CS 2210a — Data Structures and Algorithms
Assignment 5
Due Date: December 6, 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. Each road has a cost or a profit for using it and there is an initial amount k of money available to pay for using the roads. There are 3 kinds of roads: *free* roads that have no cost for using them, *toll* roads with a positive cost for using them, and *compensation* roads which earn the road user a certain amount of money. Your program will receive as input a file with a description of the road map, the starting point s, the destination e, the amount of money k available to pay for toll roads, the cost of each toll, and the amount gained by using a compensation road. The program will then try to find a path from s to e that can be traversed with the given amount of money plus the money earned from travelling the compensation 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 two 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 `RoadMap`. 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 a 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 nodes in the graph.
- `setMark(boolean mark)`: Marks the node with the specified value, either `true` or `false`. This is useful when traversing the graph to know which vertices have already been visited.
- `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, int 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 0 (if the edge represents a free road), 1 (if the edge represents a toll road), or -1 (if the edge represents a compensation road). Each edge will also have a `String` label which you can use, for example, to mark the edge as "discovery" or "back". 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.
- `int getType()`: Returns the type of the edge.
- `setLabel(String label)`: Sets the label of the edge to the specified value.
- `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 $0, 1, \ldots,$ `n`-1.
- `Node getNode(int name)`: Returns the node with the specified `name`. If no node with this name exists, the method should throw a `GraphException`.
- `insertEdge(Node u, Node v, int edgeType)`: Adds an edge of the given type 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.
- `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 four methods throw a `GraphException` if `u` or `v` are not nodes of the graph.

2.4 RoadMap

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:

- `RoadMap(String inputFile)`: Constructor for building a graph from the content of the input file; this graph represents the road map. 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 the graph representing the road map.
- `int getStartingNode()`: Returns the starting node (specified in the input file).
- `int getDestinationNode()`: Returns the destination node.

- `int getInitialMoney()`: Returns the initial amount of money available to pay tolls.
- `Iterator findPath(int start, int destination, int initialMoney)`: Returns a Java Iterator containing the nodes along the path from the `start` node to the `destination` node, if such a path exists. The amount specified in `initialMoney` plus the money earned by passing through the compensation roads must be enough to pay for all the toll roads. 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:

```
SCALE
START
END
INITIAL_BUDGET
WIDTH
LENGTH
TOLL
GAIN
RHRHRH···RHR
HXHXHX···HXH
RHRHRH···RHR
HXHXHX···HXH
⋮
RHRHRH···RHR
```

Each one of the first eight lines contains one number:

- `SCALE` 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.
- `START` is the starting vertex.
- `END` is the destination vertex.
- `WIDTH` 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 map.
- `LENGTH` is the length of the map, or the number of horizontal roads in each column of the grid.
- `INITIAL_BUDGET` is the initial amount of money available to pay for toll roads.
- `TOLL` is the amount of money that needs to be paid to use a toll road.
- `GAIN` is the amount of money received when using a compensation road.

For the rest of the file, `R` is any of the following characters: '+' or 'X'. `H` could be 'X', 'T', 'C', or 'F'. The meaning of the above characters is as follows:

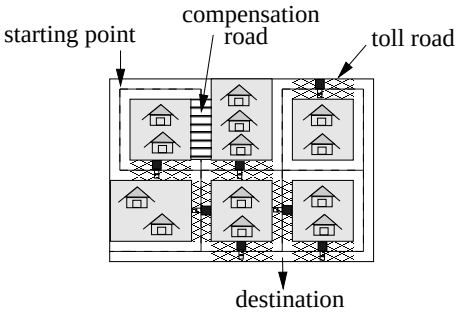
- '+': intersection of two roads, or dead end

- 'T': toll road
- 'F': free road
- 'C': compensation road
- 'X': block of houses

Each line of the file (except the first eight lines) must have the same length, so as mentioned above, the only road maps that we will consider are grid road maps. Here is an example of an input file:

```
30
0
10
1
4
3
1
-1
+F+X+T+
FXFXFXF
+T+T+F+
XXTXFXF
+F+T+T+
```

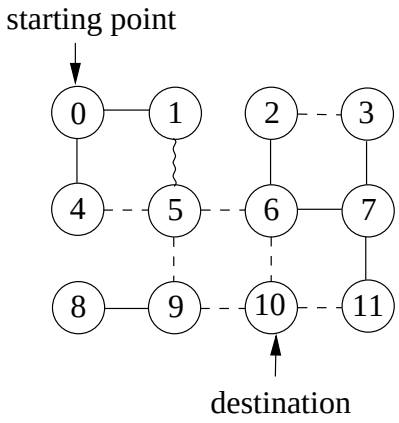
This input represents the following road map



For the above example, the initial amount of money available for paying tolls is 1 dollar; each toll costs 1 dollar, and using a compensation road earns 1 dollar (a negative value in the eight line means that the use of a compensation road earns money for the user).

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,solid edges represent free roads, and wavy edges represent compensation roads. In the `RoadMap` class you need to keep a reference to the starting and destination nodes.

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 amount of money. 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 has already used all available money to pay tolls, then no more toll-road edges can be added to it.

For example, consider the above graph and let the initial amount of money 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 at a total cost of 2. 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, and 5. Since edge $(1, 5)$ represents a compensation road, the algorithm earns 1 dollar, so the total amount of money is 2. The algorithm can then move to node 6 and then to 10 as it can pay for the two toll roads. Therefore, 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: `TestDict.java`, `DrawMap.java`, `Board.java`, `Solve.java`, `GraphADT.java`, `GraphException`, `MapException`, `house1.jpg`, `house2.jpg`, `house3.jpg`, and `house4.jpg`. Class `DrawMap` provides the following public methods that are used by class Solve.java to display the road map 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.

The `main` method is in class `Solve.java` , so to run your program you will type

```
Java Solve input_file
```

You can also type

```
java Solve input_file delay
```

where `delay` is the number of milliseconds that the program will wait before drawing the next edge of the solution.

You can use `TestDict.java` to test your implementation of the `Graph.java` class. You can also download from the course's website some examples of input files that we will use to test your program.

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 starting node to the destination 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 `RoadMap` 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.