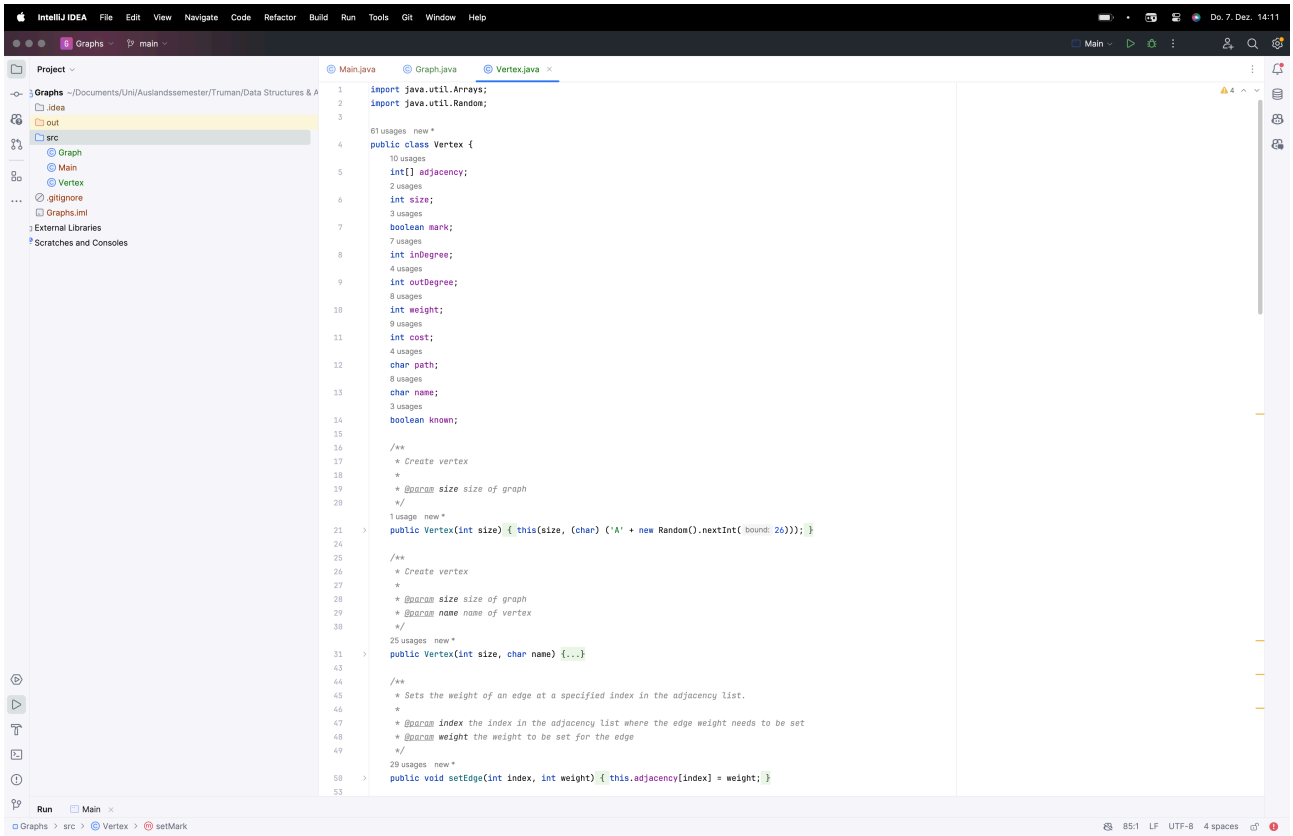
Bonus Assignment: Working with Graph

Thursday 7. December 2023

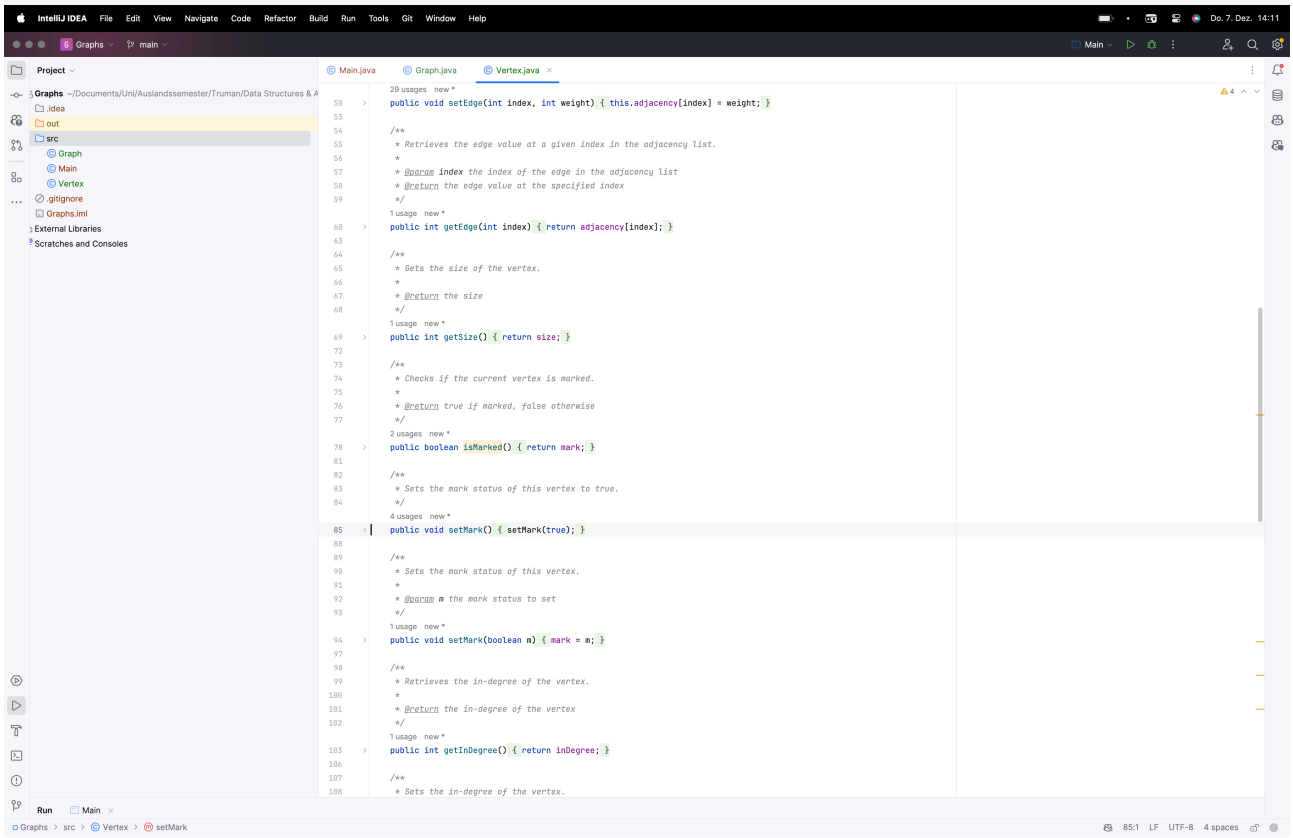
Phillip Lechenauer

Deliverable 1

I implemented the the vertex class as described in the slides.

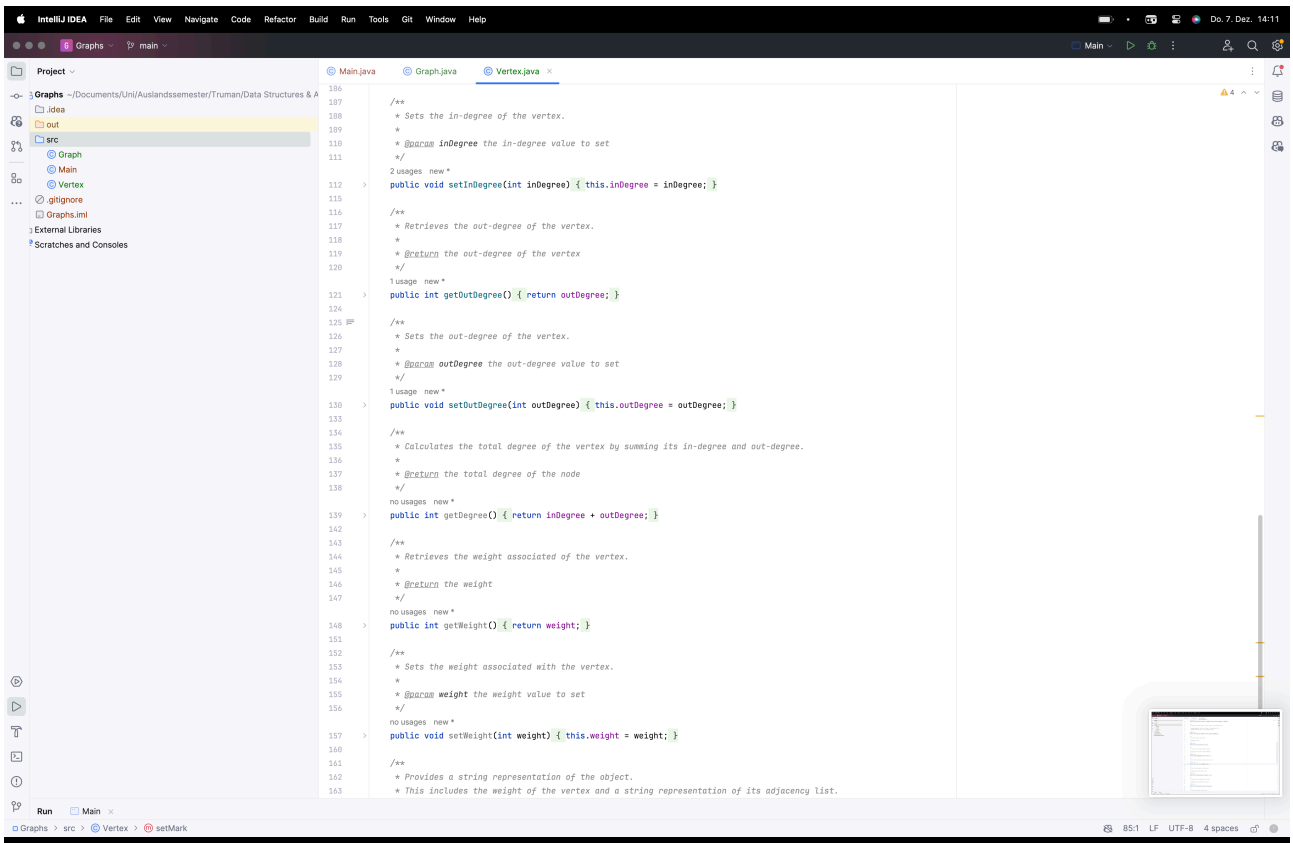


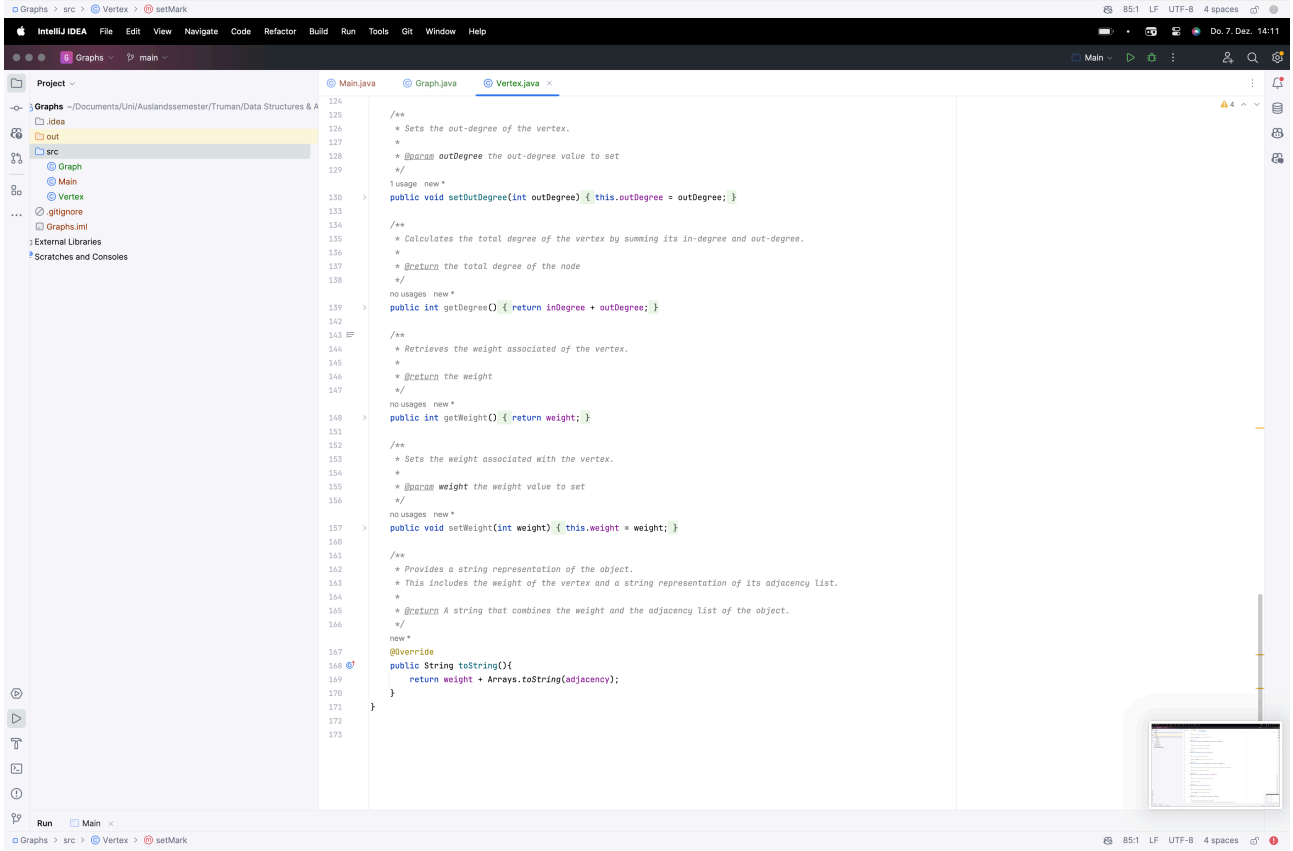
```java
import java.util.Arrays;
import java.util.Random;

61 usages  new *
public class Vertex {
    10 usages
    int[] adjacency;
    2 usages
    int size;
    3 usages
    boolean mark;
    7 usages
    int inDegree;
    4 usages
    int outDegree;
    8 usages
    int weight;
    9 usages
    int cost;
    4 usages
    char path;
    8 usages
    char name;
    3 usages
    boolean known;

    /**
     * Create vertex
     *
     * @param size size of graph
     */
    1 usage  new *
    public Vertex(int size) { this(size, (char) ('A' + new Random().nextInt( bound: 26))); }

    /**
     * Create vertex
     *
     * @param size size of graph
     * @param name name of vertex
     */
    25 usages  new *
    public Vertex(int size, char name) {...}

    /**
     * Sets the weight of an edge at a specified index in the adjacency list.
     *
     * @param index the index in the adjacency list where the edge weight needs to be set
     * @param weight the weight to be set for the edge
     */
    29 usages  new *
    public void setEdge(int index, int weight) { this.adjacency[index] = weight; }
```



```java
    29 usages  new *
    public void setEdge(int index, int weight) { this.adjacency[index] = weight; }

    /**
     * Retrieves the edge value at a given index in the adjacency list.
     *
     * @param index the index of the edge in the adjacency list
     * @return the edge value at the specified index
     */
    1 usage  new *
    public int getEdge(int index) { return adjacency[index]; }

    /**
     * Gets the size of the vertex.
     *
     * @return the size
     */
    1 usage  new *
    public int getSize() { return size; }

    /**
     * Checks if the current vertex is marked.
     *
     * @return true if marked, false otherwise
     */
    2 usages  new *
    public boolean isMarked() { return mark; }

    /**
     * Sets the mark status of this vertex to true.
     */
    4 usages  new *
    public void setMark() { setMark(true); }

    /**
     * Sets the mark status of this vertex.
     *
     * @param m the mark status to set
     */
    1 usage  new *
    public void setMark(boolean m) { mark = m; }

    /**
     * Retrieves the in-degree of the vertex.
     *
     * @return the in-degree of the vertex
     */
    1 usage  new *
    public int getInDegree() { return inDegree; }

    /**
     * Sets the in-degree of the vertex.
```

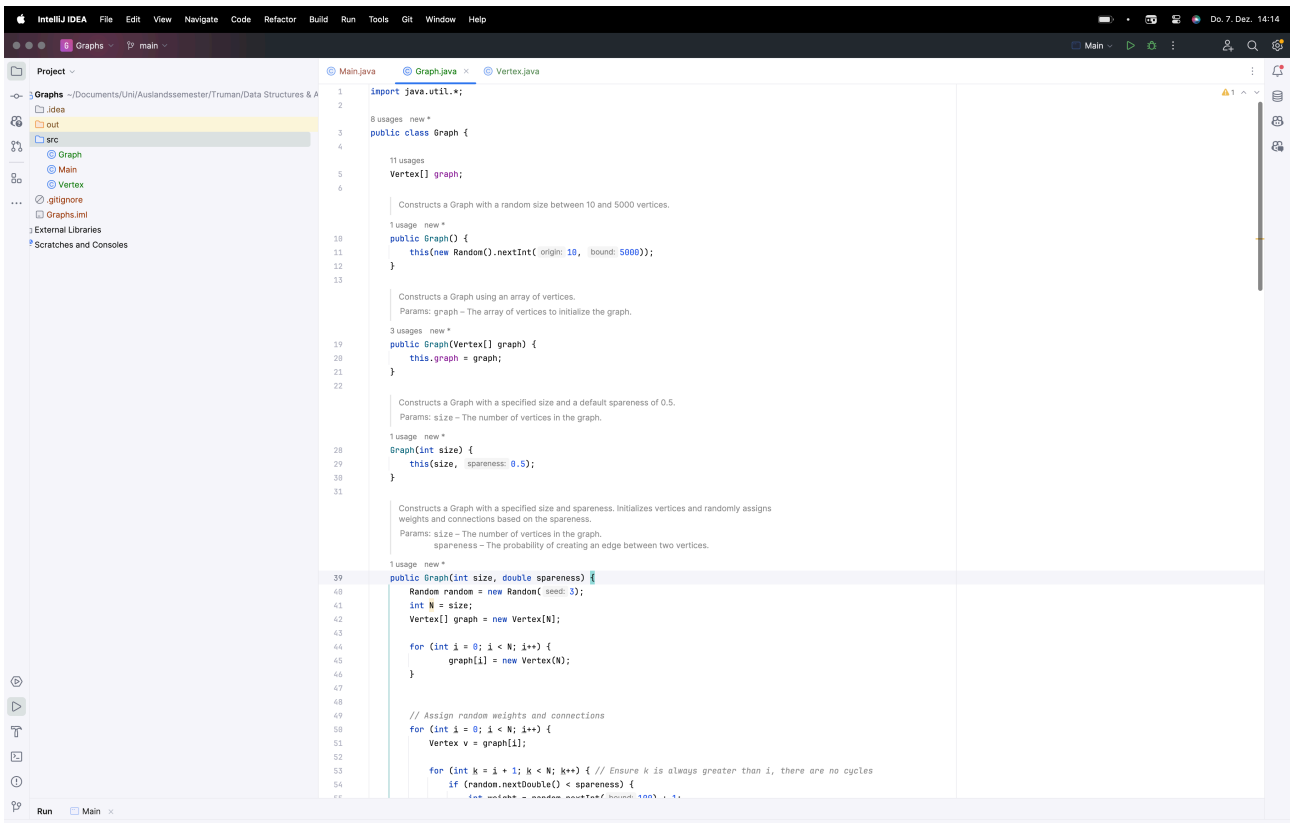
Phillip Lechenauer

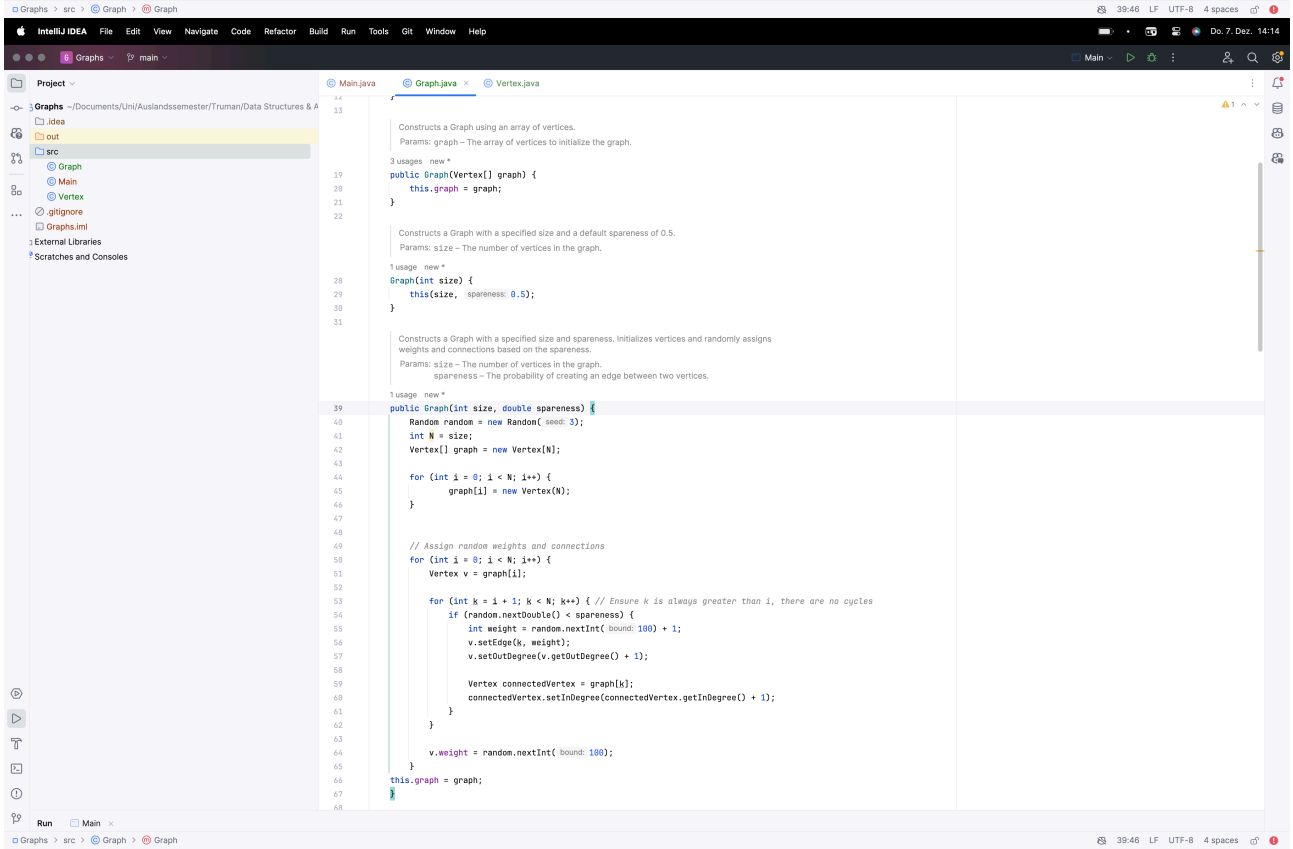




I then created an Graph class which sorted my graph and all the sorting algorithms.
It has different constructors. Some I needed for testing, others give the possibility to create own graphs and not imitate everything randomly.

Phillip Lechenauer

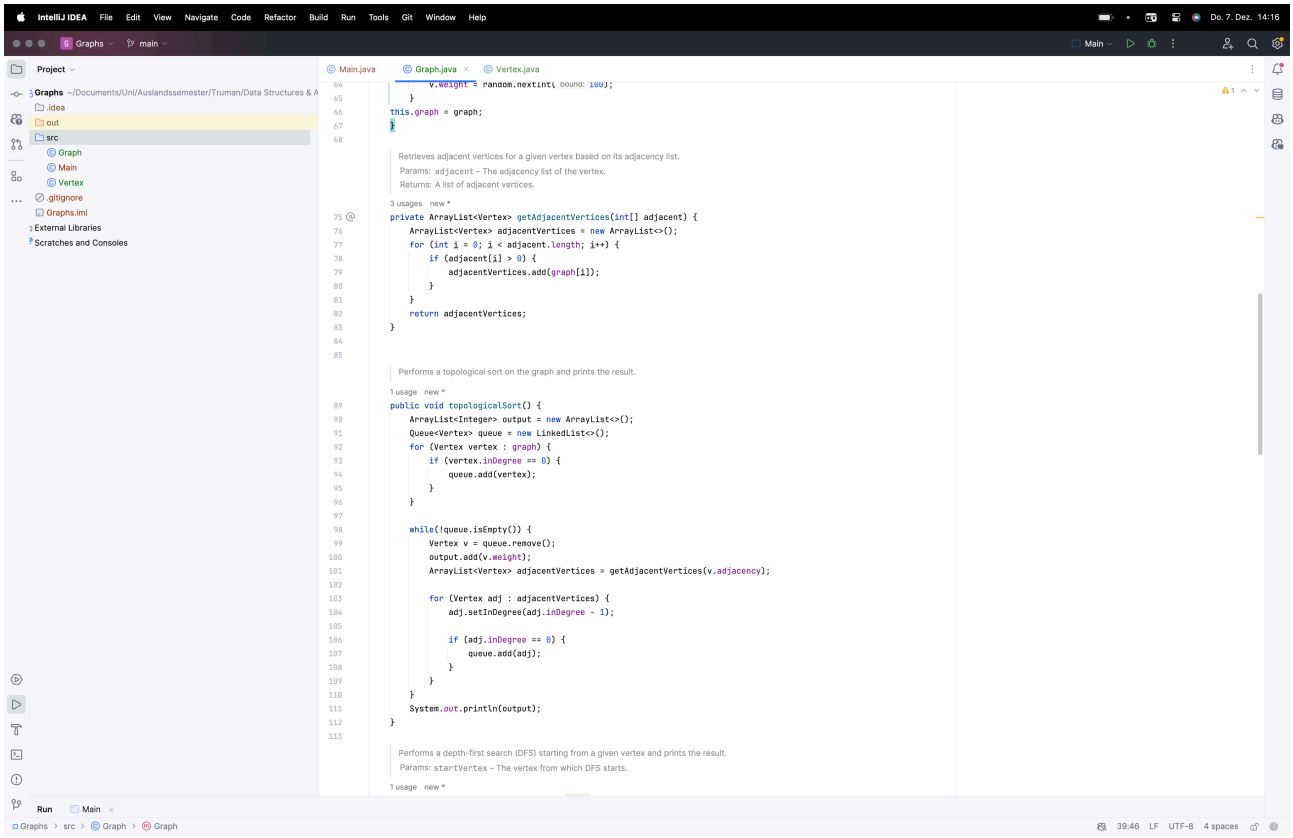




Phillip Lechenauer

Deliverable 2

I implemented the sort algorithm based on the pseudo code in the slides



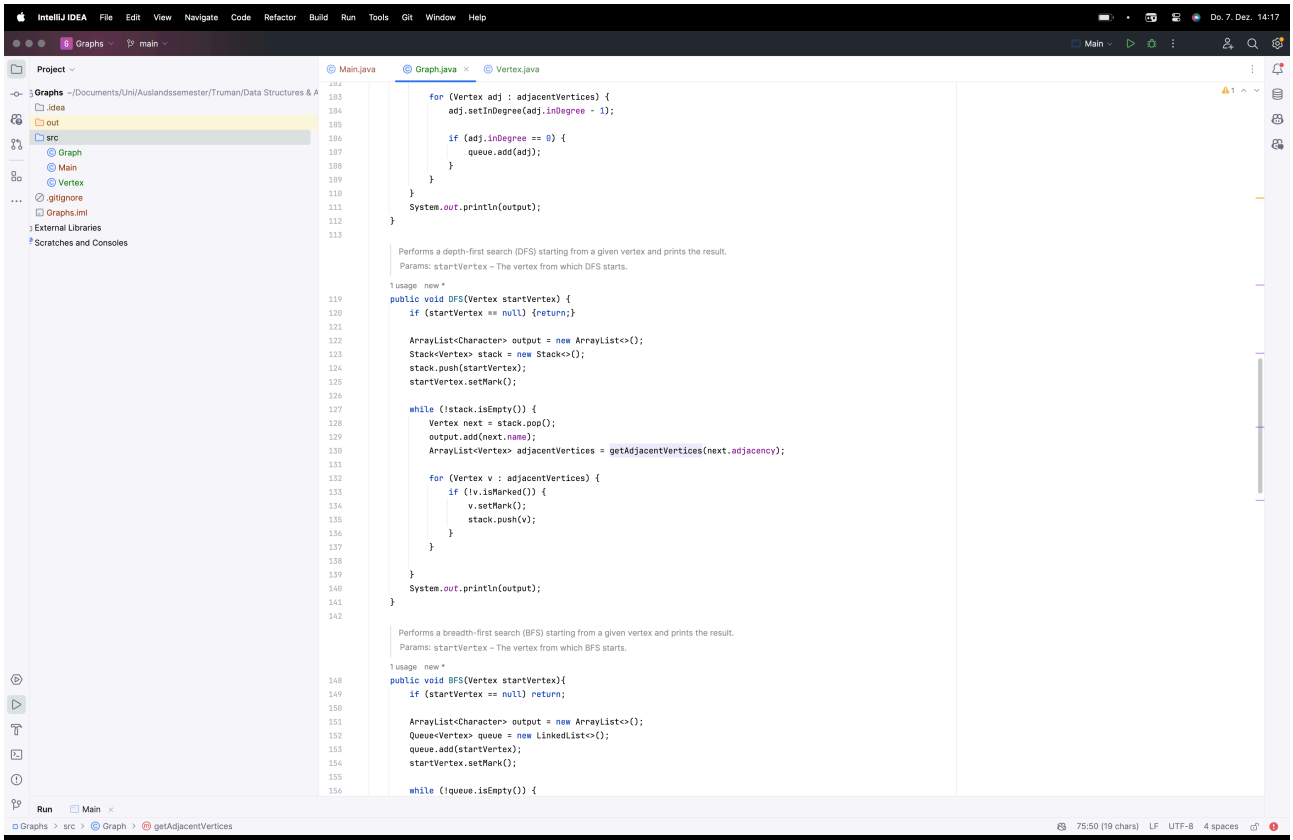
```
        v.weight = random.nextInt( bound: 100);
    }
this.graph = graph;
}

Retrieves adjacent vertices for a given vertex based on its adjacency list.
Params: adjacent – The adjacency list of the vertex.
Returns: A list of adjacent vertices.
3 usages  new *
private ArrayList<Vertex> getAdjacentVertices(int[] adjacent) {
    ArrayList<Vertex> adjacentVertices = new ArrayList<>();
    for (int i = 0; i < adjacent.length; i++) {
        if (adjacent[i] > 0) {
            adjacentVertices.add(graph[i]);
        }
    }
    return adjacentVertices;
}


Performs a topological sort on the graph and prints the result.
1 usage  new *
public void topologicalSort() {
    ArrayList<Integer> output = new ArrayList<>();
    Queue<Vertex> queue = new LinkedList<>();
    for (Vertex vertex : graph) {
        if (vertex.inDegree == 0) {
            queue.add(vertex);
        }
    }

    while(!queue.isEmpty()) {
        Vertex v = queue.remove();
        output.add(v.weight);
        ArrayList<Vertex> adjacentVertices = getAdjacentVertices(v.adjacency);

        for (Vertex adj : adjacentVertices) {
            adj.setInDegree(adj.inDegree - 1);

            if (adj.inDegree == 0) {
                queue.add(adj);
            }
        }
    }
    System.out.println(output);
}

Performs a depth-first search (DFS) starting from a given vertex and prints the result.
Params: startVertex – The vertex from which DFS starts.
1 usage  new *
```

I used the function getAdjacentVertices to get all adjacent vertices.

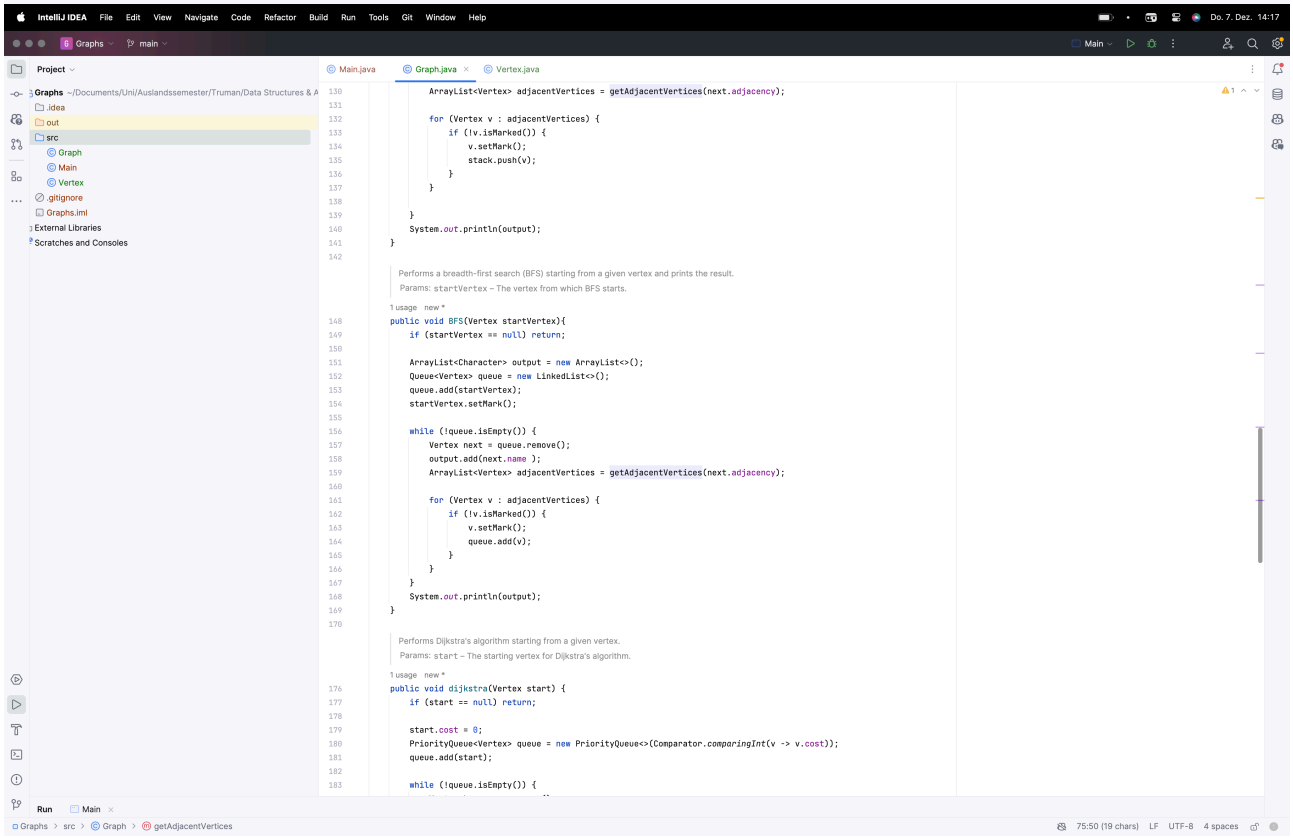
Phillip Lechenauer

Deliverable 3:

These algorithms are also based on the pseudo code in the slides. They are very similar, the main difference is the data structure it uses to store the data for the next iteration.



```
        for (Vertex adj : adjacentVertices) {
            adj.setInDegree(adj.inDegree - 1);

            if (adj.inDegree == 0) {
                queue.add(adj);
            }
        }
    }
    System.out.println(output);
}

    Performs a depth-first search (DFS) starting from a given vertex and prints the result.
    Params: startVertex – The vertex from which DFS starts.
1 usage  new *
public void DFS(Vertex startVertex) {
    if (startVertex == null) {return;}

    ArrayList<Character> output = new ArrayList<>();
    Stack<Vertex> stack = new Stack<>();
    stack.push(startVertex);
    startVertex.setMark();

    while (!stack.isEmpty()) {
        Vertex next = stack.pop();
        output.add(next.name);
        ArrayList<Vertex> adjacentVertices = getAdjacentVertices(next.adjacency);

        for (Vertex v : adjacentVertices) {
            if (!v.isMarked()) {
                v.setMark();
                stack.push(v);
            }
        }

    }
    System.out.println(output);
}

    Performs a breadth-first search (BFS) starting from a given vertex and prints the result.
    Params: startVertex – The vertex from which BFS starts.
1 usage  new *
public void BFS(Vertex startVertex){
    if (startVertex == null) return;

    ArrayList<Character> output = new ArrayList<>();
    Queue<Vertex> queue = new LinkedList<>();
    queue.add(startVertex);
    startVertex.setMark();

    while (!queue.isEmpty()) {
```

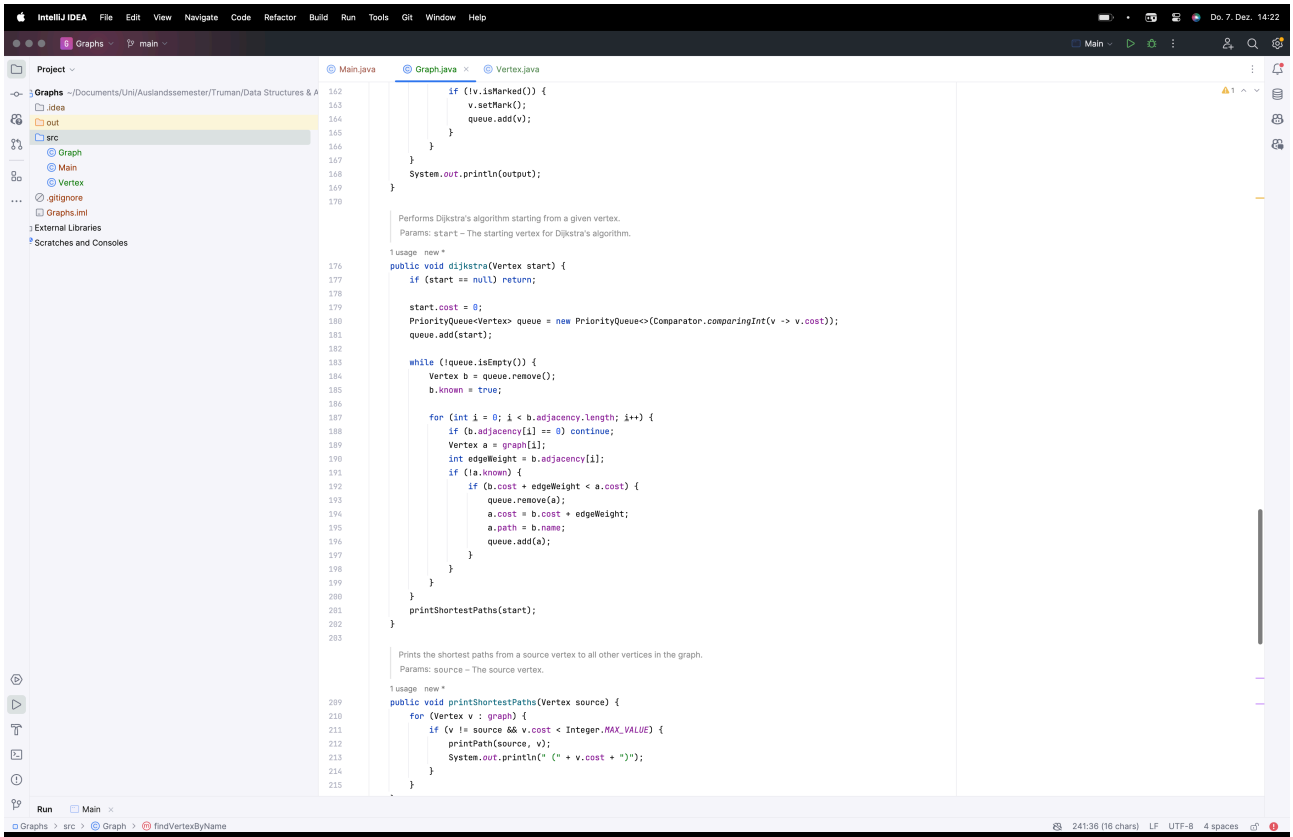


```
        ArrayList<Vertex> adjacentVertices = getAdjacentVertices(next.adjacency);

        for (Vertex v : adjacentVertices) {
            if (!v.isMarked()) {
                v.setMark();
                stack.push(v);
            }
        }

    }
    System.out.println(output);
}

    Performs a breadth-first search (BFS) starting from a given vertex and prints the result.
    Params: startVertex – The vertex from which BFS starts.
1 usage  new *
public void BFS(Vertex startVertex){
    if (startVertex == null) return;

    ArrayList<Character> output = new ArrayList<>();
    Queue<Vertex> queue = new LinkedList<>();
    queue.add(startVertex);
    startVertex.setMark();

    while (!queue.isEmpty()) {
        Vertex next = queue.remove();
        output.add(next.name );
        ArrayList<Vertex> adjacentVertices = getAdjacentVertices(next.adjacency);

        for (Vertex v : adjacentVertices) {
            if (!v.isMarked()) {
                v.setMark();
                queue.add(v);
            }
        }
    }
    System.out.println(output);
}

    Performs Dijkstra's algorithm starting from a given vertex.
    Params: start – The starting vertex for Dijkstra's algorithm.
1 usage  new *
public void dijkstra(Vertex start) {
    if (start == null) return;

    start.cost = 0;
    PriorityQueue<Vertex> queue = new PriorityQueue<>(Comparator.comparingInt(v -> v.cost));
    queue.add(start);

    while (!queue.isEmpty()) {
```

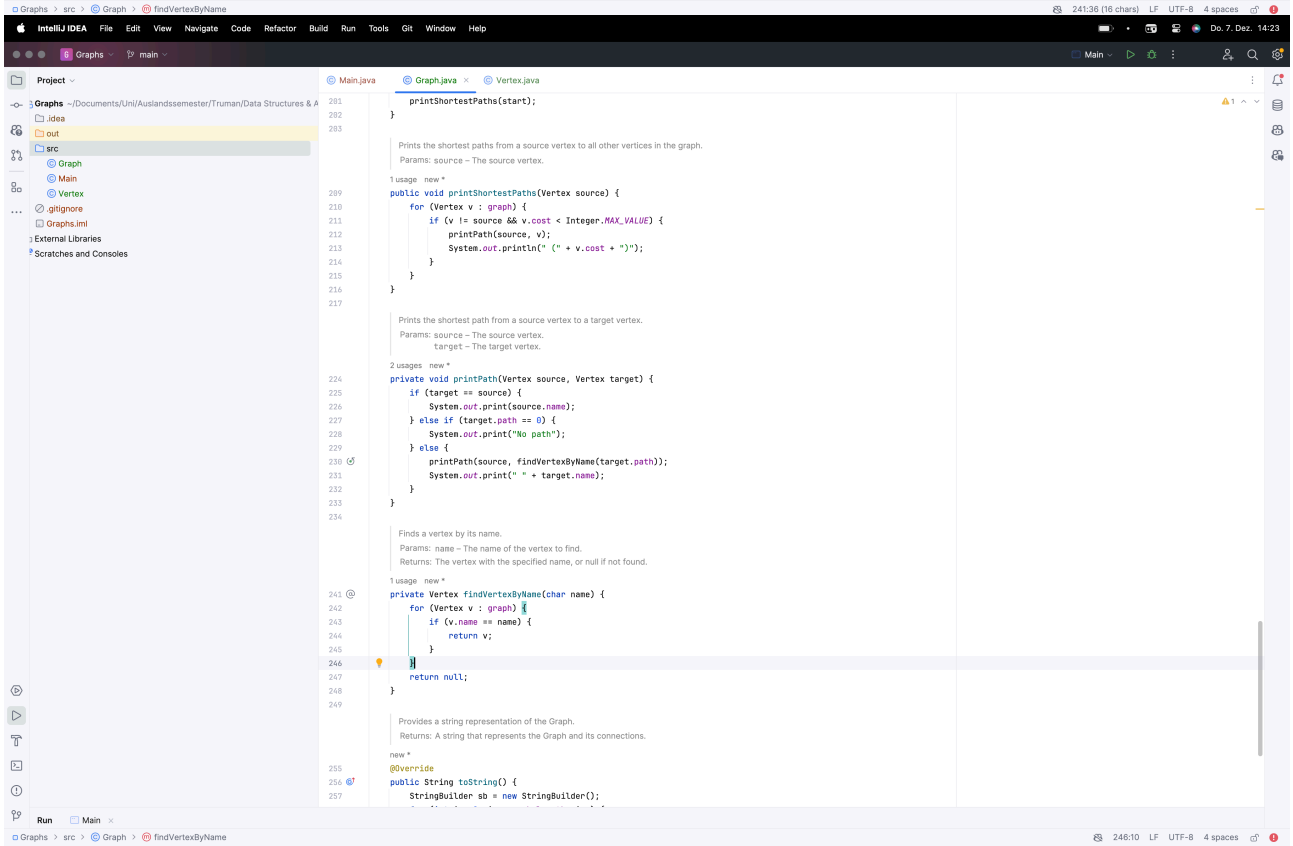
Phillip Lechenauer

Deliverable 4

I implemented the algorithm based on the Pseudo code. Then I implemented helper functions called printShortestPaths, printPath and findVertexByName which I call after sorting to print out all the shortest paths. The printing statements run into a bit of a problem if there are vertices with the same name. Because I only use uppercase letters (36), there can easily occur an duplication of names. I didn't fix it, because in the example it works, and I think it is safe to assume, that the names will be different.



```
                if (!v.isMarked()) {
                    v.setMark();
                    queue.add(v);
                }
            }
        }
        System.out.println(output);
    }

    Performs Dijkstra's algorithm starting from a given vertex.
    Params: start – The starting vertex for Dijkstra's algorithm.
    1 usage  new *
    public void dijkstra(Vertex start) {
        if (start == null) return;

        start.cost = 0;
        PriorityQueue<Vertex> queue = new PriorityQueue<>(Comparator.comparingInt(v -> v.cost));
        queue.add(start);

        while (!queue.isEmpty()) {
            Vertex b = queue.remove();
            b.known = true;

            for (int i = 0; i < b.adjacency.length; i++) {
                if (b.adjacency[i] == 0) continue;
                Vertex a = graph[i];
                int edgeWeight = b.adjacency[i];
                if (!a.known) {
                    if (b.cost + edgeWeight < a.cost) {
                        queue.remove(a);
                        a.cost = b.cost + edgeWeight;
                        a.path = b.name;
                        queue.add(a);
                    }
                }
            }
        }
        printShortestPaths(start);
    }

    Prints the shortest paths from a source vertex to all other vertices in the graph.
    Params: source – The source vertex.
    1 usage  new *
    public void printShortestPaths(Vertex source) {
        for (Vertex v : graph) {
            if (v != source && v.cost < Integer.MAX_VALUE) {
                printPath(source, v);
                System.out.println(" (" + v.cost + ")");
            }
        }
    }
```



```
    }

    Prints the shortest paths from a source vertex to all other vertices in the graph.
    Params: source – The source vertex.
    1 usage  new *
    public void printShortestPaths(Vertex source) {
        for (Vertex v : graph) {
            if (v != source && v.cost < Integer.MAX_VALUE) {
                printPath(source, v);
                System.out.println(" (" + v.cost + ")");
            }
        }
    }

    Prints the shortest path from a source vertex to a target vertex.
    Params: source – The source vertex.
            target – The target vertex.
    2 usages  new *
    private void printPath(Vertex source, Vertex target) {
        if (target == source) {
            System.out.print(source.name);
        } else if (target.path == 0) {
            System.out.print("No path");
        } else {
            printPath(source, findVertexByName(target.path));
            System.out.print(" " + target.name);
        }
    }

    Finds a vertex by its name.
    Params: name – The name of the vertex to find.
    Returns: The vertex with the specified name, or null if not found.
    1 usage  new *
    private Vertex findVertexByName(char name) {
        for (Vertex v : graph) {
            if (v.name == name) {
                return v;
            }
        }
        return null;
    }

    Provides a string representation of the Graph.
    Returns: A string that represents the Graph and its connections.
    new *
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
```

Phillip Lechenauer