Course: SYSC5709 Software Development with C

Professor: Cristina Ruiz

Date: 6/23/2020

Final Project

Students:
Tareq Dawoudiah, 7136770 (uOttawa), haiduowad
Marwan Ghalib, 100859950, marwanghalib
Griffin Barrett, 100978436, xlirate
Manojkumar Chandrasekaran (uOttawa), 101130903,
mchandra5034

GitHub Repository:
https://github.com/marwanghalib/Group_C_MAZE GENERATOR

User and Developer Manual for Maze Generator

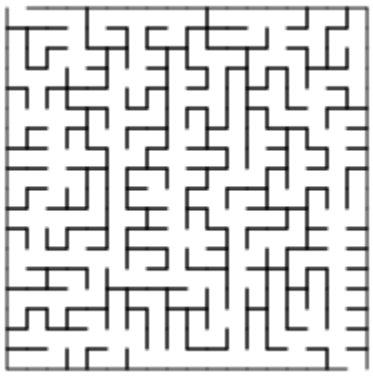
Table of Contents

Introduction:

A maze is a complex structure of interconnected passageways. There should be (at least) one way to get from a designated start location to a designated end. Typically, the path is convoluted and branched (these branches can also be branched, and often leading to dead-ends) making it not obvious to the naked eye the correct path to take (even when exposed to a God's eye view from above with all information exposed).

Mazes are even more challenging to solve when you are inside one and are only exposed to the information you can immediately see!

Different algorithms for generating mazes work in different ways. Some start with a 'solid' block and 'carves' out passages as they progress. Others start with an 'empty' space and 'build' walls. The back track recursive algorithm is a carving algorithm: It starts out as a complete grid with all boundaries and walls set and removes walls to generate the labyrinth.



Overview:

In our project, Maze Generator collects and canonicalizes the input from the user for maze generation. When generating a maze, the user should specify the size, bias, algorithm, characters used to represent the walls of the maze in the output, as well as where to send the output (stdout or print to a file).

In this project we have used Recursive Backtracking, Sidewinder and Kruskal algorithm We have also included the ability to solve a maze provided by the user as input. In this case, instead of providing console arguments, the user pipes a maze in on stdin or specifies a file path in the first argument. This maze solution will come in the form of a set of '.' marking the path form entrance to exit.

User Level Manual

The list of information and its description

Product name
Version Number
Intended use
Description of options
Compilation and installation instructions
1. Prerequisite
2. Download and setup

Description of how to use/operate the product
Troubleshooting section and instructions on how to solve problems

Product Name

Maze Generator

Version Number

0.3

Intended Use:

To create and solve maze using user added values and using different algorithm

Description of option

Short	Long	Description	Default	Arguments
-h	--help	Print a help message		0

	--width	How wide is the maze	10	1
	--height	How tall is the maze	5	1
-s	--size	how wide and tall is the maze	10 5	2
-b	--bias	How biased is the maze: >1 more horizontal hallway <1 more vertical hallway	1.0	1
-a	--alg --algorithm	Which algorithm to use	backtrack	1
-t	--tileset	What characters to use to represent the walls of the maze in the output	hedge	1
-o	--output	Where to send the output	stdout is used	1
	--solve	Solve the maze		0

Compilation and installation

Prerequisite:

gcc complier

Download and setup

1. You need to use the command line git clone https://github.com/marwanghalib/Group_C_MAZEGENERATOR.git to download the repo.
2. Once the repo is downloaded then go to cd Group_C_MAZEGENERATOR.git
3. Then run make
4. Once it compiled you need to change the directory to bin and run cmaze

Step by step with instructions below

git clone https://github.com/marwanghalib/Group_C_MAZEGENERATOR.git
cd Group_C_MAZEGENERATOR.git
make
cd bin
cmaze -options

User need to add the options to specify like height, bias, width, output file location or stdout, algorithm to solve. If the user inputs the wrong value, then the maze will exit and print help function. If user forgot to input the value, then it will take the default value and maze will be generated.

Eg ./cmaze --height --width 5 -a 0 -b 1.0 -o maze.txt

Developer Manual for Maze Generator

Overview:

In our project, Maze Generator collects and canonicalizes the input from the user for maze generation. When generating a maze, the user should specify the size, bias, algorithm, characters used to represent the walls of the maze in the output, as well as where to send the output (stdout or print to a file).

In this project we have used Recursive Backtracking, Sidewinder and Kruskal algorithm We have also included the ability to solve a maze provided by the user as input. In this case, instead of providing console arguments, the user pipes a maze in on stdin or specifies a file path in the first argument. This maze solution will come in the form of a set of '.' marking the path form entrance to exit.

User Input Collection

In the user input collection, we had used coll-args function which collects all the arguments and returns a pointer to the functions. This function collects the arguments. This function collects the arguments that user provided during the input when he runs the maze all the value are collected height, width, tiles, using agrv and once it is collected it will be sent to the first function which is initialize maze and this will return a value and then it is passed to the algorithm which is sidewinder or backtrack on the user input function. This function will also give an option to choose were the output should be displayed whether its stdout or file path

type used for all the arguments

height - int

width - int

size - int

bias - double

algorithm - char

tileset - char

output - char

Function called

```
void (*coll_args (int argc, char **argv,maze_t* maze,tile_set_t** tile_set, FILE** output_file))(maze_t*){
    //assigning default values to the variables which will reset if user input something
    int width = 10;
```

Maze Generation

In Maze generation they are three number of functions is used this will initiate, destroy, set bias

Init Maze:

This will initialize the maze object with specified width, height and start in the top left, and end in the bottom right at the address of the maze

```
int init_maze(int width, int height, maze_t *maze);
```

Set_bias:

This function is used to set the bias filed of a maze it requires.

0 < bias < inf

maze is in a valid state

ensure

if bias < 1 NOP rather than doing anything

bias is updated

maze is in a valid state

end

```
void set_bias(double bias, maze_t *maze);
```

Destroy Maze:

This function will destroy if the maze is invalid

```
int destroy_maze(maze_t *maze);
```

Set_Cross:

```
void set_cross(cell_t cell, int x, int y, maze_t *maze);
```

is

updates maze with cell at x,y relating to

corner_north, corner_west, north, and west,

ignoring doors outside the body of the maze and ignoring start and end

requires

maze is in a valid state

ensure

maze is updated

no illigal writes

maze is in a valid state

end

Set Room:

is

updates maze with cell at x,y relating to

north, south, east, and west,

ignoring doors outside of the body of the maze, and ignoring start and end

require

maze is in a valid state

ensure

maze is updated

no illegal writes

maze is in a valid state

end

```
void set_room(cell_t cell, int x, int y, maze_t *maze);
```

Tile Set

Delete Tileset:

```
This function ensure it deletes and free a tileset and ensure there is no memory is leaked
```

int delete_tile_set(tile_set_t* set);

New Tiles Set:

tile_set_t* new_tile_set(char* wall);

This function allocates and init a tileset with defaults and specifies wall sting requires. Wall is a valid null terminated string

do

copies from and does not retain argument pointer

caller still owns the string

and it ensures tile_set is in a valid state, default space is " ", default start is "<" , default end is ">" both walls and all corners are set to wall

Algorithm

Maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells. The purpose of the maze generation algorithm can then be making a subgraph in which it is challenging to find a route between two nodes.

If the subgraph is not connected, then there are regions of the graph that are wasted because they do not contribute to the search space. If the graph contains loops, then there may be multiple paths between the chosen nodes. Because of this, maze generation is often approached as generating a random spanning tree. Loops, which can confound naive maze solvers, may be introduced by adding random edges to the result during the algorithm.

Properties of Maze

Mazes have characteristics that describe them. A maze is classified as ‘perfect’ if it does not contain loops (as we will see later, the dual of a maze is a graph, and if this graph is a single tree with no cycles then it is a perfect maze. A perfect maze can also be described as a ‘simply connected’ maze.

If a maze is simply connected it is possible to solve it using a wall following algorithm. By always keeping you right hand (or left if you prefer!), against the maze wall and walking around you will walk a path that will eventually visit every location in the maze and return to the same location.

Backtracking

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Pseudo Code for Backtracking FIND-PATH(x, y)

1. if (x,y outside maze) return false
2. if (x,y is goal) return true
3. if (x,y not open) return false
4. mark x,y as part of solution path
5. if (FIND-PATH(North of x,y) == true) return true
6. if (FIND-PATH(East of x,y) == true) return true
7. if (FIND-PATH(South of x,y) == true) return true
8. if (FIND-PATH(West of x,y) == true) return true
9. unmark x,y as part of solution pat
10. return false

```
void gen_backtrack(maze_t* maze) {

    set_size_of_extra(sizeof(struct backtrack_extra), maze);

    /* Do the generation here */

    //Displaying array elements
    printf("STARTING 2-D array elements:\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            //printf("%d ", maze[i][j]);
            if (j == (4 - 1)) {
                printf("\n");
            }
        }
    }
```

Sidewinder

Sidewinder Maze Generator is very similar to the Binary Tree algorithm, and only slightly more complicated. Furthermore, the Sidewinder algorithm only needs to consider the current row, and therefore can be used to generate infinitely large mazes (like the Binary Tree).

While binary tree mazes have two of its four sides being one long passage, a Sidewinder mazes have just one long passage.

Pseudo Code for Sidewinder

In a nutshell, it goes like this:

Work through the grid row-wise, starting with the cell at 0,0. Initialize the “run” set to be empty.

Add the current cell to the “run” set.

For the current cell, randomly decide whether to carve east or not.

If a passage was carved, make the new cell the current cell and repeat steps 2-4.

If a passage was not carved, choose any one of the cells in the run set and carve a passage north. Then empty the run set, set the next cell in the row to be the current cell, and repeat steps 2-5.

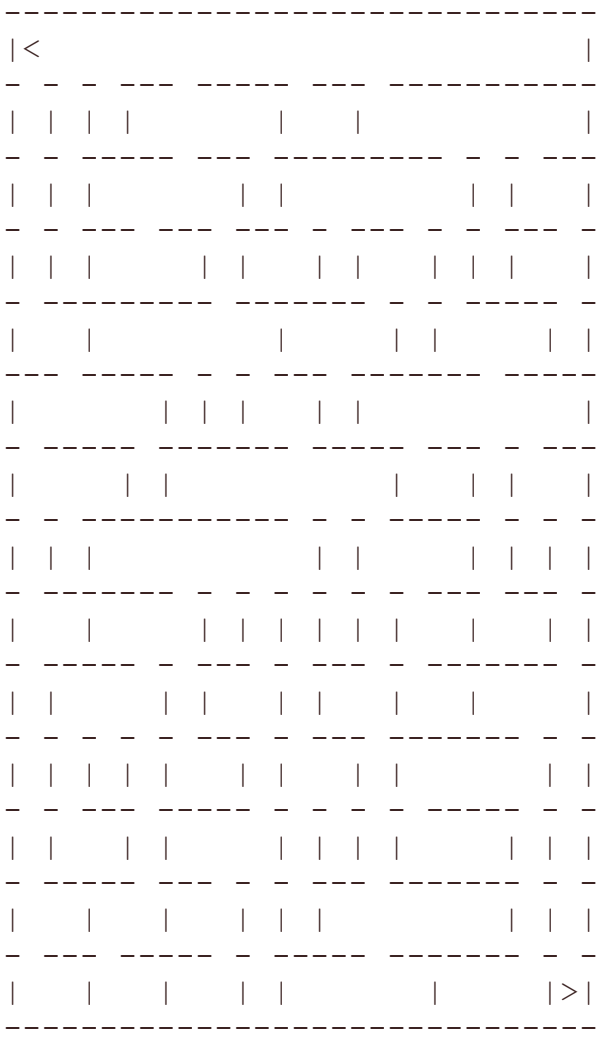
Continue until all rows have been processed.

Function to Generate Sidewinder Maze

```
// Generates a maze based on the Sidewinder algorithm
void gen_sidewinder(maze_t* maze) {

    // enables random number generation
    srand((unsigned)time(NULL));

    // start at the specified maze object starting coordinates
    struct CELL_COORDINATES *current_cell_coordinates = (struct CELL_COORDINATES*)malloc(sizeof(struct CELL_COORDINATES));
    (*current_cell_coordinates).row = maze->start_y;
    (*current_cell_coordinates).column = maze->start_x;

    // clear entire 1st row (i.e. set cell_t.east to 0 for each cell in that row)
```

This function will generate maze for sidewinder, it starts with a random number for generation and specifies the maze object with starting coordinates

Expected Output for Sidewinder with tile set, size will vary depending on the user value



Expected Output for sidewinder without tileset

```
#########
#<        #
# # ### #
# # #    #
# ### ###
# #       #
### ### #
#    #    #
# # # # #
# # # #>#
#########
```

Parser Function:

This function will take in a maze object, look at its file pointer that contains

a text file with a maze and populate the maze object data.

After running parser, you can now take this maze object and pass it to the printer to get printed

```
void parse_maze(maze_t *my_maze);
```

Print Function:

This function will print the maze, the parameters used her are the maze_t tile_set_t and the file destination

```
void print_maze(maze_t *myMaze, const tile_set_t *myTileSet , FILE* destination);
```

References:

Links for Maze algorithm references

https://medium.com/analytics-vidhya/maze-generations-algorithms-and-visualizations-9f5e88a3ae37

http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking

http://datagenetics.com/blog/november22015/index.html

https://en.wikipedia.org/wiki/Maze_generation_algorithm