Assignment 5: Navigating through a Maze

Out: 8th April 2015 **Due: 21st April 2015 at 11:55pm**

In this assignment you will use Dijkstra's shortest path algorithm to find your way out of a maze.

Provided files

1. The main "driver" program for this assignment is a Java program: MazeDraw.java. It calls the C++ program that implements a graph that you will write. It works with 4 data files.
    a. A file that describes a maze. The maze has exactly one entry and one exit.
    b. Two files that describe the path in the maze from its entry to its exit. **This file will be generated by the program.**
    c. A file that stores "forbidden" cells. The Java program allows you to click on cells in the maze to toggle them between "allowed" and "forbidden". Your shortest paths must avoid these cells.

MazeDraw.java calls the C++ program in SolveMaze.cpp using the Java Native Interface (JNI). This allows a Java program to call programs written in other languages (C++ in this case).

2. An executable file "GenerateMaze" has been provided to you. This program can be used to generate a random maze. The command below will create a maze with 40 rows and 50 columns and store it in "newmaze.txt":

```
./GenerateMaze 40 50 > newmaze.txt
```

Several maze files have been provided to you. For debugging purposes it is recommended that you generate smaller mazes (e.g. 5x5) so that the number of vertices is manageable.

3. A shell of a C++ program has been provided to you in five files:
    a. SolveMaze.h: This is an auto-generated file from Java. Do not attempt to change this file!
    b. SolveMaze.cpp: This file describes the function that reads the input maze and "forbidden" cells, creates a graph from them, finds shortest paths and writes them out to a file. It expects the file names to be the following:
        i. If the input maze is in "newmaze.txt" as generated above:
        ii. It expects the "forbidden" cells to be in the file "newmaze-blocked.txt".
        iii. It writes out the two shortest paths in "newmaze-out-with-turns.txt" and "newmaze-out-without-turns.txt".

    Study the code in SolveMaze.cpp carefully, because it works with the Graph class that you must implement. The only public functions in your Graph class should be the ones called from this code. As always you can have as many private helpers as you want.

    c. A priority queue class has been provided to you in PriorityQueue.h. This is essentially a solution to Assignment 3, except that this is a min-priority queue. You may use your own PriorityQueue class from Assignment 3 if you wish.
    d. Graph.h/.cpp: these files contain "empty" functions. **In this assignment, you must fill this class.**
4. A Makefile: this is ready to be used as-is. **You should not have to change this file in any way.**

How to build/run the solution

Within the given files, you will see a folder called "working solution". This folder contains the solution to this assignment, in the form of a pre-compiled library file (libSolveMaze.so). This working solution will allow you to check your solutions against mine.

1. Generate a maze file called “maze-10x10.txt” as follows:

   ```
   ./GenerateMaze 10 10 > maze-10x10.txt
   ```
2. Run the Java program as follows:

   ```
   java -Djava.library.path=. MazeDraw maze-10x10
   ```
3. The program prints out the lengths and number of turns in each path on the command prompt, and the window shows these paths. Compare them with your program

How to build/run the program: A complete example

4. Use the provided Makefile to build the Java and C++ program. This will generate:
   a. A shared library file libSolveMaze.so from the C++ code. This is the Linux-equivalent of a dll.
   b. MazeDraw.class from the Java code.
5. Generate a maze file called “maze-10x10.txt” as follows:

   ```
   ./GenerateMaze 10 10 > maze-10x10.txt
   ```
6. Run the Java program as follows:

   ```
   java -Djava.library.path=. MazeDraw maze-10x10
   ```

This command will run MazeDraw.class, pass it a parameter “maze-10x10” and add the current directory in the java.library.path variable so that Java can find the dll to run. With the parameter “maze-10x10” it will expect the files “maze-10x10.txt” to have the maze, “maze-10x10-blocked.txt” to have the forbidden cells, “maze-10x10-out-with-turns.txt” and “maze-10x10-out-without-turns.txt” to have the two shortest paths.

When you run this command, a window will open and it may be blank. Simply click the mouse button anywhere to see the maze and the paths. When you run this program initially you will not see any paths, simply because you have not written that code yet.

7. Clicking on any cell will mark it red as “forbidden” and the paths should automatically change if needed. Clicking on a red cell will make it “allowable” and the paths should automatically change if needed.

<u>Please try all these steps as early as possible to understand the workflow and contact me if you have any problems!</u>

What to do

For this assignment, all you have to do is write the Graph class that is being used in SolveMaze.cpp. **You should not have to change anything else.**

You must represent the maze as a graph of edges and vertices. The vertices of the graph are the centers of all the cells, each represented by a single integer (0..w*h-1 of the maze). An edge exists between two cells if and only if there is a direct way to walk from one of the cells to the other.

1. Represent this graph using adjacency maps (A first map of (vertex,map), and the internal map of (vertex,edge weight)).
2. Fill the function `setEdge` in the Graph class.

   As you can see in SolveMaze.cpp, the weight given to an edge by default is set to the direction of that edge (NORTH, EAST, SOUTH AND WEST being 0, 1, 2, 3 respectively)

Part A

1. Fill the function `getEdgeWeightWithoutTurns(int a,int b)` in the Graph class that returns the weight of the edge from vertex ‘a’ to vertex ‘b’. If such an edge exists, this function should simply return ‘1’. If the edge does not exist, it should return a large positive number.

2. Fill the function `getShortestPathWithoutTurns(int start,int end)` in the Graph class that implements the Dijkstra's shortest path algorithm to find the shortest path from the starting vertex to the ending vertex. This function will use the `getEdgeWeightWithoutTurns` function above to determine edge weights.

   This function should return a list of vertices (as integers) that are on that path, in order from "start" to "end" including those two vertices.

When you run the MazeDraw program now, you will see this shortest path in red.

Part B (read this part carefully)

In this part you will determine a shortest path through the same maze, while also trying to minimize the number of turns one must make while navigating the maze.

1. Fill the function `getEdgeWeightWithTurns(int a,int b,int current_direction)` in the Graph class that returns the weight of the edge from vertex 'a' to vertex 'b', assuming that the user is facing the provided direction at vertex 'a'. The directions NORTH, EAST, SOUTH, WEST are represented as numbers 0, 1, 2, 3 respectively.

   If an edge from 'a' to 'b' exists, the weight of that edge is determined as follows: standing at vertex 'a', count the minimum number of turns that have to be made from "current_direction" so that the user faces vertex 'b'. The adjacency list already stores the direction that the user must be facing in order to go from 'a' to 'b'. For example, if current_direction=0 (NORTH), and 'b' is to the east/west of 'a', then it will take 1 turn. If 'b' is to the south of 'a' then it will take 2 turns, while it will take 0 turns if 'b' is to the north of 'a'. The number of required turns will never be greater than 2. **This function returns 1 plus the number of turns.**

   Hint: Think about the logic required to calculate the number of turns. It is much simpler than you might imagine. Just work out a table with "from" and "to" directions with intended answers and see if you can determine the logic. **In order to get full credit, this function must be somewhat mathematical , and should not be simply a large number of if-statements.**

2. Fill the function `getShortestPathWithTurns(int start,int end,int initial_direction)` in the Graph class that works similar to `getShortestPathWithoutTurns` in Part A, except it now considers the initial facing direction of the user, and uses the `getEdgeWeightWithTurns` function above to determine edge weights.
   Hints:
   a. In addition to all the data you are keeping track of for every vertex 'x', record the direction d(x) the user would be facing when he/she arrives at that vertex along the shortest path.
   b. If you are arriving at a shortest path to 'v' from 'u', then d(v) should store the direction the user would be facing at 'v', and this is exactly what is stored in the adjacency map for edge (u→v).
   c. While evaluating vertices adjacent to the current vertex 'v', d(v) will help you to determine the cost to reach an adjacent vertex.

When you run the MazeDraw program now, you will see the "without turns" shortest path from getShortestPathWithoutTurns in red and the "with turns" shortest path from getShortestPathWithTurns in green. These paths will likely overlap in many regions.

Testing your maze strategy (for your fun, not for points)

After you have part 2 working successfully, you may ponder: how much should I weigh the reluctance to turn vs. the actual distance traversed?

You can determine this by changing `getEdgeWeightWithTurns` to return “(1-x) +x* no. of turns”, where you can vary ‘x’ from 0 to 1. The number ‘x’ will weigh the importance of turns vs. actual distance travelled per edge (which is 1). For example, if x=0 (which is effectively what `getEdgeWeightWithoutTurns` in Part 1 is) means the user does not consider turning laborious at all (and is thus looking to simply navigate the maze using as few steps as possible). If x=1 then it means the user wants to avoid a turn at all costs, no matter how long the path is. By default, you have effectively implemented x=0.5 in `getEdgeWeightWithTurns` in Part 2 (i.e. giving equal importance to shortest distance and minimum turns).

What to submit

Submit all source code files (*.cpp,*.h, *java), along with the Makefile in a single zipped file on Reggienet.