Project 4: Linked-Based Maze Traversal

Dr. Oyewole Oyekoya

Due May 8 @ 5pm

Objective:

Your objective for this project is to find the shortest path from start-point to end-point of a general link-based maze using the following algorithms: Depth-First Search, Breadth-First Search, and Dead-End Filling.

Preliminary Work:

Familiarize yourself with the provided code base. The Maze and MazeNode classes have already been defined and implemented for you in order for you to focus on algorithm design. The suggested sequence in which you should read the files is first MazeNode.hpp, then MazeNode.cpp, followed by Maze.hpp, and finally Maze.cpp. If within the code base you discover semantic or syntactical elements that are novel to you, it is **strongly** suggested that you seek to understand them before proceeding to work on the meat of this project. Try your best to consult as many quality resources as you can find. Here are some:

- http://www.cplusplus.com/reference/stl/
- https://www.programiz.com/cpp-programming/operator-overloading
- http://www.cplusplus.com/doc/oldtutorial/namespaces/

Also, brush up on how Depth-First Search and Breadth-First Search work over graphs, specifically grids. Here are some useful applied resources:

- https://www.youtube.com/watch?time_continue=487&v=W9F8fDQj7Ok&feature=emb_title
- https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

Remember that you are free to reach out to a TA if you have any explicit questions.

Task 1:

Modify the function `std::vector<MazeNode> solveDFS(Maze & a_maze)` in Solutions.cpp to implement Depth-First Search in order to find the shortest solution to a maze. The method should return an ordered vector that holds the path nodes from start-point to end-point.

Hint: use the backtracking recursion technique discussed in class.

Task 2:

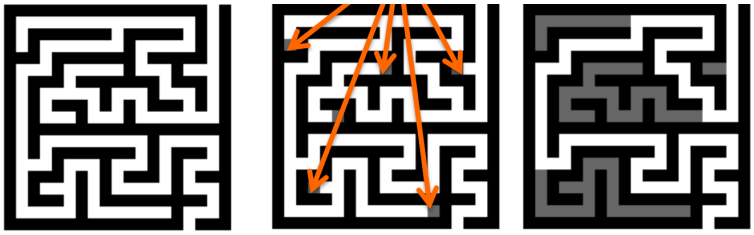
Modify the function `std::vector<MazeNode> solveBFS(Maze & a_maze)` in Solutions.cpp to implement Breadth-First Search in order to find the shortest solution to a maze. The method should return an ordered vector that holds the path nodes from start-point to end-point.

Hint: you can use a queue as a temporary container to hold nodes that you will process.

Task 3:

Modify the function `std::vector<MazeNode> solveDEF(Maze & a_maze)` in Solutions.cpp to implement Dead-End Filling in order to find the shortest solution to a maze. The method should return an ordered vector that holds the path nodes from start-point to end-point.

Hint: use a stack to hold "dead-end" nodes and update the stack with each newly acquired "dead-end" while marking non-traversable or undesirable nodes as you go. Thus, the only remaining path will be the shortest path.

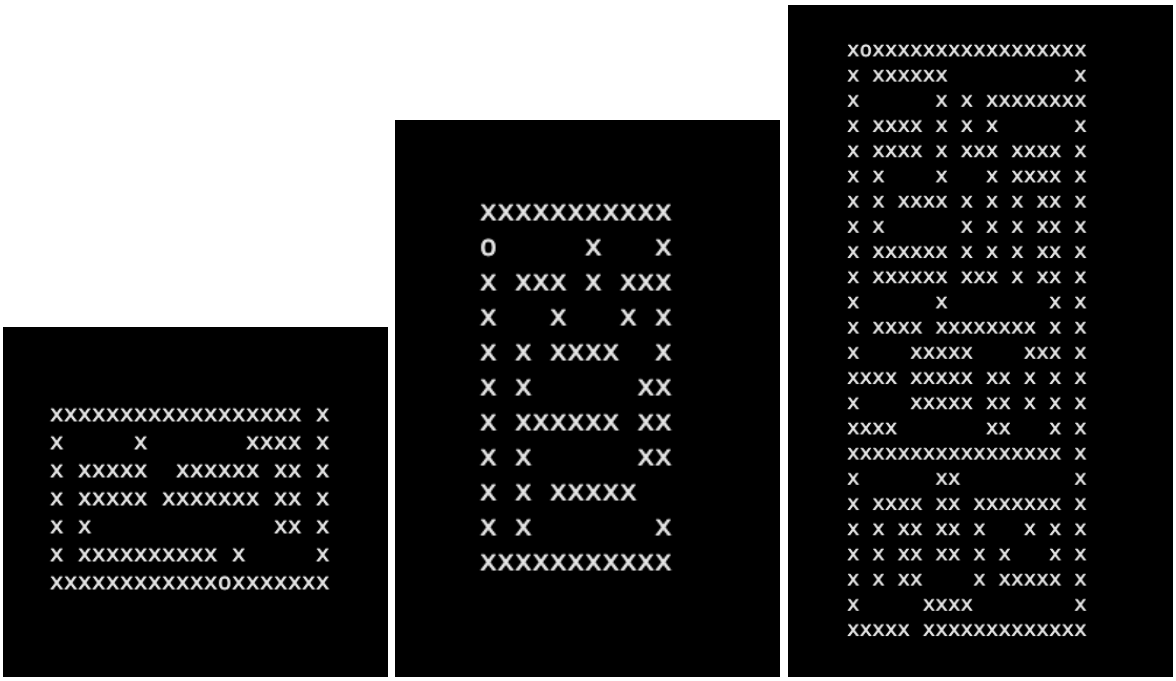


Task 4:

Modify the function `std::vector<MazeNode> solveCustom(Maze & a_maze)` in Solutions.cpp to implement your own shortest path finding algorithm. This could be an optimization of one of the algorithms you have already implemented, an algorithm that already exists, your own creation, or a combination of any or every one element from these categories. You will receive full credit by creating an algorithm that works. **However, the top 10 submissions with the lowest run-time and space overhead will receive 5 points extra credit on their assignment grade. The best submission will receive 10 points extra credit**.

Testing

Use the provided .csv files to test your functions. Here is a visual representation of each .csv file maze, where 'x's represent walls, where 'o' represents the start point, where spaces represent accessible nodes, and where the one hole in each maze perimeter represents the respective exit:



You must always implement and test you programs **INCREMENTALLY!!!**.

What does this mean?

- Implement and test one method at a time.
- For each class:
    1. Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable)
    2. Implement the next function/method and test in the same fashion.

How do you do this?

Write your own **main()** function to test your classes. In this course you will never submit your test program, but you must always write one to test your classes. Choose the order in which you implement your methods so that you can test incrementally (i.e. implement mutator functions before accessor functions). Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use **stubs**: a dummy implementation that always returns a single value for testing (dont forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Grading Rubric:

- **Correctness 80%** (distributed across unit testing of your submission)
    - A submission that implements all required classes and/or functions but does not compile will receive 40 points total (including documentation and design).
- **Documentation 10%**
- **Style and Design 10%** (proper naming, modularity, and organization)

Submission:

You will submit **the following file:**

- Solutions.cpp

Your project must be submitted on Gradescope.

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You MUST test and debug your program locally. Before submitting to Gradescope you MUST ensure that your program compiles (with g++) and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the Programming Rules document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you dont have through Gradescope. But it ran on my machine! is not a valid argument for a submission that does not compile. Once you have done all the above you submit it to Gradescope.