CS 455 Programming Assignment 3

Fall 2016 [Bono]
Due: Wed., Oct. 26, 11:59pm

Introduction

In this program you will get a chance to use recursion to solve a problem that could not be done just as easily or more efficiently with a loop. It is possible to solve this problem without recursion, but it would somewhat more complicated, and would result in the same big-O space and time requirements as a recursive solution.

Backtracking algorithms for solving a problem are ones that attempt to complete a search for a solution by constructing partial solutions. The algorithm then attempts to extend a partial solution toward completion, but if it gets stuck, the algorithm backs up (backtracks) by removing the most recently constructed part of the solution, and trying another possibility. Backtracking problems lend themselves well to recursion. In this program you'll use backtracking to try to find a path through a maze. To make it a little easier, we're giving you the outline of the algorithm to use in this document--if you want an extra challenge, try doing it without looking at the pseudo-code in the section on Recursive search algorithm.

In addition to recursion, some other recent class material that will be used in this assignment are the `LinkedList` class and file reading.

The code for this assignment uses a few new features of Java. They are: 2D arrays and inner classes (we already wrote the inner class code for you). Two-dimensional arrays were covered in section 7.6 of the textbook. Inner classes are covered in section 10.5, and their use for Listeners (part of the Java GUI system) is covered in section 10.7.2. Event handling in general is covered in Section 10.7, but we wrote all the event-handling code for you.

The assignment files

Getting the assignment files. Make a pa3 directory and cd into it. The following command will copy the necessary files to your current directory:

```
gmake -f ~csci455/assgts/pa3/Makefile getfiles
```

Note: the blurbs below do not describe what each of these classes are, and how they fit together. For more details on that, see the section on the class design.

The files in **`bold`** below are ones you create and/or modify and submit. The files are:

- **`Maze.java`** The interface has been specified for you; you need to complete the implementation of this class.
- **`MazeComponent.java`** We have written part of this for you, including the interface; you'll need to complete the implementation.
- **`MazeViewer.java`** You'll need to add the file processing code (`readMazeFile` method) to this

program.

- `MazeFrame.java` We have written this class for you. Do not change it. It creates the other major objects used in this program.
- `MazeCoord.java` We have written this class for you. Do not change it.
- **`MazeTester.java`** A test program to test your Maze class. You are not required to submit this file. This is described further in the section on Development Hints.
- `testfiles` A directory with some test data for your program. You will need to test your program on more than just these files.
- `Makefile` This has the rules for downloading and submitting the files. These rules are input to the `gmake` program. There are directions below on how and when you'll need to call gmake.
- **`README`** See section on Submitting your program for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

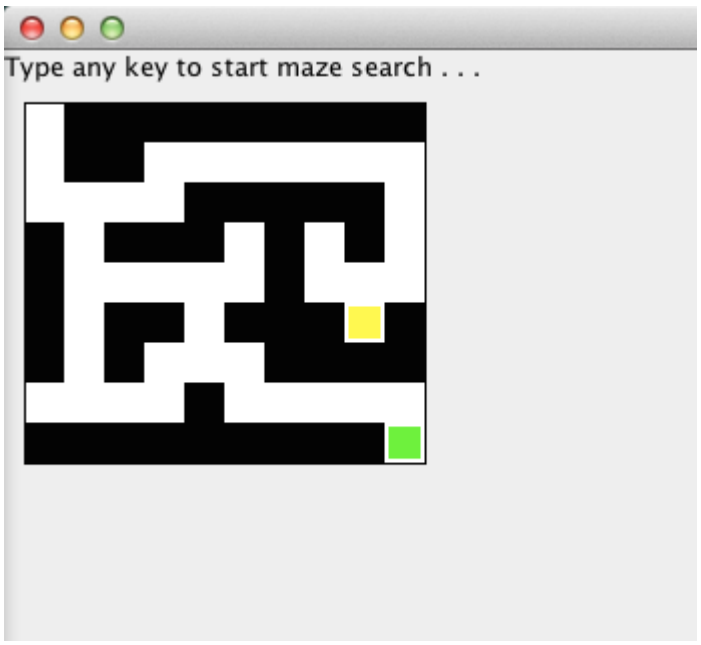
The assignment

Write a program that takes as input (from a text file) a maze with the following format:

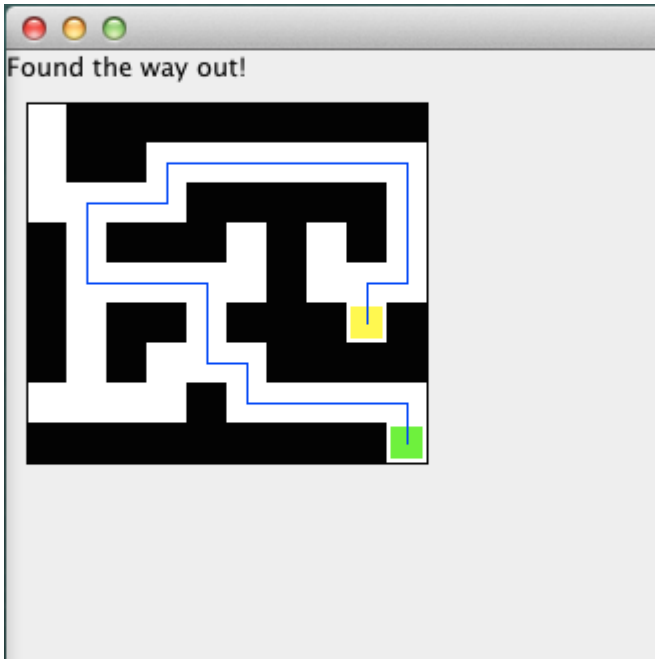
```
9 10
0111111111
0110000000
0000111110
1011101010
1000001000
1011011101
1010001111
0000100000
1111111110
5 8
8 9
```

The first line gives the total size of the maze to come; this is a maze with 9 rows and 10 columns. On the following lines, a 1 in a particular position means that there's a wall at that position. A 0 in a particular position means that that position is free. The last two lines of the file are the maze coordinates for the entry location, followed by the maze coordinates for the exit location. (Note that these maze coordinates are 0-based, just like Java arrays.)

Your program will read a maze, such as the one above, from a file whose name will be given on the java command line, and initially display it to the graphics window as follows (entry location shown in yellow and exit location shown in green):


Type any key to start maze search . . .

Then, when the user hits any key, it will display the maze with a path through it shown in blue, or just the original maze if there is no possible path from the entrance to the exit. It will also give some indication of whether there was a path (see the message in the upper-left corner of the window below). Here's example output for the maze above:


Found the way out!

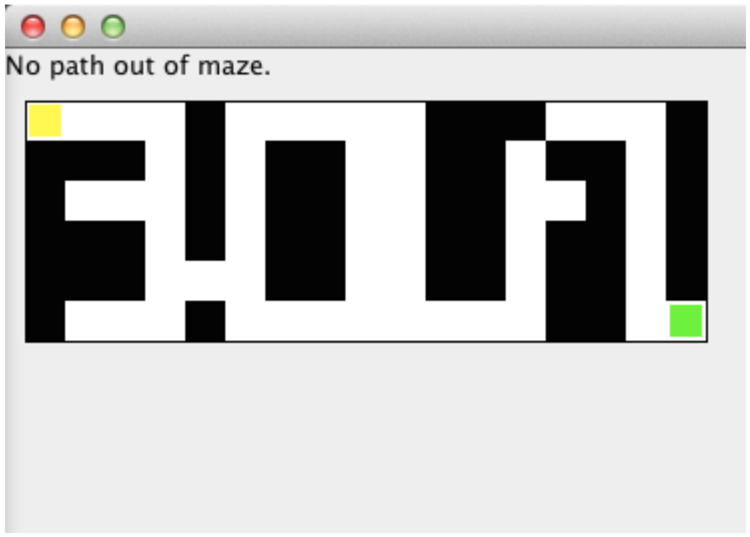
In general, a given maze may have more than one path through it. Your program just has to find one of them. Note: the above data file is available as `testfiles/medMaze`.

Here is an example of input and the corresponding output for a maze with no escape (i.e., no way to get from the upper left corner to the lower right corner.

Maze data (also available as file `testfiles/mazeNoPathCycle`):

```
6 17
00001000001110001
11101011001101101
10001011001100101
11101011001101101
11100011001101101
10001000000001100
0 0
5 16
```

Graphics window after doing the search:


No path out of maze.

Note the different message at the top of the window. You can see from this example that one cannot travel between diagonally adjacent non-wall positions; only up-down or left-right.

We are giving you the overall design to use, as well as some starter code, so you should read on before you dive into this problem.

What if a *wall* is the exit location?

The filled-in squares denoting the exit and entry locations (i.e., the green and yellow blocks) are intentionally a little smaller than the other blocks. This is so you can see whether this location was also a wall or not (because you'll be able to see the black or white around the edges). It is legal to give such an input file (there are examples of these cases in test files `medWallAtExit` and `medWallAtEntrance`). If the entry or exit location is a wall, you won't be able to find a path though that maze.

The Class Design

We have done the class design for you. Normally in a GUI program it's desirable to separate the part of the program that does the computation and stores data from the part of the program dealing with the display and input. This way it's easier to change the look and feel of the program without having to touch the code that does the computation (e.g., we could use our `Maze` class below, unchanged, with a non-graphical console-based interface). The design given here has that separation.

This is an overview of the design, stating which classes are responsible for which part of the program, and what the dependencies are between the various classes.

- **`MazeViewer`** class. This class has `main`. It reads in the maze data and sets up the graphics window with the `MazeFrame` (passing the maze data to `MazeFrame`). It depends on the `MazeFrame`.

- **`MazeFrame`** class. This `JFrame` subclass contains all the display elements and listeners in the program. The main elements are the `JLabel` for displaying the prompt and other messages, the `MazeComponent` for displaying the maze and path, and the listener to get keyboard input from the user. It creates the `Maze`, `MazeKeyListener` , and `MazeComponent` objects. (So it depends on those three other classes.) We wrote this class for you.

- **`MazeComponent`** class. This `JComponent` subclass displays the maze and the maze path. (It should look like the maze shown in the screenshots earlier in this document.) It gets all the information about what to display from the `Maze` object. Depends on `Maze` and `MazeCoord`.

- **`Maze`** class. This class has the data about a maze, and knows how to do a maze search. It doesn't do any I/O. Depends on `MazeCoord`.

- **`MazeKeyListener`** class. It has the code that responds to a user's keystrokes. It's created in the `MazeFrame` constructor. It's an inner class of `MazeFrame`, so it depends on `MazeFrame`. We wrote this class for you. Listeners are described in a little more detail below.

- **`MazeCoord`** class. Tiny immutable class for storing coordinates for a maze location. (The code is completed for you.) This way we can store a maze path as a single `LinkedList` object, where each element is a `MazeCoord`. In addition to accessors, and `toString`, it also supplies an `equals` method. Doesn't depend on any other classes.

What's a listener?A listener is a GUI component that has a function that gets called whenever a certain user action happens (e.g., a mouse click or a key pressed). It's usually associated with a particular component: e.g., the component might be a button, and the listener has the code that gets executed when that button is pressed). The application programmer fills in the body of this function for a subclass of the appropriate Java listener class.

What happens when we run `MazeViewer`?

We have written the top-level code for you. This explains the sequence of events that happen when we run the program.

`main` will read in the maze data from a file specified on the command line. It also sets up the frame (like we did in pa1). Once the frame is set up, the Java GUI system will call paintComponent, which should in turn display the maze (with no path at this point).

The frame also displays a prompt for the user to type a key to initiate the maze search. Once the user types the key, the code inside the `keyPressed` method in `MazeKeyListener` gets called.

That code in `keyPressed` calls the `Maze search` function, and once `search` returns, it then calls `repaint`, which triggers a call to `paintComponent`, which will display the maze again, this time with the path if there was one. The code also updates the message displayed to let the user know whether the search was successful. You should take a look at the `keyPressed` method, which is near the bottom of `MazeFrame.java`.

Again, the top-level control structure has already been written for you. The only part of the `main` class (i.e. `MazeViewer.java`) you will have to write is the code to read in the file. We called and wrote the method header for a method `readMazeFile` that you'll have to implement to do this. The rest of the code you'll be writing will be the implementations of the `Maze` and `MazeComponent` classes.

Maze: interface

You are required to have the following interface for the `Maze` class, which will store the internal data for a maze and a path through it, and support the following operations (the starter file for this is `Maze.java`). The parts in square brackets describe where how/where that method will be used.

`Maze(boolean[][] mazeData, MazeCoord startLoc, MazeCoord endLoc)`
: constructs a maze from the data given parameters. More details about these parameters in the `Maze` class comments, and the constructor method comments. [The code to *call* this constructor has already been written for you in the `MazeFrame` constructor.]

`boolean search()`
: searches for a path through the maze from the start position to the end position, and saves the path it finds if it finds one. Client can access the path found via getPath method. Also returns whether or not it found a path. [The call to `search` has already been written for you in the `MazeListener`.]

  Subsequent calls to `search` just return whether a path was found. I.e., the path found the first time will be saved.

`LinkedList<MazeCoord> getPath()`
: Returns the path found, or an empty list if we haven't yet called `search()` or if there is no path through the maze. [The `MazeComponent` will use this to get the path to display.]

`boolean hasWallAt(MazeCoord loc)`
: Tells us whether there is a wall at this location. [The `MazeComponent` will use this to get information about where the walls are.]

`int numRows()`
`int numCols()`
: The number of rows and number of columns in the maze, respectively. [The `MazeComponent` will use this.]

`MazeCoord getEntryLoc()`
`MazeCoord getExitLoc()`
: The entry and exit locations of the maze, respectively. [The `MazeComponent` will use this.]

Note that a `MazeCoord`, is a (row,col) pair. This is different than the coordinate system used in Java windows. Besides the units being different, the row (first number) increase vertically, and the columns (second number) increase horizontally. Put another way, in the display, rows change in the y direction and cols change in the x direction.

Maze: representation/implementation

We're leaving it up to you exactly how to represent your maze inside the Maze object. You will probably use some kind of two-dimensional array or arrays. Some of the design choices are what type of elements should be in the array (ints, chars, booleans), whether you want to pad it with virtual outer walls (more about that in the next paragraph), and whether you want a separate array for storing which part of the maze you have already searched (more about that in the next section). You will also have an instance variable for the path, which gets created by `search` and can be accessed with `getPath`.

When visiting different elements in your 2D array to find a path you might constantly have to check that you're not trying to access a position outside the bounds of the array. An alternate way to handle this issue is to pad the matrix with phantom walls all around it. In this scenario, the matrix you would create would not be the exact size given by the user, but be one bigger on all sides, where the "phantom" locations are all walls, so you'll never create a path through that part. Note: choosing this implementation tactic should not affect how the class operates for the client, since the specification for the class will not have changed. For example, it doesn't change the valid MazeCoord values we can give to `hasWallAt`, or `numRows()` and `numCols()` for a particular maze. and the Maze that is displayed is the input maze, not one with extra walls. You are not required to use this padded array idea.

Whatever representation you choose, don't forget to document it with a representation invariant comment.

Detecting cycles

A maze may have a cycle, that is, a way to loop back to a place we've already been while searching for a path. We would end up with infinite recursion if we don't detect that we have already visited a location before. There are a few ways to handle this situation in our code. One way is to use another 2D array `visited` that keeps track of which positions we've already visited in our search for a path, so we don't go around the cycles more than once. Another way, a variation on the first, involves keeping that information in the "maze" 2D array itself, by storing a special value at locations we have already visited.

Recursive Search Algorithm

This kind of problem is called a *search* problem, even though the meaning here is a little different than when we talked about finding a particular element in a collection (where we used algorithms such as linear or binary search).

You are required to use recursion to solve this problem. The top-level method is called `search()`. It will call a helper routine (private method) which does the actual recursion. Thus the top-level routine can do any initialization or cleanup that is necessary (only once). The helper routine will search from a particular location. In the pseudo-code below, we assume we have a routine which searches from a location given by row, column coordinates (r,c). if `search(r,c)` returns true, it means there is a free path from position (r,c) to the end position, o.w. it returns false.

```
*********** search (r, c): ***********

BASE CASES:

if (r, c) is a wall -- return failure
if (r, c) has already been visited -- return failure
    (i.e. that means that we think (r,c) is already on an earlier part
    of the path; if we visit it again we'll get into an infinite "loop")
if (r, c) is the final position -- then make (r,c) part of the path and
```

```
    return success

RECURSIVE CASE(S):

mark (r,c) as visited

for each position, r',c' that is adjacent to r,c:
    try to find a path from (r',c') to the end (i.e., make a recursive call).
    if that search was successful
        make (r,c) part of the path, and return success
    else try the next neighbor

if we got through the loop without finding a path, that means that
there is no path from (r,c) -- return failure.
```

Hint: one of the specifications of the Maze class is that the beginning of the path linked list (i.e., what's returned by `getPath()`) is the start of the path found, and the end of the linked list is the end of the path. If you don't write your search code carefully you may end up with the reverse path instead.

Data Files

You may assume that the data is in the correct format, as shown earlier in this document. We will put a few data files in the directory

```
~csci455/assgts/pa3/testfiles
```

Your program will get the name of the maze file from the command line. We have provided the code to get that string from the command line (the `args` argument to `main` has command line arguments). Here are a few examples of how to run the program:

```
java MazeViewer smalldata
java MazeViewer ~csci455/assgts/pa3/testfiles/bigmaze
```

Note: if you are running your program from Eclipse, you set command-line arguments by going to `Run Configurations` on the `Run` menu. Then go to the tab `Arguments` in the dialog box. In the first text box you put the file name. You can change what is the default directory it looks in for that file by changing the Working directory to something other than the Default (set this near the bottom of the dialog box).

The file reading method, `readMazeFile` (defined in `MazeViewer.java`), can throw an `IOException`. Your code will throw `FileNotFoundException` if you attempt to read a file that doesn't exist. (We wrote the handler for this exception for you in `main`.) This is a subclass of `IOException`, but we used the more general class in the `readMazeFile` header so that you wouldn't have to change this interface if you decided to use exception handling for other purposes, namely to detect malformed files. However, you are not required error-check the file format.

Development Hints

You are welcome to add any debugging print statements to be sent to `System.out`, and leave them in for your final version. Make sure to preface each such line with the string "DEBUG: ". The only use of System.out in the current program is for error messages.

You are also encouraged to write any private helper methods that will make your code clearer. (That's true for any program for this course.)

Here is a sequence of tasks for how you might develop your code:

1. Get the constructor and accessors for the Maze class working before you write or test the search function. You can see that they are working by adding a toString method (the only change to the interface that we will accept), and a MazeTester to use it to create several Maze objects whose data come from hard-coded arrays.

2. You could then proceed to implement your MazeComponent class, so that you can see that you are displaying your maze correctly. You can test this with the existing MazeViewer program, which will necessitate your implementing the file reading part now too, or instead you can test it with hard-coded data (described in more detail in the next step).

3. To test your maze display before writing your file-reading code you can create a very small hard-coded maze 2D array that you can pass into the Maze constructor (and you could have different such maze arrays that you swap in by modifying the code). To fit in with the current structure of the program, to do this you would need to make a new version of MazeViewer, because the maze array data comes from MazeViewer: MazeViewer passes the maze data into MazeFrame (which in turn creates the Maze object from it). So the new version of MazeViewer would use the hard-coded maze data instead of data from a file.

4. You could next implement the file-reading, to make it easier to test your search method on various mazes later. Make sure that these mazes you read are display correctly.

5. You could then implement the search method. It's easier to see visually if it's doing the right thing in the context of the graphical program, vs printing out the path as a sequence of locations in a MazeTester program. Also, because in general there are multiple valid paths, you won't know the exact expected output for a search (i.e., to compare with the actual results in a MazeTester). For example, you can get different paths depending on the order that you visit the neighbors. Thus your friend's program might get different results than yours does for a particular maze. However, you could write MazeTester code that tests that the path is valid, that is, that it doesn't go through walls, it only moves in certain directions, and it starts and ends in the right places.

   When debugging your search function use *very* small mazes.

6. You should also use your MazeTester to test creating multiple Maze objects and making multiple calls to the various methods in different orders. Once the constructor is called there is no restriction on the order you can call the other methods.

7. For the case of calling search twice on the same maze, you test this by running the existing MazeViewer program, and typing another key on the keyboard after the first search has been completed. The program should show the same results as when you did the search the first time.

8. Similarly, you can test multiple calls to the accessors by resizing or iconifying and deiconifying the window, since paintComponent should be calling them every time that happens. (Note: unlike pa1, this GUI displays its output fixed-size; it does not resize the diagram when the window resizes.)

README file / Submitting your program

You will be submitting completed versions of Maze.java, MazeComponent.java, MazeViewer.java, and README. Make sure your name and loginid appear at the start of each file.

Here's a review of what goes in the README: This is the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the the program you are submitting. You can also use the README to give the grader any other special information, such as if there is some special way to compile or run your program. You will also be signing the certification shown near the top of this document.

If you have downloaded all the files using the command shown near the beginning of this file, you should have a file called `Makefile` in your directory. You can submit your assignment with the command:

```
gmake submit
```

from an aludra directory containing your solution and the Makefile.
