De La Salle University - Manila
Term 2, Academic Year 2023 - 2024

In Partial Fulfillment
Of The Course Requirements
In **CSINTSY - S15**

MCO1: MazeBot

Submitted By:
Chong, Hans Kirzen
Salen, Rommel Kendric
Tuco, Kevin Bryan
Uy, Wesley King

Submitted To:
Mr. Thomas James Tiam-Lee

Submitted On:
March 6, 2023

I. INTRODUCTION

The project revolves around creating a bot that, upon reading a specific maze definition from an input text file, determines whether or not there is a viable path from starting state to goal state. The maze is a size of *n* x *n*, wherein n is at minimum 3, and at most 64, and is saved as a text file. The bot then reads through the maze, – reading '.' as empty spaces, '#' as walls, 'S' as starting state, and 'G' as the goal state – determines the coordinates of the starting state and the goal state, and then begins moving.

At most, the bot has a total of four moves: up (^), down (V), left (<), and right (>). It decides its next moves based on A.) Which moves it currently has available to it/moves that lead to an unoccupied space, B.) The A* algorithm and the heuristic of the move (the lower the heuristic, the higher its priority), and C.) Whether or not it has already visited that space before.

The heuristic of each move is determined by the distance between its current space and the goal state, calculated via the manhattan distance formula ($|x_1 - x_2| + |y_1 - y_2|$). This makes it so that the bot will, in most scenarios, move closer towards the goal state. Using this algorithm, however, the bot will likely make numerous backtracks when it finds moves that lower the heuristic but leads to dead ends, which is when the third condition triggers and prevents the bot from engaging in an endless loop.

The bot stops when it is either A.) Finds a functioning path from the starting state to the goal state, or B.) All possible moves have been made/all nearby nodes have been explored, and thus determines that there is no path available.

Afterwards, if the path was found, the algorithm will print onto a text file wherein it contains:

- The maze
- Number of states explored
- Maze with arrows showing the path from start to goal
- Order of states explored from path to goal
    - Order is in terms of when it was explored in the program; so it might not follow numerical ordering (i.e. 1, 2, 3, 4, etc.)

II. PROGRAM

1. Open the .jar file within the terminal or run the source file in a java IDE.
2. Make sure that the **maze.txt** must be in the same directory as .jar file or source file.
3. Type the <name>.txt in the terminal window. (if its not finding the file, add the folder in the filename)
4. New file is generated in the source folder if a path exists, else it doesn't.

The .java file can also be run on IDEs like VSCode; in that case just install the java extension and you will find a clickable run button above the main function.

III. ALGORITHM

```
Read first line in file to set the size
While file next line is not empty do
        save char into a 2d array
Start = find the coordinates of 'S'
Goal = find the coordinates of 'G'
Start Node = set x and y coordinates
Goal Node = set x and y coordinates
Add Start Node to Frontier

While Frontier is not empty do
        Current Node = Remove the smallest total cost from Frontier

        If Start Node is equal End Node:
                While current Node parent is not null do
                        Store current node parent into path
                Reverse path
                Save maze, path, and states explored in a txt file
        For all possible moves within the state:
                If move is impossible do
                        continue
                Next Node = set possible x and y
                Set Next Node.cost = Current node.cost + 1
                Set Next Node.heuristic = ManhattanFormula(possible x, possible y, Goal
                Node.x, Goal Node.y)
                Set Next Node.total cost or h* = cost + heuristic
                If the Next Node is already visited or it is in the Frontier with lower cost
                        Continue, back to the top of the while loop
                Add the Next Node in the Frontier
                Add the Next Node in Visited

        If there is no found path
                return null
```

IV. RESULTS AND ANALYSIS

• Discuss what situations the bot can handle. Explain why the algorithm is able to handle these cases.

The bot uses an A* search algorithm with its heuristic being the manhattan distance. In all of our test mazes, it seems to run perfectly well, however the bigger maze slows it down slightly, but still returns a path. It was able to handle cases where the goal cannot be reached due to it being blocked by walls.

The algorithm uses a priority queue to keep track of the nodes to explore next. Each node has a heuristic and the priority queue makes it so that the nodes with the lowest heuristic are explored first. The algorithm will keep exploring with that strategy until either the goal is reached or all reachable nodes have been explored. If all reachable nodes have been explored, that means that the goal isn't reachable from the start node; which handles cases where the path to the goal is being blocked or surrounded by walls.

The bot can handle scenarios with large mazes without walls since it will move towards the goal due to the manhattan distance. It can also handle mazes with a lot of obstacles due to the mechanics stated above. The algorithm can also handle looping paths; the algorithm has a check if the state or node being explored has been explored before, if it is and its heuristic is larger than when said node was first explored, it will not explore that node further. This stops the bot from infinitely checking the same states again and again.

• Discuss what situations the bot cannot handle, or what situation the bot performs poorly. Explain why these situations cannot be handled by the bot. Point out which part of the AI made it fail.

The bot performs worse with situations where there are multiple paths of equal length between the start and goal. When the bot encounters those cases, it will explore all of the said paths then pick one random path (since they all have the same length). Here's an example:

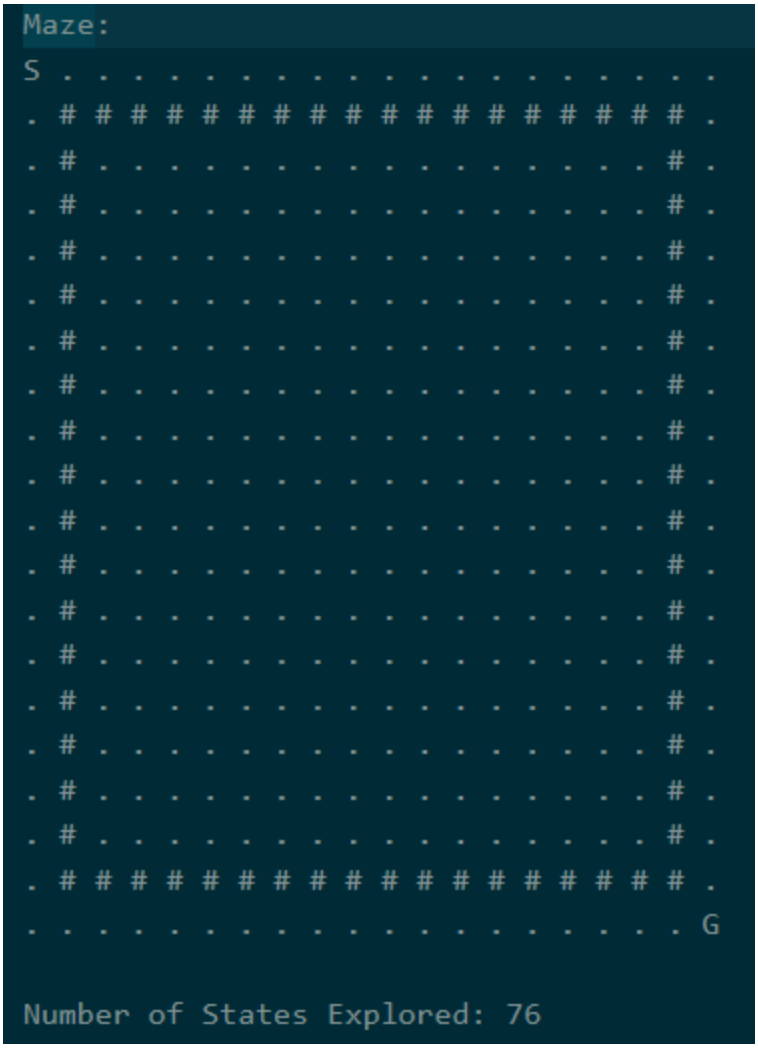

Figure 1. 2 equal path

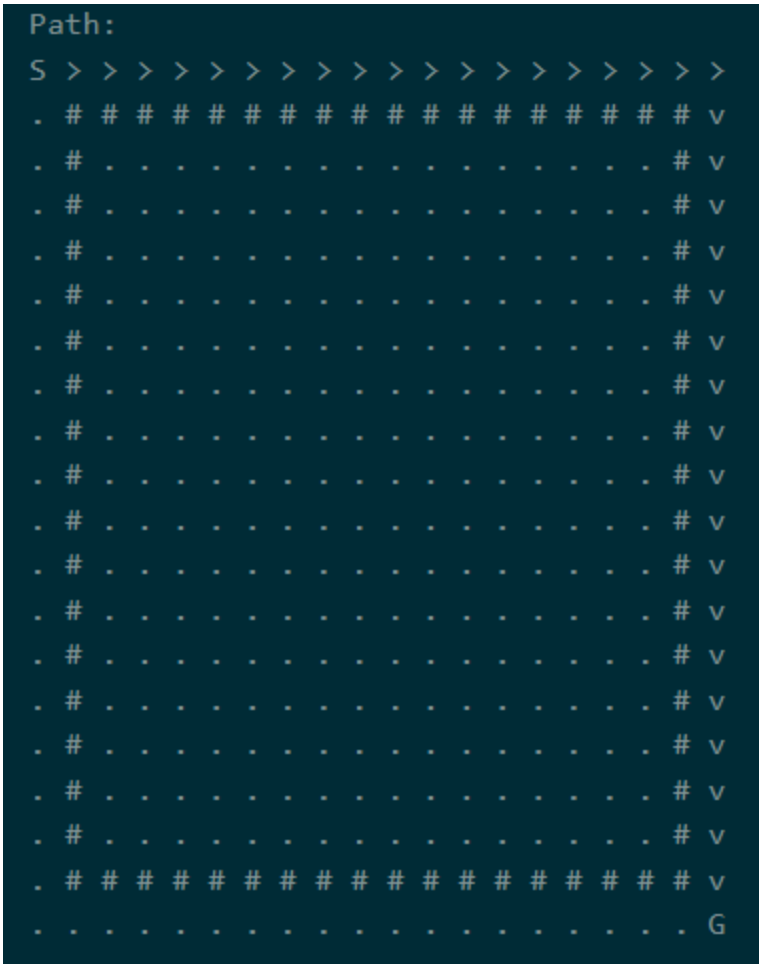

Figure 1.2. Solved path

In this test maze, the bot explored 76 states, which is the amount of reachable paths in the maze (start and goal included). After exploring both, it chose a random path to use; in this case, it chose the path going to the right.

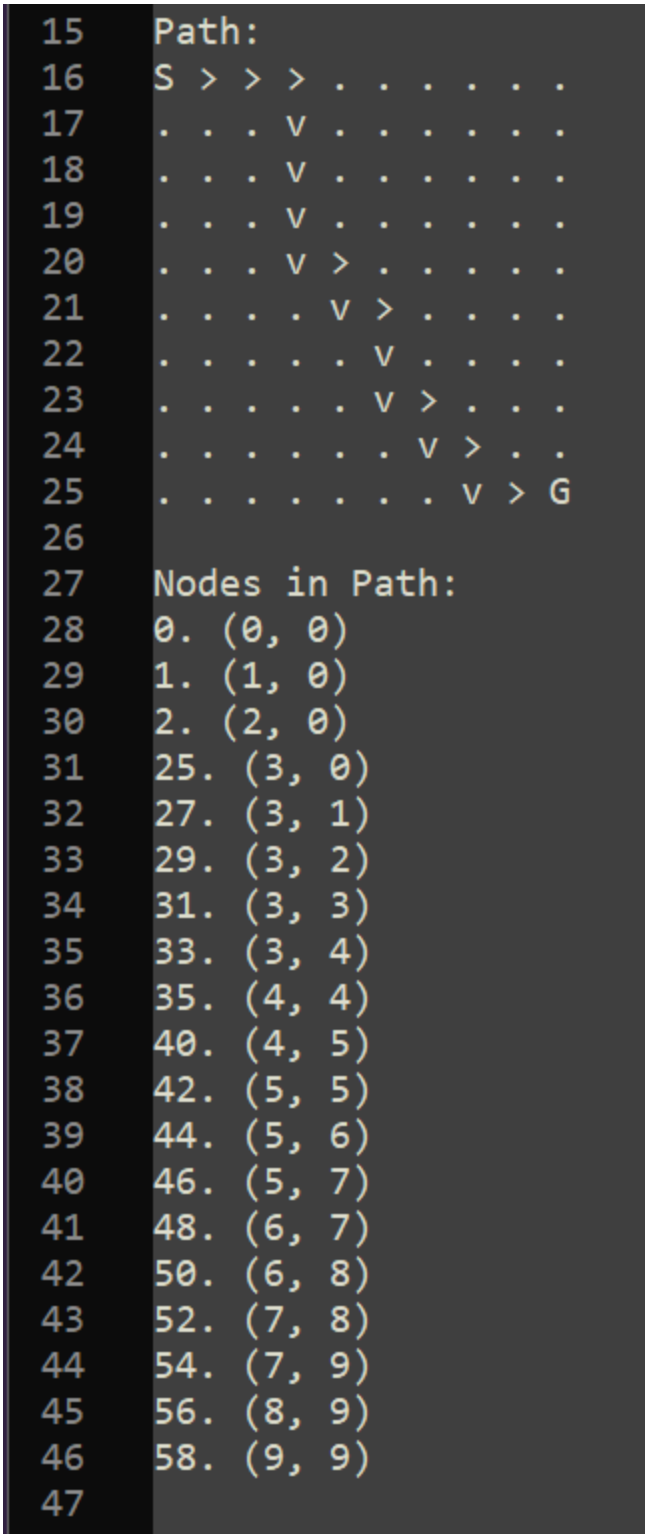

Figure 2. File Output

Same with this case where the bot explored 60 states on an empty maze before concluding with the path on the picture above.

It's not a fail, since it still returns a path, but it can cause problems in terms of speed of the bot, especially with larger mazes. Specifically, this problem is caused with how we pick the next node to explore; which is through a priority queue. When the value being compared is the same, the priority queue will pick one at random, this can cause the bot to explore paths all at once; exploring a node in one path then a node in another, etc.

V. RECOMMENDATIONS

Based on the analysis of the performance of the bot, point out the weaknesses of the bot. Identify and explain possible ways to address these weaknesses.

The weakness of this bot is that it needs to check all the possible states or nodes within the current state, making the bot run slower. if the size of the maze is too big, it could slow down the program in the long run. Another weakness is that it can perform worse when there are multiple paths of equal length between the start and goal. If this happens, the bot will always return one optimal path, but still slows the program down due to the unnecessarily explored states.

A possible solution is to check for straight lines of paths and prefer straight lines if it is going towards the goal; then check the end of said straight lines first. This can fix the problem where the program will choose to explore multiple paths with the same cost. It can also make the program run faster as it has to explore less states if it encounters straight lines, more so if the straight line goes straight to the goal node.

For the slowness coming from larger mazes, there really isn't any catch-all solution for that problem. All programs will face a bit of trouble when handling really large inputs. Instead, we can optimize the code that we already have and cut off code that is unnecessary or replace parts of the program with code that works faster.

VI. CONTRIBUTIONS OF EACH MEMBER

Chong, Hans Kirzen

- Implemented and programmed the A* Search algorithm
- Made test cases
- Helped in documentation

Salen, Rommel Kendric

- Made test cases
- Helped in documentation
- Helped in the main function

Tuco, Kevin Bryan

- Implemented the initial source file
- Implemented and programmed the A* Search

- Helped in the main function

Uy, Wesley King

- Debugging the initial source file
- Programming the scanning of Maze
- Did the Pseudo Code for documentation