Maze Escape Engineering Test

Bailey Brown

Assumptions Made For Program

I designed my program working on the following assumptions:

- Maze input is always given in .txt format
- Maze input is stored in the same folder location as my program
- Only one start point will be given
- Only one exit point will be given
- Maze is described only with the given characters in the brief ‘ A ’,’ B ’,’ x ’, ‘ . ’

Algorithm Explanation

When designing my program I broke the problem down into 3 smaller tasks:

- Reading and formatting file input
- Calculating maze solution
- Formatting and outputting Results

Reading and formatting file input

For reading the textfile I ask for the filename and then read the data in using <fstream>, to format the maze into readable data I created Grid, Node and Adjacency classes.

Each character in the maze file is defined by a node representing a tile / space in the maze, this is then stored in a grid object containing a 2 dimensional vector to represent the maze as a whole. Each node contains x and y cords that can be used to get its position in the 2d vector as well as a ‘walkable’ bool used to signify a ‘x’ tile that cant be moved through during route calculation. Cords for the nodes parent’s position and G,H & F costs are also defined for later use in the calculating maze path section.

Calculating maze solution

Once the file has been formatted into a 2d array of nodes I use an A* pathfinding algorithm to find the fastest route, I decided to use A* as it’s a efficient standard in pathfinding and its flexibility would allow for expanding of the maze system eg “m” could represent mud and a movement penalty could be simulated by a higher cost for traveling across those nodes.

My A* algorithm selects a node from the open list with the lowest f cost and adds it to the closed list to signify it’s been assessed, the algorithm then checks if this node is the exit node, if not it evaluates its adjacent nodes to see if they can be moved to and haven’t already been assessed, if so the f cost of the node is calculated and its added to the open list with its parent updated or assigned as the currently selected node.

Given the maze start node as an input and enough loops the algorithm will consider nodes further and further out from the start until a path is found, assuming the maze has a possible path.

Formatting and outputting Results

Once the path to the exit has been calculated I then identify the path taken in the `retracePath() function` by backtracking from the end exit node, I do this by inputting each nodes parent cords into the node grid. As I backtrack the difference in cords between a node and its parent are used to identify in which direction the move was made (NESW) I add this to a string and by the end of the function I have the whole path stored in the string. Once the path is generated I cout the path and use <fstream> again to write the solution to a file is asked by the user.

Testing Procedure

Once the program was complete the first tests I made were to check that my program gave the correct outputs for the quickest_route files provided, Once these were confirmed to work I then created my own series of maze files to test a few more basic mazes and then any edge cases I could think of that could cause unexpected results. The text mazes I made can be found alongside the exe and source files for reference.

After I was happy that the program could correctly handle working mazes I then made a series of mazes designed to cause one specific type of error each so I could check if my program could handle them gracefully. To do this I created my own returnError() function that would output a detected error with the maze and exit the program safely before a runtime error would occur, I then tested each of my deliberately broken mazes to check that in all situations I detected and handled the error via my returnError function.

Tests included a maze with no start point, maze with no exit point, maze with multiple start points, maze with multiple exit points and a maze with no possible path. These mazes can also be found alongside the exe and source files for reference.

```
Please enter the maze file name (eg maze1) - error test 1 no start point


 ERROR - 0 start positions given, please check maze layout and ensure one exit position is given
```

```
Please enter the maze file name (eg maze1) - error test 2 no exit point


 ERROR - 0 exit positions given, please check maze layout and ensure one exit position is given
```

```
Please enter the maze file name (eg maze1) - error test 3 multiple start points


 ERROR - multiple start positions given, please check maze layout and ensure only one start position is given
```

```
Please enter the maze file name (eg maze1) - error test 4 multiple exit points


 ERROR - multiple exit positions given, please check maze layout and ensure only one exit position is given
```

```
Please enter the maze file name (eg maze1) - error test 5 no path to exit

ERROR - No path found, please check maze layout and ensure a path from A to B is possible
```

How To Build / Run

To run my program place any maze files you wish to test in the same folder as the Maze Escape.exe and then run it. When prompted enter the name of the maze file you wish to test (caps sensitive and don't include the .txt). Once given the filename the program should return the path as a cout statement within the cmd, the program will also ask if you wish to save the solution to a file, if you enter yes a .txt file will be created in the same folder as the .exe called "xxxxx solution.txt" where xxxxx represents the original maze filename.

Program Outputs

Below are screenshots of my program outputs for the provided mazes