Avery Anderson
4/30/17

Final Project: The MiniMouse Maze Runner

In this project I set out to with a three step goal in mind. First I wanted to create a maze interface, Second I wanted to build a depth-first maze solving algorithm that would be able to solve these mazes, and third I wanted to create a breadth first method as well so I could compare the two systems. I have succeeded in creating both a maze and a mouse library, which when put together do allow me to make mazes and run depth-first algorithms, but unfortunately I was unable to get the breadth first system fully functional for reasons I'll discuss in greater depth later when I get to the breadth first section of this write-up, but the first topic of this paper should be the maze construction itself.

Building the maze interface

In order to solve mazes, I first had to figure out how I was inputting them. I chose to re-use many elements of our matrix library to help facilitate the maze making process, mainly the matrices themselves becoming my mazes, the get and set functions and the bad arguments functions. To turn a simple matrix into a maze, I had to greatly increase its size, for example, in order to translate all of the data of a three by three square maze and all of its walls, I need data for all nine maze squares as well as the twelve potential wall spaces. To accommodate this, each maze was made with rows and columns multiplied by two and subtracted by one (maze-rows = 2*rows-1), with this function every square with an even index was either a wall or a corner, and it was very easy to determine where the boundaries of my mazes were meant to be. After I had a maze making system, I had to learn how to make it easily accessible, because writing makeWall 98 times for an 8x8 maze is just unreasonable. To solve this problem I learned how to scan my maze data in from a text file. Using multiple libraries, I scanned each character of the text document, compared it to my set of special characters, and indexed whatever response was indicated. There was a slight problem about the newline character setting the maze off, but I eventually got around that by subtracting an index value whenever I encountered a whitespace.

Beyond maze creation and editing, the functions in maze.h also tell important information about elements of the maze and store important data, but more on that will come later. There are two more important things to notice about my maze system, the global variables and the pre-built mazes. In the process of building this maze, I had to pick arbitrary values that indicate the start, end and dead-ends within a maze, and found that I'd just be writing out random fives in the middle of everywhere. In order to increase the readability of the code as well as the ease of adjustment, I defined these variables at the top of my header file so they'll be consistent for every user of the system. In order to test my maze solver, I created four maze text files. Though the first maze has the same solution for both the breadth and depth search, the other ones all do a fairly good job depicting the shortcomings of the depth search and showing the universal nature of the solver, being able to solve both empty and rectangular mazes as well as winding and square ones. Maze 3 is a particularly good example of these issues, as though there are no walls in the maze, the mouse goes through every single square and gets itself stuck in a self-made dead-end before actually finding the exit. As these mazes and their solutions can be quite hard to visualize, I have also included a hand-drawn image of the mazes and their solutions as an image on the last page of this paper.

Depth First Search

The logic behind a the depth search algorithm is very simple, but actually laying the groundwork necessary for that algorithm gets a little complicated. A solver running a depth search follows the following logic: First, it has direction preferences, it wants to go in certain directions more than others. In my maze, these preferences flow right, down, left up. In order to correctly flow through these directions, I needed four functions that can characterize the ability of my move functions, my hasDirection functions. These functions make five checks to see if the mouse can move

1. If any inputs are null no move can be made
2. If the mouse is in a location it has no right to be it cannot move
3. If there is a wall in front of the mouse it cannot move that direction
4. If the spot has already been traversed and left behind it is a deadend, we do not want to go down that path
5. If a spot is already in my queue I would be going in circles if I tried to follow that path.

In order to test these assertions I simply had to write if statements using my isMazeSpace, isWall , getE and maze components for comparisons. In order to manage all of the locations and the path I took, I made two structures that I use throughout the moush.c file. A mouse containing a head pointer for my LIFO and a copy of the maze, and a spot pointer which contains row and element data, a pointer to the previous spot, and a pathlength counter which keeps track of how many squares from the start square I am. When initializing the mouse, it contains a mouse with the starting and ending locations filled in, and a spot representing the start spot of the maze in its location pointer, both of these values had to be extracted with other functions to be initialized.

After I had this framework I was finally able to make my moveMouse function. If this function finds that it can move in a direction, it will: creating a new spot pointer with the correct coordinates, putting it on the top of the LIFO and increasing my pathLength pointer. If it cannot find a move, it will return a true value, otherwise it returns false immediately after moving the mouse. If there is no move, the LIFO needs to lose elements until it can move again, this is done by my stepBack function which will remove the top element from my queue, set that square to denote a dead-end so I don't visit it again in the future, and returns zero. If it finds it has returned to the start square, it will instead return true as there is no solution to this maze.

The actual depthSearch function actually only consists of three functions, as long as the element found in my maze is not the end, I call moveMouse to try to get to the end. If I stepBack far enough to find that there is no solution to the maze, I return -1 from the middle of the loop after printing the maze for debugging's sake. If I do find a solution, the loop closes and runs my printMouse function, which copies the pathLength of my mouse's queue into its copied maze and then prints it so we can see the trail of the mouse in a sequence of increasing ints.

This function works in most situations I tested, however I have run into one problem. My initial test for maze4.txt was an 8x8 maze used in an actual micromouse competition, it is still in the file just slightly below the 6x6 maze that is actually used. When running this maze, and then a subsequent 7x7 maze, I had a segmentation fault that I could not track down, but as soon as I decreased the dimensions to 6x6 the function worked correctly. My only guess for this issue is that either my situation is too memory inefficient, it leaks a lot of memory, or some combination of the two, and ended up running out of space due to the mazes size, as I do not know what else could cause this function to crash when it is too large.

Breadth Search

My breadth search function works fairly similarly to my depth search with two main differences, first it is based off of a FIFO, and second it doesn't actually work. The logic of a breadth search is instead of going down each path one at a time, you put every option in your FIFO immediately so you can go down each path independently of each other. As soon as a path finds the end, you can know that the other paths are inefficient and close them, leaving you with just the one fastest path.

In my attempts to get breadthSearch to work I had to make a lot of changes to my code. I added a spot pointer that would keep track of the divergingNode or the location I last make a choice, as well as a character that tells me what that choice was to my spot struct. I did this as when I found a dead-end with breadth search I didn't want to have to loop back to the that node, mainly because I do not have that path anymore. Rather I wanted to just close the incorrect turns meaning that once my breadth search had finished, the only solution to the maze would be the one I found with the breadth search. This method of closing corners is slightly flawed in the fact that nested corners. Such as those found directly above the exit in maze 4 would result in dangling dead-ends, but my initial plan was to change the maze with breadth-search, and when there is only one solution left runa breadth search so I didn't have to write a new printMouse program and could clear up any stray numbers left by this unchanged deadEnds, unfortunately I never got to this part of the program.

My moveMouseBreadth function moves fairly complexly. It first checks if it is at the end, and if it is it closes all of the other paths found in the FIFO and returns 1. Otherwise it creates four null pointers, one for each direction and initializes them, but doesn't yet put them in the fifo. This happens because if I am at a node I want to save the node's location. Much like in the depth search I need to to use hasDirection functions to see if the direction should be added, unfortunately due to differences in the LIFO and FIFO structure I couldn't use the same has functions, and created copies of them without the queue tests in the bottom of the c file.

After I have tried to initialize each direction, I check to see if more than one was initialized. If so this spot has become a diverging node and is saved to the corresponding location section of the new spots and the the spots are put in the FIFO. If only one was initialized we simply put that entry in the FIFO without changing the diverging data. If no move can be made anymore, every move since the last diverging Node was simply leading to a dead end, so I can just block off that turn back at the node. After that I change the value of the space I am currently in so I never return to it, and then check to see the FIFO's size. If it is still zero after I added everything that I encountered, then there must not be a solution to the maze.

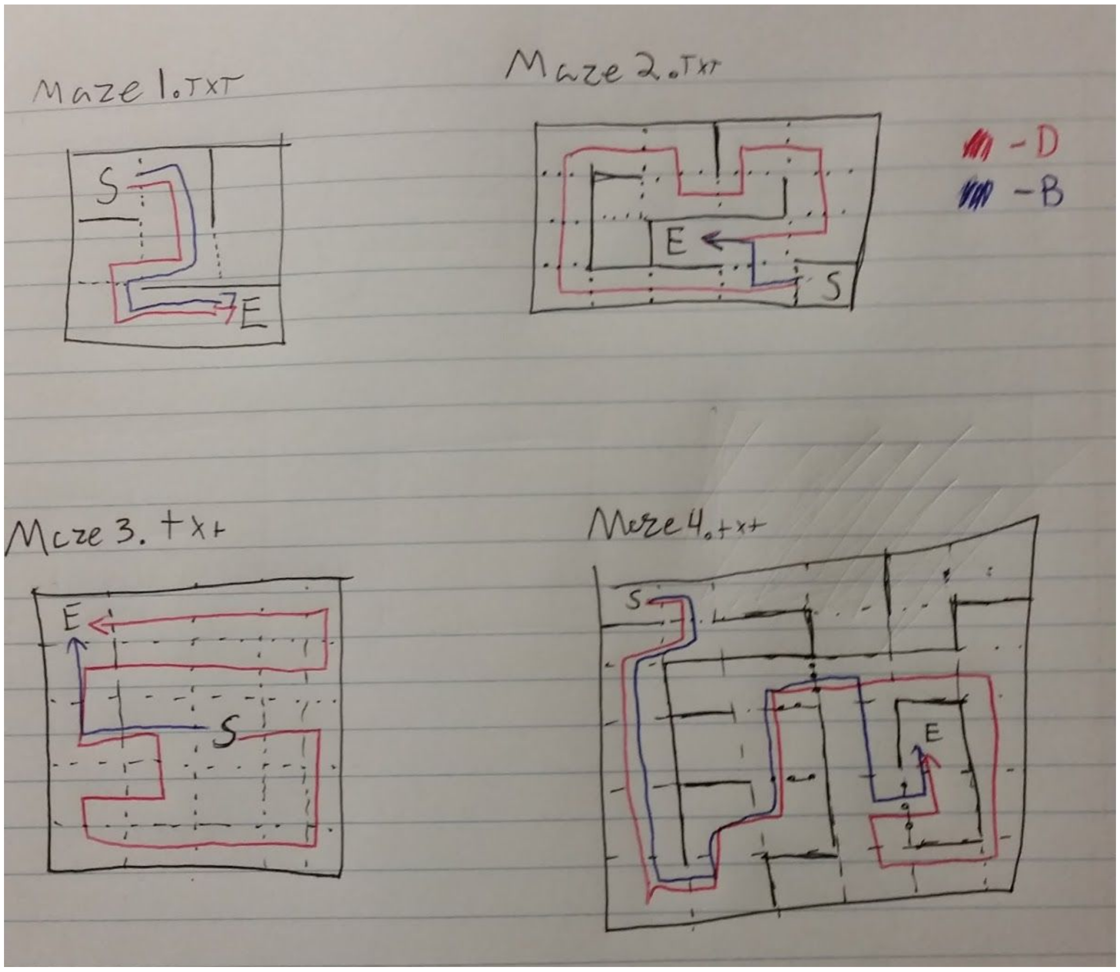
The issues with this function came from my put and get FIFO functions. My initial function which is commented out in the code made it about halfway through the code before it tried dereference a spot outside of my maze. I am fairly certain that the issue with that code was that in its last addition of code it tried to put information in a pointer that was contained by a null pointer, so it didn't exist. However my second attempt also fails with the put function as it just puts in a null function instead of the values I want, maybe I am just missing some critical part of my FIFO, but I haven't been able to piece together how to fix this section.

Running the code

I made a file minimouse that takes in command line arguments and runs one of four operations depending on what you type. After creating the executable file by typing "make" into the command line, typing “./miniMouse -h” will print out a help statement describing how to call the function. Typing “./miniMouse -M” will print out a very large set of information first outlining how to read the mazes, then it prints out brief descriptions of each pre-built maze, shows how to run a depth search on each maze, and prints each maze. Lastly “./miniMouse -d rows cols name.txt” and “./miniMouse -b rows cols name.txt” will run a depth search, and attempt to run a breadth search on the maze named name.txt and of dimension rowsxcols.

If I had more time to work on this maze solver, I would love to try to get my breadth search function working properly, though I’m sure there are at least a couple more hours of finding incorrect shifts and bad logic that I’d need to shift through once I fix my FIFO problem. I would also like to try to solve my size problem with the depth search. I pretty much glossed over it so far in the sense that I am still unsure of what actually caused it, and would like to know what actually happened to it.

Maze Sketches


Maze1.TXT
S
E
Maze2.TXT
E
S
- D
- B
Maze3.txt
E
S
Maze4.txt
S
E