CS 2210a — Data Structures and Algorithms
Assignment 5
Labyrinth
Due Date: 2011 December 6, 11:59 pm
Total marks: 20

1 Overview

For this assignment you will write a program for solving a labyrinth. The program will receive as input a file with a description of the labyrinth, and it will produce as output a path from the entrance to the exit, if any. Think of the labyrinth as a set of rooms connected by halls or corridors, some of which could be closed by walls. The labyrinth might not have a path to the exit; in that case your program is allowed to break **up to** k walls to try to reach the exit, where k is a value specified in the input file.

Internally, the labyrinth will be stored as an undirected graph. Every node of the graph corresponds to a room, and every edge corresponds to either a hall 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 six Java classes: Node, Edge, GraphException, LabyrinthException, Graph, and Labyrinth. 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 Node

This class represent a node of the graph. You must implement these public methods:

- `Node(int name)`: the constructor for the class. Creates an unmarked node with the given name. The name of a node is an integer value between 0 and $n-1$, where n is the number of vertices in the graph to which the node belongs. A node can be marked with a value that is either `true` or `false` using the following method.
- `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 vertex.

2.2 Edge

This class represents an edge of the graph. You must implement these public methods:

- `Edge(Node u, Node v, String type)`: the constructor for the class. The first two parameters are the endpoints of the edge. The last parameter is the type of the edge, which for this project can be either "hall" or "wall". Each edge will also have a `String` label. When an edge is created this label is initially set to the empty `String`.

- `Node firstEndpoint()`: returns the first endpoint of the edge.
- `Node secondEndpoint()`: returns the second endpoint of the edge.
- `String getType()`: returns the type of the edge. As mentioned above, the type of an edge is one of these 2 possible strings: “hall” or “wall”.
- `setLabel(String label)`: sets the label of the edge to the specified value;
- `String getLabel()`: gets the label of the edge.

For example let edge (u, v) represent a hall 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 “hall”.

2.3 Graph

This class represents an undirected graph. 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 $0, 1, \ldots, \texttt{n}-1$.
- `insertEdge(Node u, Node v, String edgeType)`: adds to the graph an edge connecting `u` and `v`. The type for this new edge is as indicated by the last parameters. The label of the edge is the empty `String`. This method throws a GraphException if either node does not exist or if there is already an edge connecting the given vertices.
- `Node getNode(int name)`: returns the node with the specified `name`. If no node with this name exists, the method should throw a GraphException.
- `Iterator incidentEdges(Node u)`: returns a Java Iterator 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 Labyrinth

This class represents the Labyrinth. A graph will be used to store the labyrinth and to find a solution for it. You must implement the following public methods:

- `Labyrinth(String inputFile)`: constructor for building a labyrinth from the contents of 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 Implementation Issues

3.1 Input File

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

```
S
W
L
K
RHRHRH···RHR
V V V ···V V
RHRHRH···RHR
V V V ···V V
⋮
RHRHRH···RHR
```

Each one of the first four lines contain one number: `S, W, L,` or `K`.

- `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 must 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.
- `K` is the number of walls that the program is allowed to break while looking for a solution.

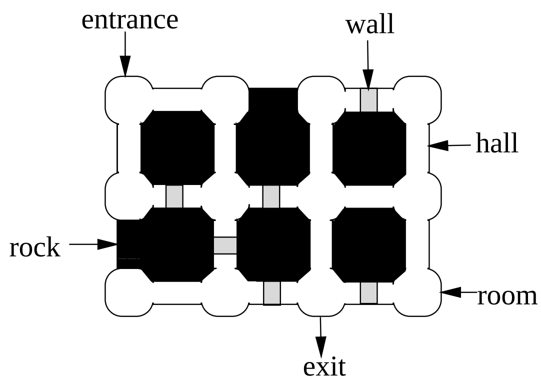
For the rest of the file, `R` is any of the following characters: ’s’, ’e’, or ’o’. `H` could be ’ ’ (space), ’h’, or ’-’, and `V` could be ’`|`’, ’ ’, or ’v’. The meaning of the above characters is as follows:

- ’s’: entrance to the labyrinth
- ’e’: exit of the labyrinth
- ’o’: room
- ’h’: horizontal wall
- ’v’: vertical wall
- ’-’: horizontal hall
- ’`|`’: vertical hall
- ’ ’: unbreakable, solid rock

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
s-o oho
| | | |
ohoho-o
  v | |
o-oheho
```

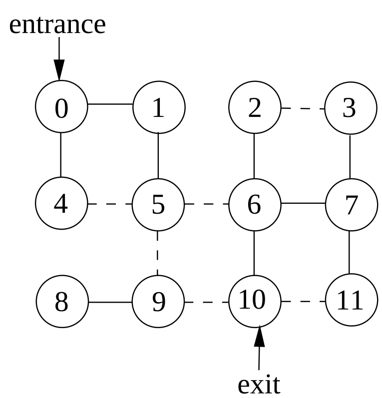
This input represents the following labyrinth



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

3.2 Graph Construction

The rooms (or nodes) are numbered consecutively, starting at zero. For example, the above labyrinth is represented with this graph:



where dotted edges represent walls and solid edges represent halls. In the `Labyrinth` class you need to keep a reference to the entrance and exit vertices.

3.3 Labyrinth solution

A solution for the labyrinth is **any** path from the entrance vertex to the exit vertex that uses at most the specified number of wall edges. 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 vertices along the path that the DFS traversal has followed. If the current path already has the maximum allowed number of wall edges, then no more wall edges can be added to it.

For example, consider the above graph and let the number of allowed wall edges in the solution be 1. Assume that the algorithm visits first vertices 0, 4, and 5. As the algorithm traverses the graph, all visited vertices get marked. While at vertex 5, the algorithm cannot next visit vertices 6 or 9, since then two walls would have been broken by the current path. Hence, the algorithm goes next to vertex 1. However, the exit cannot be reached from here, so the algorithm must go back to vertex 5, and then to vertices 4 and 0. Note that vertices 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 vertex 0 to vertices 1, 5, and 6 (the edge from 5 to 6 can be followed because when at vertex 5, the current path is 0, 1, 5, which has no wall edges yet). 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`.

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 Solve.java. Class DrawLab provides the following public methods that you will use 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 vertices.

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

You can also download from the course's website some examples of input files that we will 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 `Vector` 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 a `Vector` 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 variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose values do not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared `private` (not `protected`), 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 `Labyrinth` class: 4 marks.
- Coding style: 2 marks.
- `Graph` implementation: 4 marks.
- `Labyrinth` implementation: 4 marks.

8 Handing In Your Program

You are required to fill out and sign the Assignment Submission Form, which can be downloaded from

`http://www.csd.uwo.ca/courses/CS2210a/submForm.html`

You must also print an assignment ticket. Put the submission form, assignment ticket, and a hard copy of your program in an envelope labelled with your name. Drop the envelope in the CS2210 locker (third floor of the Middlesex College Building) by the due date.

You must also submit an electronic copy of your program. To submit your program, you must run the `submit` program on your Java files. Please put your code for the assignment in a directory called `Assignment5`, and issue the command

```
submit cs2210 Assignment5
```

Please do not put your code in sub-directories; this will make it easier for the markers. When you submit your program, we will receive a copy of it with a datestamp and timestamp. You can submit your program more than once if you need to. We will take the latest program submitted as the final version, and will deduct marks accordingly if it is late.

After you submit your assignment you should receive an email message from the submissions system acknowledging the receipt of your work. You can also check whether your assignment was received by filling out the electronic form at

`http://www.csd.uwo.ca/undergrad/Courses/ereceipt.shtml`

It is your responsibility to ensure that your assignment was received by the system.