CSC 301 Project 2: DFS/BFS Fall 2025

Due: Friday, October 17th

In this project, you will implement both DFS and BFS to solve various mazes. You will be required to implement your solutions in their respective locations in the provided project 2 files. You must submit your `project2.hpp` and `project2.cpp` code on Gradescope.

Contents

1 Problem Statement

You will have two primary tasks in this project: implement functions `DFS` and `BFS` that can solve an input maze. Your code is tested in the main function of the `maze.cpp` file.

You must submit your `project2.hpp` and `project2.cpp` code online on Gradescope. If you worked with a partner, submit a single version of the code that is associated with both partners on Gradescope.

1.1 `C++` stack and queue Classes

`C++` has built in `stack` and `queue` classes as part of the standard library. I strongly recommend using the `C` Library Reference (https://cplusplus.com/reference/clibrary) as a good resource for looking up documentation **and examples** of most `C++` functions/classes.

In particular, you will want to look at the behavior of the following methods:

- `std::reverse`
- `stack`
  - `push`
  - `top`
  - `pop`
- `queue`
  - `push`
  - `front`
  - `pop`

1.2 DFS/BFS Implementation

The bulk of your submission will concern the functions `DFS` and `BFS` in `project2.cpp`. These functions should take as input an adjacency list represented as a `vector<Vertex>` containing the `Vertex` objects making up the full maze.

The output of these functions should be a path represented as a `vector<int>` containing the `labels` of the vertices in the path from start to exit in order. Note, the path will need to start at the starting vertex of the maze and end at the exit vertex to be valid.

The main function in `maze.cpp` will test your functions on all of the five provided mazes. You can compile with the `make` command, and run the code with the command:

```
./maze
```

This will simply print how many tests you passed. If you want more detailed info about the failed tests, as well as ascii drawings of the mazes and your paths through them, you can turn on verbosity with the -v command **(you should check this output before submission!)**, i.e.,

```
./maze -v
```

2 Provided Code

Your goal in this project will be to implement both DFS and BFS to solve several given mazes. To aid you in this, you have been provided with a fully functioning `Vertex` class along with functions for creating, printing, and testing the mazes.

2.1 Vertex Class

The fully functioning `Vertex` class is in `Vertex.hpp` and `Vertex.cpp`. This class has 4 class attributes:

- `int label`: the label of the given vertex **(must always be the index of this Vertex in the adjacency list)**
- `vector<int> neighbors`: the vector of the labels of the neighboring vertices
- `bool visited`: a flag for marking a vertex as visited
- `int previous`: the label of the previous vertex in the path

Along with these is an overloaded `operator ==` that will compare two `Vertex` objects based on their `labels` (which should be unique within a graph), and a `printVertex` function that can be used to more cleanly print info about a vertex (it returns a string that can be sent to `cout`).

2.2 `maze.cpp`

You have also been provided with a number of functions for creating, printing, and testing the mazes in the `maze.hpp` and `maze.cpp` files:

- `createMaze`: this function takes in 0, 1, 2, 3, 4, or 5 and outputs the vector of 0s (open rooms) and 1s (walls) representing the maze. This is used internally by the code in `maze.cpp`. The mazes are:

    0. A small open room for debugging. Note that BFS should find a straight path across this room, while DFS probably wont.
    1. A small maze for debugging.
    2. A large maze.
    3. A large zig-zag map.
    4. A specially designed test that will guarentee different behaviors for DFS and BFS. For this maze, DFS cannot find the shortest path, while BFS must find the shortest path.
    5. A randomly generated maze (uses Wilson's algorithm implemented in `wilson.hpp` and `wilson.cpp`). *Note: this functionality has not been added yet. It may get added this semester in a later update, or it may not.*

- `findStartExit`: this function is used internally by the code in `maze.cpp` to find the label of the start and exit vertex of any maze created by the rules of the `createMaze` function (i.e., start is in first row and exit is in last).

- `printRoomNums`: this function will generate a string that cleanly prints the `label`s of every open room in a maze. **This function may be useful for debugging.**

- `makeAdjList`: this function takes in a maze created by `createMaze` and returns the corresponding `vector<Vertex>` adjacency list.

- `checkMaze`: this function takes in a maze and a path and checks to see if the path is valid.

- `printMaze`: this function takes in a maze and a path and creates a string that prints the maze in ascii: `X` are walls, `s` is the start, `e` is the exit, `o` is the path, and spaces are open rooms. Any other characters correspond to invalid path issues: `R` are repeated vertices on the path, `G` are ghosts where the path go through walls, and `!` indicates when the start or exit is not on the path. **This function may be useful for debugging.**

- `testMaze`: this function will test both DFS and BFS on a specified maze and return whether the tests were successful (as a `pair<bool,bool>`). If `verbosity` is set to `true`, it will print useful info when path issues are encountered.

- `main`: the main function will run all of the tests for all of the mazes. You should feel free to edit your local copy of this function to test your code as you write it.

3 Pair Programming

You are required to work in assigned pairs for this project.

- Your group should submit only one version of your final code. Have one partner upload the code and report through Gradescope and associate both partners with the submission. Make sure that both partner's names are clearly indicated at the top of all submitted files.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating'). (You may work together remotely by sharing screens.)
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

4 Readable Code

You are expected to produce high quality code for this project, so the code you turn in must be well commented and readable. A reasonable user ought to be able to read through your provided code and be able to understand what it is you have done, and how your functions work.

The guidelines for style in this project:

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code.

5 Submission

You **must** submit your `project2.cpp` and `project2.hpp` code online on Gradescope. Your group should only submit one version of your completed project, associate both partners with the submission, and indicate clearly the names of both partners at the top of all submitted files. Attribute help in the header of your `.cpp`.