CSE 1322L – Assignment 5 (Fall 2025)

Introduction

A path-finding algorithm is an algorithm which attempts to find a path between 2 points on some search space. Of the many pathfinding algorithms currently in existence, one of them is called “Depth-First Search”, or DFS. DFS works by exploring branches in the search space as deeply as possible, only backtracking once it hits a dead end, whereupon it will explore a different branch. This process repeats until all branches are explored or until the intended destination is reached.

As an example, suppose you enter a maze and you wish to find its exit. If you follow the DFS algorithm, upon reaching a split on the road, you always take the rightmost path. If you find another split, you again take the rightmost path. You must keep doing this until either of the following happens:

- You find the exit
- You reach a dead end
- You reach a spot you’ve already been to before

If you reach the exit, then you are done. In either of the other cases, you must go back to the previous split on the road and then follow the second rightmost path. If you’ve already visited that one, then you follow the third rightmost path, and so on.

You may know a variant of DFS which is sometimes called the “right hand rule”: if you enter a maze and you always walk with your right hand touching the right wall, you may eventually find an exit.

DFS is not guaranteed to find an exit if the maze is infinite, nor is it guaranteed to find the shortest path to the exit. However, if we assume a finite search space (such as a maze in a computer), then we are guaranteed to find an exit; even if it’s not the shortest one. DFS is usually implemented using a data structure called a “stack”. The call stack, which is the part of a program which keeps track of which methods have been called and in what order, is more than enough to implement DFS.

In this assignment, you will write a program which prompts the user for a maze, its starting point, and its end point. You will then write a method which uses DFS to determine if a path exists between both points and, if so, display instructions to the user on how to go from one point to the other.

The maze will always be square in shape, and it will contain only 4 different symbols:

- o: the starting point
- x: the ending point
- .: empty spaces
- +: walls

An 8x8 maze could look like this:

```
o..+++..

++.++..+

+.....+x

..+.+.+.

.+..+...

.+.++.++

.+..+.+.

.++++...
```

In the maze above, the starting point is at the top left, whereas the end point is on the last column of the third row. DFS can go through empty spaces as it searches for the ending point, but it cannot go through walls.

To store the maze information, you will use a 2D array. By convention, the first dimension will be used to store the row while the second dimension will store column. Thus, given the maze above:

```
maze[0][0] = “o”; //starting point
maze[2][7] = “x”; //ending point
```

Requirements

The features described below must be in your program:

- A class Tile with the following members:
  - A boolean field called “visited”
  - A String field called “symbol”
  - **Tile(String)**: utilizes the argument to initialize the “symbol” field. Sets visited to “false”
  - Getters for both fields
  - A setter for “visited”
- Your driver must have the following static methods:
  - **boolean validateRow(String, int)**: This method receives a String which represents a row in the maze, returning true if the row is valid and false

otherwise. A valid row contains either "." or "+" and its length matches the width of the maze; the second argument should be the maze's width.

  - If the length of the first argument does not match the second argument, return false. Otherwise, this method **recursively** checks each character in the first argument, returning true if it only contains either "." or "+" and false otherwise
  - **This method must be fully recursive**. Using any type of iterative loops (e.g.: FOR, WHILE, DO-WHILE), calling another method which solves this problem iteratively, or accessing any fields outside the method's scope **will lose you points**. (Arguments passed to the method are in-scope)

- **String pathfinder(Tile[][], int, int)**: This method uses **recursion** and Depth-First Search to find a path from the starting point to the end point. If a path is found, the method returns a string containing instructions on how to reach the goal from the starting point. If no path can be found, the method returns an empty string. The instructions are a sequence of arrows followed by the goal symbol. For example:

  →↓↓↓↓→→→x

  These instructions mean that, to reach the goal, you must go to the right once, down four times, and right 3 times. To print these arrow symbols, use the variables below in your program:

```
String leftArrow = "\u2190";
String upArrow = "\u2191";
String rightArrow = "\u2192";
String downArrow = "\u2193";
```

  The first argument of pathfinder() is the maze, whereas the second and third argument are the coordinates which pathfinder() is currently evaluating; they must be row and column, respectively.

  To achieve this, pathfinder() should do the following:

  - If the coordinates are out of the maze's bounds, return an empty string
  - If the coordinates point to a Tile that has already been visited or to a wall tile, return an empty string
  - If the coordinates point to the goal Tile, return "x"
  - Otherwise, set the current Tile as "visited"
  - Recursively visit another Tile. Tiles must be visited in the following order: Down, Right, Up, Left. If recursively vising a tile returns a non-empty string, return the appropriate arrow concatenated with the returned string. For example, assuming we are visiting the tile one row below the current one:

```
String down = pathfinder(maze, row + 1, column);
if(!down.isEmpty()) return “↓” + down;
```

- If none of the directions return a valid path, return an empty string
- <u>**This method must be fully recursive**</u>. Using any type of iterative loops (e.g.: FOR, WHILE, DO-WHILE), calling another method which solves this problem iteratively, or accessing any fields outside the method’s scope <u>**will lose you points**</u>. (Arguments passed to the method are in-scope)

- **void main()**: The driver must prompt the user for a valid maze using validateRow() and then run pathfinder() on it to figure out if a path exists between the starting and ending points.
  - Prompt the user for a maze size. Mazes must have a size of 2 or greater; reject all other sizes by prompting the user for a new size.
  - Create a 2D Tile array that is {size} x {size}, where {size} is the information the user just entered
  - Ask the user to enter a row in the maze. Use validateRow() with the string the user enters along with the {size} above to check if the user entered a valid row. Reject invalid rows by asking for a new row
  - Every time the user enters a valid row, create Tile objects with the symbols in the String that the user just entered and place them in the 2D array. For example, in a maze that is 5x5, assuming the user is entering row 0, if they enter the following string:

    ```
    “.+.+.”
    ```

    Create the following Tile objects:

    ```
    Tile one = new Tile(“.”);
    Tile two = new Tile(“+”);
    Tile three = new Tile(“.”);
    Tile four = new Tile(“+”);
    Tile five = new Tile(“.”);
    ```

    And place them at the following locations:

    ```
    maze[0][0] = one;
    maze[0][1] = two;
    maze[0][2] = three;
    maze[0][3] = four;
    maze[0][4] = five;
    ```

  - Perform the two steps above for each row in the maze, until the 2D array is completely full.
  - Print the current maze to the user (you may use a nested loop)

        - Next, the starting point must be placed. Ask the user in which row and column they wish to place the starting point on. If either point is out of the maze's bounds, ask the user again. Otherwise, create a Tile object with a "o" as its symbol and place it in the 2D array at the row and column the user specified
        - Next, the ending point must be placed. Perform the step above again with the following modifications:
            - The symbol must be "x" instead of "o"
            - If the user tries to place the ending point in the same spot as the starting point, ask the user for a new pair of coordinates
        - Print the maze one last time
        - Pass the 2D array and the coordinates of the starting point (row and column) to pathfinder()
            - If pathfinder() returns an empty string, tell the user there is no path from the starting point to the end point
            - Otherwise, print pathfinder()'s instructions

Deliverables

- Assignment5.java (driver)
- Tile.java

Considerations

- You will get partial credit for partial work, as long as the rubric permits it. If you yourself near the deadline and it's too late to ask for help, try to do as much as you can.
- Despite what the deliverables above say, you can submit all of your classes in a single file.
- **Recursion is a hard topic**. Start the assignment as soon as you can, so you can ask for help if you get stuck.
- Recall that every recursive solution has at least 1 base case and 1 recursive case
    - **Base case**: a case for which you already know the answer, which is simply returned.
    - **Recursive case**: a case which tries to simplify the problem, calling the method again with different arguments. These new arguments must progress the solution towards a base case.
- You will have to make use of String methods to code pathfinder() and validateRow(). You can find its documentation here.

Sample Output (user input in red)

```
[Maze Pathfinder]
Enter size of maze. Must be 2 or greater: -2
Enter size of maze. Must be 2 or greater: 0
Enter size of maze. Must be 2 or greater: 1
Enter size of maze. Must be 2 or greater: 8
Maze set to 8x8

Enter maze row by row. Row must be made of either walls (+) or empty
spaces (.) and must be of length 8
Enter row 0: ...+++..
Enter row 1: ++.++..+
Enter row 2: +.....+x
Row must be made of either walls (+) or empty spaces (.) and must be
of length 8
Enter row 2: +.....++
Enter row 3: ..+.+.+.
Enter row 4: .+..+...
Enter row 5: .+.++.+++
Row must be made of either walls (+) or empty spaces (.) and must be
of length 8
Enter row 5: .+.++.++
Enter row 6: .+..+.+*
Row must be made of either walls (+) or empty spaces (.) and must be
of length 8
Enter row 6: .+..+.+.
Enter row 7: .++++...

Here is your current maze:
. . . + + + . .
+ + . + + . . +
+ . . . . . + +
. . + . + . + .
. + . . + . . .
. + . + + . + +
. + . . + . + .
. + + + + . . .

Placing starting point
Place starting point in which row?: 0
Place starting point in which column?: 10
Invalid column: Must be between 0 and 7
```

```
Placing starting point
Place starting point in which row?: 0
Place starting point in which column?: 0
----
Placing ending point
Place ending point in which row?: 0
Place ending point in which column?: 0
Cannot place ending point on starting point
Placing ending point
Place ending point in which row?: 2
Place ending point in which column?: 7

Here is your final maze:
o . . + + + . .
+ + . + + . . +
+ . . . . . + x
. . + . + . + .
. + . . + . . .
. + . + + . + +
. + . . + . + .
. + + + + . . .

Press any key to run pathfinder...

To go from 'o' to 'x', follow these steps:
→→↓↓→→→↓↓→→↑↑x
```

Sample Output (user input in red)

```
[Maze Pathfinder]
Enter size of maze. Must be 2 or greater: 5
Maze set to 5x5

Enter maze row by row. Row must be made of either walls (+) or empty
spaces (.) and must be of length 5
Enter row 0: .....
Enter row 1: .....
Enter row 2: +++++
Enter row 3: .....
Enter row 4: .....
```

```
Here is your current maze:
. . . . .
. . . . .
+ + + + +
. . . . .
. . . . .

Placing starting point
Place starting point in which row?: 4
Place starting point in which column?: 4
----
Placing ending point
Place ending point in which row?: 0
Place ending point in which column?: 0

Here is your final maze:
x . . . .
. . . . .
+ + + + +
. . . . .
. . . . o

Press any key to run pathfinder...

There is no path from 'o' to 'x'
```

Sample Output (user input in red)

```
[Maze Pathfinder]
Enter size of maze. Must be 2 or greater: 11
Maze set to 11x11

Enter maze row by row. Row must be made of either walls (+) or empty
spaces (.) and must be of length 11
Enter row 0: ...........
Enter row 1: .+++++++++.
Enter row 2: .........+.
Enter row 3: ++++++++.+.
Enter row 4: .........+.
Enter row 5: .+++++++++.
Enter row 6: .........+.
Enter row 7: ++++++++.+.
Enter row 8: .........+.
Enter row 9: .+++++++++.
Enter row 10: ...........

Here is your current maze:
. . . . . . . . . . .
. + + + + + + + + + .
. . . . . . . . . + .
+ + + + + + + + . + .
. . . . . . . . . + .
. + + + + + + + + + .
. . . . . . . . . + .
+ + + + + + + + . + .
. . . . . . . . . + .
. + + + + + + + + + .
. . . . . . . . . . .

Placing starting point
Place starting point in which row?: 0
Place starting point in which column?: 10
----
Placing ending point
Place ending point in which row?: 0
Place ending point in which column?: 9

Here is your final maze:
. . . . . . . . . x o
. + + + + + + + + + .
. . . . . . . . . + .
+ + + + + + + + . + .
. . . . . . . . . + .
. + + + + + + + + + .
. . . . . . . . . + .
+ + + + + + + + . + .
. . . . . . . . . + .
. + + + + + + + + + .
. . . . . . . . . . .

Press any key to run pathfinder...

To go from 'o' to 'x', follow these steps:
```

↓↓↓↓↓↓↓↓↓↓←←←←←←←←←←↑↑→→→→→→→→↑↑←←←←←←←←↑↑→→→→→→→→↑↑←←←←←←←←↑↑→→→→→→→→
→x