Markus Arendt (538148)
Joshua Widmann (537813)

Laboratory 10: Getting from A to B

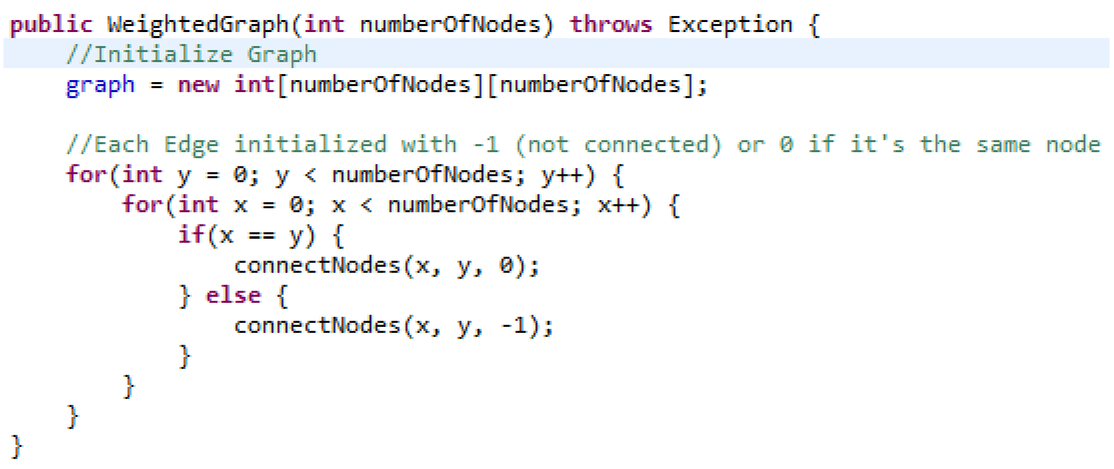
(http://people.f4.htw-berlin.de/~weberwu/info2/labs/ExerA.shtml)

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 decided to use a adjacency matrix to represent our WeightedGraph. This matrix constists of an two-dimensional Integer array.

By creating a new graph you have to pass the number of nodes/vertices into the constructor to initialize the array.

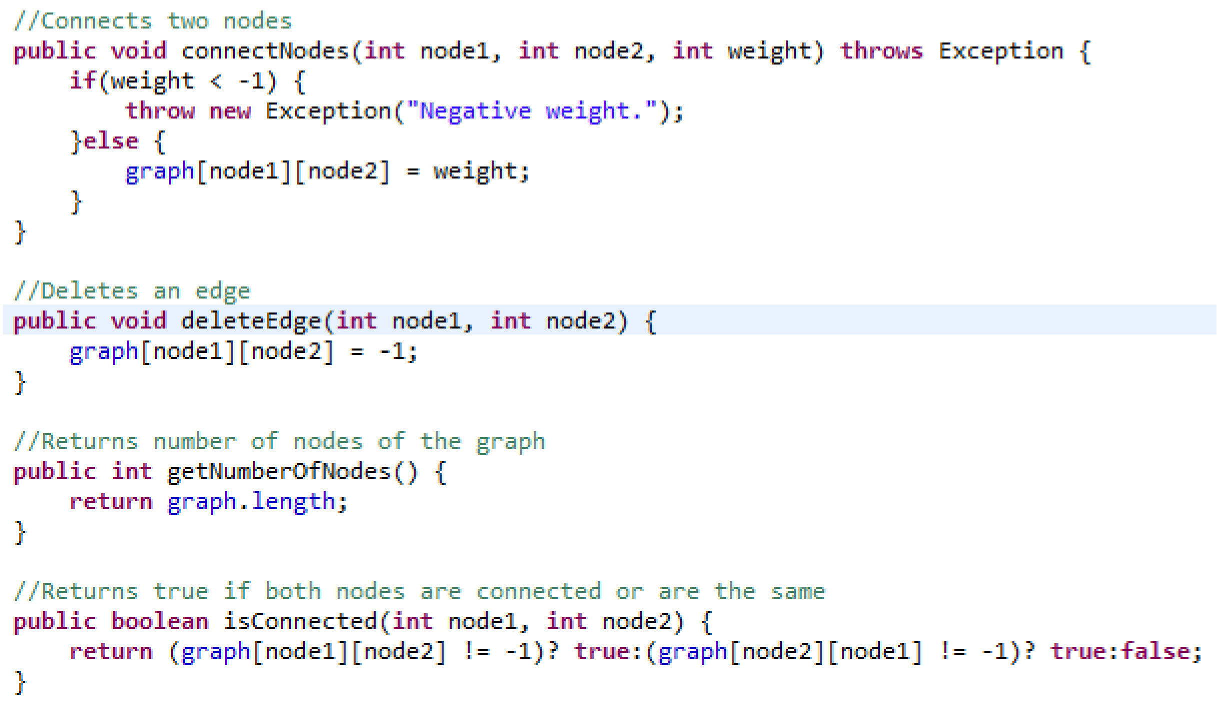
The array then gets initialized with "-1" which means there's no connection. If it's the same node its getting initialized with 0 weight.



```
public WeightedGraph(int numberOfNodes) throws Exception {
    //Initialize Graph
    graph = new int[numberOfNodes][numberOfNodes];

    //Each Edge initialized with -1 (not connected) or 0 if it's the same node
    for(int y = 0; y < numberOfNodes; y++) {
        for(int x = 0; x < numberOfNodes; x++) {
            if(x == y) {
                connectNodes(x, y, 0);
            } else {
                connectNodes(x, y, -1);
            }
        }
    }
}
```

We added additional methods to the data type to be able to add nodes/vertices and check different things.



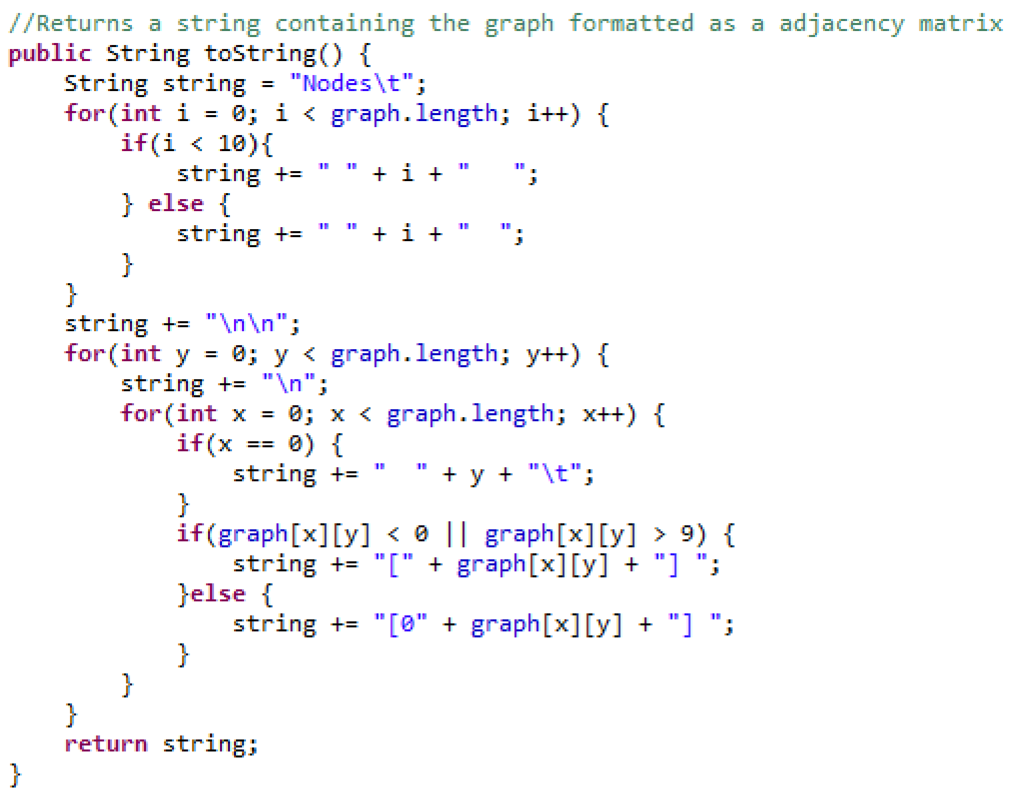
```
//Connects two nodes
public void connectNodes(int node1, int node2, int weight) throws Exception {
    if(weight < -1) {
        throw new Exception("Negative weight.");
    }else {
        graph[node1][node2] = weight;
    }
}

//Deletes an edge
public void deleteEdge(int node1, int node2) {
    graph[node1][node2] = -1;
}

//Returns number of nodes of the graph
public int getNumberOfNodes() {
    return graph.length;
}

//Returns true if both nodes are connected or are the same
public boolean isConnected(int node1, int node2) {
    return (graph[node1][node2] != -1)? true:(graph[node2][node1] != -1)? true:false;
}
```

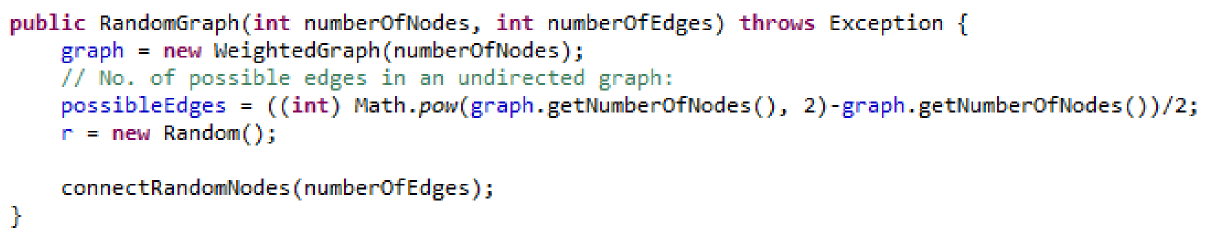
Then we also added a toString() method to be able to print the graph tot he console for testing reasons.



```
//Returns a string containing the graph formatted as a adjacency matrix
public String toString() {
    String string = "Nodes\t";
    for(int i = 0; i < graph.length; i++) {
        if(i < 10){
            string += " " + i + "   ";
        } else {
            string += " " + i + "  ";
        }
    }
    string += "\n\n";
    for(int y = 0; y < graph.length; y++) {
        string += "\n";
        for(int x = 0; x < graph.length; x++) {
            if(x == 0) {
                string += "  " + y + "\t";
            }
            if(graph[x][y] < 0 || graph[x][y] > 9) {
                string += "[" + graph[x][y] + "] ";
            }else {
                string += "[0" + graph[x][y] + "] ";
            }
        }
    }
    return string;
}
```

2. **While one partner is doing this, the other one should 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. Use the WeightedGraph your partner is constructing.**

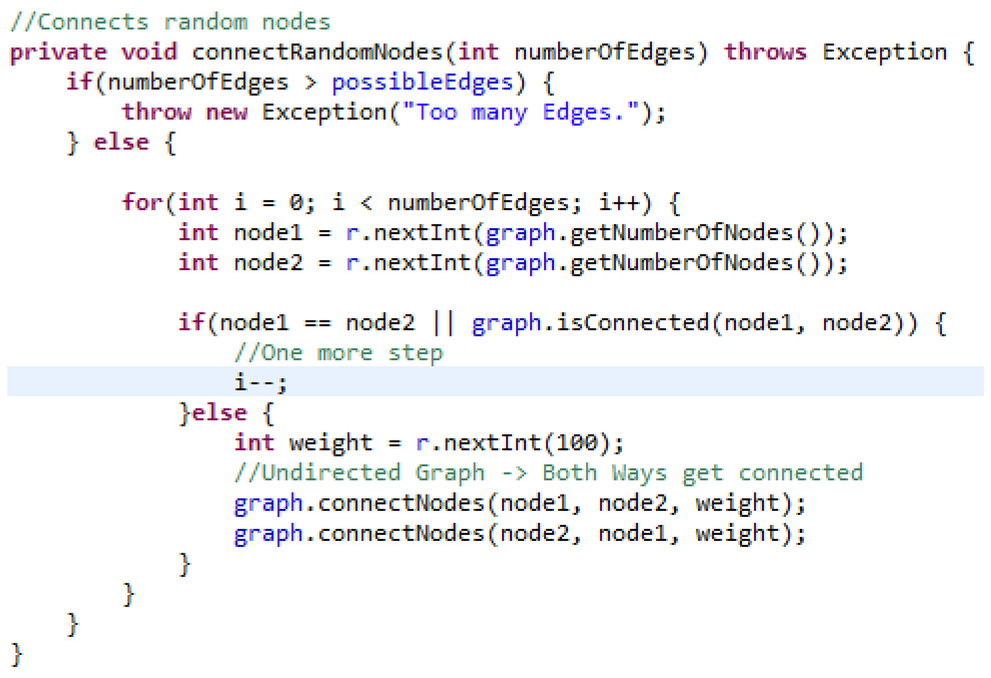
To create a random graph, you have to pass the wished number of nodes and edges to the constructor.



```java
public RandomGraph(int numberOfNodes, int numberOfEdges) throws Exception {
    graph = new WeightedGraph(numberOfNodes);
    // No. of possible edges in an undirected graph:
    possibleEdges = ((int) Math.pow(graph.getNumberOfNodes(), 2)-graph.getNumberOfNodes())/2;
    r = new Random();

    connectRandomNodes(numberOfEdges);
}
```

The constructor then uses the WeightedGraph data type to create a graph and calls the “connectRandomNodes()” method to connect random nodes.



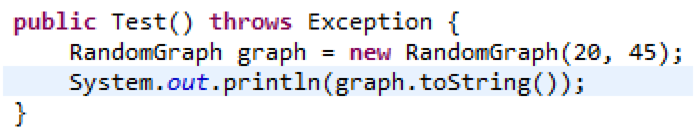
```java
//Connects random nodes
private void connectRandomNodes(int numberOfEdges) throws Exception {
    if(numberOfEdges > possibleEdges) {
        throw new Exception("Too many Edges.");
    } else {

        for(int i = 0; i < numberOfEdges; i++) {
            int node1 = r.nextInt(graph.getNumberOfNodes());
            int node2 = r.nextInt(graph.getNumberOfNodes());

            if(node1 == node2 || graph.isConnected(node1, node2)) {
                //One more step
                i--;
            }else {
                int weight = r.nextInt(100);
                //Undirected Graph -> Both Ways get connected
                graph.connectNodes(node1, node2, weight);
                graph.connectNodes(node2, node1, weight);
            }
        }
    }
}
```

If the user input on the number of edges was wrong (too high) we throw an exception. Otherwise we choose random numbers as nodes and connect both with the method “connectNodes()”.

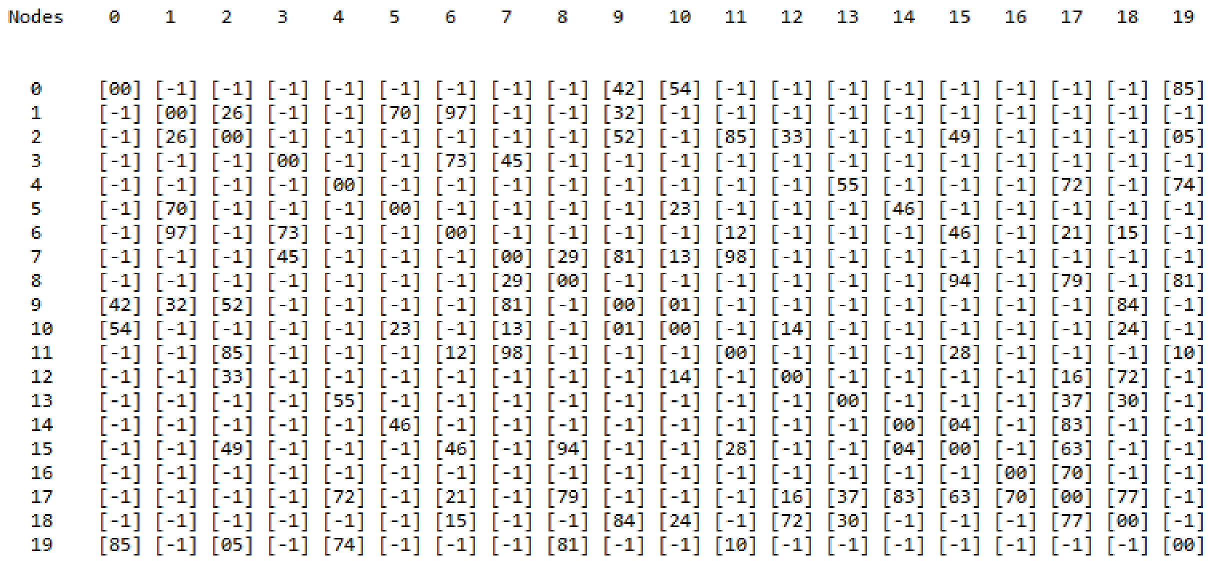
In our test method we now check out if this already works.



```
public Test() throws Exception {
    RandomGraph graph = new RandomGraph(20, 45);
    System.out.println(graph.toString());
}
```

We create a random graph with 20 nodes and 45 edges and use the toString() method to print it to the console.

The output looks like that:



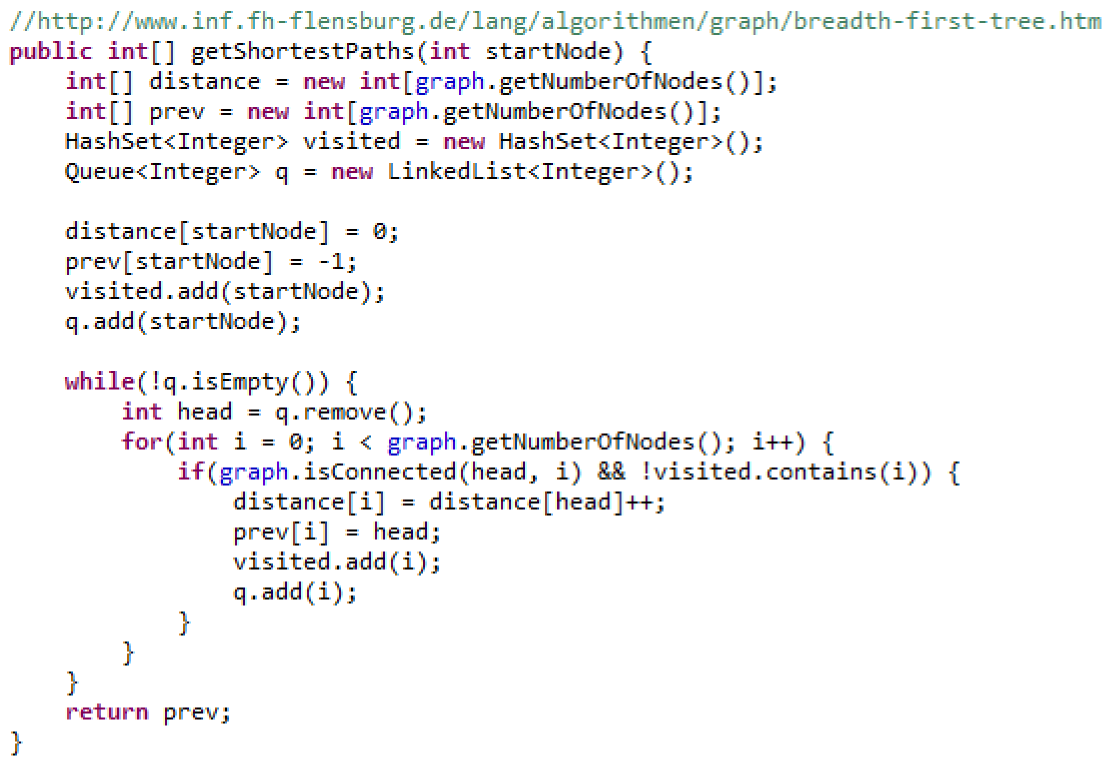
```
Nodes    0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19


  0     [00] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [42] [54] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [85]
  1     [-1] [00] [26] [-1] [-1] [70] [97] [-1] [-1] [32] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1]
  2     [-1] [26] [00] [-1] [-1] [-1] [-1] [-1] [-1] [52] [-1] [85] [33] [-1] [-1] [49] [-1] [-1] [-1] [05]
  3     [-1] [-1] [-1] [00] [-1] [-1] [73] [45] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1]
  4     [-1] [-1] [-1] [-1] [00] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [55] [-1] [-1] [-1] [72] [-1] [74]
  5     [-1] [70] [-1] [-1] [-1] [00] [-1] [-1] [-1] [-1] [23] [-1] [-1] [-1] [46] [-1] [-1] [-1] [-1] [-1]
  6     [-1] [97] [-1] [73] [-1] [-1] [00] [-1] [-1] [-1] [-1] [12] [-1] [-1] [-1] [46] [-1] [21] [15] [-1]
  7     [-1] [-1] [-1] [45] [-1] [-1] [-1] [00] [29] [81] [13] [98] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1]
  8     [-1] [-1] [-1] [-1] [-1] [-1] [-1] [29] [00] [-1] [-1] [-1] [-1] [-1] [-1] [94] [-1] [79] [-1] [81]
  9     [42] [32] [52] [-1] [-1] [-1] [-1] [81] [-1] [00] [01] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [84] [-1]
  10    [54] [-1] [-1] [-1] [-1] [23] [-1] [13] [-1] [01] [00] [-1] [14] [-1] [-1] [-1] [-1] [-1] [24] [-1]
  11    [-1] [-1] [85] [-1] [-1] [-1] [12] [98] [-1] [-1] [-1] [00] [-1] [-1] [-1] [28] [-1] [-1] [-1] [10]
  12    [-1] [-1] [33] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [14] [-1] [00] [-1] [-1] [-1] [-1] [16] [72] [-1]
  13    [-1] [-1] [-1] [-1] [55] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [00] [-1] [-1] [-1] [37] [30] [-1]
  14    [-1] [-1] [-1] [-1] [-1] [46] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [00] [04] [-1] [83] [-1] [-1]
  15    [-1] [-1] [49] [-1] [-1] [-1] [46] [-1] [94] [-1] [-1] [28] [-1] [-1] [04] [00] [-1] [63] [-1] [-1]
  16    [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [00] [70] [-1] [-1]
  17    [-1] [-1] [-1] [-1] [72] [-1] [21] [-1] [79] [-1] [-1] [-1] [16] [37] [83] [63] [70] [00] [77] [-1]
  18    [-1] [-1] [-1] [-1] [-1] [-1] [15] [-1] [-1] [84] [24] [-1] [72] [30] [-1] [-1] [-1] [77] [00] [-1]
  19    [85] [-1] [05] [-1] [74] [-1] [-1] [-1] [81] [-1] [-1] [10] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [00]
```

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?

That's the part we couldn't get running. We searched the internet for algorithms and pseudocode but nothing worked as expected and we were not able to implement it as we wanted to.

The only algorithm we could get working was the breadth-first search from [redacted].



```
//http://www.inf.fh-flensburg.de/lang/algorithmen/graph/breadth-first-tree.htm
public int[] getShortestPaths(int startNode) {
    int[] distance = new int[graph.getNumberOfNodes()];
    int[] prev = new int[graph.getNumberOfNodes()];
    HashSet<Integer> visited = new HashSet<Integer>();
    Queue<Integer> q = new LinkedList<Integer>();

    distance[startNode] = 0;
    prev[startNode] = -1;
    visited.add(startNode);
    q.add(startNode);

    while(!q.isEmpty()) {
        int head = q.remove();
        for(int i = 0; i < graph.getNumberOfNodes(); i++) {
            if(graph.isConnected(head, i) && !visited.contains(i)) {
                distance[i] = distance[head]++;
                prev[i] = head;
                visited.add(i);
                q.add(i);
            }
        }
    }
    return prev;
}
```

But this algorithm is just able to give the shortest amount of steps to each node in the graph from a specific node.

4. Meanwhile, your partner writes a method that takes a graph, picks two vertices at random, and finds the cheapest path between the two.

We have nothing here, since this is even more complicated than the one above.

Source code:

WeightedGraph:

```
public class WeightedGraph {
	private int[][] graph;

	public WeightedGraph(int numberOfNodes) throws Exception {
		//Initialize Graph
		graph = new int[numberOfNodes][numberOfNodes];

		//Each Edge initialized with -1 (not connected) or 0 if it's the same
node
		for(int y = 0; y < numberOfNodes; y++) {
			for(int x = 0; x < numberOfNodes; x++) {
				if(x == y) {
					connectNodes(x, y, 0);
				} else {
					connectNodes(x, y, -1);
				}
			}
		}
	}

	//Connects two nodes
	public void connectNodes(int node1, int node2, int weight) throws Exception
{
		if(weight < -1) {
			throw new Exception("Negative weight.");
		}else {
			graph[node1][node2] = weight;
		}
	}

	//Deletes an edge
	public void deleteEdge(int node1, int node2) {
		graph[node1][node2] = -1;
	}

	//Returns number of nodes of the graph
	public int getNumberOfNodes() {
		return graph.length;
	}

	//Returns true if both nodes are connected or are the same
	public boolean isConnected(int node1, int node2) {
		return (graph[node1][node2] != -1)? true:(graph[node2][node1] != -1)?
true:false;
	}

	//Returns a string containing the graph formatted as a adjacency matrix
	public String toString() {
		String string = "Nodes\t";
		for(int i = 0; i < graph.length; i++) {
			if(i < 10){
				string += " " + i + "   ";
			} else {
				string += " " + i + "  ";
			}
		}
		string += "\n\n";
		for(int y = 0; y < graph.length; y++) {
			string += "\n";
			for(int x = 0; x < graph.length; x++) {
				if(x == 0) {
					string += "  " + y + "\t";
				}
				if(graph[x][y] < 0 || graph[x][y] > 9) {
					string += "[" + graph[x][y] + "] ";
				}else {
					string += "[0" + graph[x][y] + "] ";
				}
			}
		}
		return string;
	}
}
```

RandomGraph:

```
public class RandomGraph {
	private WeightedGraph graph;
	private int possibleEdges;
	private Random r;

	public RandomGraph(int numberOfNodes, int numberOfEdges) throws Exception {
		graph = new WeightedGraph(numberOfNodes);
		// No. of possible edges in an undirected graph:
		possibleEdges = ((int) Math.pow(graph.getNumberOfNodes(), 2)-
graph.getNumberOfNodes())/2;
		r = new Random();

		connectRandomNodes(numberOfEdges);
	}

	//Connects random nodes
	private void connectRandomNodes(int numberOfEdges) throws Exception {

		if(numberOfEdges > possibleEdges) {
			throw new Exception("Too many Edges.");
		} else {

			for(int i = 0; i < numberOfEdges; i++) {
				int node1 = r.nextInt(graph.getNumberOfNodes());
				int node2 = r.nextInt(graph.getNumberOfNodes());

				if(node1 == node2 || graph.isConnected(node1, node2)) {
					//One more step
					i--;
```

```
                        }else {
                              int weight = r.nextInt(100);
                              //Undirected Graph -> Both Ways get connected
                              graph.connectNodes(node1, node2, weight);
                              graph.connectNodes(node2, node1, weight);
                        }
                  }
            }
      }

      //Calls the toString() method of the graph
      public String toString() {
            return graph.toString();
      }

      //http://www.inf.fh-flensburg.de/lang/algorithmen/graph/breadth-first-
tree.htm
      public int[] getShortestPaths(int startNode) {
            int[] distance = new int[graph.getNumberOfNodes()];
            int[] prev = new int[graph.getNumberOfNodes()];
            HashSet<Integer> visited = new HashSet<Integer>();
            Queue<Integer> q = new LinkedList<Integer>();

            distance[startNode] = 0;
            prev[startNode] = -1;
            visited.add(startNode);
            q.add(startNode);

            while(!q.isEmpty()) {
                  int head = q.remove();
                  for(int i = 0; i < graph.getNumberOfNodes(); i++) {
                        if(graph.isConnected(head, i) && !visited.contains(i)) {
                              distance[i] = distance[head]++;
                              prev[i] = head;
                              visited.add(i);
                              q.add(i);
                        }
                  }
            }
            return prev;
      }
}
```

Test:

```java
public class Test {

	public static void main(String[] args) throws Exception {
		new Test();
	}

	public Test() throws Exception {
		RandomGraph graph = new RandomGraph(20, 45);
		System.out.println(graph.toString());

		int[] path = graph.getShortestPaths(0);

		for(int i = 0; i < path.length; i++) {
			System.out.print(i+":");
			System.out.println("["+path[i]+"]");
		}
	}
}
```