MazeADT

Nikolas Provost and Perrin Ruth

April 2018

1 Introduction

In this project we created a maze ADT in the files maze.c and maze.h. This ADT consists of two data types: tree and maze. The maze data type takes information from a text file that may be provided by the user. The tree is a data type that works with the maze to find a solution using a depth first search. This has other files as well, namely solveMaze.c and test_maze.sh (the rest are text file mazes). The function solveMaze is compiled by the function "make", reads a text file from the user, solves the maze, and prints a user-friendly view of the solved maze. This is able to take any reasonably sized maze (stack overflow) as well as any start and end points. This checks if there is a solution, tells the user if there is a solution, and shows said solution. This also checks for multiple start/end points and produces an error if there is not exactly one of each. A test_maze script is included that tests solveMaze with several different maze situations. This also gives an example of how the function solveMaze can be used. Once this is done "make clean" can be used to remove the .o and solveMaze files that were compiled.

2 Using the ADT

The user must start by creating a maze text file for the ADT to read. The maze must be created with spaces denoting free space in the maze, S for the maze start point, E for the maze end point, and any other character will denote a wall (X was used for clarity purposes in our test files). The ADT will automatically find the maze size based on the farthest reaching points in the text file. The maze can only have one start-point and one end-point in order to be accepted. Past this, running the solveMaze program will first ask for the user to enter the name of the created text file, which must be within the same folder as solveMaze. If any errors are found within the maze file, the particular error is stated to the user. Once the maze file is validated by the program, solveMaze begins a depth first search in order to solve the maze, and the solution path will be displayed to the user in a user-friendly view. This view denotes @ as wall, space as free space, * for the correct path, S as the start-point, and F as the end-point. This view does not show incorrect paths taken by the solveMaze program. Within the solveMaze.c file, the PrintMaze(m) function can be uncommented to enable an in-depth computer stored view of the maze to be printed. This version of the maze denotes 0 as unvisited free space, 1 as wall, 2 as start-point, 3 as end-point, -1 as visited path that was deemed incorrect, and -2 as the correct path.

3 Implementation and testing

In this project we made the library run through two main steps, converting the maze into a computer-friendly format, and solving with a tree search. Then the code was tested by the script described in the introduction.

The first part was created for the project checkpoint. This takes data from a text file as discussed above by running a function "maze * newMaze(FILE * mz)". This function first works by reading the text file to find the size of a matrix that would be needed to hold the text file. This was difficult since the rows could have different lengths. To counteract this, the longest row was found. The matrix was then created to have as many rows as the text file and the maximum number of columns in a row. The values outside the text file are then initialized as walls so that they don't affect the maze. Then this read through the text file character by character to see if there was an empty space, start point, or end point. The main difference from before is this now works with an integer data array rather than a boolean one. This is helpful because it allows for a more detailed expression of data to show different values other than walls (1) and not walls (0). This allows for the tree search to add values for more cases such as already explored regions (-1) and regions in the current path (-2). Also, there is a function, "deleteMaze(maze * mz)" that frees the maze.

The other datatype that was implemented in maze.c was the tree datatype. This works by modeling the maze as a tree graph and searching by a depth first search. The tree works as a LIFO which only needs a tail pointer since we are only observing the last node. Each node has a parent pointer so that it is easier to backtrack, and it has one pointer to its child node since it only has one child since it holds a direct path. This tree is created through the function "tree * plantTree(maze * mz)". Then, the tree can be solved through a function "int findSol(tree * tr, maze * mz)". This moves down one node at a time until the end is reached or a dead-end. To know what path is accessible it turned out to be easiest to carry over the information into the maze itself. This happened by adding -1 into the maze it means the spot was already explored and -2 for the current path. This stops the risks of repeating the maze in case of a loop, and it also leaves the path in case it is correct so that it can be repeated when printed. Also, this checks for no solution. This works since eventually because if and only if it can't find a path it will to try to backtrack from the start node which gives a parent pointer of zero.

To first test the code we tried to make a function printMaze work. This was to see if we could get the code to behave as expected in the process of adding more code to the ADT. The biggest issue with this was naming the maze pointers because they were declared with name maze which confused the compiler. Thus, most maze pointers were renamed to mz. Once the code was finalized, the shell script was added to test various mazes for inputs. This took what hopefully covers the corner cases of the code. This checked a bad file name not trying to continue through the code. This also checked the situation of multiple start and end points and seeing if the code understood that issue. Then, we checked different difficult cases for the solver including different sizes, non-fixed start and end points, and loops within the maze. Then we used Valgrind to check for memory leaks as well in case the delete functions were poorly implemented.

Lastly, there are functions that cause the maze to be printed in its current form. This converts it to readable form as discussed above.

4 Future Planning

The future of this ADT would be mainly focused around optimization. With more time, different ways of selecting the pathing logic would be implemented to find the shortest and quickest path for any maze, especially if the maze has multiple solution paths. This could be done through varying the directional preference of the depth first search, or by implementing a breadth first search. This type of search would progress one step at a time through all available paths until path termination rather than following one path a time.