FA17 MP9 - Maze

Created by Zhang, Yihao, last modified on Nov 02, 2017

Due Saturday Nov 11

Introduction

The goal of this MP is to implement a maze solver using a recursive depth-first search, as well as a couple of functions that can be used to verify a correct solution. You will be given text files that contain the unsolved mazes in the following format:

```
11 11
%%%%%%%%%%%
S %       %
% % %%%%% %
% % %   % %
% % % % %%%
% %   %   %
% %%%%%%% %
% % %     %
% % % % %%%
%     %   E
%%%%%%%%%%%
```

The first two numbers are the width and height of the maze, followed by the structure of the maze. ‘%’ represent walls within the maze while a space represents an empty cell. ‘S’ indicates the starting position of the maze and ‘E’ indicates the ending position of the maze.

Your program will attempt to solve the maze by finding a path from ‘S’ to ‘E’ by only moving up, down, left, or right (no diagonal moves or moving through walls), print out the maze marked with the solution path and all cells visited during the search. A solved maze will look something like this:

```
%%%%%%%%%%%
S.%       %
%.% %%%%% %
%.% %   % %
%.% % % %%%
%.%   %   %
%.%%%%%%% %
%.%~%...  %
%.%~%.%.%%%
%.....%...E
%%%%%%%%%%%
```

‘.’ represent the cells that make up the solution for the maze and ‘~’ represent the cells that aren't a part of the solution but were visited during the search.

The Pieces

- main.c - contains the code to call the functions you implement in maze.c as well as the code for reading the maze input file.
- maze.h - contains the function definitions, structures, and some helpful macros you can use.
- maze.c - contains all the functions you have to write. Only code in this file will be graded.
- Makefile - A file that tells the make command how to compile the files into a single program.

Please note that when we autograde your problem we will be using the original versions of main.c, maze.h, and the Makefile. However, you can change these to test your code, just don't submit the changed version.

Details

The reading of the maze file name from the program arguments is done for you in main.c. You may assume the user will never input an incorrect maze file name. A maze is contained in the structure maze_t (defined in maze.h). This structure contains information about the maze such as its width, height, start position, end position, and the actual layout of the maze. The layout for the maze is stored in the parameter called cells. Cells is a 2D array (char ** cells) where the first index selects a **row** and the second index selects a **column** (maze->cells[row][col]). You can assume the maze won't have any cycles or loops (i.e. if you continue to move forward along a path in the maze you'll never end up in a place you've already visited) and will have neither a width or height dimension greater than 50.

There are four functions you need to implement for this MP:

```
maze_t * createMaze(char * fileName);
```

Given the name of the file containing the maze, this function should correctly allocate memory for the maze_t structure and allocate memory for the cells. This function should then parse the given file and fill in all the parameters of the maze structure, including the cells parameter. This function must return a pointer to the maze structure you allocate and fill.

```
void destroyMaze(maze_t * maze);
```

Given a pointer to a maze structure this function should free all memory associated with the maze.

```
void printMaze(maze_t * maze);
```

Given the structure that represents the maze, print out the maze in a human readable format to the console (stdout). This means the maze should be printed in the same format as the mazes shown in the introduction section. (Print each row followed by a newline)

```
int solveMazeMahhatanDFS(maze_t * maze, int col, int row)
```

This year for simplicity we are **NOT** using Manhattan distance. The only thing we are going to use is Depth First Search.

Given the structure containing the maze and a current row and column recursively solve the maze from this position using the DFS algorithm. This function should be the recursive function, but if you feel it's necessary you can write a recursive helper function that's called by this. This function should change the contents of the 2D array (cells) by indicating which cells are along the solution path and which cells were visited during the search. If the maze is unsolvable this function should return 0 otherwise it should return 1. You should NOT overwrite spaces that contain walls, the start, or the end.

These four functions are called for you in main.c, all you need to do is implement the four functions as specified.

Depth-First-Search (DFS) Algorithm

The recursive DFS algorithm, like many recursive algorithms, consists of two main parts, the base case and the recursive part:

Base case - when we should stop recursion.

Recursive part - calls the same algorithm to assist in solving the maze.

Recursive Part

Since the DFS algorithm you have to implement must be recursive we should look at the problem as a set of sub-problems. Recursive algorithms are sometimes exhaustive and in this application this means we need to find a solution path by searching through all the possible paths within the maze.

Assume there's an algorithm that already finds some path within the maze, let's call it getPath(), which gets us from the start position to column = 1, row = 2.

```
%%%%%
S.% %
%.% %
%   E
%%%%%
```

What we need to know now is if there's a path from (1, 2) to the end. If there is we know there's a path from the start to the end because we already know there's a path from start to (1,2).

To find a path from (1, 2) to the end, we can just ask getPath() to try to find a path from the left, right, up, and down positions of (1, 2):

- getPath(col = 0, row = 2) *Left*
- getPath(col = 2, row = 2) *Right*
- getPath(col = 1, row = 1) *Up*
- getPath(col = 1, row = 3) *Down*

Generalizing this, we can call getPath() recursively to move from any location in the maze to adjacent locations. In this way, we move through the maze.

Base Case

Now we need to know when to stop making recursive calls. There are two base cases we need to consider, the goal state when we reached the end of the maze and invalid positions within the maze.

Invalid positions may be things like positions outside the bounds of the maze, positions that are wall, or positions we've already visited.

Both Parts Together

Now that we have the base cases and recursive part we can put it all together into one algorithm:

1. If (col, row) outside bounds of the maze return false
2. if (col, row) is not an empty cell return false
3. if (col, row) is the end of the maze return true
4. set (col, row) as part of the solution path in the maze
5. if (solveMaze(left of (col, row)) == true) return true
6. if (solveMaze(right of (col, row)) == true) return true
7. if (solveMaze(up of (col, row)) == true) return true
8. if (solveMaze(down of (col, row)) == true) return true
9. unmark (col, row) as part of solution and mark as visited
10. return false

This algorithm can be modified and still function correctly. For instance, instead of checking if a (col, row) is outside the bounds as a base case, we can only call solveMaze recursively on spaces we already know to be valid.

Backtracking in this context

Backtracking is the idea that once all the paths starting from the current cell are found and none contain the end, we will begin looking for new paths from the previous cell (moving back one cell). These new paths won't contain the current cell because we've already searched all paths that contain it. To clarify, imagine the run-time stack. Each time we call solveMaze a new function is pushed onto the stack. Once that function reaches step 10 (return false) this means we've searched all possible paths that start from this cell. So after we return false, the function that was called for this cell is popped off the stack and we return to the previous calling function (the previous cell). The previous cell will then start looking for paths in a different direction. If this cell can't find any paths that contain the end it will return false, have its function popped off the stack, and we'll move back to the previous cell. Basically, for each recursive call to solveMaze we are making a guess for the next cell on the solution path, if this guess is incorrect backtracking allows us to undo the guess and make a new one.

Here's an illustration of what a depth first search algorithm does.

As you saw in the description of DFS the order in which we explore positions is set as left, up, right down. But you can use other combinations of directions to cover this four directions, as long as you can solve the maze.

If you would like more information on depth-first-searches, best-first-searches, and heuristic searches google and wikipedia are your friend 😬

https://en.wikipedia.org/wiki/Depth-first_search

Specifics

- Your program must be written in C in the file maze.c provided for you.
- You must implement createMaze, destroyMaze, printMaze, and solveMazeManhattanDFS correctly.
- Your routine's return values and outputs must be correct.
- Don't change the function definitions for the functions you have to implement.
- Be sure to access the maze as row first then column. (maze->cells[row][column])
- Your code must be well commented, including introductory paragraph.

Building and Testing

The commands to build your project and compile your code. Make clean will first remove any compiled code (object files and the mp9 binary) from the MP folder. Make will then compile everything.

```
make clean
make
```

Once compiled, run the program from mp9 directory as:

```
./mp9 [file_name]
```

We also give you the gold outputs for these mazes in the tests/outputs directory. Please note that your maze may NOT look exactly the same after you solve it depending on how you handle the order you search the maze in when multiple directions have the same Manhatta distance to the end. The gold version, will search in the following order: left, down, right, up. Feel free to write your own debug and helper functions in maze.c.

You may also find using valgrind helpful to check for memory leaks (see if your createMaze/destroyMaze is correct). To run your code with valgrind:

```
valgrind ./mp9
```

Grading Rubric

Functionality (90%)

- 20% - createMaze works correctly
- 10% - destroyMaze works correctly
- 10% - printMaze works correctly
- 50% - solveMazeManhattanDFS works correctly

Style (5%)

- Code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Comments, clarity, and write-up (5%)

- Introductory paragraph explaining what you did (even if it's just the required work)

Some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.

Slides from lab: MP9.pptx

No labels