CPS 112: Computational Thinking with
Algorithms and Data Structures
Homework 4

Handed out: October 2nd, 2018 (Tuesday)
Midway Checkpoint Due: 11:59pm October 17th, 2018 (Wednesday)
Late submissions accepted with penalty until 11:59 pm October 19th, 2018 (Friday)

Final Due Date: 11:59 pm October 24th, 2018 (Wednesday)
Late submissions accepted with penalty until 11:59 pm October 26th, 2018 (Friday)

If you run into trouble or have questions, arrange to meet with me or a tutor!

Before you submit:

- Please review the homework guidelines available on Canvas
- Read the assignment very carefully to ensure that you have followed all instructions and satisfied all requirements.
- Make sure to compile and thoroughly test all of your programs before you submit your code. Any file that does not compile will receive a 0!

Submitting

- Create a zip file named like this: *yourusernameHWX.zip* (with your username and replacing the *X* with the assignment number) that contains a directory named *yourusernameHWX*, which contains all of your .java files, your debug log, and any other files necessary for the assignment.
- Upload your coversheet as a PDF (NOT IN THE ZIP FILE). Feel free to use the coversheet template files.

In this homework you will create a program that solves mazes using the stack and queue data structures we have been exploring in class.

NOTE: This is a two week project! To help you pace your work, there is a midway checkpoint. Your final submission should contain the *entire project*. The whole thing will be graded, including the problems that appear before the midway checkpoint. **The final score for Parts 1-3 will be the average of your score at the checkpoint and your score for the final submission.** This is to allow you the chance to encounter difficulties and recover or simply to get credit for improving on your initial solutions to these problems.

Part 1 (6 pts): Representing the Maze

A typical maze has walls, a start point, and an end-point. The goal is to find a path from the start to the end that doesn't cross any walls. Your first task is to create a class to represent a maze in your program. The maze will be input as a text file. The first line of the file will contain the number of rows and the number of columns separated by a space. The rest of the file will specify the maze layout, one row of the maze per line, using the following characters to mark different components of the maze:

- walls: # (hash mark)
- open spaces: . (period)
- start: o (lowercase 'O')
- goal: * (asterisk)

You may assume that every maze has only one start and one goal. The edges of the maze are meant to be impassable, even if there is no # to indicate as such. You may assume the file ends with a newline character. **Think carefully about the design of this class with an eye toward efficiency. Some design choices will make many of these class methods very efficient while other choices will make many of the methods very inefficient.**

In *Maze.java* create a Maze class that stores the layout of the maze. The constructor of Maze should take a filename as a parameter. It should open that file, and store the maze layout contained therein, in a GridLocation[][] as you see fit. **The Maze constructor should *not* catch file-not-found exceptions; just declare that the constructor may throw them**. Your Maze class should have the following methods:

- getNumColumns() – returns the number of columns in the maze
- getNumRows() – returns the number of rows in the maze
- getStartLocation() – returns a GridLocation containing the row and column of the start
- getGoalLocation() – returns GridLocation containing the row and column of the goal
- getSquare(GridLocation) – returns the character representing the square at the given position
- toString() – returns a string representation of the maze (a string representing the maze's layout with newline characters separating rows, as in the maze files). Handy for debugging!

The GridLocation class is essentially the same one used in the FlyWorld game in homework 2 with the Creature stuff removed. For this homework, you have been given a jar file and Makefile to facilitate your compiling/running code to use GridLocation. Although you cannot edit GridLocation directly (because it's a .jar file), you may find it useful to extend that class inside Maze.java. Make sure to test your Maze class thoroughly before moving on by writing a `main` method in Maze.java. When you have it mostly working, be sure to test with *mazefile1, mazefile2* and some mazes you make up on your

own!

Compiling and Running:
When you compile and run you'll need to provide java a reference to the included .jar file. At this point you can compile like this:

```
javac -cp .:./GridLocation.jar Maze.java
```

and you can run like this (assuming Maze has a main method):

```
java -cp .:./GridLocation Maze mazefile1.txt
```

A makefile has been provided to you to help make it easier. You can have it compile for you like this:

```
make part1
```

Part 2 (7 pts): Solving the Maze (sort of)
Before you start coding, let's spend some time thinking about the algorithm you'll be using.
To start with, you'll just determine whether a maze *has* a solution. Later on you will actually give a path through the maze. To accomplish this you'll have to keep track of some things. You'll keep an "agenda" of locations that have been deemed reachable but that still need to be processed (see description below). You'll also have to somehow keep track of which locations have already been added to the agenda so you don't bother with them again. Here is a rough outline of the algorithm:

1. Start with an empty agenda and add the start location to it. Mark the start location as added.
2. Is the agenda empty? If it is, we've run out of options and the maze is unsolvable. Return false.
3. Otherwise, get a location from the agenda.
4. Check if that location is the goal. If it is, there is a solution! Return true.
5. Otherwise, that location is a reachable "open space" square. Find all adjacent locations (to the north, south, east, and west) that are not walls, outside the maze, or already added and:
    a. Add them to the agenda.
    b. Mark them as added.
6. Go to 2.

Note that this algorithm does not specify exactly how the agenda works – that's on purpose! As you'll see, you can substitute different concrete agendas into this algorithm and get different behavior.

Before you move on, I recommend working out a small example by hand just to get a sense for how the algorithm works. Write down a small maze and perform this algorithm by hand, keeping track of the contents of the agenda and which locations have been added. For now, assume the agenda works like a stack (it stores and retrieves locations in a FILO manner).

When you have one or more examples worked out, send a picture of them to me!

StackAgenda

Ready to start coding? First you'll need to work out the details of the agenda. Begin by designing an abstract class called Agenda in *Agenda.java*. It should have the following *abstract* methods:

- addLocation(GridLocation) – adds a location (row, column position) to the agenda
- getLocation() – removes and returns a location from the agenda
- isEmpty() – returns true if the agenda is empty
- clear() – removes all items from the agenda

- toString() - returns a string representation of the agenda

Now, in *StackAgenda.java* implement a concrete subclass of Agenda: StackAgenda. It should behave as advertised, storing and retrieving locations in a FILO manner. Feel free to make use of Java's standard stack structure in `java.util.Stack`. You should thoroughly test your StackAgenda class all by itself before you incorporate it into the more complicated algorithm! You'll need to figure out how to compile and run your code.

MazeSolver

Now it's time to create a MazeSolver class in *MazeSolver.java.* Its constructor should take an Agenda as a parameter and store it as an instance variable. It should use this provided Agenda when solving the maze.

The most important method of MazeSolver is solveMaze(Maze). This method takes a Maze and determines whether it has a solution (whether there is some path from the start to the goal that doesn't cross any walls). It should implement the algorithm discussed above, using the Agenda that was given when the solver was created (don't forget to clear the agenda at the beginning of the method!).

I **highly** recommend that you plan your maze solver on paper **before** you start typing code.
Think about how you will concretely accomplish each of the steps of the algorithm and develop a clear sense of what it ought to do. What kinds of data structures will you employ? What will the logical structure of your code look like (loops, conditionals, etc.)? The implementation will go much more smoothly if you have a clear picture in your head of what you are trying to create!

You might want to write a method which generates the neighbors of a given node. You can put that method wherever you think it fits best!

Test It

At this point you should test your maze solver thoroughly. Create a file *SolveMaze.java* which defines a main method. It should take the filename of a maze file as a command line argument. The main function should create a Maze using the given filename, a StackAgenda, and a MazeSolver using that agenda. If the Maze constructor throws a FileNotFoundException then print an error message and quit. Otherwise, it should use the MazeSolver to solve the maze and print the result.
Hint: if you are encountering strange behavior, it may help to print out the locations your MazeSolver is adding/removing from the agenda and compare them to an example you work out by hand...just to see if it is behaving the way you think it ought to.

Test your code thoroughly using *mazefile1* and *mazefile2*, and any other examples that you come up with yourself. Make sure your code is correct before moving on.

Part 3 (7 pts): Really Solving the Maze (no, really)

In Part 2, you got part of the way to solving the maze. You can determine *if* there was a solution. But really you haven't solved the maze until you have a path from the start to the finish. In this part you will modify your solveMaze method so that, instead of returning true or false, it returns an

ArrayList<GridLocation> that represents this path: a sequence of adjacent locations, starting at the start position and ending at the goal position. If there is no solution, it should return an empty list.

Here is an idea about how to accomplish this. Whenever you add a location to the agenda, you know it is reachable (it is adjacent to a reachable square), but you haven't been keeping track of the location *from which* it is reachable. If only you knew, for each location, the location that added it to the agenda in the first place, you'd be in good shape (I would **discourage** the use of a HashMap for this. Once you found the goal, you could find the location that added the goal to the agenda, and then the location that added that location, and so on tracing all the way back to the start. This sequence of locations (in reverse) is a path from the start to the goal.

Again, *you should develop a plan of attack before you start coding!* Take this general idea and think through what you will need to implement it. Do an example by hand to see what sorts of issues arise. Only when you have a clear vision for where you want to go should you start coding. If you are having trouble developing that clear vision, come talk to me! I'm happy to help.

Once your solveMaze method is changed, make sure to change your main function as well to correctly print out the new and improved results. Test your changes thoroughly using the provided maze files and other examples of your own design.

The makefile provides a `midway` directive. Using that directive your project should compile correctly and it should be run by calling the `run-midway` directive.

Turn in your work so far. You score on these three parts will be averaged between your checkpoint submission and your final submission. If you are ahead of schedule, keep going! There's plenty more to do. If this first part of your project still needs work, don't panic! You still have time to recover. That said, you should come talk to me immediately so we can work together to get you back on track!

---Midway Checkpoint---

Part 4 (2 pts): Another Solver

You've already created a maze solver using a stack for an agenda. But that's not the only way to organize your data while solving a maze. Using different agendas will result in different maze solving algorithms.

In *QueueAgenda.java* implement a concrete subclass of Agenda called QueueAgenda. Unsurprisingly, it should store and retrieve items in a FIFO manner. You may use Java's standard linked-list structure in java.util.LinkedList or the ArrayDeque (being careful to store and retrieve in FIFO manner).

Modify your main function to now take two command line parameters. The first is the name of the maze file, and the second is either “s” or “q” to indicate which type of agenda to pass to the MazeSolver constructor (if it's anything else you can print an error and quit).

Test your program thoroughly using both agendas to be sure it behaves sensibly!

Part 5 (3 pts): Adding Graphics

In this section we'll add pretty graphics to make it easier to visualize what the maze solver is doing. Most of the heavy lifting has been done for you, but you will have to make a few small modifications to make it all work together. Take a look at *MazeGUI.java* to familiarize yourself with the MazeGUI class. You will use a MazeGUI to graphically display the maze.

Alter your MazeSolver's solveMaze method to additionally take a MazeGUI as a parameter (and store it as an instance variable).

The solveMaze method should use the MazeGUI in the following ways:

- Whenever you pull a location off of the agenda, you should call the pause method. This will pause execution for approximately 1 second so you can see the action step-by-step). Then call the visitLoc method, passing in the location that was just taken off the agenda. This will color that location dark grey.
- Whenever you add a location to the agenda, you should pass that location to the addLocToAgenda method. This will color the added location light grey.
- When you are constructing the solution path at the end, for each location in the path you should call the addLocToPath method. This will color the solution path gold.

I have provided a main function with MazeGUI so you should now use that rather than *SolveMaze.java* to run the code. You should look at this function to see what parameters it takes.

Test the whole program on several maze files!

Your program should compile correctly with:
```
make final
```

and it should run successfully (at least) with:
```
make run-final
```

Part 6 (5 pts): Efficiency

Now that your maze solver is working correctly, it's time to go back over it with an eye toward improving efficiency. You may have to significantly re-think some aspects of your design – or you may find that you have already taken these concerns into account in your initial design. In any case, carefully assess your code and be open to pursuing alternative approaches. If you are having trouble thinking of alternatives, come chat with me! I am happy to be a sounding board.
In particular, to get full credit for this part you should achieve the following complexities, in terms of n, the number of locations in the maze (i.e. $n = width*height$).

- In Maze...
  - The constructor and toString should be O(n).
  - All other methods should be O(1).
    - *Hint: you might need to re-think what instance variables Maze should have.*

- In both agendas...
  - All methods should be O(1), in some cases, amortized O(1).
- In MazeSolver...
  - You should be able to check whether a location has been added to the agenda in O(1) time.
  - When making the path, you should get each location's previous location in O(1) time.
    - *Hint: Think carefully about the structures you are using to store this information. Consider alternative approaches that would offer fast access to the data.*
  - The overall complexity of solveMaze should be O(*n*).

As you make changes, make sure you keep testing to verify that you are not breaking anything!

Part 7 (5 pts): Complexity Analysis

In *mazes.pdf* write a thorough, precise worst-case analysis of SolveMaze, in terms of *n*, the number of locations in the maze (*width*height*). As you did in Homework 3, you should refer to specific line numbers, analyze small blocks of code, and then combine those analyses to analyze larger blocks and so on. Your prose should guide the reader through the logic of your analysis. Imagine that one of your classmates is reading this, and you want to fully convince them that your analysis is correct. Notes:

- Analyze the code you actually wrote, not what you wish it were! If you didn't figure out how to achieve all of the goals of Part 6, analyze what you have instead. Furthermore, even if you think you did achieve those goals, look at your code with an objective, critical eye. Who knows? You might find that you missed something important.
- Just stating the answer is not enough. Based on Part 6 we both know what it's *supposed to be*. Your job is to thoroughly convince your reader that the complexity is what you say it is.
- Your score in this part is based upon the correctness of your conclusions, the quality of your argument, and the quality of your writing.

Part 8 (5 pts): Experiments and Discussion

In this part you will explore differences between the StackAgenda and the QueueAgenda in terms of the number of locations visited (colored dark grey) and the quality of the final solution.

First, consider the number of locations visited. Provide two maze files, *fastqueue.txt* and *faststack.txt* with the following properties, respectively. It will probably take some experimentation to find them!

- The QueueAgenda visits (colors dark grey) at most 5 squares while the StackAgenda visits at least 25.
- The StackAgenda visits no more than 5 locations while the QueueAgenda visits at least 25.

Hint: Not all mazes consist of narrow corridors; use as many or as few walls as you like.

In *mazes.pdf* add answers to the following two questions.

- Consider the relative efficiency of the two maze solvers (i.e. the number of locations visited). Based on the above examples, it is clear that neither one is *always* more efficient. What properties of mazes are important in determining which one is more efficient? In what kinds of mazes is the queue-based agenda more efficient and in what kinds is the stack-based agenda more efficient?

- Now consider the quality of the solutions, regardless of how long it takes to find it. Does one agenda always find a shorter path from the start to the goal? If so, clearly explain why that one is always best. If not, describe properties of the maze that make one or the other agenda produce a better solution. In either case, **supply at least one example maze** that helps to illustrate your points and demonstrates the differences between the two algorithms.

---Final Submission (due 11:59pm March 7th)---

This is the end of the project! Your final submission of the *entire project* will be graded. Your score for the first three parts will be the average of your checkpoint submission and your final submission. Make sure you've tested everything thoroughly and fixed as many problems with your midway checkpoint submission as you can (feel free to come see me if you are stuck on something). Go back through your code to look for opportunities to improve readability or elegance. Make sure you've clearly and thoroughly expressed your answers to the questions in Parts 7 and 8.

Submit all your source code, maze files, mazes.pdf, and cover sheet.