Assignment 02 – Freak on a Leash

*Authors: Andrew Molony and Amy Eddins*

2/9/11

**Abstract:** The overall design for the Freak on a Leash project will be covered, as well as explanations for the coding of individual methods, alternative strategies, justification for choosing the strategy put into place based off of relative trade-offs, and how clients use the methods. The project itself consists of a program that can solve mazes using both depth-first and breadth-first testing strategies, as well as a driver program. Special attention will be paid to design of the project, with broad-based explanations of class construction and usage being offered.

1. **– Design Problem**

The design team was presented with the task of creating a program that could explore a maze using both depth-first and breadth-first strategies. The program must be robust, meaning that it should be reusable. Efficiency of the program should be taken into account for both strategies, either by reducing the bigO or by increasing the accuracy of the searching method (thus reducing actual time for completion of code).

A driver program must be included. A file directory of mazes will be given, and the driver program must successfully read the mazes from the directory, and use both depth-first and breadth-first strategies to solve them. Maze files will be proceeded by the size (in row x column format) on the line above, and all lines below the maze are comments which should not be read in by the driver. Mazes will follow the format “x” for a closed space, “-“ for an open space, “S” for the start, and “F” for the finish (case-sensitive). When “exploring” the maze, the program should turn any space that has been examined into a “.”, unless the space is the starting point or finishing point.

The output of the program should show the original maze, as well as the solved maze, with the correct path displayed with “o” indicators through the maze. The output should include the original maze, output for both strategies, the number of cells explored, and the number of steps to the correct point. The finish point, but not the starting point, is included in the number of steps that is to be outputted. Output should be repeated for each maze in the directory. Formatting should be clear and easy to read. Also, output should be provided displaying the correct path, in coordinate form, from the starting location to the finish location. Each coordinate should be connected to the following coordinate by an arrow, such that the path follows the form (0,0) 🡪 (0,1).

1. **– Class Design Alternatives**

Many solutions were considered during design of the project. It was generally accepted that the design team would have the following classes: Depth, a class that would implement a depth-first testing methodology, Breadth, a class that would implement a breadth-first testing methodology, and the driver program, called MazeSolver. This general program structure was easy to determine. Each method implementation would be completed by a different member of the design team, and their strategy for implementation would differ slightly.

**2.1 – Depth Class**

Very few alternatives for the Depth class were considered. It was viewed as a fairly straight-forward pseudo-code implementation, although it was immediately understood that the actual implementation of code would be inherently difficult. However, there were some alternatives as to how the code should be implemented.

The class would need a stack variable that would keep track of the path taken as a stack, as is characteristic of all depth-first testing strategies. Row and column variables may also be necessary, in order to have an easily-referenced variable representing the total number of rows and columns. Variables for keeping track of the amount of steps taken to the finishing point, as well as the total number of cells explored, would be necessary per design description (see Section 1.0). A variable representing the current position was also determined to be necessary, so that the current position could easily be referenced within methods where the current location was important to determining the direction to be taken. A starting position variable would also more than likely be necessary, as the starting position would have to be calculated by the program, and therefore must be held in variable form. Locations would be represented by Arrays with two values, one for row and one for column.

The class would naturally need a method that would search the maze for the starting location. This method would be utility method – the user does not need access to this method. A method that would explore the maze for the correct path to the finish using depth-first testing would be necessary in order for the class as a whole to be useful. It was immediately agreed upon that the simplest way for completing design instructions (Section 1.0) would be to implement a method that would complete the maze exploration, marking all explored cells with a “.”, and output a stack with the correct path. This method would be called by another method that would then mark the correct path with “o” symbols and output the maze solution as a string. The maze exploration method could potentially need auxiliary methods that tested for whether or not it is possible for the current position to move right, left, up, or down. However, this could also be implemented within the method itself, if necessary.

It was also a requirement to provide the user with a “path” from start to finish in coordinate form, of the form (0,0) 🡪 (1,1) 🡪 (1,2) etc. In order for this to be accomplished, some sort of collection or array would be necessary in order to hold the values of the stack so that they would not be lost when “popped” off to determine the path. The values would be returned as a String.

**2.2 – Breadth Class**

Besides the BreadthSearch class two separate classes were created to implement the BreadthSearch class. One called the Maze class, which represents a maze by a double string array, and a Position class which contains the position of the y and x coordinates on the maze and a position object representing the previous position on the maze. It was considered to only create a separate maze class and save positions from that maze double array, but after the position class was considered this was decided to be the best design decision because the Position class could be used separately to hold integers, be used to add and subtract positions, and hold links to the previous position so as to print out the solution in the end. The Maze class could be kept separately to save the maze as a string and change the string input as needed while going through the Breadth search.

1. **– Design Decision and Implementation of Code**

**3.1 – Depth Class**

Implementation of the Depth class was fairly complex. The variables established in the design alternatives section (Section 2.1) were all necessary. A depthScan(String[][] mazeIn) method was implemented. This method took a two-dimensional string array as the parameter. This string array was the maze itself. The code would first determine the starting location by calling a startSearch(row, column) method, where the parameters were the numbers of rows and columns. This method loops through the array until it finds a location with a string equal to “S”, and then returns the location as an array of form [row, column]. If no location is found, it returns an array of [-1, -1].

The depthScan method then loops through the maze, starting at the start location. It first tests to see if the current location is marked by an “F” – if so, exit the loop and return the stack, as the search is completed. Then it tests to make sure the stack isn’t empty – if the stack is empty, there is no solution, so return the empty stack as an indicator of this. It then tests to see if it is possible to move right, then down, then left, then up. If any of these steps is possible, it takes the step in the appropriate direction, and marks the location in the maze with a “.” (if it is not marked by an “F”), as well as sets the current location to the appropriate location. If none of these steps is possible, it takes a step backwards by popping the top location off the stack, and setting the current location the location underneath it. These tests are done by calling auxiliary methods, which each function similar by testing the appropriate location (either right, down, left, or up) to see what string it contains. If it contains a passable string, the method returns true, indicating that it is possible for the current locator to move one step in the appropriate direction.

The solutionPath() method has simplistic design. It takes the ArrayList object created by the DepthScan method, and loops through the objects contained within it. Each object is inserted into the proper format, so that the output string is in the desired format (see Section 2.1). This String output is then returned to the user.

Taking a macro view, this method will return a correct path to the Finish point, although the path itself may not be the fastest path possible. The depthScan method will actually be called by the depthSearch method, which has the same parameters. The depthSearch method will use the stack result returned by the depthScan method to go through the maze and mark the correct path with “o” indicators, and return the maze to the user as a String with several lines, as opposed to a double array. This was done to increase ease of method implementation in the driver class, so that the implementing class would not need to loop through the result to print it to standard output, but could simply call the method.

**3.2 – Breadth Class**

The search method in the BreadthSearch class uses and while loops that searches right, down, left, and up, while adding each movable spot to the queue. At the end of the loop it pulls from the queue if the spot is not movable. The loop ends when the finish spot is found and all throughout the loop it is saving positions into a solution array. The setSolution method starts at the finish position in the solution array and tracks back getting the previous positions; therefore, saving only the correct solutions into a separate array.

1. **– Implementing the Classes**

A representation of a maze as a double String array is necessary for implementation of the depth class. “Open” cells, or cells that are passable, should be denoted with the symbol “-“. “Closed” cells, or cells that are impassable, should be denoted with the symbol “x”. Starting and Finishing locations should be denoted with “S” and “F”, respectively. All symbols are case sensitive. An example of a maze is given (not in String[][] array form).

Maze: S – x F  
 - - x -  
 x - - -  
 - - x x

The constructor for the Depth class lacks parameters. To return a string object that represents the order of the solution path, call depthSearch(String[][] maze), where the parameter is identical to the depthScan method. A correct path to the finish location, if possible, will be returned. All cells that were examined will be marked with a “.” symbol, and all cells that are part of the correct path will be marked with a “o” symbol. The following output would be returned if the previous String[][] object was given as a parameter.

Output: SoxF  
 -oxo  
 xooo  
 --xx

Following a call to the depthSearch method, the amount of locations explored can be returned by the numExplored() method as an integer. Also, the amount of steps taken to reach the finish can be returned by the steps() method. If the depthSearch method has not yet been called, or it has been called, the steps method will return 0.

The solutionPath() method returns the correct path to the solution spot as a String. If there is no correct path to the solution (correct path does not exist), an empty string will be returned. The depthSearch method must be called prior to calling the solutionPath method, else incorrect output will be returned.

The MazeSolver driver loops through each file and prints out the solutions using both search methods. The number of steps in the solution does not count the start position.

Example output (Breadth-first search):

Maze file: maze1.txt

Original maze:

Sx-xx

---xx

-x---

-x-x-

x--xF

Explored maze:

Sx.xx

oooxx

.xooo

.x.xo

x..xF

Number of cells explored: 14 (56%)

Solution:

(0,0)->(1,0)->(1,1)->(1,2)->(2,2)->(2,3)->(2,4)->(3,4)->(4,4)

Number of steps in solution: 8