Programming Assignment #6: Pathogen

COP 3503, Spring 2020

Due: Sunday, April 12, *before* 11:59 PM

Abstract

In this assignment, you will determine all safe and valid paths to an exit in an arbitrary ASCII maze that is potentially riddled with 19-nCoV (the coronavirus). You are required to take a backtracking approach to this problem, which means that by completing this assignment, you will attain a deeper understanding of the general structure of recursive backtracking algorithms.

This program requires you to modify the maze backtracking code I presented in class, and so it will also give you some experience reading, comprehending, and modifying a fairly sophisticated piece of code that someone else has written – something you'll be doing quite a lot if you end up pursuing a career in software engineering.

As you begin to work on this assignment, you might find it helpful to refer to my backtracking notes in Webcourses. In particular, see the section titled, "Basic Anatomy of a Backtracking Algorithm."

Credit goes to Marty Power, one of my long-time TAs, for proposing this as an assignment idea (although I sprinkled the 19-nCoV on top). Thank you, Marty!

Deliverables

Pathogen.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Problem Statement

In this assignment, you will write a function that takes in a 2D char array representing an ASCII maze and returns a HashSet of all possible paths from a person's starting position in the maze to its exit.

Movement through these mazes is restricted to four directions: up, down, left, and right. All the mazes we pass to your backtracking function in this program are guaranteed to abide by the following restrictions:

- The 2D char array will be a rectangle (which means that all rows in the array will have the same length).
- The 2D char array passed to your backtracking function will contain only the following characters:
  - '@' – This character denotes the person in the maze. There will be exactly one of these characters in the maze (no more and no fewer), and it could appear anywhere in the maze.
  - 'e' – This is the exit that the person in the maze is trying to reach. There will always be exactly one exit in the maze (no more and no fewer), and it could appear anywhere in the maze.
  - '#' – This character denotes a wall. The person in the maze cannot step on one of these spaces.
  - '*' – This character denotes the coronavirus. The person in the maze must not walk on these spaces.
  - ' ' – This character denotes a space where the person in the maze is allowed to walk.

There are no guarantees regarding the number of walls in the maze or the locations of those walls. It's possible that the border of the maze might not be made up entirely of walls (i.e., it might contain spaces), in which case, it's up to you to make sure the person in the maze does not go outside the bounds of the 2D array. Note also that the initial positions of the '@' and 'e' characters may vary from maze to maze. The coronavirus might not be present in every maze, and if it is, it could occur in multiple locations or just one.

2. Path Format

Consider, for example, the maze shown below (far left diagram). For the purposes of this assignment, there are exactly two valid paths from the '@' character to the exit:

```
###########
# #@    *##
#   # ## ##
# ### ## ##
#    e   ##
##### ## ##
#####     #
###########
```

Original Maze

```
###########
# #.    *##
#...# ## ##
#.### ## ##
#.....   ##
##### ## ##
#####     #
###########
```

Solution #1

```
###########
# #...  *##
#   #.## ##
# ###.## ##
#    .   ##
##### ## ##
#####     #
###########
```

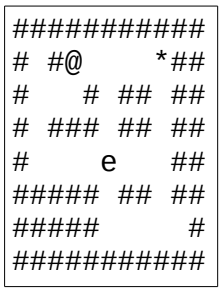
Solution #2

When creating a string to denote a path, we indicate each directional movement from the person's starting position using a single character: 'u' (up), 'd' (down), 'l' (left), or 'r' (right). The characters in each string must be separated with a single space. There should be no spaces at the beginning or end of the string. For example,

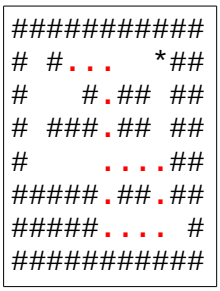
the strings representing the two paths above are as follows (and should be placed into a HashSet to be returned by the wrapper method that calls your backtracking method):

"d l l d d r r r r"
"r r d d d"

Note that you should never produce a path that causes the person in the maze to step on an instance of the coronavirus. Also, the person in the maze can never step on the same square more than once. For example, in the maze shown above, it might be tempting to pass over the exit and journey around the loop in the bottom-right corner of the maze before landing on the exit again. That would get the person from the starting position to the exit, but that path would be invalid because it would cause the person in the maze to touch some position twice (namely, the position with the exit): once when passing over the exit the first time, and once when the person finishes the given path. This invalid path is denoted by the following red breadcrumb trail:



Original Maze



Invalid Path

3. Method and Class Requirements

Implement the following methods in a class named `Pathogen`.

`public static HashSet<String> findPaths(char [][] maze)`

This method receives a 2D char array representing an ASCII maze and returns a HashSet of strings representing all safe and valid paths from the starting position ('@') to the exit ('e') using the format described above. The *maze* array will be non-*null* and non-empty and is guaranteed to be well-formed according to the guarantees given in the problem statement above. If there are no solutions, you should return an empty HashSet (rather than returning *null*). Please note that when your method returns, the *maze* array must be in the same state that it was in when this method was called initially, without any changes.

`public static double difficultyRating()`

Return a double on the range 1.0 (ridiculously easy) to 5.0 (insanely difficult).

`public static double hoursSpent()`

Return a realistic estimate (greater than zero) of the number of hours you spent on this assignment.

4. Special Requirements (*Super Important!*)

You must directly modify the backtracking code I've distributed with this assignment. In doing so, you should not drastically re-factor the code (i.e., you should not change it to be more object-oriented or to take a radically different backtracking approach). One of the purposes of this assignment is to force you to delve into an existing solution and figure out how it works, and the code you submit should reflect that.

Constructing the output string for each path should be a linear operation. So, string concatenation is strictly forbidden in this assignment. Each time you add a direction to your running path, it must be an O(1) operation. When it comes time to add a string to your HashSet, generating that string must be an O(k) operation (where *k* is the length of the string). To accomplish that, you might want to read up on Java's StringBuilder class.

The solution you submit must use a recursive backtracking algorithm. (You cannot submit a non-backtracking solution for this problem.) The algorithm you submit must backtrack when an obviously infeasible state is reached, rather than continuing to add to a path that is already guaranteed to be invalid, such as the one denoted by the red breadcrumb trail above.

Also, the algorithm you submit must be written in such a way that it generates any given path through the maze *at most* once. If it generates a particular path multiple times, the HashSet can certainly help get rid of those duplicates, but your runtime will necessarily suffer for having produced the same path repeatedly. Be sure to avoid this.

5. Style Restrictions (Same as in Program #1) (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed ***above***

the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

6. Compiling and Testing on Eustis (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of my test cases:

```
javac Pathogen.java TestCase01.java
```

2. To run this test case and redirect your output to a text file:

```
java TestCase01 > myoutput01.txt
```

3. To compare your program's output against the sample output file I've provided for this test case:

```
diff myoutput01.txt sample_output/TestCase01-output.txt
```

   If the contents of *myoutput01.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

   Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

4. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *Pathogen.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

7. Grading Criteria and Miscellaneous Requirements

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

100%	Passes test cases. Your program must use backtracking in order to be eligible for credit. A non-backtracking solution will not receive credit, even if it passes all test cases. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.

Important Note! Additional point deductions may be imposed for poor commenting and whitespace. Significant point deductions may be imposed for violating the style restrictions listed above. You should also still include your name and NID in your source code.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! Programs that do not compile on Eustis will receive zero credit. When testing your code, you should ensure that you place *Pathogen.java* alone in a directory with the test case files (source files, the *sample_output* directory, and the *test-all.sh* script), and no other files. That will help ensure that your *Pathogen.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

Important! You might want to remove *main()* and then double check that your program compiles without it before submitting. Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! Your program should not print anything to the screen. Extraneous output is disruptive to the grading process and will result in severe point deductions. Please do not print to the screen.

Important! No file writing. Please do not read or write to any files from *Pathogen.java*.

Important! **Please do not create a java package.** Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! **Name your source file, class(es), and method(s) correctly.** Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to implement a required method, or failing to make certain methods public, private, static, and/or non-static (as required), may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. Please reflect carefully on the kinds of edge cases that might cause unusual behaviors for any of the methods you're implementing.

Test your code thoroughly. Please be sure to create your own test cases and thoroughly test your code. You're welcome to share test cases with each other, as long as your test cases don't include any solution code for the assignment itself.

Start early! Work hard! Ask questions! Good luck!