CME 211: Homework 5

Due: Friday, November 8, 2019 at 4:30pm (Pacific)

Background

Maze generation and solution algorithms are an interesting application of graph theory. They can also be practical if you are stuck in a Halloween corn maze! A simple solution is the right hand wall follower algorithm as shown in Figure 1. You start with your hand on the right wall and follow along the wall until you exit the maze.

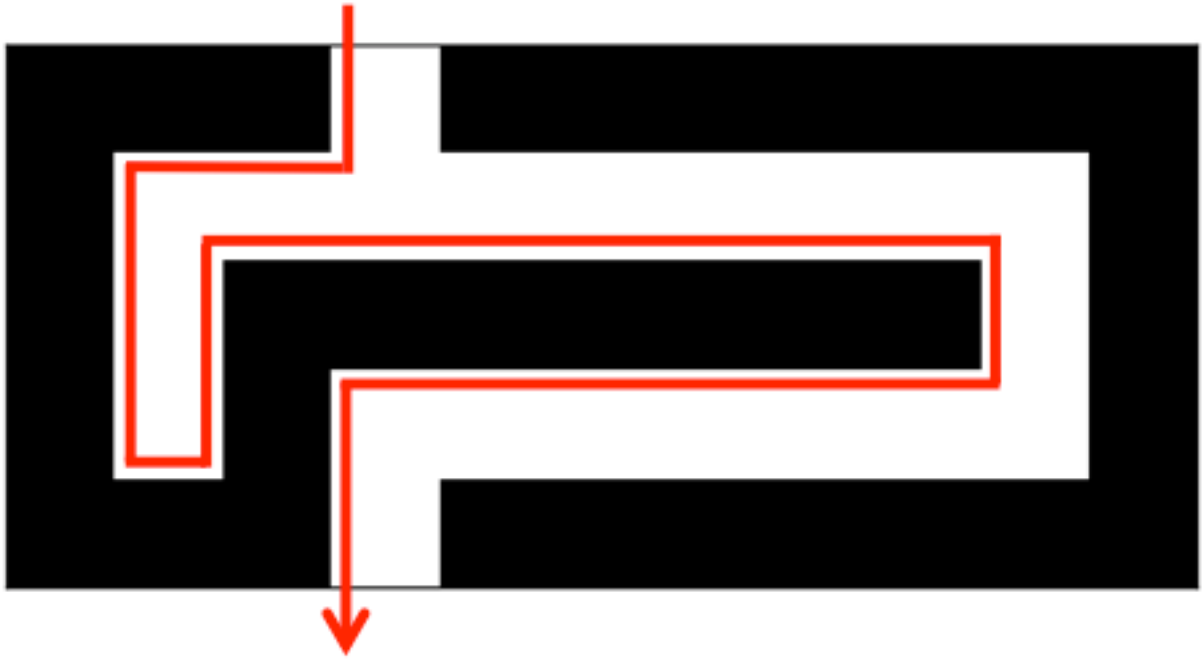

Figure 1: Right hand wall follower solution for `maze1.txt`

Preparation

- **This assignment requires that you have completed CME211 HW0.**
- Make sure you have an up-to-date local clone of your CME211 homework repository.
- Create directory named `hw5` at the top level of your CME211 homework repository.
- All work for this homework goes inside of the `hw5` directory.

Assignment (60 pts functionality, 30 pts design, 10 pts writeup)

Write a C++ program that reads a maze, computes a solution using the right hand wall following algorithm, and stores the solution in a file. If you are having difficulty developing your approach for the algorithm, think about a state machine approach. Given your current location and direction, and your current surroundings, change direction and/or move to a new location.

The maze will be stored in your program as a 2D array as illustrated in Figure 2. In each maze file the first line contains the number of rows and columns respectively, and each subsequent line contains the row and

column index of a wall location. So to read the maze you should confirm there is sufficient storage space in the array, initialize all of the values in the array to a constant value, and then for each line with a row and column pair you should change the value at that location in the array to indicate the presence of a wall. The mazes will always have one entrance on the top row and you will know you have exited the maze when you reach the last row.

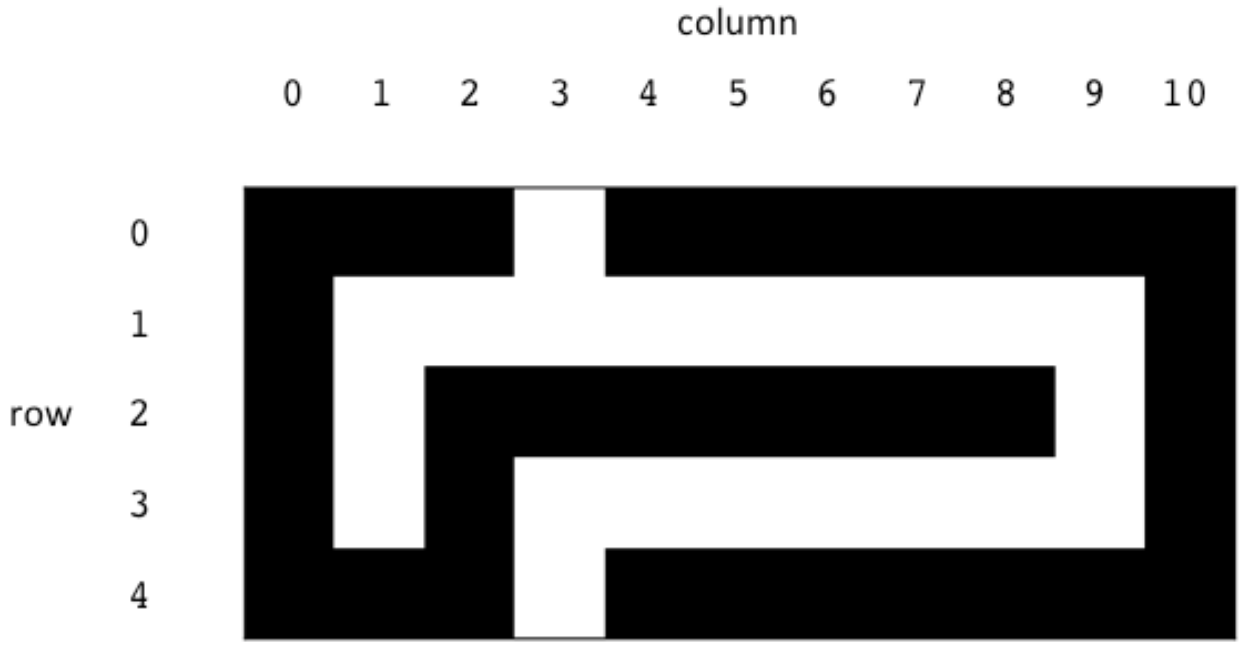

Figure 2: Representation of `maze1.txt` in terms of rows and columns of a 2D array.

The program should take two command line arguments to get the name of the maze file and the name of the solution file:

```
$ ./mazesolver
Usage:
 ./mazesolver <maze file> <solution file>
$ ./mazesolver maze1.txt solution1.txt
```

For a solution file you should write the row and column index of your positions as you progress through the maze. So for example, the solution file `solution1.txt` for `maze1.txt` should look like this:

```
0 3
1 3
1 2
1 1
2 1
3 1
2 1
1 1
1 2
...
4 3
```

Put your program in a file called `mazesolver.cpp`.

In addition to your solver in C++, write a Python program to test your solution. The program should take

as command line arguments the name of a maze file and the name of a maze solution file. The program should read the files and confirm that you enter the maze via the opening in the top row, that you do not cross any walls during your trip through the maze, and that you do in fact reach the last row. Provide output to the screen that the solution is either valid or invalid:

```
$ python checksoln.py maze1.txt solution1.txt
Solution is valid!
```

Put your code in a file called `checksoln.py`.

Summary of requirements

1. Write a C++ program in a file called `mazesolver.cpp` that implements the right hand wall following algorithm. Your program should:
   - Confirm that appropriate command line arguments were provided and if not provide a usage message and exit.
   - Verify that appropriate static array storage is available for storing the maze. You can setup your static array for the largest maze file provided, but should still have your program confirm at runtime that the array is large enough. We emphasize that you should store the maze in a **static array**!
   - Find the maze entrance at the opening in the first row and store this as your first position in the solution file.
   - Use the right hand wall following algorithm to move through the maze without going through any walls, storing each position in the solution file. We expect that a valid path should not involve duplicated adjacent entries, i.e. if we treat each valid position in a path as a unit time-step, we shouldn't force the user of our directions to "wait" or "remain" at a position without moving.
   - Exit on the last row and store this as your last position in the solution file.
   - Your code must compile without warnings under the compile command on `rice`:

   ```
   $ g++ -std=c++11 -Wall -Wconversion -Wextra -Wpedantic mazesolver.cpp -o mazesolver
   ```
2. Write a Python program in a file called `checksoln.py` to verify that your maze solution is valid. Your program should:
   - Determine if appropriate command line arguments were provided and if not provide a usage message and exit.
   - Store the maze in an appropriate NumPy array.
   - Read the solution and make sure the maze was properly entered on the first row, each position change is valid (i.e. you move one position at a time, don't go through a wall, and stay within the bounds of the maze), and that you reach the exit of the maze on the last row.
   - Print feedback about whether the solution is valid or invalid.
3. Write a README file documenting your work. Your README file should include:
   - Brief statement of the problem.
   - Description of your C++ code.
   - Brief summary of your code verification with 'checksoln.py'.

(Pedantic) Compiler Flags

Note that the GNU compiler `g++` *allows* variable length arrays as a nonstandard compiler extension. However, we ask that you *not* use this feature for this assignment. If one adds the option `-Wpedantic` to the compilation command, it will trigger warnings if any nonstandard compiler extensions are used; we will use this when grading your assignment. Please ensure you understand what it means to declare a static array (i.e. length is *known at compile time*); if you read in the dimensions of the maze and then use these input values to determine how much storage to allocate in an array, you are *not* using static arrays!

Includes and Modules

In your C++ program, you may use the functionality from the following set of files: `fstream`, `iostream`, and `string`. You should not need any additional `includes`. In your Python program, you may import any module that we have covered in class to date, including for example `numpy`; you should think carefully about what functionality may be useful in your solution.

Performance and Testing

We will verify your solution using the provided input files. Your C++ program should run in under a second, and the Python program shouldn't take more than a few seconds.

Homework submission

Please be very careful with directory and filenames, as points will be deducted if you do not follow the directions. To be clear you should have a minimum of these files in the `hw5` directory in your GitHub repository:

- `checksoln.py`
- `mazesolver.cpp`
- `README`
- `solution1.txt`
- `solution2.txt`
- `solution3.txt`

Do not commit temporary files produced by your text editor. We will deduct points if these instructions are not followed. When working in teams and professional environments, it is important to keep your repositories free of extraneous files.

Whatever files you have in your GitHub repository at the deadline will be considered your final submission.