CS 2210a — Data Structures and Algorithms
Assignment 5
Path Planning

Due Date: 2014 December 1, 11:59:59 PM

Total marks: 20

1 Overview

For this assignment you will write a program that given a road map, it will find a path between two specified points, if such a path exists. Some of the roads in the map are free and some are toll roads; the use of a toll road costs 1 dollar. Your program will receive as input a file with a description of the road map, the starting point s, the destination e, and the amount of money k available to pay for toll roads. The program will then try to find a path from s to t that uses at most k toll roads.

Your program will store the road map as an undirected graph. Every edge of the graph represents a road and every node represents either the intersection of 2 roads or the end of a dead-end road. There are two special nodes in this graph denoting the starting point s and the destination e. A modified depth first search traversal, for example, can be used to find a path as required.

2 Classes to Implement

You are to implement at least six Java classes: `Node`, `Edge`, `GraphException`, `Graph`, `MapException`, and `Map`. 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)`: This is the constructor for the class and it creates an *unmarked* node (see below) with the given name. The name of a node is an integer value between 0 and $n-1$, where n is the number of nodes in the graph.

  A node can be marked with a value that is either `true` or `false` using method `setMark`. This is useful when traversing the graph to know which vertices have already been visited.
- `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 "free" (if the edge represents a free road) or "toll" (if the edge represents a toll road). 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 either "free" or "toll".
- `setLabel(String label)`: Sets the label of the edge to the specified value; you can use this method, for example, to label an edge as "discovery", or "back".
- `String getLabel()`: Gets the label of the edge.

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 the public methods specified in the `GraphADT` interface plus the constructor. These public methods are described below.

- `Graph(n)`: Creates a graph with `n` nodes and no edges. This is the constructor for the class. The names of the nodes are $0, 1, \ldots, \mathtt{n}-1$.
- `insertEdge(Node u, Node v, String edgeType)`: Add an edge of the given type to the graph connecting `u` and `v`. The label of the edge is the empty `String`. This method throws a `GraphException` if either node does not exist or if in the graph 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`.
- `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 nodes `u` and `v` are adjacent; it returns *false* otherwise.

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

2.4 Map

This class represents the road map. As mentioned above, a graph will be used to store the map and to find a path from the starting point to the destination. For this class you must implement the following public methods:

- `Map(String inputFile)`: Constructor for building a road map from the content of the input file. If the input file does not exist, this method should throw a `MapException`. Read below to learn about the format of the input file.
- `Graph getGraph()`: Returns a reference to the graph representing the road map. This method throws a `MapException` if the graph is not defined.

- `Iterator findPath()`: Returns a Java Iterator containing the nodes along the path from the starting point to the destination, if such a path exists. If the path does not exist, this method returns the value `null`. For example for the road map 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` is the scale factor used to display the map on the screen. Your program will not use this value. If the map appears too small on your screen, you must increase this value. Similarly, if the map is too large, choose a smaller value for the scale.
- `W` is the width of the map. The roads of the map are arranged in a grid. The number of vertical roads in each row of this grid is the width of the labyrinth.
- `L` is the length of the map, or the number of horizontal roads in each column of the grid.
- `K` is the number of toll roads that the program is allowed to use in the path from the starting point to the destination.

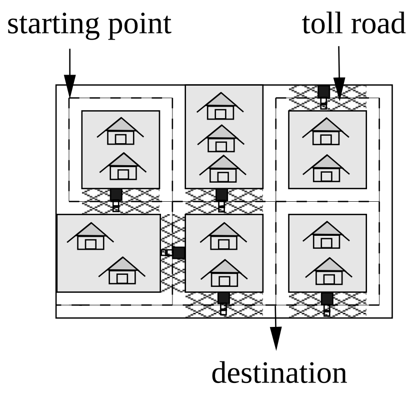
For the rest of the file, `R` is any of the following characters: 's', 'e', or 'o'. `H` could be ' ' (space), 'h', or '-', and `V` could be '`|`', ' ', or 'v'. The meaning of the above characters is as follows:

- 's': starting point
- 'e': destination
- 'o': intersection of two roads
- 'h': horizontal toll road
- 'v': vertical toll road
- '-': horizontal free road
- '`|`': vertical free road
- ' ': block of houses

There is only one starting point and one destination, 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
```

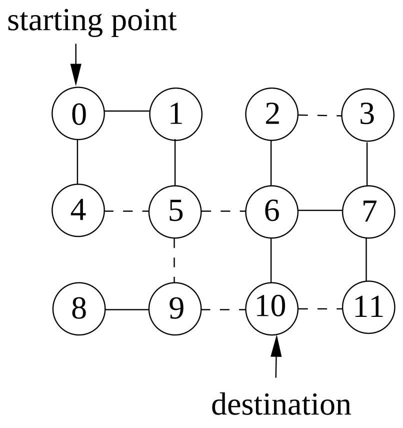
This input represents the following road map



As specified in the fourth line of the input file, for this road map up to one toll road could be used to try to reach the destination.

3.2 Graph Construction

The road map is represented as a graph in which nodes are numbered consecutively, starting at zero from left to right and top to bottom. For example, the above road map is represented with this graph:



where dotted edges represent toll roads and solid edges represent free roads. In the `Map` class you need to keep a reference to the starting point and destination vertices.

3.3 Finding a Path

Your program must find **any** path from the starting vertex to the destination vertex that uses at most the specified number of edges representing toll roads. If there are several such paths, 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 toll-road edges, then no more toll-road edges can be added to it.

For example, consider the above graph and let the number of allowed toll-road 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 toll roads would have been used by the current path. Hence, the algorithm goes next to vertex 1. However, the destination cannot be reached from here, so the algorithm must go back to vertex 5, and then back 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 toll-road 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 the following files: `DrawMap.java`, `Board.java`, `Solve.java`, `GraphADT.java`, `house1.jpg`, `house2.jpg`, `house3.jpg`, and `house4.jpg`. Class `DrawMap` provides the following public methods that you will use to display the labyrinth and the solution computed by your algorithm:

- `DrawMap(String inputFile)`: Displays the road map on the computer's screen. The parameter is the name of the input 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 `Map` class to find the required path. `Solve.java` also shows how to use the iterator returned by the `Map.findPath()` method to draw the solution found by your algorithm. You can use `Solve.java` to test your implementation of the `Map.java` class. Class `Board.java` is an auxiliary class for `DrawMap`, and `GraphADT.java` contains the Graph ADT. The `.jpg` files are used to display the map on the screen.

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`, 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 `Map` class: 4 marks.
- Coding style: 2 marks.
- `Graph` implementation: 4 marks.
- `Map` implementation: 4 marks.

8 Handing In Your Program

You must submit an electronic copy of your program through OWL. Please **DO NOT** put your code in sub-directories. Please **DO NOT** submit a .zip file containing your code; submit the individual .java files.

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. Please send me an email if you make multiple submissions so we can ensure that your last submission is marked.