1   Overview

In this assignment you are required to write a program that finds a path between two specified points in a road map, if such a path exists. The road map has 3 kinds of roads: *public* roads that have no cost for using them, *private* roads that have a positive cost or toll for using them, and *reward* roads that pay the user a certain amount of money for using them (as the user of such roads is required to perform a certain task, like cleaning the road). Your program will receive as input a file with a description of the road map, the starting point s, the destination e, an amount of money k that the program initially has for paying for the use of private roads, the cost of using each private road, and the amount gained by using a reward road. Your program will try to find a path from s to e that can be traversed with the given amount of money plus the money earned from using the reward 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 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 public road), 1 (if the edge represents a private road), or -1 (if the edge represents a reward road).

- `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.

2.3 Graph

This class represents an undirected graph. You must use an **adjacency list** or an **adjacency matrix** representation for the graph as mentioned in the video lecture (**edge list structure is not allowed**). For this class, you must implement all the public methods specified in the provided `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`. 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 three methods also throw a `GraphException` if `u` or `v` are not nodes of the graph.

2.4 RoadMap

This class represents the road map. A graph will be used to store the map and to try 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 input file specified in the parameter; 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 (this value and the next two values are 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 of a path from the `start` node to the `destination` node as specified above, if such a path exists. The amount specified in `initialMoney` plus the money earned by passing through the reward roads must be enough to pay for all the private 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
WIDTH
LENGTH
INITIAL_BUDGET
TOLL
GAIN
RHRHRH···RHR
HXHXHX···HXH
RHRHRH···RHR
HXHXHX···HXH
⋮
RHRHRH···RHR
```

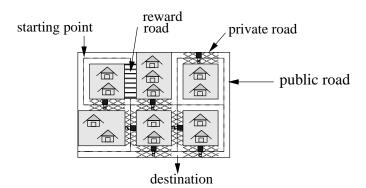Each one of the first eight lines of the input file contains one number:

- `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 private roads.
- `TOLL` is the amount of money that needs to be paid to use a private road.
- `GAIN` is the amount of money received when using a reward road.

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

- '+': denotes either and intersection of two roads, or a dead end
- 'T': denotes a private road (**T**oll required)
- 'F': denotes a public road (**F**ree to use)
- 'C': denotes a reward road (**C**ash received from using it)
- 'X': denotes a block of houses

Each line of the file (except the first eight lines) must have the same length because, as mentioned above, the only road maps that we will consider are grid road maps. Here is an example of an input file (also see sample inputs in the `inputs.zip` file):

```
30
0
10
4
3
1
1
1
+F+X+T+
FXCXFXF
+T+T+F+
XXTXTXF
+F+T+T+
```
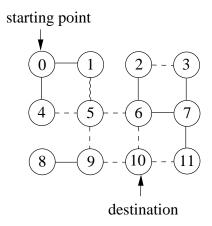
This input represents the following road map

For the above example, the initial amount of money available for paying tolls is 1 dollar; each toll costs 1 dollar, and using a reward road earns 1 dollar.

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:

starting point

destination

where dotted edges represent private roads, solid edges represent public roads, and wavy edges represent reward 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 private-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 vertex 0, then 4, and then 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 private 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 the algorithm steps back, as otherwise the algorithm will not be able to find a solution. Next, the algorithm will move from vertex 0 to vertex 1, and then to 5. Since edge $(1, 5)$ represents a reward road, the algorithm earns 1 dollar, so the total amount of money that the algorithm has is 2. The algorithm can then move to node 6 and then to 10 as now it can pay for the two private 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 Class Provided

The following files are provided: `TestGraph.java`, `DrawMap.java`, `Board.java`, `Path.java`, `GraphADT.java`, sample input files, and several image files. The `main` method is in class `Path.java` , so to run your program in the command line, you will type

    java Path input_file
Or, you can also pass a second parameter `delay` to your program

```
    java Path 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 `TestGraph.java` to test your implementation of the `Graph.java` class. You can also use the example input files in `inputs.zip` to test your program.

5   Hints

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, for instance, 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 unnecessary instance variables must be used in your code.
- All instance variables must be declared `private`, to maximize information hiding.

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.
- `RoadMap` 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 or other compressed file containing your code; submit the individual .java files. Remember that the TAs will test your program use the same environment settings as department's computer.

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.

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