: 16(4.0) 4(5.0) 14(3.0) 9(4.0) 18(6.0)
Vertex 0 has edges to: 3(9.0) 13(4.0) 12(8.0) 18(8.0) 11(9.0) 7(1.0)
Vertex 6 has edges to: 14(2.0) 15(9.0) 13(4.0) 18(7.0)
Vertex 18 has edges to: 2(9.0) 0(8.0) 9(4.0) 8(6.0) 7(9.0) 6(7.0)

The shortest path between 5 and 2 is:
5 -> 15 -> 2
The shortest path costs: 9.0
The shortest path needs 2 steps.

The cheapest path between 11 and 19 is:
11 -> 14 -> 2 -> 19
The cheapest path costs: 8.0
The cheapest path needs 3 steps.
```

Sometimes the shortest path was also the cheapest path.

```
Vertex 4 has edges to: 16(1.0) 6(5.0)
Vertex 5 has edges to: 7(9.0) 19(9.0) 9(6.0) 10(7.0) 17(4.0)
Vertex 3 has edges to: 12(2.0) 2(2.0) 19(6.0)
Vertex 11 has edges to: 7(3.0)
Vertex 17 has edges to: 14(5.0) 13(2.0) 5(4.0) 7(8.0)
Vertex 13 has edges to: 15(3.0) 17(2.0) 1(6.0)
Vertex 19 has edges to: 2(7.0) 5(9.0) 1(7.0) 3(6.0)
Vertex 7 has edges to: 5(9.0) 6(5.0) 12(4.0) 2(7.0) 11(3.0) 17(8.0)
Vertex 16 has edges to: 10(3.0) 4(1.0) 16(3.0) 8(9.0)
Vertex 14 has edges to: 0(3.0) 17(5.0) 15(1.0)
Vertex 2 has edges to: 19(7.0) 3(2.0) 7(7.0)
Vertex 9 has edges to: 5(6.0) 8(4.0) 1(1.0)
Vertex 15 has edges to: 15(9.0) 13(3.0) 1(7.0) 14(1.0) 8(9.0) 6(5.0)
Vertex 1 has edges to: 18(7.0) 15(7.0) 19(7.0) 1(8.0) 13(6.0) 9(1.0)
Vertex 10 has edges to: 0(9.0) 16(3.0) 5(7.0)
Vertex 12 has edges to: 3(2.0) 7(4.0) 6(2.0)
Vertex 8 has edges to: 9(4.0) 15(9.0) 16(9.0)
Vertex 0 has edges to: 10(9.0) 14(3.0) 0(3.0)
Vertex 6 has edges to: 4(5.0) 7(5.0) 12(2.0) 15(5.0)
Vertex 18 has edges to: 1(7.0)

The shortest path between 11 and 8 is:
11 -> 7 -> 5 -> 9 -> 8
The shortest path costs: 22.0
The shortest path needs 4 steps.

The cheapest path between 11 and 8 is:
11 -> 7 -> 6 -> 15 -> 8
The cheapest path costs: 22.0
The cheapest path needs 4 steps.
```

5. (For the curious) Doing singing and dancing GUIs for these exercises gets boring, doesn't it? And drawing a graph in 2 dimensions is an extremely complicated task. So instead, note that there are a number of implementations for these algorithms on the Internet. There is a mathematician who wrote so many papers with so many colleagues, that people began bragging about having written a paper together with someone who wrote a paper together with..... whom? Find out who the mathematician was! Can you figure out my number? There is also a popular actor who has acted in many films. What was his name again?

Implementing the algorithms and making them work was really exhausting so we needed something refreshing. This exercise got our curiosity.

We decided to search for that mathematician and found the following:

https://en.wikipedia.org/wiki/Erd%C5%91s_number
"The **Erdős number** describes the "collaborative distance" between mathematician Paul Erdős and another person, as measured by authorship of mathematical papers."

That woke up our interest even more and we decided to research further.

We found this page that had a lot of information about the Erdös number: https://www.oakland.edu/enp/ (**The Erdös Number Project)**

There we also found a way to compute someone's **Erdös** number: https://oakland.edu/enp/compute/

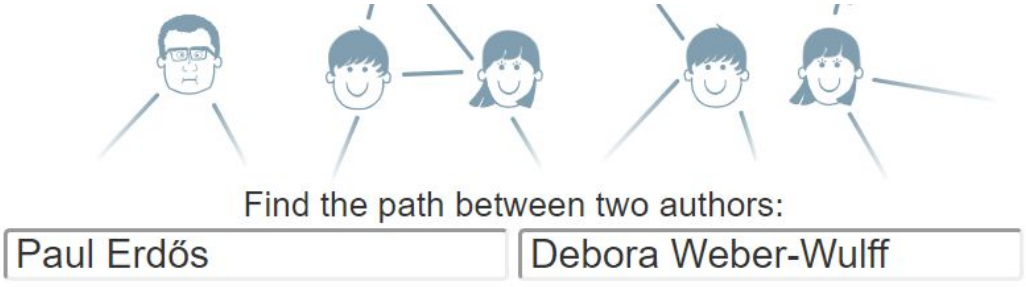
We wanted to try and find out Prof. Weber-Wulff's number so we spent some time clicking and reading through pages. We stumbled upon this enormous file containing coauthors of Paul Erdos:
https://files.oakland.edu/users/grossman/enp/Erdos1.html
We couldn't find the number there so we continued our search.

The following page probably had information about it, but it required some sort of subscription and it didn't let us search. https://mathscinet.ams.org/mathscinet

Finally! We found a working tool here: https://www.csauthors.net/distance

https://www.csauthors.net/distance/paul-erdos/debora-weber-wulff

We found the number we were looking for. (according to that page)



Paul Erdős
co-authored 8 papers with
Ronald L. Graham
co-authored 3 papers with
Hongyu Chen
co-authored 1 paper with
Horst Lichter
co-authored 1 paper with
Debora Weber-Wulff
distance = 4

The popular actor was Kevin Bacon. This took us a while, but we also found the internet page which Prof. Weber-Wulff showed us in class some time ago.

https://oracleofbacon.org/

Reflections:

Pavel

I found this exercise difficult, but after going through multiple resources I got a much better understanding about representing Graphs in Java. I learned about the varargs feature in Java and also how the adjacency list works. It was interesting to see that we can randomize the graph on each run and see the vertices get connected randomly. Getting the shortest and cheapest path methods was tough, but it was nice that we had other groups to collaborate with. After implementing the algorithms, it was also great to read and get to know more about the Erdős number.

Cansu

I think that the exercises are getting harder and harder. I really enjoyed the lectures about this thema, the exercise was also a good idea, but it was to hard for me. I think without the help from Sophia, Dennis, Juri and also Pavel, I would have still more questions about all this. I found a lot of informations online, but they were not useful or to complicated.

Pre-Lab

Pavel:

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

https://www.ida.liu.se/opendsa/Books/TDDD86F19/html/GraphIntro.html
https://runestone.academy/runestone/books/published/pythonds/Graphs/Objectives.html
https://en.wikipedia.org/wiki/Graph_(abstract_data_type)
https://www.geeksforgeeks.org/abstract-data-types/

void Graph ()
void addVertex(Vertex v)
void addEdge(Vertex v1 , Vertex v2)

void removeVertex (Vertex v)
void removeEdge (Vertex v1 , Vertex v2)
Collection getNeighbors (Vertex v)
int getNumberOfVertices()
int getNumberOfEdges()
`adjacent`(*G*, *x*, *y*): tests whether there is an edge from the vertex *x* to the vertex *y*;
`neighbors`(*G*, *x*): lists all vertices *y* such that there is an edge from the vertex *x* to the vertex *y*;
`get_vertex_value`(*G*, *x*): returns the value associated with the vertex *x*;
`set_vertex_value`(*G*, *x*, *v*): sets the value associated with the vertex *x* to *v*.

A **weighted graph** is simply a graph whose edges have weights. The only significant difference is the addition of operations to support edge weights

addEdge(Vertex v1 , double w, Vertex v2)
getEdgeWeight(Vertex v1 , Vertex v2)
setEdgeWeight(Vertex v1 , double newWeight, Vertex v2)

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.

Shortest path:
https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/
https://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/

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?

Implementation with both adjacency matrix and list:
https://www.ida.liu.se/opendsa/Books/TDDD86F19/html/GraphImpl.html

Implementation with an adjacency list:
https://algorithms.tutorialhorizon.com/graph-implementation-adjacency-list-better-set-2/

4. Briefly review generating random numbers.

https://www.geeksforgeeks.org/generating-random-numbers-in-java/

```
import java.util.Random;

public class generateRandom{

    public static void main(String args[])
    {
        // create instance of Random class
        Random rand = new Random();

        // Generate random integers in range 0 to 999
        int rand_int1 = rand.nextInt(1000);
        int rand_int2 = rand.nextInt(1000);
```

Code:

```
public interface WeightedGraph {
//https://stackoverflow.com/questions/3158730/java-3-dots-in-parameters
void addVertex(Vertex...n);
void addEdge(Vertex start, Vertex destination, double weight);
void addEdgeHelper(Vertex start, Vertex destination, double weight);
void printEdges();
boolean hasEdge(Vertex start, Vertex destination);
void resetVisitedVertices();
}
import java.util.LinkedList;

public class Vertex {
public int n;
public String name;
private boolean visited; //keeping track if the Vertex has been visited
LinkedList<EdgeWeighted> edges
public Vertex(int n, String name) {
```

```
this.n = n;
this.name = name;
this.visited = false;
this.edges = new LinkedList<>(); // a vertex has a linked list of edges
}
public LinkedList<Vertex> getAdjVertices() {
LinkedList<Vertex> adjVertices = new LinkedList<Vertex>();
for(EdgeWeighted edge : edges) {
if(n == edge.start.n)
adjVertices.add(edge.end);
if(n == edge.end.n)
adjVertices.add(edge.start);
}
return adjVertices;
}
public boolean isVisited() {
return visited;
}
public void visit() {
visited = true;
}
public void unvisit() {
visited = false;
}
}

public class EdgeWeighted implements Comparable<EdgeWeighted>{

                                                    public Vertex start;
                                                     public Vertex end;
```

```
        public double weight;

        public EdgeWeighted(Vertex start, Vertex end, double weight){

            this.start = start;

            this.end = end;

            this.weight = weight;
        }
    // From here https://stackabuse.com/graphs-in-java-dijkstras-algorithm/
        @Override
        public String toString() {

            return String.format("(%s -> %s, %f)",start.name, end.name, weight);
        }
    // We need the compareTo method to be able to deal with exercise 4.
    // We are comparing "this" edge to another edge and checking which one has
    //
        @Override
        public int compareTo(EdgeWeighted otherEdge) {

            if(this.weight > otherEdge.weight)

                return 1;

            else


                return -1;
        }
}
```

```
public class RandomGraph {
                                                    public static void main(String[] args) {

            MyWeightedGraph myWeightedGraph = new MyWeightedGraph(true);
                                                                int vertices = 20;
                                                                 int edges = 45;

                                                myWeightedGraph.RandomGraph(vertices, edges);

    myWeightedGraph.printEdges();

    // choose two random vertices
                                                  Vertex v = myWeightedGraph.getRandomVertex();
                                                 Vertex v2 = myWeightedGraph.getRandomVertex();
                                        // find the cheapest path between the two random vertices
                                                 myWeightedGraph.getCheapestPath(v, v2);
                                                  myWeightedGraph.resetVisitedVertices();
                                                                System.out.println();
                                         // find the shortest path between the two random vertices
                                                 myWeightedGraph.getShortestPath(v, v2);
                                                  myWeightedGraph.resetVisitedVertices();

                                                                                    }
}

import java.util.Arrays;
import java.util.Collection;
```

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

public class MyWeightedGraph implements WeightedGraph{
                                        // using a Set will not allow duplicated vertices
                                        private Set<Vertex> vertices;
                                    // is the Graph directed (digraph) or undirected?
                                        private boolean directed;
                                        Random r = new Random();

                                                    // Constructor
                                    public MyWeightedGraph(boolean directed) {

                                                this.directed = directed;

                                                vertices = new HashSet<>();
                                                }

                                                    @Override
                                            public void addVertex(Vertex...n) {

                                                vertices.addAll(Arrays.asList(n));
                                                }
                                                @Override
```

```
public void addEdge(Vertex start, Vertex end, double weight) {

vertices.add(start);

vertices.add(end);

// no duplicated edges

addEdgeHelper(start, end, weight);

if(!directed && start != end) {

addEdgeHelper(end, start, weight);

}

}

@Override
public void addEdgeHelper(Vertex start, Vertex 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(Vertex vertice : vertices) {

LinkedList<EdgeWeighted> edges = vertice.edges;



                if(edges.isEmpty()) {

System.out.println("Vertex " + vertice.name + " has no edges.");

                continue;

                }

System.out.print("Vertex " + vertice.name + " has edges to: ");
```

```
for(EdgeWeighted edge : edges) {

System.out.print(edge.end.name + "(" + edge.weight + ") ");

}

System.out.println();

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

for(Vertex vertice : vertices) {

vertice.unvisit();

}
```

```
}

public LinkedList<Vertex> getCheapestPath(Vertex start, Vertex end) {

LinkedList<Vertex> direction = new LinkedList<Vertex>();
// 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<Vertex, Vertex> changedAt = new HashMap<>();
changedAt.put(start, null);

// Keeps track of the shortest path we've found so far for every vertice
HashMap<Vertex, 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 (Vertex 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) {

Vertex 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) {

for(Vertex vertice = end; vertice != null; vertice = changedAt.get(vertice)) {

direction.add(vertice);

if(vertice == start)
```

```
                break;

            }

            Collections.reverse(direction);



            System.out.println("The cheapest path between "
            + start.name + " and " + end.name + " is:");

            String path = "";

            int steps = -1;

            Vertex prefVertice = null;

            double weight = 0.0;

            for(Vertex 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 cheapest path costs: " + weight);

        System.out.println("The cheapest path needs " + steps + " steps.");
        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);
                                            }
                                        }
                                    }
                                }
                    // changed from DijkstraShortestPath to getShortestPath
                                                                //
https://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes-for-un
weighted-graph
                    public LinkedList<Vertex> getShortestPath(Vertex start, Vertex end) {



                                                                //

                                        HashMap<Vertex, Vertex> changedAt = new HashMap<>();



                                        LinkedList<Vertex> direction = new LinkedList<Vertex>();
```

```
Queue q = new LinkedList();

Vertex currentVertice = start;

q.add(currentVertice);

start.visit();

while(!q.isEmpty()) {

currentVertice = (Vertex) q.remove();

if(currentVertice.equals(end)) {

break;

}

else {

for(Vertex vertice : currentVertice.getAdjVertices()) {

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;

    }
```

```
for(Vertex vertice = end; vertice != null; vertice = changedAt.get(vertice)) {

direction.add(vertice);

}

Collections.reverse(direction);



System.out.println("The shortest path between "
+ start.name + " and " + end.name + " is:");

String path = "";

int steps = -1;

Vertex prefVertice = null;

double weight = 0.0;

for(Vertex 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 shortest path costs: " + weight);

    System.out.println("The shortest path needs " + steps + " steps.");

            return direction;
        }
    private Vertex closestReachableUnvisited(HashMap<Vertex, Double>
shortestPathMap) {

        double shortestDistance = Double.POSITIVE_INFINITY;

        Vertex closestReachableVertice = null;

        for(Vertex 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 RandomGraph(int vertices, int edges) {

        //reset the current

        this.vertices = new HashSet<>();
```

```
for(int i = 0; i<vertices;i++) {

this.vertices.add(new Vertex(i, Integer.toString(i)));

}

for(int i = 0; i<edges;i++) {

Vertex vert1 = getRandomVertex();

Vertex vert2 = getRandomVertex();


addEdge(vert1, vert2, r.nextInt(9)+1);


}
}

public Vertex getRandomVertex() {

int index = 0;

int size = vertices.size();

int counter = r.nextInt(size);


for(Vertex v : vertices) {
```

```
                if(index == counter)

                    return v;

                    index++;

                }

            return null;
            }

    public EdgeWeighted getEdge(Vertex V1, Vertex V2) {

        boolean isChild = false;

        for(Vertex vertice : V1.getAdjVertices()) {

            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;
        }
}
```