CS 2210 — Data Structures and Algorithms
Assignment 5
Due Date: December 7, 11:55 pm
Total marks: 20

1 Overview

For this assignment you will write a program for finding a path to exit a labyrinth. The program will receive as input a file with a specification of the labyrinth, and it will produce as output a path from the entrance to the exit, if any exists. Think of the labyrinth as a set of rooms connected by corridors, some of which could be closed by walls. There are 3 types of walls: brick walls (displayed by the program in red), rock walls (displayed by the program in blue), and metal walls. The labyrinth might not have a path to the exit; in that case your program is allowed to break some of the walls to try to reach the exit. The program is given two types of bombs to break walls: blast bombs and melt bombs. A blast bomb can break a brick wall, but not a metal wall; a rock wall needs 2 blast bombs to be broken. A melt bomb can break a metal wall, but not a brick or a rock wall. The program will be given k_1 blast bombs and k_2 melt bombs, where k_1 and k_1 are specified in the input file.

Your program will represent the labyrinth as an undirected graph. Every node of the graph corresponds to a room, and every edge corresponds to either a corridor that can be used to go from one room to the other or to a wall that the program might decide to break. There are two special nodes in this graph corresponding to the entrance and the exit. A modified depth first search traversal, for example, can be used to find a solution for the labyrinth.

2 Classes to Implement

You are to implement at least four Java classes: Node, Edge, Graph, and Solver. You can implement more classes if you need to, as long as you follow good program design and information-hiding principles.

You must write all code yourself. You cannot use code from the textbook, the Internet, other students, or any other sources. However, you are allowed to use the algorithms discussed in class.

For each one of the classes below, you can implement more private methods if you want to, but you cannot implement additional public methods.

2.1 Class Node

This class represent a node of the graph. This class has two instance variables:

- `int name`: The name of a node is an integer value between 0 and $n - 1$.
- `boolean marked`: initially the value of this variable is `false`.

You must implement these public methods:

- `Node(int nodeName)`: Creates an unmarked node with the given name.
- `setMark(boolean mark)`: marks the node with the specified value.
- `boolean getMark()`: returns the value with which the node has been marked.
- `int getName()`: returns the name of the node.
- `boolean equals(Node otherNode)`: returns true of `this` node has the same name as `otherNode`; returns false otherwise.

2.2 Class Edge

This class represents an edge of the graph. This class has 3 instance variables:

- `Node firstEndpoint, Node secondEndpoint`: The two nodes connected by the edge.
- `int type`: The type of an edge can be
  - 1: corridor
  - 2: brick wall
  - 3: rock wall
  - 4: metal wall

You must implement these public methods:

- `Edge(Node u, Node v, int edgeType)`: Creates and edge of the given type connecting nodes `u` and `v`. For example let edge (u, v) represent a corridor of the labyrinth. The first endpoint of this edge is node u and the second endpoint is node v; the type of the edge is 1.
- `Node firstEndpoint()`: returns the first endpoint of the edge.
- `Node secondEndpoint()`: returns the second endpoint of the edge.
- `int getType()`: returns the type of the edge.
- `setType(int newType)`: sets the type of the edge to the specified value.
- `equals(Edge otherEdge)`: returns true if `this Edge` object connects the same two nodes as `otherEdge`.

2.3 Class Graph

This class represents an undirected graph. You need to determine what instance variables are needed in this class. You must only use the instance variables needed to implement the methods specified below. You must use use an adjacency matrix or an adjacency list representation for the graph. For this class, you must implement all and only the public methods specified in the `GraphADT` interface and the constructor. These public methods are described below.

- `Graph(n)`: creates an empty graph with `n` nodes and no edges. This is the constructor for the class. The names of the nodes are $0, 1, \ldots,$ `n`-1.
- `insertEdge(Node u, Node v, int edgeType)`: adds to the graph an edge connecting nodes `u` and `v`. The type for this new edge is as indicated by the last parameters. This method throws a GraphException if either node does not exist or if there is already an edge connecting the given nodes.
- `Node getNode(int name)`: returns the node with the specified `name`. If no node with this name exists, the method should throw a GraphException.
- `ArrayList incidentEdges(Node u)`: returns a list storing all the edges incident on node `u`. It returns null if `u` does not have any edges incident on it.
- `Edge getEdge(Node u, Node v)`: returns the edge connecting nodes `u` and `v`. This method throws a `GraphException` if there is no edge between `u` and `v`.
- `boolean areAdjacent(Node u, Node v)`: returns *true* if and only if nodes `u` and `v` are adjacent.

The last three methods throw a `GraphException` if `u` or `v` are not nodes of the graph.

2.4 Class Solver

This class represents the Labyrinth. You must think about which instance variables are needed in this class. A graph will be used to store the labyrinth and to find a solution for it. You must implement the following public methods:

- `Solver(String inputFile)`: constructor for building a labyrinth from the input file. If the input file does not exist, this method should throw a `LabyrinthException`. Read below to learn about the format of the input file.
- `Graph getGraph()`: returns a reference to the graph representing the labyrinth. Throws a Labyrinth Exception if the graph is not defined.
- `Iterator solve()`: returns a java Iterator containing the nodes along the path from the entrance to the exit of the labyrinth, if such a path exists. If the path does not exist, this method returns the value `null`. For example for the labyrinth described below the Iterator returned by this method should contain the nodes `0, 1, 5, 6,` and `10`.

3 Input File

The input file is a text file with the following format:

```
S
W
L
K1
K2
RHRHRH···RHR
V*V*V*···V*V
RHRHRH···RHR
V*V*V*···V*V
⋮
RHRHRH···RHR
```

Each one of the first five lines contain one number: `S, W, L, K1`, or `K2`.

- `S` is the scale factor used to display the labyrinth on the screen. Your program will not use this value. If the labyrinth appears too small on your monitor, you can increase this value. Similarly, if the labyrinth is too large, choose a smaller value for the scale.
- `W` is the width of the labyrinth. The rooms of the labyrinth are arranged in a grid. The number of rooms in each row of this grid is the width of the labyrinth.
- `L` is the length of the labyrinth, or the number of rooms in each column of the grid.
- `K1` is the number of blast bombs that the program is allowed to use while looking for a solution.
- `K2` is the number of melt bombs that the program is allowed to use while looking for a solution.

For the rest of the file, `R` is any of the following characters: 'e', 'x', '*', or 'o'. `H` could be '*', 'b', 'r', 'm', or '-', and `V` could be '|', '*', 'B', 'R', or 'M'. The meaning of the above characters is as follows:

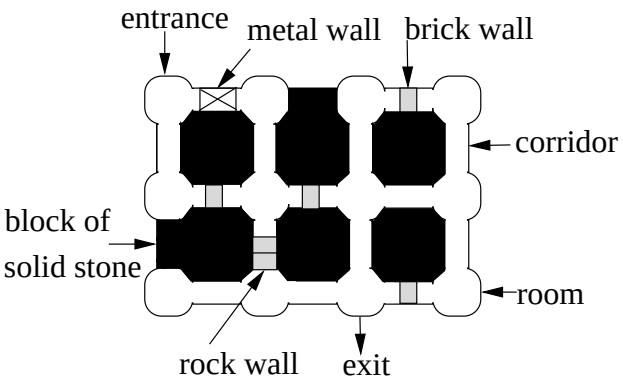
- 'e': entrance to the labyrinth
- 'x': exit of the labyrinth
- 'o': room

- 'b': horizontal brick wall
- 'r': horizontal rock wall
- 'm': horizontal metal wall
- 'B': vertical brick wall
- 'R': vertical rock wall
- 'M': vertical metal wall
- '-': horizontal corridor
- '|': vertical corridor
- '*': unbreakable, solid stone block

There is only one entrance and one exit, and each line of the file (except the first four lines) must have the same length. Here is an example of an input file:

```
30
4
3
1
1
emo*obo
|*|*|*|
obobo-o
**R*|*|
o-o-xbo
```

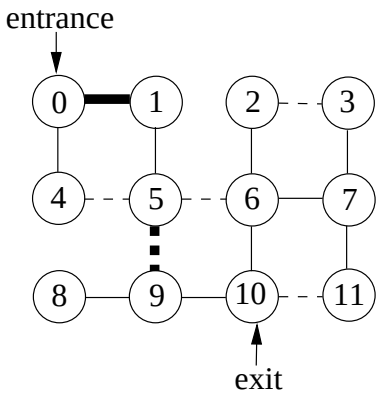
This input represents the following labyrinth



In this labyrinth up to one brick wall and one metal wall could be broken to try to reach the exit.

3.1 Graph Construction

The rooms (or nodes) are numbered consecutively, starting at zero, from left to right and top to bottom. For example, the above labyrinth is represented with this graph:



where dotted edges represent brick walls, thick dotted edges represent rock walls, thick solid edges represent metal walls, and solid thin edges represent corridors. In the `Solver` class you need to keep a reference to the entrance and exit nodes.

3.2 Solving the Labyrinth

A solution for the labyrinth is **any** path from the entrance node to the exit node that uses at most the specified number of bombs of each type. If there are several solutions for the labyrinth, your program might return any one of them.

The solution can be found, for example, by using a modified DFS traversal. While traversing the graph, your algorithm needs to keep track of the nodes along the path that the DFS traversal has followed. If the current path already has used the maximum allowed number of bombs, then no more edges representing walls can be added to it.

For example, consider the above graph and let the number of blast bombs be 1 and the number of melt bombs be also 1. Assume that the algorithm visits first nodes 0, 4, and 5. As the algorithm traverses the graph, all visited nodes get marked. While at node 5, the algorithm cannot next visit nodes 6 or 9, since then two brick walls would have been broken by the current path. Hence, the algorithm goes next to node 1. However, the exit cannot be reached from here, so the algorithm must go back to node 5, and then to nodes 4 and 0. Note that nodes 1, 5 and 4 must be unmarked when DFS traces its steps back, otherwise the algorithm will not be able to find a solution.

Next, the algorithm will move from node 0 to node 1 (this uses the melt bomb) and then to nodes 5, and 6 (the edge from 5 to 6 can be followed because when at node 5, the current path is 0, 1, 5, which has no brick wall edges yet, so we can use the blast bomb to break the brick wall between 5 and 6). From 6 the exit 10 is reached on the next step. So, the solution produced by the algorithm is: `0, 1, 5, 6,` and `10`. Note that the path 0, 1, 5, 9, 10 is not a valid solution as edge (5,9) represents a rock wall that needs two blast bombs to be broken.

You do not have to implement the above algorithm if you do not want to. Please feel free to design your own solution for the problem.

4 Code Provided

You can download from the course's website two files: DrawLab.java and Lab.java. Class DrawLab provides the following public methods that are used to display the labyrinth and the solution computed by your algorithm:

- `DrawLab(String labyrinthFile)`: displays the labyrinth on the screen. The parameter is the name of the labyrinth file.

- `drawEdge(Node u, Node v)`: draws an edge connecting the specified nodes.

Read carefully the code of class `Lab.java` to learn how to invoke the methods from the `Solver` class to find the solution for the labyrinth. `Lab.java` also shows how to use the iterator returned by the `Solver.solve()` method to draw the solution found by your algorithm on the screen. You can use `Lab.java` to test your implementation of the `Solver.java` class.

You can also download from the course's website some other java classes that you will need (Board.java, GraphADT.java, GraphException.java, LabyrinthException.java) and some examples of input files that you can use to test your program. We will also post a program that we will use to test your implementation for the `Graph` class.

5 Hints

You might find the `ArrayList` and `Stack` classes useful. However, you do not have to use them if you do not want to. Recall that the java class `Iterator` is an interface, so you cannot create objects of type `Iterator`. The methods provided by this interface are `hasNext(), next()`, and `remove()`. An Iterator can be obtained from an `ArrayList` or `Stack` object by using the method `iterator()`. For example, if your algorithm stores the path from the entrance of the labyrinth to the exit in a `Stack S`, then an iterator can be obtained from `S` by invoking `S.iterator()`.

6 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and white spaces should be used to improve readability.
- No instance variable should be used unless they contain data which needs to be maintained in the object from call to call.
- All instance variables should be declared `private` to maximize information hiding. Any access to the variables should be done with accessor methods.

7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- Tests for the `Graph` class: 4 marks.
- Tests for the `Solver` class: 4 marks.
- Coding style: 2 marks.
- `Graph` implementation: 4 marks.
- `Solver` implementation: 4 marks.