Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Maze #3 (A)

Due 11/3

Overview

This homework will give you more practice with file operations, memory allocation, and recursions (if you model your solution after the code provided by the instructor in maze.c). You are in charge of a special mission. In particular, your job is to guide a special agent through a maze, beginning at some source location and reaching some destination location, taking a shortest path. Based on a pre-processed satellite images of the maze, where the walls of maze are stored as 'X' and the paths are stored as ' ', you want to provide the directions that the special agent, who has been air-dropped to the source location within the maze, can take to reach the destination location with the shortest distance because time is critical. In this exercise, you will write the following functions:

1. `dfs_shortest_path_directions(…)`: Given a file containing the satellite "image" of a rectangular maze, compute and store the directions to be taken by the agent to reach the destination location in the shortest distance, starting from a source location. The directions will be stored in a different file.
2. `simulate_movement(…)`: Given a file containing the satellite "image" of a rectangular maze, the directions to be taken by the special agent in another file, simulate the moves of the agent, count the number of locations visited by the agent, and print an "image" representing the maze traversed by the agent.
3. The `main(…)` function, which, depending on a user-given option, calls the function in (1) or (2) accordingly.

Please read this homework description together with the figures in sample5.pdf.

Input file format

Just like in HW09 and HW10, we are concerned with only rectangular mazes. The mazes that we would consider are the mazes that are produced by the program amaze.c or expanded by a program in HW10. You may assume that if a given file exists, it contains a valid maze.

Output file format

There are two files that you will output:

1. A file containing the directions for the first function
2. A file containing the visited maze for the second function.

For the directions file, it contains a stream of characters 'N', 'S', 'E', 'W' (all upper case), which stand for the north (up), south (down), east (right), west (left) directions, respectively. A 'N' ('S') character, for example, means that the agent should move north (south) by 1 row. A 'E' ('W') character means

that the agent should move east (west) by 1 column. No other characters, including space and newline characters, are allowed in the file. In other words, if we use the vi editor to open a direction file, the editor should say that it is an incomplete file. For the example sample5, a directions file for a shortest distance path from the source location (0,5) to the destination location (12,11) is in sample5.dir. An alternative directions file is in sample5.dir2. For the visited maze, the format is somewhat similar to an input file for a maze. The difference is that that locations that have been visited along the path taken by the agent (in the simulation) should have the space character ' ' (`PATH`) replaced by '.' (VISITED). For the directions file sample5.dir (or sample5.dir2), the path taken the agent is shown in the file sample5.dir.visited (or sample5.dir2.visited).

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents	

maze.c	functions	**`dfs_shortest_path_directions`**`(char * maze_file, char * directions_file, Coord source, Coord destination)`
		→ *return type:* int
		• Two filenames are given to the function:
		  • `maze_file`: This file contains the given maze
		  • `directions_file`: This file is to be written with the directions that guide an agent from a source location to a destination location using a shortest path
		  `visited_file`
		  Both the `source` and `destination` are given as the third and fourth parameters of the function, respectively.
		• Return -1 and leave the function in the following conditions:
		  • File does not exist
		  • Could not read and store a maze into a `Maze` struture
		  • The source and destination locations are invalid (not in the range and not a `PATH`)
		  • Cannot open the directions file for writing
		  ⚠ The direction file should not be creaed if you return -1 from the function because of the aforementioned reasons.
		  ⚠ Beyond this stage, you should not return -1, and the directions file has been created.
		• Assume that the maze is either generated from amaze.c in HW09 or expanded from such a maze in HW10 if the file exists.
		• The function should determine the directions that would allow the agent to use only `PATH` locations to reach the destination location from the source location using a shortest path.
		• The function should determine the fewest number of steps for the agent to reach the destination location from the source location. As each step is represented by a character, the steps taken by the agent should be output to the directions file.
		• Return the number of steps taken by the agent. (Note that the number of steps taken by the agent should be the same as the size of the directions file. If for some reason, you cannot write a direction character into the directions file, you should still, at this stage, return the number of steps taken by the agent.)
		• ⚠The agent should not take a step that would let him/her collide with a `WALL` or go out of bound.
		• ⚠For the path taken to be the shortest, the agent should never re-visit a location.
		• Example: For maze sample5, integer 22 is returned by the `dfs_shortest_path_directions(…)` function for source location (0,5) and destination location (12,11). The output directions file is sample5.dir or sample5.dir2.

file	contents	

		`simulate_movement``(char * maze_file, char * directions_file, char * visited_file, Coord source, Coord destination)`
		→ *return type:* int
		• Three filenames are given to the function:
		  • `maze_file`: Given maze
		  • `directions_file`: Directions file
		  • `visited_file`: This file should be used to print the visited maze
		• Two locations (as Coord's) are also given to the function. They are the `source` location and the `destination` location.
		• Return -1 in and leave the function in the following conditions:
		  • Maze file does not exist
		  • Can't allocate memory to store the maze
		  • Direction file does not exist
		  • Source and destination locations are invalid (not in the range and not a `PATH`)
		  • Cannot print the visited maze to `visited_file`
		  ⚠When you return -1 from the function because of the aforementioned reason, the output file for the visited maze (i.e., `visited_file`) should not be created.
		• Assume that the maze is either generated from amaze.c in HW09 or expanded from such a maze in HW10 if the file exists.
		• The following applies when the maze file, directions file exist, and the source and destination locations are valid.
		• At the beginning, the agent is positioned at the source location. That location will always be visited regardless of the contents of the directions file.
		• Depending on the direction character read from the directions file, the agent moves North or South by a row, or East or West by a column. The simulation terminates when
		  • Reach the end of the directions file: If the agent is at the destination location, you output the visited maze into the third file, with each location visited by the agent now represented by '.'. Return the number of locations that have been visited.
		  • Reach the end of the direction file: If the agent is not at the destination location, you still output the visited maze into the third file, with each location visited by the agent now represented by '.'. Return -1.
		  • Encounter an illegal direction:
		  • the character is not one of the'N', 'S', 'E', 'W'
		  • the direction will take the agent out of bound
		  • the direction will take the agent to a location with `WALL`.
		  You output the maze that was visited by all legal directions so far, and return -1.
		• The only time you return a number other than -1 is when the directions file allows the agent to move the source to destination, and the visited maze could be printed to the `visited_file`.
		• Example: For sample5 and directions file sample5.dir (sample5.dir2), integer 23 is returned by the `simulate_movement(…)` function. The file storing the visited maze is sample5.dir.visited (sample5.dir2.visited).

test_maze.c	functions	**`main`**`(int argc, char * argv[])`
		→ *return type:* int
		• The `main(…)` function expects "-s" or "-m" to supplied as `argv[1]`. If not, the executable should exit with `EXIT_FAILURE`.
		• If the "-s" is the first argument to the executable, it means that you should perform simulation to verify that the agent can reach the destination using the directions provided. We expect 7 arguments to follow the "-s" option. The arguments, in order, should be
		  • The maze file
		  • The directions file
		  • The output file for the visited maze
		  • The row coordinates of the source location
		  • The column coordinates of the source location
		  • The row coordinates of the destination location
		  • The column coordinates of the destination location
		  ⚠ You call the simulation function only if you have sufficient arguments. Also, you have to perform string-to-integer conversions of the last 4 arguments into source

file	contents	

and destination coordinates.
We suggest that you use `strtol(…)` to perform the conversion and use the second parameter of `strtol(…)` to check whether the string being converted contains only valid symbols for an integer (base 10). You call the simulation function only if the source and destination coordinates are integers converted from strings that contain only valid symbols. Otherwise, you should exit with `EXIT_FAILURE`.
Example: When the following command is issued

```
./maze -s maze_file directions_file visited_file 5 0 10 101
```

where 5 and 0 are the row and column coordinates of the source location, respectively, and 10 and 101 are the row and column coordinates of the destination location, the simulation function should be called. Note that it is the role of the simulation function to check whether the coordinates are valid for the input maze_file. The following commands are used to produced sample5.dir.visited, sample5.dir2.visited, sample5.tdir.visited, respectively:

Command	Output on screen
`./maze -s sample5 sample5.tdir sample5.tdir.visited 0 5 12 11`	39
`./maze -s sample5 sample5.dir sample5.dir.visited 0 5 12 11`	23
`./maze -s sample5 sample5.dir2 sample5.dir2.visited 0 5 12 11`	23

- The following would not result in calling the simulation function because one of last four arguments is a string with invalid symbols.
  `./maze -s maze_file directions_file visited_file 5 0abc 10 101`
- If the "-m" is the first argument to the executable, it means that you should determine the directions for the agent.
  The following arguments, in order, should be
  - The maze file
  - The directions file for storing the direcitons computed by your function
  - The row-column coordinates of the source and description locations (see the description in the "-s" option)

  You call the shortest-path function to determine the directions only if you have sufficient arguments and that the coordinates are integers converted from strings with only valid symbols. Otherwise, you should exit with `EXIT_FAILURE`.

Command	Output on screen
`./maze -m sample5 sample5.dir 0 5 12 11`	22

- ⚠ Immediately after the `dfs_shortest_path_directions(…)` and `simulate_movement(…)` functions, you should print to `stdout` the returned value of the called function with the following pintf format: "%d\n".
- ⚠ Write the simulation function before the shortest-path function.
  In a sense, the simulation function helps to test whether your shortest-pathh function is partially correct.
  The `main(…)` function is designed to help you to write a useful function (like the shortest-path function) and a testing function (like the simulation function).
- ⚠ Only two functions defined in maze.c that would be called by the `main(…)` functions are the two functions declared in maze.h.
- In the test_maze.c provided to you, we use option "-t" to call the function `find_path_from_top_entrance_to_bottom_exit(…)`.
  You may remove this part of the program in your final submission, or you may keep it. We will not evaluate your program with the "-t" option.

2. ⚠ test_maze.c should contain only the `main(…)` function and nothing else. The only two user-defined functions that test_maze.c would call in the `main(…)` function are the two functions declared in maze.h: `dfs_shortest_path_directions` and `simulate_movement(…)`.
3. ⚠ Always close all files you have opened and free all memory you have allocated before eixting the function.
4. Submissions must meet the code quality standards and the course policies on homework and academic integrity.

Hint (optional)

(You may skip over this if you want to think of your own solution and do not model your solution after the `find_path_from_top_entrance_to_bottom_exit(…)` function provided by us.)

The "DFS" in the name is the acronym for Depth First Search, which is typically used to describe a method based on recursion, because a recursive function always seeks to call itself until it reaches a terminating condition, when we have the largest depth of the recursion.

In maze.c, we provide the code for finding a path from the top left-most entrance to a maze to the bottom right-most exit of the said maze. This function takes in three parameters: the maze file, the directions file, and also a visited maze file that records which locations in the maze have been visited to find the directions. The `find_path_from_top_entrance_to_bottom_exit(…)` function uses a recursive helper function called `_pathfinder_helper(…)` to find a path.

In this helper function, we mark the current location VISITED. If the current location is the bottom exit, we found a path (this is the terminating condition). We will explain how we output the path later.

If we have not reached the destination location, we explore the 'N', 'S', 'E', and 'W' of a current location using recursions. If the location to the 'N' is within bound and it is a path, we visit all locations reachable from there by recursively calling the helper function. This is achieved by providing the updated current location (with the curr decremented because we are going 1 row up) and also that the distance from the top entrance is incremented by 1. If the destination location can be reached by going north, we print the step taken to be 'N'. Otherwise, we proceed to try 'S', 'E', and 'W'. If none of these give us a path to the destination location, we return -1.

How is the path from the top entrance to the bottom exit printed? When we reach the bottom exit (the terminating condition), we know how many steps it takes to reach there from the top entrance. Let count be the number of steps, we first output count number of '.' characters to the directions file. These '.' characters will be replaced by the actual directions later on. Now, the position index is 1 character after the last invalid direction character '.' in the directions file.

Therefore, when after a recursive call of the `_pathfinder_helper(…)`, we know that the bottom exit has been found, we have to move the position index 1 position backward, write the actual direction taken to reach the bottom exit. After the actual direction taken is output to the directions file, the position index is again 1 position after the most recent step. We therefore have to move the position index 1 position backward again. Now the the position index of the directions file is again 1 position after the last invalid direction character '.' in the directions file.

(Note that there are other ways to change how the directons are determined and written to the directions file such that we do not have to perform so many fseek operations.)

For the maze in sample5, the top left-most entrance is at (0,5) and the bottom right-most exit is at (12,11). Using the `find_path_from_top_entrance_to_bottom_exit(…)` function, we found the directions and stored the directions in sample5.tdir. The locations in the maze that have been explored in order to find this path are stored in sample5.tdir.explored. To demonstrate how the recursion works, the computation tree (incomplete) to find the path is shown in sample5.pdf. We start from location (0,5) and a distance from the top entrance of 0. (You may uncomment the first statement of the `_pathfinder_helper(…)` and use the screen output to help you with tracing through the explored maze.)

Please read the following paragraphs with sample5.pdf. Part (1) of the figure shows the path from (0,5) to (7,7) after 17 steps. We continue from (7,7) and reach location (9,7). Since we explore in the order of 'N', 'S', 'E', and 'W' directions, we call `_pathfinder_helper(…)` with the 'S' direction because an invalid location is in the 'N' direction (that location has been visited). We would reach

(11,5), which would allow us to explore in the 'N', 'S', and 'W', directions (see part (3)), but none of them would allow us to reach the bottom exit.

The recursion will now find its way back to location (9,7), and we would now explore in the 'E' direction. When we reach (7,11) (see part (4)) in a distance of 25 steps from the source, we would explore the 'N' direction first, which would not lead us to the bottom exit. We would then explore the 'E' direction, which allow us to reach the bottom exit. We return the value 38, the number of steps it takes to reach the bottom exit, and along the way of "backtracking" to the caller functions, print the directions to the directions file.

Note that the `find_path_from_top_entrance_to_bottom_exit(…)` function relies on many functions that you have written for HW10. You have to copy those functions over into maze.c before you can compile the given test_maze.c and maze.c. Please note that these functions you are copying over should be declared and defined in maze.c. Do not attempt to declare these functions in maze.h, because you are not submitting maze.h.

Let maze be the executable (see later for how the program should be compiled), the command

```
./maze -t sample5 sample5.tdir sample5.tdir.explored
```

should print 38 on the screen and output the directions to sample5.tdir, and the explored maze to sample5.tdir.explored. If you have written the simulation function and the `main(…)` function correctly, the command

```
./maze -s sample5 sample5.tdir sample5.tdir.visited 0 5 12 11
```

should print 39 on the screen and output the visited maze to sample5.tdir.visited.

Note also that `_pathfinder_helper(…)` is also declared as a static function. This means that this function can be used by functions in maze.c. It cannot be used by functions in other files. The advantage of declaring a function to be static is that you do not have to worry that the function has a name that would clash with functions declared and defined in other files.

You can try to change the order in which the neighboring locations of the current location are explored. You may get different directions, different length of the path, and also different explored maze.

How can you modify the `find_path_from_top_entrance_to_bottom_exit(…)` function for the implementation of your `dfs_shortest_path_directions(…)`? First, you have to recognize that you have to write a different helper function. In `_pathfinder_helper(…)`, you do not explore a location after it has been visited. However, that is not true when you are looking for the shortest path. Let's take the maze in sample5 as an example. In sample5.pdf part (1), let's examine the directions explored at location (1,5). At (1,5), we have to explore 'N', 'S', 'E', and 'W'. However, we just came from (0,5), so, the recursive call in the 'N' direction is not attempted. What we have is the recursive call in the 'S' direction. Upon the completion of that recursive call, we have 38 being returned. Therefore, we skipped over the exploration in the 'E' and 'W' directions.

Now, imagine if the returned value is -1. In that case, `_pathfinder_helper(…)` would still skip over the exploration in the 'E' direction, because (1,6) has been visited. We would explore only in the 'W' direction. In fact, (1,6) has a distance of 48 from (0,5) (see part (5) in sample.pdf). However, (1,6) has a shortest distance of 1 from (1,5) (and a shortest distance of 2 from (0,5)). In other words, in the determination of a shortest path, having visited a location earlier should not prevent the location from being explored again. As long as we can reach the location with a shorter distance, we should always explore in that direction to see whether we can find an alternative path to the destination location with a shorter distance. (Do you see a new terminating condition for your new recursive function?)

Similarly, in `_pathfinder_helper(…)`, we reach (5,6) with a distance of 48. However, (5,5) has a distance of 13 from the top-entrance. Therefore, we can reach (5,6) with a shorter distance of 14.

In other words, we cannot stop exploring a visited location in the new helper function. We will explore a visited location as long as we can reach a visited location with a shorter distance. You would therefore have to store the current best distance of every location from the source location. We suggest that you use a two-dimensional array of int's to store those current best distances. You would also have to initialize these distances properly so that you do not miss out a shortest path. You may assume that the product of the dimensions of a maze is no larger than `INT_MAX`. Upon the complete execution of the recursive helper function, this array of int's should contain the shortest distances from the source locations to all other locations in the maze.

Another challenge is for you to print the directions of the shortest path from the source to the destination. You probably have to write another recursive helper function to trace a path from the destination to the source. You should use the same two-dimensional array of int's (storing the shortest distances to all locations in the maze from the source location) to help you in tracing a path from the destination to the source. When you reach the source, you will start returning from all the recursive calls of this helper function. Now is the time to output the directions to the directions file.

Compile

To compile:

```
gcc test_maze.c maze.c -o maze
```

test_maze.c is a skeleton containing function call to compute a path from the top left-most entrance to the bottom right-most exit of a maze. The path finder function (see the description given above) is in the skeleton maze.c file. The path finder function also assumes certain functions written in HW10. You should be able to compile the program if you copy these functions from your HW10 to maze.c.

Submit

To submit HW11, type `264submit HW11 maze.c test_maze.c` from inside your hw11 directory.

> In general, to submit any assignment for this course, you will use the following command:
>
> `264submit` *ASSIGNMENT* *FILES…*
>
> Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).
>
> To retrieve your most recent submission, type `264get --restore` *ASSIGNMENT* (e.g., `264get --restore hw11`).
>
> To retrieve an earlier submission, first type `264get --list` *ASSIGNMENT* to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get --restore -t` *TIMESTAMP* *ASSIGNMENT*.
>
> Scores will be posted to the Scores page after the deadline for each assignment.

Pre-tester ●

The pre-tester for HW11 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw11
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

1. **Can we use functions from previous homeworks?**
   Yes, there are many other functions that would be useful to your program. For example, the `malloc_maze(…)` and `free_maze(…)` functions in HW10 are probably needed in this assignment. You should go through HW09 and HW10 to select the important functions that you want to include in this assignment. Moreover, you will probably need other functions, but we will not specify them here. All these functions should be declared and defined in maze.c. (⚠Note that you cannot declare these additional functions in maze.h because you are not submitting maze.h) Note that a function must be declared (the declaration states the return type and the input parameters) before it is called by another function.

2. **What do you mean that "we will explore a visited location as long as we can reach a visited location with a shorter distance?"**

Consider the following (partial) maze, which is a part of a bigger maze:

```
XXXXXX
X    X
X XX X
X    X
X XXXX
```

Let's assume that we start from position (3,1) of this partial maze. Also assume that we always try to explore the maze using the following order: 'N', 'S', 'E', 'W'. Based on the given helper function, we would have explored various locations of the partial maze with the following numbers of steps:

```
XXXXXX
X2345X
X1XX6X
X0987X
X1XXXX
```

We will not explore the location (3,2) using the 'E' direction from (3,1) because (3,2) has been visited earlier using the 'N' direction. However, for the shortest path problem, it is clear that we can reach (3,2) with 1 step to (3,2) from (3,1). So, if you want to use a recursive function to solve the shortest path problem, you would have to re-visit (3,2) and replace the value 9 with 1.

```
XXXXXX
X2345X
X1XX6X
X0187X
X1XXXX
```

Similarly, you will also re-visit (3,3), (3,4), and (2,4) because you can reach there with a shorter distance than previously recorded.

```
XXXXXX
X2345X
X1XX4X
X0123X
X1XXXX
```

However, you will not continue to explore (1,4) from (2,4) because you cannot reach there with a shorter distance.

Updates

10/15/2017	Draft
11/1/2017, 11/2/2017	Added clarification on why we should re-explore a location that has been visited before.