Programming Assignment 3

● Graded

Student

Mohammed Ali Abdul-nabi

Total Points

20 / 20 pts

Autograder Score

10.0 / 10.0

Passed Tests

Compiled Succesfully

Maze Tests

Graph Tests

Question 2

Graph Implementation **4** / 4 pts

2.1 **Class GraphNode.java** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.25 pts Missing items or methods

– 0.5 pts Incorrect

2.2 **Class GraphEdge.java** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.25 pts Missing items or methods

– 0.5 pts Incorrect

2.3 **Class Graph.java** **3** / 3 pts

✔ **– 0 pts** Correct

– 0.5 pts improper exception handling

– 1 pt Not all methods in GraphADT implemented/ extra methods implemented

– 2 pts Mostly incorrect implementaion for methods

– 3 pts No adjacency matrix/ list representation used for the Graph

– 3 pts Completely Incorrect or missing

Question 3

Class Maze Implementation — 4 / 4 pts

✔ **– 0 pts** Correct

– 0.5 pts improper exception handling

– 1 pt methods return objects are not in the requested format and or type

– 1.5 pts Incorrect Constructor

– 2 pts Incorrect solve method

– 4 pts Completely Incorrect or missing

Question 4

Coding Style — 2 / 2 pts

4.1 Meaningful names for variables and constants: All instance variables are private; only public methods are as specified in assignment — 0.5 / 0.5 pts

✔ **– 0 pts** Correct

– 0.25 pts Partially satisfied requirements

– 0.5 pts Completely ignored

4.2 Readability: Good Indentation — 0.5 / 0.5 pts

✔ **– 0 pts** Correct

– 0.5 pts Missing indentation.

– 0.5 pts unstructured/ unformatted code

4.3 Code Comments : Comments for instance variables and methods; comments within code to explain alogorithms — 1 / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts inadequate comments

– 0.5 pts Missing comments.

– 1 pt No comments.

– 0 pts Correct

– 1 pt little to no comments

– 0.5 pts Commented out code left in

Autograder Results

Autograder Output

{"score": 2, "output": "Code compiled successfully", "output_format": "simple_format", "visibility": "visible", "st

Compiled Succesfully

Maze Tests

Graph Tests

Submitted Files

Graph.java Download

```java
import java.util.*;

public class Graph implements GraphADT {

    private Map<GraphNode, List<GraphEdge>> adjacencyList; // Adjacency list
    private Map<Integer, GraphNode> nodes; // Map to store nodes by their names

    // Constructor: initializes the graph with n nodes and no edges
    public Graph(int n) {
        adjacencyList = new HashMap<>();
        nodes = new HashMap<>();
        for (int i = 0; i < n; i++) {
            GraphNode node = new GraphNode(i);
            adjacencyList.put(node, new ArrayList<>());
            nodes.put(i, node);
        }
    }

    // Inserts an edge connecting nodes u and v
        @Override
        public void insertEdge(GraphNode u, GraphNode v, int edgeType, String label) throws
GraphException {
                if (!nodes.containsKey(u.getName()) || !nodes.containsKey(v.getName())) {
                        throw new GraphException("One or both nodes do not exist.");
                }

                for (GraphEdge edge : adjacencyList.get(u)) {
                        if (edge.secondEndpoint().equals(v)) {
                                throw new GraphException("Edge already exists between these nodes.");
                        }
                }

                GraphEdge edge = new GraphEdge(u, v, edgeType, label);
                adjacencyList.get(u).add(edge);
                adjacencyList.get(v).add(edge); // Add to both nodes' lists for undirected graph
        }


    // Returns the node with the specified name
    @Override
    public GraphNode getNode(int name) throws GraphException {
        if (!nodes.containsKey(name)) {
            throw new GraphException("Node does not exist.");
        }
        return nodes.get(name);
    }

    // Returns an iterator of all edges incident on the node u
    @Override
```

```
49      public Iterator<GraphEdge> incidentEdges(GraphNode u) throws GraphException {
50          if (!nodes.containsKey(u.getName())) {
51              throw new GraphException("Node does not exist.");
52          }
53
54          List<GraphEdge> edges = adjacencyList.get(u);
55          if (edges.isEmpty()) {
56              return null; // No incident edges
57          }
58          return edges.iterator();
59      }
60
61      // Returns the edge connecting nodes u and v
62      @Override
63      public GraphEdge getEdge(GraphNode u, GraphNode v) throws GraphException {
64          if (!nodes.containsKey(u.getName()) || !nodes.containsKey(v.getName())) {
65              throw new GraphException("One or both nodes do not exist.");
66          }
67
68          // Search for the edge in u's adjacency list
69          for (GraphEdge edge : adjacencyList.get(u)) {
70              if (edge.secondEndpoint().equals(v)) {
71                  return edge;
72              }
73          }
74
75          throw new GraphException("No edge exists between the given nodes.");
76      }
77
78      // Checks if nodes u and v are adjacent
79          @Override
80          public boolean areAdjacent(GraphNode u, GraphNode v) throws GraphException {
81                  if (!nodes.containsKey(u.getName()) || !nodes.containsKey(v.getName())) {
82                          throw new GraphException("One or both nodes do not exist.");
83                  }
84
85                  for (GraphEdge edge : adjacencyList.get(u)) {
86                          if (edge.secondEndpoint().equals(v) || edge.firstEndpoint().equals(v)) {
87                                  return true;
88                          }
89                  }
90
91                  return false;
92          }
93
94  }
95
```

GraphEdge.java Download

```java

public class GraphEdge {
	private GraphNode origin;
	private GraphNode dest;
	private int type;
	private String label;

	// Constructor to initialize an edge with endpoints, type, and label
	public GraphEdge(GraphNode origin, GraphNode destination, int type, String label) {
		this.origin = origin;
		this.dest = destination;
		this.type = type;
		this.label = label;
	}

	// Returns the first endpoint of the edge
	public GraphNode firstEndpoint() {
		return this.origin;
	}

	// Returns the second endpoint of the edge
	public GraphNode secondEndpoint() {
		return this.dest;
	}

	// Returns the type of the edge
	public int getType() {
		return this.type;
	}

	// Sets the type of the edge
	public void setType(int type) {
		this.type = type;
	}

	// Returns the label of the edge
	public String getLabel() {
		return this.label;
	}

	// Sets the label of the edge
	public void setLabel(String label) {
		this.label = label;
	}
	}

```

GraphNode.java Download

```java

public class GraphNode {

//	I need 2 variables, name and mark
private int name;
private boolean mark;

	public GraphNode(int name) {
		this.name = name;
		this.mark = false;
	}


//	setters and getters, should be fun
	public void mark(boolean mark) {
	this.mark = mark;

	}

	public boolean isMarked() {
		return mark;
	}

	public int getName() {
		return name;
	}

}

```

Maze.java

Download

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.HashMap;


public class Maze {
    private Graph mazeGraph;                        // stores the maze as a graph
    private HashMap<GraphNode, String> visitedNodes;    // keeps track of nodes alredy visited
    private HashMap<GraphEdge, String> edgeprocesseds;  // stores if edge is 'discovery' or 'back'
    private int entranceNode;                       // the entranc node of the maze
    private int exitNode;                           // exit of the maze
    private int availableCoins;                     // how many coins are available for doors

    public Maze(String inputPath) throws MazeException {
        try {
            BufferedReader reader = new BufferedReader(new FileReader(inputPath));
            readInput(reader);                      // reads the file and parses the maze data
            reader.close();                         // closes the file after reading
            visitedNodes = new HashMap<>();         // initialize map for visited nodes
            edgeprocesseds = new HashMap<>();       // init edge state map
        } catch (Exception e) {
            throw new MazeException("Unable to initialize maze: " + e.getMessage());
        }
    }

    //Check if the graph was made and return an error if it hasn't
    public Graph getGraph() throws MazeException{
        if (mazeGraph != null){
                    return mazeGraph;
            }
            return null;
    }

    //Check if there is a solution, if not return null
    public Iterator<GraphNode> solve() {
        try {
            return DFS(availableCoins, mazeGraph.getNode(entranceNode));
        } catch (GraphException e) {
            return null;
        }
    }

    private Iterator<GraphNode> DFS(int remainingCoins, GraphNode currentNode) throws
GraphException {
```

```
49        visitedNodes.put(currentNode, "visited"); // marks current node as visited
50        List<GraphNode> path = new ArrayList<>(); // list to store path to the exit
51
52        if (currentNode.getName() == exitNode) { // if current node is exit node
53            path.add(currentNode); // add the exit node to path
54            return path.iterator(); // return the solution path
55        }
56
57        Iterator<GraphEdge> edges = mazeGraph.incidentEdges(currentNode); // get all edges connected
    to the node
58        while (edges.hasNext()) { // go through each edge
59            GraphEdge edge = edges.next(); // take the next edge
60            if (!edgeprocesseds.containsKey(edge) || !edgeprocesseds.get(edge).equals("discovery") &&
    !edgeprocesseds.get(edge).equals("back")) {
61                GraphNode adjacentNode;
62                if (currentNode == edge.firstEndpoint()) { // get the other endpoint
63                    adjacentNode = edge.secondEndpoint();
64                } else {
65                    adjacentNode = edge.firstEndpoint();
66                }
67
68                if (!visitedNodes.containsKey(adjacentNode) ||
    !visitedNodes.get(adjacentNode).equals("visited")) {
69                    int coinsLeft = remainingCoins; // coins left for traversal
70                    if (edge.getLabel().equals("door")) {
71                        coinsLeft -= edge.getType(); // decrease coins if door
72                    }
73                    edgeprocesseds.put(edge, "discovery"); // mark edge as discovered
74
75                    if (coinsLeft >= 0) { // check if enough coins are left
76                        Iterator<GraphNode> subPath = DFS(coinsLeft, adjacentNode); // recursive DFS call
77                        if (subPath != null) { // if path is found
78                            path.add(currentNode); // add current node to path
79                            while (subPath.hasNext()) { // add all nodes from subpath
80                                path.add(subPath.next());
81                            }
82                            visitedNodes.put(currentNode, ""); // reset current node's visited status
83                            return path.iterator(); // return the full path
84                        }
85                    }
86                } else {
87                    edgeprocesseds.put(edge, "backtrack"); // backtrack edge if already visited
88                }
89            }
90        }
91
92        visitedNodes.put(currentNode, ""); // unmark node when backtracking
93        return null; // no path found, return null
94    }
95
96
97    private void readInput(BufferedReader reader) throws IOException, GraphException {
```

```
98      int width = 0, height = 0; // stores the maze dimensions
99      String line = reader.readLine();
100     int lineCount = 0, nodeIndex = 0, charIndex = 0, gridLineCounter = 0;
101
102     while (line != null) { // read each line of input
103        try {
104           if (lineCount == 4) {
105              mazeGraph = new Graph(width * height); // initialize the graph
106           }
107
108           if (lineCount == 1) {
109              width = Integer.parseInt(line); // get maze width
110           } else if (lineCount == 2) {
111              height = Integer.parseInt(line); // get maze height
112           } else if (lineCount == 3) {
113              availableCoins = Integer.parseInt(line); // read coins budget
114           } else if (lineCount > 3) { // lines describing rooms and connections
115              charIndex = 0;
116              for (int i = 0; i < line.length(); i++) {
117                 char currentChar = line.charAt(i); // read each character
118                 if (currentChar == 's') {
119                    entranceNode = nodeIndex; // set entrance node
120                    nodeIndex++;
121                 } else if (currentChar == 'x') {
122                    exitNode = nodeIndex; // set exit node
123                    nodeIndex++;
124                 } else if (currentChar == 'o') {
125                    nodeIndex++; // regular room
126                 } else if (Character.isDigit(currentChar) || currentChar == 'c') {
127                    int start = -1, end = -1; // calculate start and end nodes
128                    if (lineCount % 2 == 0) {
129                       start = (charIndex - 1) / 2 + gridLineCounter * width;
130                       end = (charIndex + 1) / 2 + gridLineCounter * width;
131                    } else {
132                       start = charIndex / 2 + (gridLineCounter - 1) * width;
133                       end = charIndex / 2 + gridLineCounter * width;
134                    }
135
136                    if (currentChar == 'c') {
137                       insertEdge(start, end, 0, "corridor"); // insert corridor edge
138                    } else {
139                       insertEdge(start, end, Character.getNumericValue(currentChar), "door"); // insert door
    edge
140                    }
141                 }
142                 charIndex++;
143              }
144              if (lineCount % 2 == 0) gridLineCounter++; // update grid counter
145           }
146
147           line = reader.readLine(); // read next line
148           lineCount++;
```

```
149             } catch (NumberFormatException e) {
150                 throw new IOException("Invalid input format"); // error if input invalid
151             }
152         }
153     }
154 
155     private void insertEdge(int node1, int node2, int cost, String processed) throws GraphException {
156                 GraphNode nodeu = mazeGraph.getNode(node1);
157                 GraphNode nodev = mazeGraph.getNode(node2);
158         mazeGraph.insertEdge(nodeu, nodev, cost, processed);
159     }
160 }
161 
```