CS201: Data Structures and Algorithms

Assignment 1b

Version 1

Introduction

For your second programming assignment, you are to implement a maze creation algorithm and a maze solving algorithm using your stack and queue classes.

The executable

Your executable must be named:

```
amaze
```

This single executable will incorporate both algorithms. The executable will take command-line arguments and will create, solve, and/or display a maze. Here are some example invocations:

```
$ amaze -c 3 3 m.data        #create a maze, storing it in m.data
$ amaze -d m.data            #draw the maze in m.data
-------------
    |       |
-   -   -   -
|       |   |
---------   -
|
-------------
$ amaze -s m.data m.solved   #solve, storing solution in m.solved
$ amaze -d m.solved          #draw the solved maze
-------------
  0 | 3   4 |
-   -   -   -
| 1   2 | 5 |
---------   -
|         6
-------------
$ amaze -v
Written by Hailey Hackwell
$
```

where `$` is the system prompt. In the example, there are three rows and three columns for a total of nine cells; the following walls have been removed between cells:

the *bottom* wall of `(0,0)`
the *right* wall of `(0,1)`
the *bottom* wall of `(0,1)`
the *bottom* wall of `(0,2)`
the *right* wall of `(1,0)`
the *bottom* wall of `(1,2)`
the *right* wall of `(2,0)`
the *right* wall of `(2,1)`
the *right* wall of `(2,2)`

Note that the left wall of cell `(0,0)`, the entrance, and the right wall of the bottom right cell, the exit, are always missing.

The executable must handle the following options:

option	*action*
`-v`	give author's name;the program exits immediately after the name is printed
`-s III OOO`	solve the maze in file `III` placing solution in file `OOO`
`-c RRR CCC MMM`	create a maze with `RRR` rows and `CCC` columns, placing the maze in file `MMM`
`-r NNN`	seed a pseudo-random number generator with `NNN`; ifthe `-r` option is not given, use a random seed of 1
`-d III`	draw the created maze (`-c`) or draw the solved maze (`-s`)found in file `III`

Options may be combined. The commands:

```
amaze -c 3 3 m -s m p -d p
amaze -d p -c 3 3 m -s m p
amaze -s m p -d p -c 3 3 m
```

are all equivalent. In general, creation happens before solving, which happens before drawing.

Here is a program that features some handy-dandy option-handling code that you may use verbatim without credit: `options.c` Options may come in any order, but there will be at most one create, one solve, and one draw command.

Approach

You must create *MAZE* and *CELL* classes. If you do not, I won't help you figure out problems in your code. A *MAZE* object should hold the number of rows and columns plus a two dimensional array of *CELL*s, at a minimum. A *CELL* object should hold its location in the matrix (row, column) as well as information about its walls and what value the cell contains.

In building a maze, perform a depth-first traversal of a completely walled-in maze. Remove the wall between the current location and a randomly-chosen, unvisited neighbor. Then traverse further from the chosen neighbor (and so on) until you find no unvisited neighbors. At this point, go back to the previous spot and try again (and so on). If you use a stack for this task, 'going back' will happen automatically.

In solving a maze, use a breadth-first search off the maze starting in the upper left corner. Continue searching until the lower right corner is reached. Mark the squares searched with step number (mod 10). The upper left corner is always marked with a zero. The solving process stops when the lower right-hand corner is reached. Use a queue for this task. If you have any cells left on the queue when the exit is reached and they have step numbers, you should remove the step numbers from those cells.

Constraints

Use `srandom` and `random` to generate pseudo-random numbers.

When adding a reachable neighbor in searching for a solution, add the neighbor with the lower row number. In the case of a tie in the row number, break the tie with the lower column number.

When removing a wall in building a maze, use one random number (i.e. one call to `random`) to index into an array (or equivalent) of sorted, eligible candidates. For example, if *count* is the number of candidates, then `random() % count` would generate a random, but legal, index. Sort the candidates by lower row number (break ties with the lower column number). There can, at most, be four eligible candidates.

The lowest row number is the index of the top row of the maze. The lower column number is the index of the leftmost column of the maze.

System requirements

See the *previous* assignment.

Testing

I will be testing your maze implementation against my implementation. I will also be substituting my modules for your modules and vice versa. Therefore, it is important for you to adhere to the public interfaces found in any supplied `.h` files.

Project compilation

You must provide a file named *makefile*, which responds properly to the commands:

```
make
make test
```

```
make clean
make valgrind
```

The `make` command compiles your program, which should compile cleanly with no warnings or errors at the highest level of error checking (the `-Wall and -Wextra options`). The `make test` command should test your program, the `make clean` command should remove object files, and the `make valgrind` command should test an executable with *valgrind*. Valgrind should report no memory errors or leaks.

The correctness and efficiency of your makefile will be tested. You must have the correct dependencies and your makefile should not perform any unnecessary compilations.

You must implement your modules in C99 and you must use the `-std=C99` option to *gcc* in your makefile.

Documentation

All code you hand in must be attributed to its author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Submission

You will hand in (electronically) your code for the preliminary assessment and for final testing with the commands:

```
submit cs201 lusth compile
submit cs201 lusth test1
submit cs201 lusth assign1
```

Change log

`1b`	clarified the termination condition to prevent an extra step
`1a`	changed termination condition for solving to lower right corner