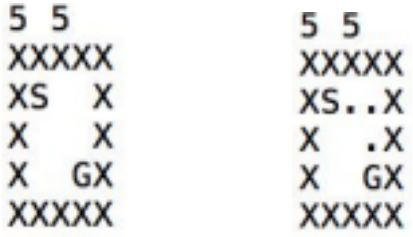
CSC220 Lab12
Graph Algorithms

The goal of this week's lab is:

1. Practice working with graphs
2. Practice working with stacks and queues
3. Practice using breadth-first search (BFS) and depth-first search (DFS)
4. Practice using a nested class
5. Practice I/O

This week, you are asked to write a program to help Pacman solve mazes. We will apply BFS and DFS search strategies to find a path from one location to another in any enclosed field of obstacles. We will also compare the results to see which delivers the shortest path. The field is given as input, represented as a plaintext file. The start and end point are indicated, as well as the layout of the field (the location of the walls and obstacles). **A similar text file is produced as output, annotated with the path from the start-point (S) to the end-point (G)**. If no such path exists, the input and output text files are identical. The following figure shows an example input and output, where Pacman's search strategy here gives the shortest path:



Pacman is free to move from his current location to any adjacent open space. This includes the space directly above, below, left and right of where he is. It does **not** include diagonally adjacent spaces. If any of the adjacent spaces are a wall, Pacman cannot travel in that direction. The path that your maze solver finds MUST be a **connected** path (it can't skip spaces and have Pacman "jumping" over walls or empty spaces).
You can assume that all input mazes will be rectangular. All of the border positions around the perimeter of the field will be walls (the field is fully enclosed).

To solve this problem, you must first encode the maze as a graph, and perform a search strategy from Pacman's starting location. The graph here can be represented by

a 2D array of nodes. In lab, we will populate the graph and prepare the output file. For the assignment, you will implement search.

Part 0

- Create a new Java project and call it **Lab12**.
- Create a package inside your project and call it **lab12.**
- Download the **Lab12-Assignment12** ZIP folder from BB and copy in the following files:
    - **Pacman.java (inside code/)** You will be working on developing methods for this class to help Pacman solve mazes.
    - **mazes/** This directory holds the input mazes we will be working with, and the solutions. You have learned how to add a text file to your project in assignment 4. If you are in doubt consult the instruction for assignment4 on Blackboard

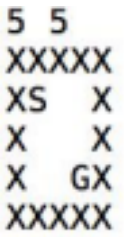
- Note the following directory for later:
    - **pacman/** This holds some Python scripts you can use to visualize your mazes.

Part 1 – Pacman **constructor**

The constructor for this class receives an input and output filename. The input name is

used to read a maze from a file with the given input name and the output name will be used by our solve methods to output the solved maze to a file.
The constructor has already been started for you. To finish the implementation, **you must finish writing the method buildGraph()** which initializes member variables and populates the graph according to the input file. This method uses a **BufferedReader** to visit the contents of the file line by line.

The input files are in the following form:



The first line contains two numbers, separated by a space. The first number is the height of the field, and the second is the width. The rest of the lines contain the layout of the field. The characters have the following meaning:

- **X:** A single wall segment. Pacman cannot travel on to or through walls.
- **S:** The starting point of the path that we are trying to find (where Pacman starts). This is an open space (no wall). Be sure to set the position of startX and startY after S has been visited.
- **G:** The ending point of the path we are trying to find (where Pacman wants to be). This is an open space (no wall).
- **(space)**: An open space on the field. Pacman is free to travel in any open space, assuming he can get there.

Part 2 – writeOutput **method**

This method is responsible for writing out the contents of the maze to file as pointed to by **outputFileName**. Output the height and width at the top, just like the input. The output should also have the same layout of the walls and the same start and end points.

After completing the assignment, some space characters will be replaced with dots ('.'). For now, the input and output files should be identical. **There is a single newline character after the last 'X' in the output.**

You must produce output in the exact format specified (for grading purposes). You will lose up to 10% if the format of your output file is wrong even though your solution is correct.

PrintWriter can be used to create and write to a file in Java. The method provides the incantation for you. Do not modify the path of the file.

Part 3 – getNeighbors **method**

This method receives a given node currentNode and returns an ArrayList neighbors of all unvisited neighboring nodes (or, successors) by looking at the node to the north, south, west, and east of it (i.e. all four cardinal directions). Visit only those neighbors of currentNode that have not been visited yet. If a neighbor is visited, mark them as such,

set its parent (to who?), and add it to neighbors. Take care that your neighbor checks do not go out of bounds (you should look at the class to see what methods can help you).

When visiting neighbors, we will maintain the following order: **(1) north, (2) south, (3) west, and (4) east**. While this order is not necessary for a BFS search strategy (why not?), it can change the outcome of DFS (why?). **Therefore, it is important that we are consistent with this order.**

Part 4 – backtrack **method**

This method receives a given node end and follows the path of nodes until it reaches the starting node S. The content of each node visited is modified to become the dot character (“.”)

Part 5 – solveBFS **method**

This method implements the breadth-first search (BFS) strategy Pacman will use to solve the maze. Recall that we use a **queue** data structure to implement BFS. Your first operation on this data structure should be inserting the starting node S.

Hint: the methods you just implemented in part 1 and 2 will prove very handy. After you have found the solution path, call writeOutput() to output the solution to file.

Part 6 – solveDFS **method**

This method implements the depth-first search (DFS) strategy Pacman will use to solve the maze. Recall that we use a **stack** data structure to implement DFS. Your solution here will be very similar to Part 3.

Part 7 – Testing your search

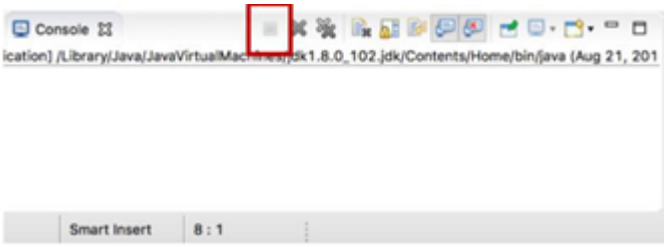
A collection of test mazes have been provided for you in the mazes/ directory. You should already be familiar with some of the input maze files, e.g. demoMaze.txt or tinyMaze.txt. The BFS solution is written to *BFSSol.txt and the DFS solution *DFSSol.txt.

You should compare your output (by writing it to file using writeOutput() or invoking the toString()) with these solution files. You can use a diff utility like https://text-compare.com/ to compare the files. Because of the convention we are using when visiting neighboring nodes (north, south, west, then east), the files should be identical. If not, you must go back and debug your code.

How to debug your code?

1. Use the Eclipse debugger you learned about during the first lab.
2. If you see JavaStackOverflow, that means that you have an infinite recursive call and your recursive call is filling up the “call stack” (we talked about this concept in class). Go back and debug the method that is causing the problem.
3. Infinite loops! How would you know you have an infinite loop? As you should know from CSC120, if you have an infinite loop in your code, your code will not stop running. An easy way to inspect that in Eclipse is to look

at your console window, if your code is done running the console should look like the following:



If the little square marked above is red and continues to stay red, that means your code has an infinite loop!

Part 8 – for your fun (after finishing assignment)

You can check out the Pacman tool (courtesy of John DeNero, UC Berkeley and Miriah Meyer, University of Utah).

Important note: You need python to be able to use this tool, if you don't have it, you can try this in the lab. All computers in the lab have python and you can follow the instruction below to use it.

Pacman tool

The pacman tool is not necessary for the successful completion of this assignment. However, it will help you visualize your mazes and verify that your solutions are correct. Downlaod pacman.zip to help you visualize your text-form mazes. You can also use this tool to play Pacman using the arrow keys!

Grab the "pacman" folder from the Lab12-Assignment12 ZIP. From the command line, change directories to that folder (using cd). To run pacman, type the following command... ...on a Linux/Mac machine:

/usr/bin/python pacman.py -l <path/to/mazefile>

...on a Windows machine:

python pacman.py -l <path/to/mazefile>

In these commands <path/to/mazefile> is the path to a file containing a maze, for example:

python pacman.py -l demoMaze.txt

where "demoMaze.txt" is a file containing a maze layout (such as the one above). This assumes that "demoMaze.txt" is in your pacman directory (it is included in the zip file). To test your own maze solutions, I recommend putting the solution maze file in the pacman directory. If your maze files are instead all in a different directory, you can provide the full path to the file.

Auto pacman

The above command will display the maze and the path (if there is one), but pacman will not move. This is designed so that you can examine the path without pacman eating it. You can control pacman with the arrow keys if you wish. If you want to see pacman follow the path automatically, add some additional command-line arguments:

python pacman.py -l demoMazeSol.txt -p auto

The -p auto tells pacman to follow the path laid out in the maze.

Zoom

If your maze is too small or too large, you can change the size of the pacman window with the -z command. For example:

python pacman.py -l demoMazeSol.txt -p auto -z
0.5 python pacman.py -l demoMazeSol.txt -p
auto -z 2

You can use any number after -z to increase or decrease the size of the window.

Remarks

- Make sure to submit your assignment by (re-)uploading your **Lab12** folder into your **csc220-cXXXX** folder by the deadline.
- **For all your assignments, please start early and seek help early (either from the instructor or the TAs).**