French Azerbaijani University
Arash Habibi
Second year
2017-2018

Advanced C Programming

Project : Maze resolution

1 Statement of problem

1.1 Mazes

A *maze* is a set of paths, typically from an entrance point to an exit point (figure 1). The player must find one path from the entrance to the exit.

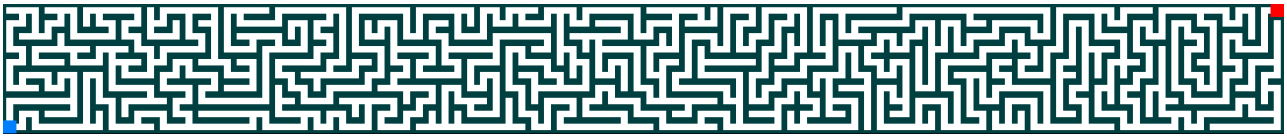

FIGURE 1 – A *maze* with one entrance point (in blue at the left) and one exit point (in red at the right).

In *perfect* (or *simply connected*) mazes, for any two points A and B, there is one and only one path connecting A to B (figure 2). In unperfect mazes, (figure 3) there may be several paths between two points (between the red point and the blue point) or there may be no paths between two points (between the green point and the other two points).

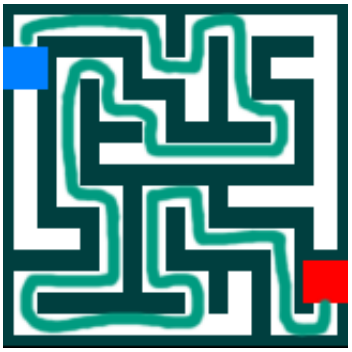

FIGURE 2 – A perfect maze

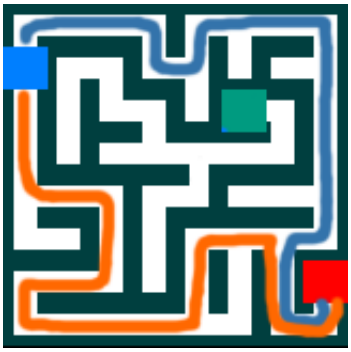

FIGURE 3 – An unperfect maze

1.2 Maze generator

We already have at hand a maze generator, capable of producing perfect rectangular mazes of any size. You will find that maze generator, in the form of a library `(libMaze.a)` on the *Moodle* platform. Download this library and the `include` folder and place them in your project's folder. In the same folder, open an empty file called `main.c` and fill it with the following code.

```
#include <stdio.h>
#include "Maze.h"
int main()
{                                       // Maze is a new type defined in Maze.h
    Maze mz = MZ_new(8,8,MZ_HARD);      // generation of a difficult 8x8 maze.
    MZ_saveImg(mz,"myFirstMaze.ppm");   // save the maze in an image file
    MZ_free(mz);                        // free maze.
    return 0;
}
```

Make sure that `libMaze.a` is in the current directory and compile the program with :

```
gcc main.c -I include -L. -lMaze -o maze
```

— `-I include` tells the compiler that it can find the header files in the `include` folder ;

— `-L.` tells the linker that it can find the library files in the current folder.

Now, running the `./maze` program will generate an 8×8 maze as in figure 2, and will put the result in `myFirstMaze.ppm`. Opening it with your image viewer will show you your first maze [1]. This program only produces the maze. It does not find the way from the entrance to the exit. That will be *your* part of the work.

1.3 The aim of this project

The aim of this project is to design and write a program capable of finding the solution of a perfect rectangular maze as in figures 1 and 2. Some maze solution algorithms can access the global shape of the maze (as if they were looking at the maze from the top). But our aim is to find a solution based only on local information (as if we were inside the maze and could only see our local environment). There will be three main stages in this project.

The specification : In this first stage, you need to elaborate the global structure of the program without being disturbed by implementation issues. At the end of this first stage (November 17th), you will hand us a set of header files containing data structure definitions and function prototypes. For each function prototype, you will define the meaning of each function parameter, and the relation between the function's output (return value or side effect) and its inputs (parameters). In other words, you will explain what you think the function should do.

Implementation : In this second stage, we will provide you with header files, and you will need to implement the corresponding functions. At the end of this second stage (December 1st), you will hand us a program which will use the `libMaze.a` library in order to solve perfect rectangular mazes of any size.

Final stretch : You may carry out the first two stages wherever you wish (at home or at the University), alone or with the help of other students. At this point, I strongly recommend you NOT to plainly copy the codes of other people (friends or internet), but to ask for explanations, to understand and to internalize this code and to write your own version of the work. Otherwise, it will be extremely difficult to go through this third stage. This third stage will take place at the university in limited time. You will be asked to change your code in order to obtain new results.

2 A bit of reflection

When solving a problem, the very first stage is to think about this problem in order to find the best way to represent it. In particular, we need to know how to represent a maze. We also need to know the exact meaning of the *solution* of a maze. Here are a few elements that will hopefully help you through this first stage.

1. Function `MZ_new` has three arguments. The third one (in here `MZ_HARD`) suggests that this maze is difficult to solve and you may find it rather easy. But the difficulty level is measured from the point of view of the programmer, not from the point of view of the player. We will see that in further detail.

2.1 The maze representation

2.1.1 *Cells*

A perfect rectangular maze can be represented as a set of *cells*. The maze in figure 4 is composed of 8×8 cells. Each cell can be characterized by its integer coordinates. The coordinates of the upper-left cell is (0,0). In figure 4, the coordinates of the entrance point (blue) is (0,1), and the coordinates of the exit point (red) is (7,6).

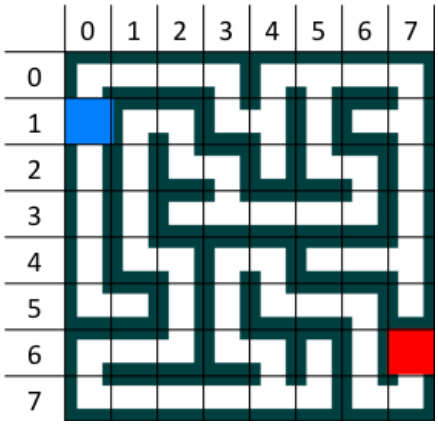

FIGURE 4 – A maze composed of 8×8 cells

2.1.2 *Neighbors* and *linked* cells

Two cells are called *neighbors* if and only if they are next to each other. Most cells have 4 neighbors. Border cells have 3 neighbors and corner cells have 2. Moreover, each cell is *linked* to one or several neighbors. Two neighbor cells are *linked* if and only if they are not separated by a wall. We can go directly from one cell to any of its linked neighbor cells. For example in figure 4, the entrance point (0,1) has two linked cells : (0,0) and (0,2). Note that (0,1) and (1,1) are neighbors but are not linked, because they are separated by a wall. The exit point (7,6) is only linked to one cell : (7,7).

2.2 The *solution* of the maze

> Solving a maze means exhibiting a sequence of linked cells. The first cell of the sequence must be the entrance cell and the last cell must be the exit cell.

Strictly speaking, you could exhibit this sequence by simply printing the coordinates of these cells. But it would be much better if you could draw this solution on the image file.

2.3 The tools at your disposal

The `libMaze` library provides you with a certain number of tools. As you have experienced in your maze generation program, the `libMaze.a` library generates an object of type `Maze` (such as `mz`). These objects are not mere graphical representations of a maze, but contain all the useful information that you need in order to solve the maze :

— The position of the entrance cell ;

— The position of the exit cell ;

— The links between a given cell and its neighbors.

The functions that enable you to extract those informations, and to display the results are described in appendix A.

3 Tasks to carry out

We will start with rather easy guided tasks, and as we progress, you will be able to make your decisions without any help.

3.1 Read the `libMaze` library

I strongly recommend that you read appendix A. It may even be relieving for you to test each function and check whether the results are coherent.

3.2 *Cell* type (6 points)

One of the most important objects that we will be manipulating are *cells*. For the moment, the functions of the `libMaze.a` library are a bit awkward because they only use integers. For example they use an array of 6 integers : 1 3 6 7 2 4 to designate three cells : (1,3) (6,7) and (2,4). So in a new `Cell.h` file, define a new type called `Cell` and write prototype functions which enable you to :

1. create and return a new `cell` from its components ;
2. print the components of a `cell` on the terminal (for debugging) ;
3. check if two cells have the same position (return 1 if they do and 0 otherwise) ;
4. have an easier interaction with the functions of `libMaze.a`. These functions deal with cells, but in practice, the parameters and return values are integers or integer arrays. You need to write functions that deal really with cells. For example :

   ```
   // For a given maze object mz, this function returns a cell object
   // representing the position of the entrance cell.
   cell CL_entrance(Maze mz);
   ```

   Of course, `CL_entrance` will call `MZ_entrance`, but it will be much more easier to use. Do something similar for the other functions, specifically : `MZ_exit`, `MZ_linkedCells` and `MZ_setSolution`.
5. Let us assume that we create mazes of the simple type (call `mz = MZ_new(X,X,MZ_SIMPLE);`) where the exit point and the entrance point are only two steps away from each other (figure 5). Write the algorithm that will enable you to find the path from the entrance to the exit in the case of such simple mazes.
6. Write the prototypes of any other functions that you may find useful for that algorithm.

3.3 Medium level (7 points)

Let us tackle a more difficult problem. In medium level mazes, the entrance and the exit are more than two steps away, but on the path linking both cells, there are no branching (as in figure 6). When you start at the entrance cell (Cell (0,7)), the choice is clear since there is only one linked neighbor which is the second cell (Cell (1,7)). But all the other cells in the path (for example cell (0,4) in figure 7), have two linked neighbors : one that we have never explored (cell (0,3)) and one that we just left (cell (0,5)). For human beings, the choice is clear : we choose the one that we have not explored. But your program only gets a set of linked cells. We need to enable your program to distinguish between unexplored cells and the ones that it has already tried.

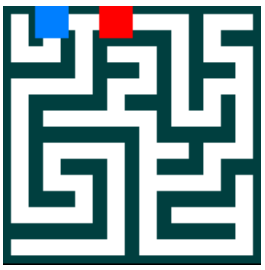

FIGURE 5 – A simple level maze

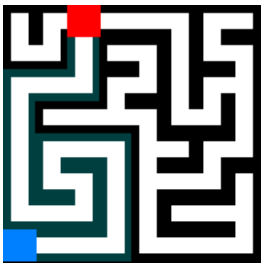

FIGURE 6 – A medium level maze

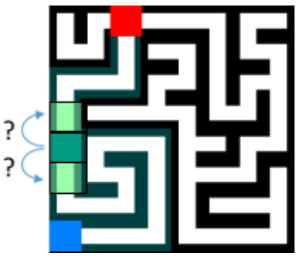

FIGURE 7 – Each cell on the path (for example the green cell), has two linked cells. Which one should we choose for the step ?

1. Write the necessary function prototypes, possibly with a new data type if necessary. If you have defined a new type, explain :
   — in which way this data type enables you to memorize the explored cells ;

   — why the chosen data type is better than other possible types.
2. Write the prototype of a function `unExploredLinkedCells` that, for a given cell `c` (and any other information that you may find necessary) produces an array containing the unexplored linked cells of `c` and the number of these cells.
3. Write the algorithm that will enable you to find the path from the entrance to the exit in the case of medium level mazes.
4. Write the prototypes of any other functions that you may find useful for that algorithm.

3.4 Difficult level (7 points)

Now let us tackle the more challenging problem of branching mazes. Let us assume that your program arrives at a cell where there are several possible unexplored paths. Of course it does not know which is the correct path. So it has to try them all, until it has found a way to the exit cell. Each time your program reaches a *dead-end* (a cell with no unexplored linked neighbors), it has to go backwards until it finds unexplored cells. So your program needs :

— to memorize the cells that lead to the exit point ;
— NOT to memorize the cells that lead to a *dead-end* ;
— to remember which were the cells that it explored the latest, in order to be able to go backwards and search other paths.

1. Write the necessary function prototypes, possibly with a new type if necessary. If you have defined a new type, explain :
   — in which way this data type enables you to accomplish the previous tasks ;
   — why the chosen data type is better than other possible types.
2. Write the algorithm that will enable you to find the path from the entrance to the exit in the case of hard level mazes (arbitrary entrance and exit cells).
3. Write the prototypes of any other functions that you may find useful for that algorithm.

I am expecting you to make an archive file from all your header files, and to hand us this file on the *Moodle* platform before Friday November the 17th at midnight.

A `libMaze.a` library

The `Maze.h` header file in the `include` folder describes the possible operations on mazes.

— `Maze MZ_new(int nbx, int nby, int diffic);`
Allocates and builds a perfect rectangular maze with `nbx` x `nby` cells and returns an object of type `Maze` which contains the useful information : entrance, exit and linked cells. The value of diffic can be :
   — `MZ_SIMPLE` : Easy level : the entrance cell and exit cell are two steps away from each other ;
   — `MZ_MEDIUM` : Medium level : the path between the entrance cell and the exit cell contains no branching. You simply have to follow the walls.
   — `MZ_HARD` : Difficult level : the entrance cell and exit cell are chosen at random.

— `void MZ_free(Maze mz);`
Frees the memory allocated by maze `mz`. After the return of this function `mz` can no more be used.

— `void MZ_size(Maze mz, int *nbx, int *nby);`
After the return of this function, `nbx` and `nby` point respectively to the width and the height of maze `mz`. This width and height are expressed in number of cells. In the case of the maze in figure 4, the following code :

```
int width, height;
MZ_size(mz, &width, &height);
printf("%d %d\n", width, height);
```

would output `8 8`

— `void MZ_entrance(Maze mz, int *nx, int *ny);`
After the return of this function `nx` and `ny` represent the coordinates of the entrance point of `mz`. In the case of the maze in figure 4, the following code :

```
int x,y;
MZ_entrance(mz, &x, &y);
printf("(%d,%d)\n", x, y);
```

would output `(0,1)`

— `void MZ_exit(Maze mz, int *nx, int *ny);`
After the return of this function `nx` and `ny` represent the coordinates of the exit point of `mz`. In the case of the maze in figure 4, the following code :

```
int x,y;
MZ_exit(mz, &x, &y);
printf("(%d,%d)\n", x, y);
```

would output `(7,6)`

— `int MZ_linkedCells(Maze mz, int nx, int ny, int linked_cells[]);`
For a given maze `mz`, a given cell `(nx,ny)` in this maze, and an array of at least 8 integers `linked_cells`, this function fills `linked_cells` with the coordinates of the linked neighbors of cell `(nx,ny)` and returns the number of linked neighbors. In the case of the maze in figure 4, the following code :

```
int linked_cells[8];
int nb_links = MZ_linkedCells(mz, 3, 3, linked_cells);
for( int i=0; i<2*nb_links; i++)
   printf("%d ", linked_cells[i]);
```

would output `3 2 2 3 4 3` which means that cell (3,3) has three neighbors : (3,2) (2,3) and (4,3). Beware `nb_links` is not the number of relevant elements in `linked_cells` but the number of linked neighbors (in our case : 3 and not 6).

— `void MZ_setSolution(Maze mz, int solution_cells[], int nb_cells);`
For a given maze `mz`, for a given array `solution_cells` of integers filled by you, and representing a proposed solution and for a given integer `nb_cells`, the length of the solution path, this function memorizes the solution path and will draw it in the image file at the next call of `MZ_saveImg`. It does NOT check if the solution is correct.

— `void MZ_saveImg(Maze mz, char *filename);`
For a given maze `mz`, and a character string `filename`, this function saves the maze `mz` in an image file. If a solution path was proposed with `MZ_setSolution`, then this solution will be represented in orange in the image file. In the case of the maze in figure 4, the following code :

```
int solution[] = {0,1,0,2,0,3,0,4,0,5,1,5};
MZ_setSolution(mz, solution, 6);
MZ_saveImg(mz,"maze.ppm");
```

proposes a solution with 6 cells : (0,1) (0,2) (0,3) (0,4) (0,5) (1,5), which, of course, is not the correct solution, but `MZ_saveImg` produces the image represented on figure 8.

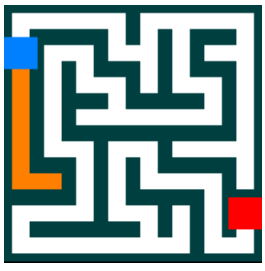

FIGURE 8 – The proposed (wrong) solution is composed of cells (0,1) (0,2) (0,3) (0,4) (0,5) (1,5).