CS 2210a — Data Structures and Algorithms
Assignment 5
Bus Routes
Due Date: 2015 December 8, 11:59:59 PM
Total marks: 20

1 Overview

For this assignment you will write a program that given a map of bus lines (from now on we will call this just a "map"), it will find a path between two given points that requires at most a pre-specified number of bus line changes, if such a path exists. Your program will receive as input a file with a description of the bus lines, the starting point S, the destination D, and the allowed number k of bus line changes. The program will then try to find a path from S to D that requires at most k bus changes.

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

The following figure shows an example of a map with three bus lines, "a", "b", and "c". The starting point is marked S and the destination is D. In all the maps that we will consider for this assignment, the streets will be arranged in a grid. Furthermore, only buses of one bus line will run on any given street (so the same street cannot be labeled with two different bus lines).

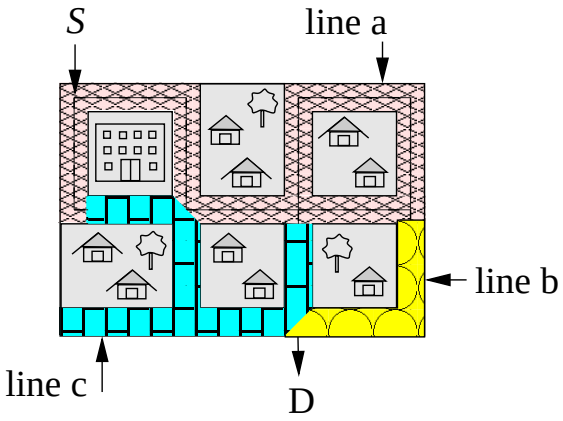

Figure 1: A map of bus lines.

The above map is represented with the following graph. Each edge is marked with the bus line that it represents. The nodes of the graph are numbered consecutively, starting at zero from left to right and top to bottom, so for a graph with n nodes, the nodes are numbered $0, 1, 2, \ldots, n-1$.

If, for example, we want to find a path from S to D that uses only one bus line change, a possible solution is the path $0, 1, 5, 6, 10$. Note that there might be several paths as required; your program will need to find only one. Observe that the path $0, 4, 5, 6, 10$ is not a valid solution as it requires three bus changes: at node 4 a change is needed from line "a" to line "c", at node 5 a second bus change is needed from line "c" to line "a", and at node 6 a third bus change is needed from line "a" to line "c". Even though this last path is formed by edges belonging to only two bus lines, three bus line changes are needed.

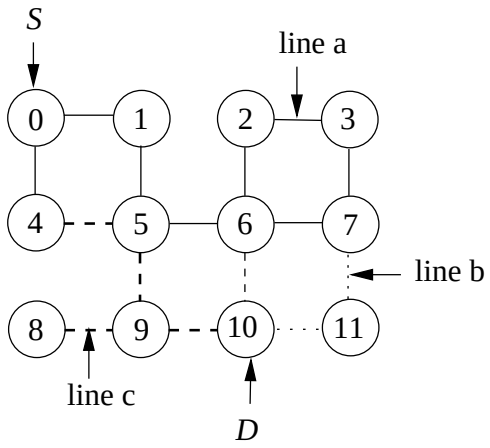

Figure 2: The graph representation for the above map.

2 Classes to Implement

You are to implement at least four Java classes: `Node`, `Edge`, `Graph`, 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 busLine)`: The constructor for the class. The first two parameters are the endpoints of the edge. The last parameter is the bus line to which the street represented by the edge belongs. Each bus line has a name that consists of a single letter, for example ”a”, ”b”, and ”c” might be the names of three bus lines.
- `Node firstEndpoint()`: Returns the first endpoint of the edge.
- `Node secondEndpoint()`: Returns the second endpoint of the edge.
- `String getBusLine()`: Returns the busLine to which the edge belongs.

2.3 Graph

This class represents an undirected graph. You must 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.
- `insertEdge(Node u, Node v, String busLine)`: Adds an edge connecting `u` and `v` and belonging to the specified bus line. 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 map of bus lines. 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 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 the graph representing the 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 map and path described above the Iterator returned by this method should contain the nodes `0, 1, 5, 6,` and `10`.

In this class you will need to store a reference to the starting and destination nodes.

3 Implementation Issues

3.1 Input File

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

```
C
W
H
```

```
K
RLRLRL···RLR
L L L ···L L
RLRLRL···RLR
L L L ···L L
⋮
RLRLRL···RLR
```

Each one of the first four lines contains one number:

- `C` is the scale factor used to display the map on the screen. Your java code will not use this value; it is used by the programs supplied to you. 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 streets of the map are arranged in a grid. The number of vertical streets in each row of this grid is the width of the map.
- `H` is the length of the map, or the number of horizontal streets in each column of the grid.
- `K` is the number of bus line changes allowed in the path from the starting point to the destination.

For the rest of the file, `R` is any of the following characters: '0', '1', or '+'. `L` could be ' ' (space), or a letter. The meaning of the above characters is as follows:

- '0': starting point
- '1': destination
- '+': intersection of two streets
- ' ': block of houses
- *letter*: street belonging to the bus line whose name is specified by the *letter*

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 representing the map shown in Figure 1:

```
30
4
3
1
0a+ +a+
a a a a
+c+a+a+
  c c b
+c+c1b+
```

The fourth line of this input file specifies that for this map at most one bus line change could be used to try to reach the destination.

3.2 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 bus line changes. A bus line change happens when in the path there are two adjacent edges belonging to different bus lines. If there are several paths from the starting vertex to the destination with at most the allowed number of bus line changes, your program can return any one of them.

The path can be found, for example, by using a modified depth first search (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 the maximum allowed number of bus line changes, then no more bus line changes can be added to it.

For example, consider the above graph and let the number of allowed bus line changes 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 bus changes would have been used in the 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 the DFS traversal 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. From 6 the exit 10 is reached on the next step while changing buses only once, as required. So, the solution produced by the algorithm is: $0, 1, 5, 6, 10$.

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`, `GraphException`, `MapException`, `house1.jpg`, `house2.jpg`, `house3.jpg`, and `house4.jpg`. Class `DrawMap` provides the following public methods that you will use to display the map and the solution computed by your algorithm:

- `DrawMap(String inputFile)`: Displays the 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. `Solve.java` contains the `main` method for your program so you should use it 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.

To run your program you need to 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 every edge of your solution.

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 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 or .rar 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.