Maze

Due: Wednesday, September 18, 2019 at 11:59pm

Note: Please make sure you read the handout all the way through before getting started or going to hours!

0 Web Handout

This year we have introduced a web-native handout for each of our assignments. Regardless of whether you are using those or the pdf, we would really appreciate your feedback!

1 Introduction

Oh jeepers! While Tom was scoping the deep waters s-Pacifically for colorful octopuses to photograph for his underwater photo collection, he encountered a ghastly C-monster! In order to escape, Tom must create a maze of kelp to stump the C-monster and swim away to safety. Complete this assignment and help Tom flee the C-monster. Shrimply put, his life is in your hands!

2 Assignment

This C programming assignment contains two parts: first you will write a program *generator*, which generates mazes; then you will write a program *solver*, which solves those mazes.

To get started, run:

```
cs0330_install maze
```

This will copy the stencil for this project into **`~/course/cs0330/maze`**.

Note: You might find it useful to run and test bits of code in an **online C compiler**. This is a super helpful to make sure each part of your code does what you think it's doing. Use it!

3 Stencil

You do **not** need to modify **`generator.h`** or **`solver.h`**. You **will** need to modify **`generator.c`**, **`solver.c`**, **`common.c`**, and **`common.h`**. We recommend implementing the functions (and structs) in the following order:

Start with **`common.h`**:

1. **maze_room** struct

Next in **common.c**:

2. **initialize_maze** function
3. **is_in_range** function
4. **get_neighbor** function

Then in **generator.c**:

5. **get_opposite_dir** function
6. **shuffle_array** function
7. **drunken_walk** function
8. **encode_room** function
9. **encode_maze** function
10. **main** function

Finally in **solver.c**:

11. **create_room_connections** function
12. **decode_maze function**
13. **dfs** function
14. **print_pruned_path** function
15. **main** function

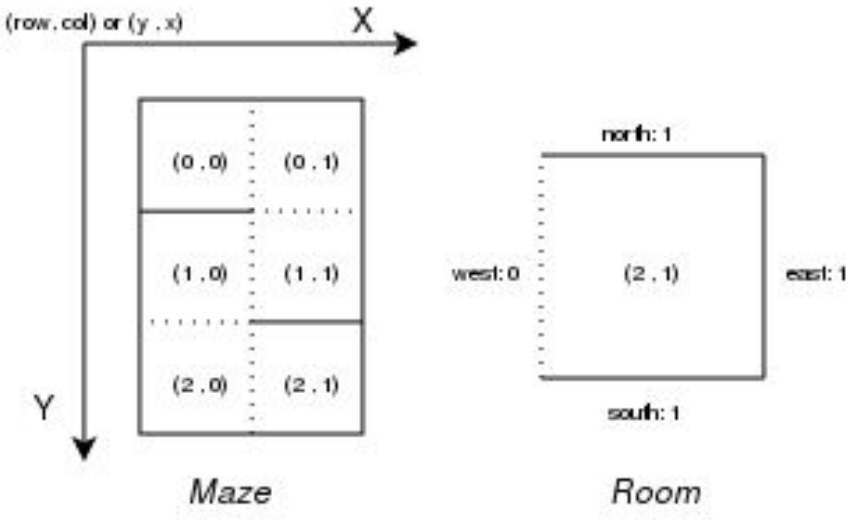
NOTE: This order is simply a recommendation. Feel free to implement these functions in whatever order you find most intuitive!

4 Maze Structure

In the Life lab, you learned how to index into a one-dimensional array to represent a two-dimensional array. In this project, however, we will be representing mazes as actual two-dimensional arrays.

Room indices start as **(0,0)** at the **top left corner**, and grow as you move down and to the right. This means the lower-right corner of a **10 × 25** maze would have coordinates **(9,24)**.

Each room will have either a **wall** (stored as a **1**) or an **opening** (stored as a **0**) for each possible direction (North, South, West, East). Connections should be consistent between rooms, and rooms on the edges should always have a wall in that direction (or two walls, in the case of corners).



You **must** use a **2D array** to represent your maze. The stencil declares maze arrays as

```
struct maze_room maze[num_rows][num_cols]
```

This shows up in the signature of each function, so be sure to build your maze array in the same way. **Always make sure the first index is the row, and the second index is the column.**

For each room, you will need to keep track of the following:

- the **`row`** and **`column`** of the room.
- whether or not the room has been visited.
- for each connection of the room, whether that connection is a wall, opening, or is uninitialized.

Make sure you initialize all values.

NOTE: You may notice there is already a `maze_room *next` field in the `maze` struct. This will only become relevant when you are implementing your solver, so don't pay attention to it for now.

5 Generator

The first part of this project will be to generate a maze.

5.1 Algorithm

There are a few ways to generate a maze, but the simplest uses the recursive *drunken-walk algorithm*. This algorithm chooses a random order in which to visit each room of the maze, and then creates walls based on that order.

This algorithm will guarantee there is a path from any room to any other room. The reason for this is because any room it encounters which it hasn't visited will be given a connection to the previous room. No room will ever be "blocked off" from the chain of visited rooms, so all rooms will be accessible from each other.

Pseudocode:

```
drunken_walk(row, col):
    r = rooms[row][col]
    set r.visited to true
    for each direction dir in random order:
        if (row + row_offset of dir, col + col_offset of dir) is out of bounds:
            store a wall in r at direction dir
        else:
           neighbor = rooms[row + row_offset of dir][col + col_offset of dir]
            if neighbor has not yet been visited:
                store an opening in r at direction dir
                drunken_walk(neighbor.row, neighbor.col)
            else:
                opposite_dir = the opposite direction of dir
                if neighbor has an initialized value in direction opposite_dir:
                    store that value in r at direction dir
                else:
                    store a wall in r at direction dir
```

5.2 Random Number Generation

To ensure that the maze generated by your program is different every time, you'll need to use C's **`rand()`** function, which takes no arguments and returns an integer between **`0`** and **`RAND_MAX`**. To get a random number between **`0`** and **`n-1`**, you can take **`rand() % (n)`**.

The **`rand()`** function is actually a *pseudorandom number generator,* meaning that it outputs a consistent sequence of values when given a particular seed value. By default, **`rand()`** has a seed value of 1, so **unless you change this, your program will generate the same sequence of random numbers each time it is run.**

To change the seed value, include the line **`srand(time(NULL))`** at the beginning of your **`main()`** function.

To randomize the order of directions through which you will search, declare the directions in some fixed order in an array. Then the following algorithm can be used to shuffle that array in-place:

```
shuffle_array(A[n]):
    for i from 0 to n-1:
        choose a random number r between i and n-1, inclusive
        switch A[i] and A[r]
```

This procedure produces all possible orderings with equal probability.

6 Maze Encoding, Decoding, and Translating

6.1 Encoding

In order for you to save the mazes you generate you need some way to represent them in a file. (Your **`generator`** will be writing to this file, and your **`solver`** will be reading from it). Since each room has four connections, each of which can be in one of two states (wall or opening), there are $2^4 = 16$ possible configurations a room can be in. Therefore, you will be using a number from **`0`** to **`15`** to represent each possible room connection configuration. We will call this number the **room encoding**.

We can use one bit to represent each connection (total of four), so we'll be using the four lowest-order bits of an int. (Note that an int is made up of 32 bits. We are only using 4 out of the 32 bits in the int, and will be ignoring the 28 highest-order bits.)

Each of these four bits will represent a connection. Specifically,

- the highest-order bit represents the **east** connection
- the next-highest bit represents the **west** connection
- the next-lowest bit represents the **south** connection
- the lowest-order bit represents the **north** connection

As an example, a room with walls to the east, west, and north, and an opening to the south would be represented as **`1101`** in binary, which is equal to **`13`** in decimal, so **`13`** would be its room encoding.

NOTE: Your maze MUST conform to this specification.

6.2 Decoding

Given a room encoding as an **int**, you will need to be able to extract the connections. This can be done using bit-level operations. The most relevant operator will be the bitwise **AND** (**&**), which compares each bit of the two operands. If both bits are 1, then the corresponding resultant bit is set to 1. Otherwise, it is set to 0. Some examples are:

```
    1011 (11)              1011 (11)
  & 0010 ( 2)            & 0100 ( 4)
    0010 ( 2)              0000 ( 0)

In C:  11 & 2 = 2          11 & 4 = 0
```

We can use the **&** operator to extract the value of a particular bit from a room encoding by using a *bit mask*, or an integer whose binary representation consists entirely of zeros except for a particular bit (or bits). Some examples include **4 (0100)** and **1 (0001)**.

To check whether the **i**th bit of a value is set (zero-indexing), you can just **&** it with 2^i.
(e.g. **if (x & 4) {printf("third bit is set")}**).

6.3 Translating Between Representations

Your **generator** will be writing the encoded maze to a file, and your **solver** will be reading it. We have chosen to write each room encoding in hexadecimal, since the numbers **0** through **15** are all one character long in hex. Here is an example of what an encoded maze will look like:

```
597333331395397313333313b
c6339595adccd639633b59639
cd53286a70ac619c5333a639c
c4a59e5396969cc6ad51b53ac
ce5a632bc5a5ac61b4ac5a738
432339ddcc5a5adc5ad6a5958
cd5396accccdc58cc5239c6ac
cccd633accc4aec6a639cc59c
68c43339cccc5949719cc6acc
7a6a7332a6a6a6a63a6a633ae
```

We have provided for you the functions responsible for reading and writing the encoded maze to and from a file (**write_encoded_maze_to_file** and **read_encoded_maze_from_file** respectively). You are responsible for correctly encoding the maze before calling **write_encoded_maze_to_file**, and correctly decoding the encoded maze returned by **read_encoded_maze_from_file**.

To debug, we encourage you to try out **very small mazes first**, and try using the encoded maze to **draw it by hand**!

NOTE: We encourage you to read over each of these functions we provided to see how we are reading and writing from the files!

7 Solver

The second part of this project is to write a solver that will solve the mazes the you (or we) generate. Please make sure your generator works before starting on this section. (see Support section)

7.1 Algorithm

Your program should employ a *depth-first search*. Such a search begins at the maze's start room and explores adjacent, accessible rooms recursively.

Beginning with the indicated room, this algorithm repeatedly chooses a path from each room and follows that path until it reaches a dead end, at which point it backtracks and tries a new path. This process continues until all paths have been explored or the destination is found. The following is pseudocode for this algorithm.

```
dfs(row, col):
    if (row, col) are the coordinates of the goal
        return true
    mark the room at [row][col] as visited
    for each direction dir:
        neighbor = rooms[row + row_offset of dir][col + col_offset of dir]
        if the connection in direction dir is open and neighbor is unvisited:
            if dfs(neighbor.row, neighbor.col) is true
                return true
    /* if the program reaches this point then each neighbor's branch
        has been explored, and none have found the goal. */
    return false
```

7.2 Solver Output

Your program should print a list of rooms to the given output file. The room coordinates should be formatted in `<row>, <col>` format when printed to the solution file.

To write to a file, use fprintf:

```
fprintf(FILE *file, char *content)
```

In the fprintf function, “file” is a pointer to the file where we want to write, and “content” is the string (**`char`** pointer) that we want to write.

Here’s an example of what the first several rows of your solution file might look like.

```
PRUNED
0, 0
1, 0
1, 1
1, 2
2, 2
...
```

We expect your solver to produce two different modes of output:

- **Pruned Mode:** Your program outputs the coordinates of only the final route from beginning to end. Your program should first print the line **“`PRUNED`”**. The program should then print the coordinates of each room on the solution path as described earlier.

  To do this, build a list of rooms as you search, and print out each room in the list when you reach the destination room. You can accomplish this using pointers! Use the provided **`next`** pointers in your room structs to maintain a linked list of rooms - when you move from room *A* to room *B*, set room *A*’s pointer to room *B*.

- **Full Mode:** Your program outputs the entire path traversed up until the goal is reached. Your program should first print the line **“`FULL`”**. The program should then print each room’s coordinates when first visiting that room, and after each recursive call that returns false. This will print the path from start to finish, including “backtracking” after dead ends.

Note: Depending on how your solver algorithm searches the maze, there can be multiple valid `FULL` solutions.

The choice should be made when your program is compiled. This is done using preprocessor macros. Macros are defined using the `gcc` compiler flag **`-D<macro>`**, which defines **`<macro>`** for the preprocessor. For example, to add the macro **`PIZZA`** to your program, add the flag **`-DPIZZA`**. In your Makefile, you’ll see the flag **`-DFULL`** in the command for the **`solver_full`** target, which defines the macro **`FULL`** for that target.

To write code that will execute only when a specific macro is defined, refer to the example below:

```
#ifdef FULL
printf(<something>);
#else
printf(<something else>);
#endif
```

The above code fragment executes the **printf(<something>)** statement only if the macro **FULL** is defined, and executes the **printf(<something else>)** statement otherwise. You can also use the macro **#ifndef <macro>** to execute code only if **<macro>** is not defined.

Your program should print the entire search if a macro **FULL** is defined and print only the path to the exit otherwise. Rooms should be printed with format **<row>, <col>** on its own line with no parentheses with the upper-left corner of the maze corresponding to coordinate `(0, 0)`. If the start and end rooms happen to be the same, your output should contain the room only once.

8 Input and Output

In this assignment, you will need to work with files to represent and solve your mazes. The C **<stdio.h>** library contains several definitions that enable you to easily write to or read from files. Included in these definitions is a **FILE** struct, which represents a file within a C program.

8.1 Opening a File

To use files (reading, writing, etc), you need to first open it. The **fopen()** function opens a file, returning a pointer to a **FILE** struct that corresponds to the desired file.

```
FILE *fopen(char *filename, char *mode)
```

The desired file is indicated by `filename`. The `mode` argument refers to how the file will be used; if you intend to write to the file, this value should be **"w"**, and if you intend to read from the file, it should be **"r"**.
If the desired file does not exist it will be created. If an error occurs, **NULL** is returned.

8.2 Writing to a File

In fact, you can write data to any file (not just standard error) using the **fprintf()** function. This function works in very much the same way as **printf()**.

```
int fprintf(FILE *stream, char *format, ...)
```

The only difference is that **fprintf()** takes an additional argument: the **FILE *** that you obtained with **fopen()**.

8.3 Closing a File

After your program has finished writing to or reading from a file, it should close that file. Do this with the function **`fclose()`**.

```
int fclose(FILE *fp)
```

This function returns `0` if no error occurred and EOF (a negative value) otherwise.

9 Error Checking

Throughout your code you will be using library calls, like **`fopen`**, **`fprintf`**, and **`fclose`**. These functions may fail, and may return an error. **You are expected to check for errors every time you make a library call.** When a library call returns an error, use **`fprintf`** to write an error message to **`stderr`**, and then stop program execution by executing **`return 1`** from **`main()`**.

```
if (fclose(f)) {
      fprintf(stderr, "[Error message goes here.]\n");
      return 1;
}
```

Most functions will return a certain value to denote an error. Find out what those values are with **`man <function>`**, or looking it up in official online documentation. Your **`main()`** function should return `0` if it completed execution normally, or `1` if it exited on encountering an error.

Note: If you are calling `fprintf` to write to `stderr`, you do not need to error check it.

10 Compiling and Running

10.1 Compiling

You have been provided a *Makefile*, a text file that contains scripts for compiling, running, or cleaning up projects (for example). In order to test the first half of the assignment, you will only need **`generator`** and **`clean`**.

The make command only builds files that have been modified since the last build and allows you to split up your build process (e.g. splitting up generator and solver binaries to be built).

Command	Function

make <target>	Builds a particular target. If no target is specified, it will build the first target (in this case, the target **all**).
make clean	Removes any previously built targets
make generator	Builds your generator program
make solver	Builds your solver program with no macros defined (i.e. the program should print pruned output)
make solver_full	Builds your solver program with the **FULL** macro defined (i.e. the program should print its full exploration path).
make all (or just **make**)	Builds EVERYTHING (your **generator**, **solver**, and **solver_full** programs)
make clean all	Shorthand for running **make clean** followed by **make all**

10.2 Running

Once you have compiled the generator portion of the project, you can run it with the following commands:

Command	Arguments
./generator	`<output maze file> <rows> <cols>`
./solver	`<input maze file> <rows> <cols> <output solution file> <starting row> <starting col> <ending row> <ending col>`
./solver_full	`<input maze file> <rows> <cols> <output solution file> <starting row> <starting col> <ending row> <ending col>`

Note: Mazes that are very large may cuase segmentation faults, so we will not test your code with a maze of a size larger than 250 x 250.

10.3 Support

Command	Function
`cs0330_maze_generator_demo`	Demonstrates the expected behavior of your **`generator`** program.
`cs0330_maze_solver_demo`	Demonstrates the expected behavior of your **`solver`** program.
`cs0330_maze_solver_full_demo`	Demonstrates the expected behavior of your **`solver_full`** program.
`cs0330_maze_validator`	This program will check your maze for errors, such as inconsistent or missing walls and inaccessible areas. Please make sure to validate your generated maze before moving on.

11 Grading

Your grade for this project will be calculated as follows:

Generator	40%
Solver	40%
Error Handling	10%
Style	10%
Total	100%

Both generator and solver will be graded based on Code Correctness (15 pts) and Functionality (25 pts).

- **Code Correctness:** no part of your code relies on undefined behavior, uninitialized values, or out-of-scope memory; your program compiles without errors or warnings. *Only* **`cs0330_maze_validator`** *will be used during grading*. (i.e It's ok if your maze doesn't work in the visualizer.)
- **Functionality:** your code produces correct output, and does not crash for any reason. It does not terminate due to a segmentation fault or floating point exception.

- **Error Handling:** your code performs error checking on its function inputs and outputs, and exits gracefully in all situations. No input to your generator or solver should cause a segmentation fault.
- **Style:** your code should look nice! Use appropriate whitespace and indentation and well-named variables and functions. Your code should be reasonably factored, and functions should not be too long. Make sure your functions have header comments, and if they already do then don't delete them!

Your programs should perform error checking on their input, with one exception: if your solver program successfully opens a maze file, you may assume that the file contents form a correctly-formatted maze. Your program should not crash for any reason; before you hand in your project, make sure that your program does not terminate due to a segmentation fault or floating point exception.

Consult the **Style Guide** for some pointers on C coding style. Note that you can run a style formatting script in order to make your code match *some* of the style specifications. To use the script, run the command

```
cs0330_reformat <file1> <file2> ...
```

To reformat all `.c` and `.h` files in your current directory, you may run:

```
cs0330_reformat *.c *.h
```

Check the style guide for more information.

Note: the reformat script should only be used on .c and .h files

See the table below for guaranteed grade cutoffs. If you do not meet the threshold for a given letter grade, you may still receive that grade after Professor Doeppner applies a curve (you will only ever be curved up).

Grade	Requirements
A	Consistently generate valid mazes AND correct solutions (both FULL and PRUNED), and does not segfault under most circumstances.
B	Consistently generate valid mazes AND correct solutions (at least one of FULL or PRUNED is correct), and does not segfault under most circumstances.
C	Consistently generate valid mazes OR correct solutions (at least one of FULL or PRUNED is correct), and may often segfault.
Failing	You can get your project checked off for a C up until the next project deadline

12 Handing In

To hand in your project, run the command

```
cs0330_handin maze
```

from your project working directory. You should hand in **ALL FILES** (**`common.c`**, **`common.h`**, **`generator.c`**, **`generator.h`**, **`solver.c`**, **`solver.h`**, along with any support code you may have written), a **`Makefile`**, and a **`README`**. Your **`README`** should describe the organization of your programs and any unresolved bugs.

If you wish to change your handin, you can do so by re-running the script. Only your most recent handin will be graded.

Important note: In order to get grades back to the class in a timely manner, we will have to start grading before the final deadline to submit an assignment for any credit has passed (6 days after the on-time deadline). Since students may hand in multiple times, and we always grade the most recent handin, we need a way of knowing not to start grading students who are planning to hand in again later.

If you have already handed in your assignment, but plan to hand in again after the TAs start grading at noon on Saturday, September 21st, you must run **`cs0330_grade_me_late maze`** *to inform us. You must run the script by noon on 9/21*. **If you run this script, you will get grades back later than other students.**

If you do not run this script, the TAs will proceed to grade whatever you have already handed in, and you will receive a grade report with the rest of the class that is based on the code you handed in before we started grading. It would be unfair to ask your UTAs to re-grade your new code after they've already put time and effort into grading your original handin.

Exercise caution when running this script: by running it, you forfeit the privilege of on-time feedback. If something changes, you can run the script with the **`--undo`** flag (before noon on 9/21) to tell us to grade you on-time and with the **`--info`** flag to check if you're currently on the list for late grading.

These instructions apply to all projects unless otherwise stated on the handout; the deadline to run the script will be noon on the Saturday after each assignment is due.