Assignment 7: PacMan! (Graph Pathfinding)

Graph Pathfinding

We have been asked to create a tool to help [PacmanLinks to an external site.](https://en.wikipedia.org/wiki/Pac-Man" \t "_blank) solve mazes. More generally, we must create a tool that can find the shortest path from one location to another in any enclosed field of obstacles. We are given the field as input, represented as a simple text file. The start-point and end point are indicated in this text file, as well as the layout of the field (the location of walls). We must produce a similar text file, but with the shortest path from the start-point to the end-point added to the field. If there is no path at all, the path indicators will simply not exist in the output.

To solve this problem, you must represent the maze as a graph, and perform a breadth-first search from Pacman's starting location.

Requirements

**PathFinder class:**

Create a class PathFinder in package assignment07, with the following method:

public static void solveMaze(String inputFile, String outputFile)

This method will read a maze from a file with the given input name, and output the solved maze to a file with the given output name. You must use the filenames exactly as is (do not change the directory or path). We will provide the full path to files we want to read/write in our tests. See required specifications below. *This method must use a graph and graph pathfinding to solve the problem.*

Your program may contain any other methods or classes necessary for your solution (it is recommended you create a separate Graph and Node classes). You will need a queue for breadth-first search. Implement one using your linked list implementation, or feel free to use Java's [LinkedListLinks to an external site.](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html" \t "_blank) which implements [QueueLinks to an external site.](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html" \t "_blank) (or any other Java stdlib class that implements Queue, of which there are several). **Make sure you submit all files necessary to run your program.**

Input

The input files are in the following form:

5 5  
XXXXX  
XS  X  
X X  
X GX  
XXXXX

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).

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.

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).

An example of reading numbers (height and width) form a file, assuming you're using a Scanner or BufferedReader named input to read the file:

  String[] dimensions = input.readLine().split(" ");  
  height = Integer.parseInt(dimensions[0]);  
  width = Integer.parseInt(dimensions[1]);

Output

The output of your program is a similar file. 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. It will, however, replace some of the open spaces (space characters) with dot characters ('.').

The spaces that you replace with dots are the spaces on the shortest path from the start point to the goal point. For example, given the above input, the following output may be produced:

5 5  
XXXXX  
XS..X  
X .X  
X GX  
XXXXX

Some of the empty spaces are now replaced with dots, connecting a path from 'S' to 'G'. There is a single newline character after the last 'X' in the output.

There are multiple shortest path solutions to the above example. Any connected path from 'S' to 'G' with length of 4 is a correct solution. For example, another acceptable solution to the above maze is:

5 5  
XXXXX  
XS X  
X. X  
X..GX  
XXXXX

If there is no path from 'S' to 'G', simply do not place any dots in the output file; print the original maze.

You can download a[zipped file of mazes](https://utah.instructure.com/courses/807886/files/136736319/download?wrap=1)[Download zipped file of mazes](https://utah.instructure.com/courses/807886/files/136736319/download?download_frd=1)and their solutions to try out your code.

An example of creating a file and writing to it in Java is below (where outputFile is a String). It will create the specified file if it does not exist. The example uses the try-with-resources statement that you can read about online if you haven't seen it before. You are of course welcome to implement this however you wish though. However, make sure to use the exact String outputFile passed in to your solveMaze method (do not modify the path of the file).

try(PrintWriter output = new PrintWriter(new FileWriter(outputFile))) {  
     output.println(height + " " + width);  
     // write more data here  
}

 Rules

* The path cannot go through or on top of walls.
* The path must be connected (no skips or jumps).
* Diagonally-adjacent spaces are not connected.
  + *Only up, left, down, right*
* If no path exists, the output file will have no dots.
* If multiple shortest paths exist, any of them are valid.
* Must produce output in exact format specified (for grading purposes).
* Ideally, your program must run fast (30 seconds or less) on all mazes we test. This includes file input, graph construction, pathfinding, and file output. The biggest test maze is 100x100 (see randomMaze in the [zipped file of mazes](https://utah.instructure.com/courses/807886/files/136736319/download?wrap=1)[Download zipped file of mazes](https://utah.instructure.com/courses/807886/files/136736319/download?download_frd=1)). There will be other test mazes of similar size.

Program guidelines

Other than the required PathFinder class with method solveMaze (see above), the design of your program is completely up to you. Download [TestPathFinder.java](https://utah.instructure.com/courses/807886/files/140636658?wrap=1)[Download TestPathFinder.java](https://utah.instructure.com/courses/807886/files/140636658/download?download_frd=1) as an example of running your program. Read the comments in the code.

GUI tool

**The GUI 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. Download [PacmanApp.java](https://utah.instructure.com/courses/807886/files/140637223?wrap=1)[Download PacmanApp.java](https://utah.instructure.com/courses/807886/files/140637223/download?download_frd=1)  and add it to a **new java project** (for simplicity, don't mix it with your code).  To run it, edit the filename in its main method to point to one of your solved mazes.

Testing

The mazes I have provided will not guarantee that your algorithm works. Develop your own test mazes, or move the start and goal points in the provided mazes to convince yourself that your algorithm can find any path.

Finishing up

Your code should be well-commented (Javadoc comments are required) and formatted such that it is clear and easy to read. You must have at least: comments for every method you write, describing what the method does, what the arguments are (and what they mean), and what the return value is, as well as any special cases that the method handles or can run in to. You should also have comments on any line or block of code that is not self-explanatory.

Zip your source code files (no .class files, source code only) and submit them through Canvas. Be sure to include all files necessary for running your code. And, as always, don't forget your assignment.properties file.

And have fun!!!

Rubric

**Some Rubric**

| Some Rubric | | |
| --- | --- | --- |
| **Criteria** | **Ratings** | **Pts** |
| This criterion is linked to a Learning OutcomePasses autograder test cases | |  |  | | --- | --- | | **75 pts**  **Full Marks** | **0 pts**  **No Marks** | | 75 pts |
| This criterion is linked to a Learning OutcomeStudent provided tests have good code coverage | |  |  | | --- | --- | | **10 pts**  **Full Marks** | **0 pts**  **No Marks** | | 10 pts |
| This criterion is linked to a Learning OutcomeCode is easy to read/well commented | |  |  | | --- | --- | | **5 pts**  **Full Marks** | **0 pts**  **No Marks** | | 5 pts |
| This criterion is linked to a Learning OutcomeParticipated in code review | |  |  | | --- | --- | | **10 pts**  **Full Marks** | **0 pts**  **No Marks** | | 10 pts |
| Total Points: 100 | | |