

Maze

Due: Wednesday, September 22, 2021 at 11:59pm EST

0 TA Hours Policy and Ed Post Expectations	2
1 Introduction	2
2 Assignment	2
2.1 Installation	2
3 Stencil	3
4 Maze Structure	3
5 Generator	5
5.1 Algorithm	5
5.2 Random Number Generation	6
6 Maze Encoding, Decoding, and Translating	6
6.1 Encoding	6
6.2 Decoding	7
6.3 Translating Between Representations	8
7 Solver	8
7.1 Algorithm	8
7.2 Solver Output	9
8 Input and Output	11
8.1 Opening a File	11
8.2 Writing to a File	11
8.3 Closing a File	11
9 Error Checking	12
10 Compiling and Running	13
10.1 Compiling	13
10.2 Running	13
10.3 Debugging	14
10.4 Support	14
11 Grading	16
12 Handing In	17

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

0 TA Hours Policy and Ed Post Expectations

You are responsible for reading through the [TA Hours Policy](#) and the [Ed Post Guidelines](#). Please join the [course Ed](#), which will be critical to your success in this course.

1 Introduction

Cruisin' on down Main Street, you're relaxed and feeling good. Next thing that you know, you're seeing a red blood cell in your neighborhood! Now that you've successfully cleaned the Magic 3school Bu3 ([Lab 1](#)), Mr. Twizzle wants to take you on an adventure to learn more about human anatomy. Mr. Twizzle shrinks you and your classmates down to explore Ralph's circulatory system. Unfortunately, Liz gets lost along the way in the maze of blood vessels. You'll need to generate a maze and find a path through Ralph's arteries to get back to Liz!

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.

Tips:

- You might find it useful to run and test bits of code in an [online C compiler](#). This is super helpful to make sure each part of your code does what you think it's doing. Use it!
- After reading the handout in completion, be sure to read the [FAQ for maze on Ed](#). This will help answer a lot of questions you may have.
- Complete [Lab01](#) to learn how to use **gdb**, which will be your best friend (and debugger) throughout the course.
- The TAs will be holding a gear up on **Monday, 9/13 at 8 pm** via Zoom. Join us to get some tips on how to get started, or to ask any questions on the project! Please refer to Ed for the Zoom link, or the course website for the slides and recording if you are unable to attend.

2.1 Installation

If this is your first time installing an assignment, or you need a refresher, please refer to the [Github & Gradescope Guide](#). This will walk you through how to get setup with Git and install an assignment.

The Github assignment link can be found [here](#). Be sure to make commits frequently so you won't lose your work!

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`; however, please **do not** modify the given function headers within these files to ensure that your submission works correctly with the autograder. 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`:

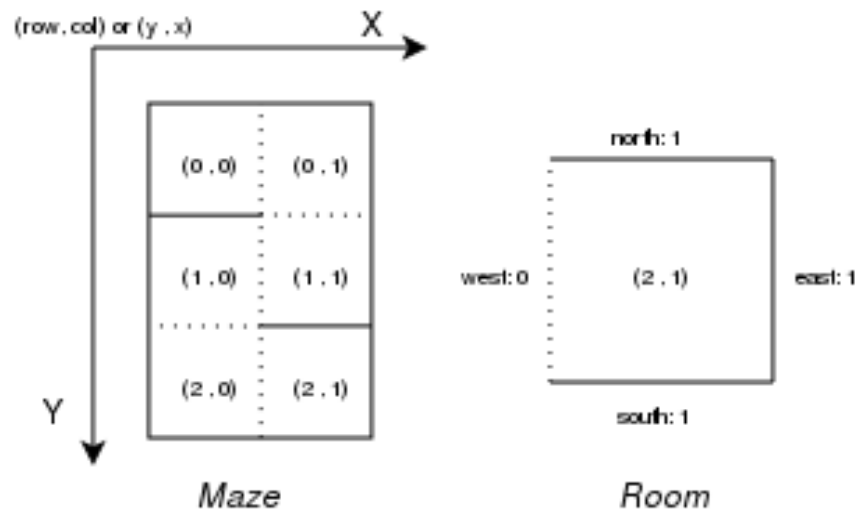
11. `create_room_connections` function
12. `decode_maze` function
13. `dfs` function
14. `print_pruned_path` function
15. `main` function

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

4 Maze Structure

The room at the **top left corner** of the maze has an index value of `(0,0)`. Room index values 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 a connection in each of the four cardinal directions (North, South, West, East). A connection is either a **wall** (represented by a **1**) or an **opening** (represented by a **0**). Connections should be consistent between rooms, i.e. adjacent rooms should share the same type of connection (wall or opening) between them, and rooms on the edges of the maze 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 yet to be determined. Think about how you might initialize connections between rooms to a value that is neither a wall nor an opening.

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, 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 encounters a wall or opening in direction opposite_dir:
                    store that value in r at direction dir
                else:
                    store a wall in r at direction dir
```

NOTE: `row_offset` and `col_offset` refer to the offset required to get the neighbor of the current room in the desired direction. For example, North would require a row offset of -1. `Dir` is the four cardinal directions (North, South, West, East).

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. This line makes use of your computer's clock to set the seed, and since time is always changing the seed also continuously changes (which is what we want).

To randomize the order of directions through which you will search, use the following algorithm to shuffle an 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, the **Knuth/Fisher-Yates Algorithm**, produces all possible orderings with **equal probability**. It is imperative that you use this method for correct randomization, and **points will be taken off** if this algorithm is not followed. For an in-depth explanation on why simple shuffling does not work as well, click [here](#). Use the [online compiler/gdb](#) to make sure your randomizing is happening correctly.

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, so we 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 **encode_room** would return **13** for that room

NOTE: Your maze MUST conform to this specification

6.2 Decoding

Now that we've encoded a room's connections as an **int**, you will need to decode this representation to get the connections for each room. 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:

$$\begin{array}{r} 1011 \text{ (11)} \\ \& 0010 \text{ (2)} \\ \hline 0010 \text{ (2)} \end{array}$$

$$\begin{array}{r} 1011 \text{ (11)} \\ \& 0100 \text{ (4)} \\ \hline 0000 \text{ (0)} \end{array}$$

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 . Within the scope of this assignment, it is acceptable to use the numerical value itself (8 instead of computing 2^3).

As an example, the following uses a bit mask for the third lowest-order bit:

```
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
cd5396accccd58cc5239c6ac
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 as an array of `ints` before calling `write_encoded_maze_to_file`, and correctly decoding the encoded maze returned by `read_encoded_maze_from_file`. You are **not** responsible for converting the `int` encoding into a hexadecimal encoding.

To debug, we encourage you to try out **very small mazes first**, and try using the encoded maze to **draw it out 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 you (or we) generate. Please make sure your generator works before starting on this section. You may test your implementation against our demo (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 a 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 stream 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 **maze_room** structs to maintain a linked list of rooms - when you move from room *A* to room *B*, set room *A*'s **next** 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.

NOTE: The autograder will not accept submissions that print the paths in different formats. Make sure to include the PRUNED and FULL headers in your solution.

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.

A file should be opened before you read or write to it and then closed after the file is no longer needed. In the scope of this project, this should only need to be done once for each file.

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.
- You can use the **FILE *** returned by **fopen()** to pass into other functions now, like **fprintf()** (described below!).

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. Note that in C, the file will be closed automatically when the program terminates normally. You can close the file 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;
}
```

Alternatively, you can exit the program from any point (not just the `main()` function) with the `exit` function

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

Most functions will return a certain value to denote an error. Find out what those values are with `man <function>` or by looking it up in official online documentation. **Your `main()` function must 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.

Ensure that no inputs to your program will result in a segfault. In the case of an error, print an error message and ensure that the correct value is returned. The exact content of the error message is not important as long as it is informative. You can also look at the given functions `write_encoded_maze_to_file` and `read_encoded_maze_from_file` as an example for how to properly error check. When running the solver program, you will **not** need to check that the dimensions of the inputted maze file match the dimensions given.

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 `make generator` and `make 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).

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

10.2 Running

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

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

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

10.3 Debugging

To debug a program, first compile it into an executable as explained above. Then, you can pass the executable to GDB and debug it.

If we wanted to debug our generator program, for example, we might run the following commands

```
> make generator
> gdb generator
```

Then in GDB we run the program

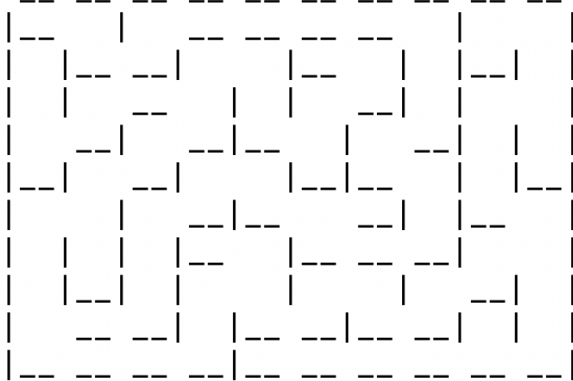
```
> r <mazefile> <num_rows> <num_cols>
```

For a much more detailed explanation of how to debug C programs with GDB, please refer to our [GDB Cheatsheet](#). Also feel free to come to TA hours and learn more about GDB. Our TAs will be more than happy to show you how to use it!

10.4 Support

In our stencil, we have also provided the following support binaries:

<u>Command</u>	<u>Function</u>
<code>./cs0330_maze_generator_demo</code>	Demonstrates the expected behavior of your generator program.

<code>./cs0330_maze_solver_demo</code>	Demonstrates the expected behavior of your solver program.
<code>./cs0330_maze_solver_full_demo</code>	Demonstrates the expected behavior of your solver_full program.
<code>./cs0330_maze_validator</code>	This program will check your maze and solution paths for errors, such as inconsistent or missing walls, inaccessible areas, and paths that jump around. When you are done with your generator, please make sure to validate your generated maze before moving on.
<code>./cs0330_maze_visualizer</code>  Sample output (above)	<p>This program was made by a student last year and prints out a visualization of a generated maze (from the maze file created by your generator program). This is not meant to replace the validator and is not used in grading. We are providing it only because it can be fun to see the actual mazes you are generating.</p> <p>Note: You must make the file executable by running the following command before trying to run the visualizer</p> <pre>chmod +x cs0330_maze_visualizer</pre> <p>Usage: <code>./cs0330_maze_visualizer <input maze file> <num rows> <num cols></code></p>

Troubleshooting:

- If you are unsure about the usage of an executable, try running it with no arguments, and the program will tell you what arguments to provide.
- If running any of the above programs gives you an error that says something along the lines of “Exec format error. Binary file not executable,” you can:
 - Make sure you’re running your code in a Linux environment, either on your VM or on the department machines.
 - Run the bash command: `chmod +x <file name>`

- This will allow the file to be executable. Copying the files from Git sometimes does not preserve the permission bits of a file.

11 Grading

Your grade for this project will be calculated as follows.

Generator	40%
Solver	40%
Error Handling	20%
Total	100%

The autograder grades your generator and solver for the following:

- **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. `cs0330_maze_validator` is what's used by the autograder to test your maze. There are no other "hidden" scripts that we'll use to test it.
This means you can run `cs0330_maze_validator` locally, and do not have to upload your submission to Gradescope each time you want to test it! The Gradescope autograder will just run the test script with many different inputs and edge cases.
- **Functionality:** your code produces correct output, and does not crash for any reason. It does not terminate due to a segmentation fault.
- **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.

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.

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	Autograder Score	Requirements
A	[100]	Consistently generates valid mazes AND correct solutions (both FULL and PRUNED), and does not segfault under most circumstances.
B	[70,100)	Consistently generates valid mazes AND correct solutions (at least one of FULL or PRUNED is correct), and does not segfault under most circumstances.
C	[40,70)	Consistently generates valid mazes OR correct solutions (at least one of FULL or PRUNED is correct), and may often segfault.
NC	[0,40)	Does not consistently generate valid mazes OR correct solutions.

12 Handing In

You should hand in **your entire Github repository**, which should include **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 made multiple branches, make sure you submit the correct one.

If this is your first time handing in, or you need a refresher, please refer to the [Github & Gradescope Guide](#). This will walk you through how to upload your work to Gradescope using your Git repository created in Installation.

If the autograder does not seem to reflect your local changes, be sure to have the correct branch selected, and push your latest changes to it. The autograder on Gradescope will give you your score within minutes. You are allowed to resubmit as many times as you like before the deadline to improve your score.