Design Plan - Maze

James Viner

Project Summary

Maze is a program that reads a maze in from a file with a specific format and outputs the map with the shortest path between two specified points represented by periods.

Features Targeted

Doors

Doors are going to be implemented as two separate tiles, open doors and closed doors, which have the same weight as a single "turn" or the weight of two "turns" respectively. I don't foresee any additional complexity involving the transition from closed to open door given that used tiles will be marked with a period instead of their respective door state.

Wall-digging

Walls by default are implemented as a space that exists in the ASCII-representation of the maze, but don't actually need to be put into the graph given that they cannot be traversed. This is different from boundary nodes that I will be using to surround the entire graph, that will connect to all traversable nodes and be used for the purpose of maze validation. The act of allowing wall-digging will involve first generating the map by default with walls excluded from the graph and running the validate_maze function to first tell if it's a legal maze in the first place. Assuming that passes, a second iteration through the textual representation of the maze will be done and the walls will be added in as nodes with valid edges to each other node in the graph, albeit with a priority eleven times as low as an empty space.

Water

Water is going to be implemented in the same way as a space block, but with the same weight as 3x lower priority.

Architecture

Data - Graph

Individual nodes to represent tiles in the maze will be stored inside of a graph so as to allow arbitrary edges to be placed to surrounding nodes, representing the valid path and the respective weight from one tile to another. Given that each tile can be traversed into and out of in a maximum of four directions, a fully-surrounded tile will have eight total edges with inbound and outbound weights respective relationship between it and its neighbors. Part of the data payload of a given node will be its index into a character array holding the original ASCII-representation of the maze so that the valid path to the exit can be iterated over by indexing into that array and changing nodes on the path to periods.

Data - Validation

For the purposes of validation, a rectangular box large enough to encompass the entire maze will be created to surround it with 'boundary' tiles, with the program filling in the holes left in the space between the original maze and the edge of the boundary with empty space tiles. This will mean that for all mazes, if there is a path to the top left node (a boundary tile), the maze is invalid.

Significant Functions

```
bool validate_maze(graph *g)
```

Before the actual pathfinding is done between the start and end tiles, first a path will attempt to be created from the start tile to the origin of the graph (the node at at the top left, which will always be a boundary). Given that boundaries by default have a very high priority, if there is an invalid maze as a result of an opening not enclosed by walls, the path to the outer limits of the maze will eventually be found and walked up to the top left of the grid. In the case that walls completely encompass the maze area, a valid path to the origin will be impossible. This function returns true if the maze is valid and false otherwise.

```
graph *s load_maze(FILE *fo, char *maze, bool doors, bool
passable_walls, bool water)
```

This function is responsible for doing the heavy lifting of first figuring out what the maximum height and width of the maze is so that it can be wrapped in a boundary, storing the textual representation of the maze (with boundaries added) into the variable char *maze, and then turning all non-wall tiles into nodes in a graph with their respective edges to adjacent tiles. Then, validate_maze will be called to ensure a path from the start to the finish can be made. In the case that wall-destruction is allowed, walls will have to be added as nodes after this process before pathfinding can be done.

```
void print_maze(graph *g)
```

Once a valid path is found using Liam's implementation of Dijkstra's algorithm, the original symbols making up the spaces between the start and end of the shortest valid path will have been replaced by periods by the load_maze function. All that is left to do is print the array back to the console.

User Interface

The user will interface with the program by passing a specified maze file on the command line and receive either the simplest path drawn with periods or just the original maze if a valid path cannot be made from start to finish.

General Approach

1. Reading maze file into an array and adding boundary
2. Printing the new maze array to the console
3. Loading non-wall nodes into graph (includes water and door functionality)
4. Dijkstra's algorithm to find shortest path
5. Replacement of non start/finish tiles on path with "."
6. Printing solved maze to console
7. Going back and implementing walls as passable nodes