Presented to the College of Computer Studies

De La Salle University - Manila

Term 2, A.Y. 2022-2023

In partial fulfilment of the course

In CSINTSY S14

Major Course Output 1: MazeBot

Submitted by:

Balderosa, Ernest

Caasi, Samantha Nicole

Marcellana, John Patrick

Noche, Zach Matthew

Submitted to:

Thomas James Tiam-Lee, PHD

March 04, 2023

I. Introduction

The Major Course Output 1: Mazebot of Introduction to Intelligent Systems course has a task to implement an ***n x n*** maze, where n has a range of 3 to 64. The elements of the maze have a starting point ***s***, goal point ***g***, walls ***#***, and empty spaces ***(.)*** which are configured in a text file input. This program also features a bot that is only allowed to step on an empty space, and it is limited to moving solely in four cardinal directions. The main task is to implement an algorithm that will guide the bot from the starting point to the goal point. Furthermore, this algorithm must search through the given states using the algorithm and provide the optimal path from the start to the goal. The number of states that were explored by the bot must be displayed as well as the total cost needed to reach the goal.

The algorithm used in the program is called A*. This algorithm features priorities wherein the state with the lowest total cost will be seen as the highest priority. It was initially designed to traverse a graph problem and it is still widely used today (Ravikiran, 2023). The priority cost is determined by the total move cost from the starting point up to the given state plus the heuristic cost—the estimated number of moves or costs it will take before reaching the goal state that is often determined by a heuristic function.

For this specific problem, the group decided to use the Manhattan Distance to compute for the heuristic cost of a given state. The Manhattan Distance is the metric distance between two points in an N-dimensional space (Szabo, 2015). This heuristic function will be efficient as it will give a lesser cost if the bot is closer to the goal point and otherwise when it is farther away.

II. `Program

In this section, the group will first discuss the visual representations made in the program in Table 1. Afterwards, the group will show and elaborate on the different functionalities of the program and present step-by-step instructions on how to run the program.

A. Visual Representation

Table 1 shows the visual representation of the maze. The maze has four major elements namely: the start, the goal, the walls, and empty spaces. These representations are seen in the graphical user interface of the program.

Table 1. Maze Visual Representation

Start ***s***	represented by a yellow panel with a 2D drawing of a robot and the "Start" text
Goal ***g***	represented by a green panel and the "Goal" text
Walls ***#***	represented by black panels
Empty Spaces (**.**)	represented by white panels
Paths/Spaces searched	a trail of green panels
Optimal Path	a trail of yellow panels

B. Functionalities

The group was able to develop the program with the functionalities: a file selector, a maze generator, graphical user interface. The file project consists of (7) seven classes, with the ***MazeBot*** class being the driver class.

[1] ***A file selector.*** Normally, a program that requires file input would read the file from a pre-defined directory. This functionality extends the ability of the user to choose a custom maze from a desired file destination. This is given the assumption that ***maze.txt*** exists in the current directory of the program. If ***maze.txt*** does not exist, the program will generate the sample maze found on the MCO1 specifications to ensure that there will be a maze to use the file chooser for. Furthermore, even with the assumption that ***maze.txt*** will always be in a valid format, the program will display "Invalid Maze" in the case of invalid file input.

[2] ***A maze generator.*** This functionality provides customizability to a certain extent, allowing the user to generate a maze within the valid range of 3 to 64. The user can also input the probability, from 1 to 100, of the number of walls in the maze, each spawned at a random location. The locations of the start and goal states are also randomly generated by default. However, should the user wish to adjust it, there is also an option for the user to specify the coordinates of the start and goal states. When generated, the program will show a preview of the newly generated maze.

[3] ***GUI.*** For the user's convenience, the program is GUI-based. Upon running the program, the user will be prompted to pick a file using the **[1]** ***<u>file selector</u>.*** The program will proceed to read the file and will launch a GUI with the following buttons and their respective uses:

Table 2. Buttons or Actions Available in the Program

[A] ***Move bot***	The prerequisite for using this button is by using the **[C]** ***<u>A*Search</u>*** button first; failure to do so will prompt the user with "*No Path found. Please initiate my search algorithm*". By using this button, the program will visually represent the MazeBot traversing the optimal path found by the algorithm. The rate at which the bot moves from one space to another is specified by the user in milliseconds. A recommended rate to use would be 200 milliseconds on smaller ***n x n*** maze sizes, faster rates are preferred otherwise.

[B] ***Reset***	This button will reset the maze back to its initial state.
[C] ***A*Search***	This button will visually represent how our algorithm works (*see **Table 1** for reference*). It will show the states explored by the algorithm looking for the goal state and afterwards, highlighting the optimal found by the algorithm.
[D] ***Generate Random Maze***	This button will launch the **[2]** ***maze generator***. The description of this function is described above.
[E] ***Pick New Maze***	This button will prompt the user if they would like to restart the program. If yes, the program will launch the **[1]** ***file selector***, otherwise, there will be no action done.

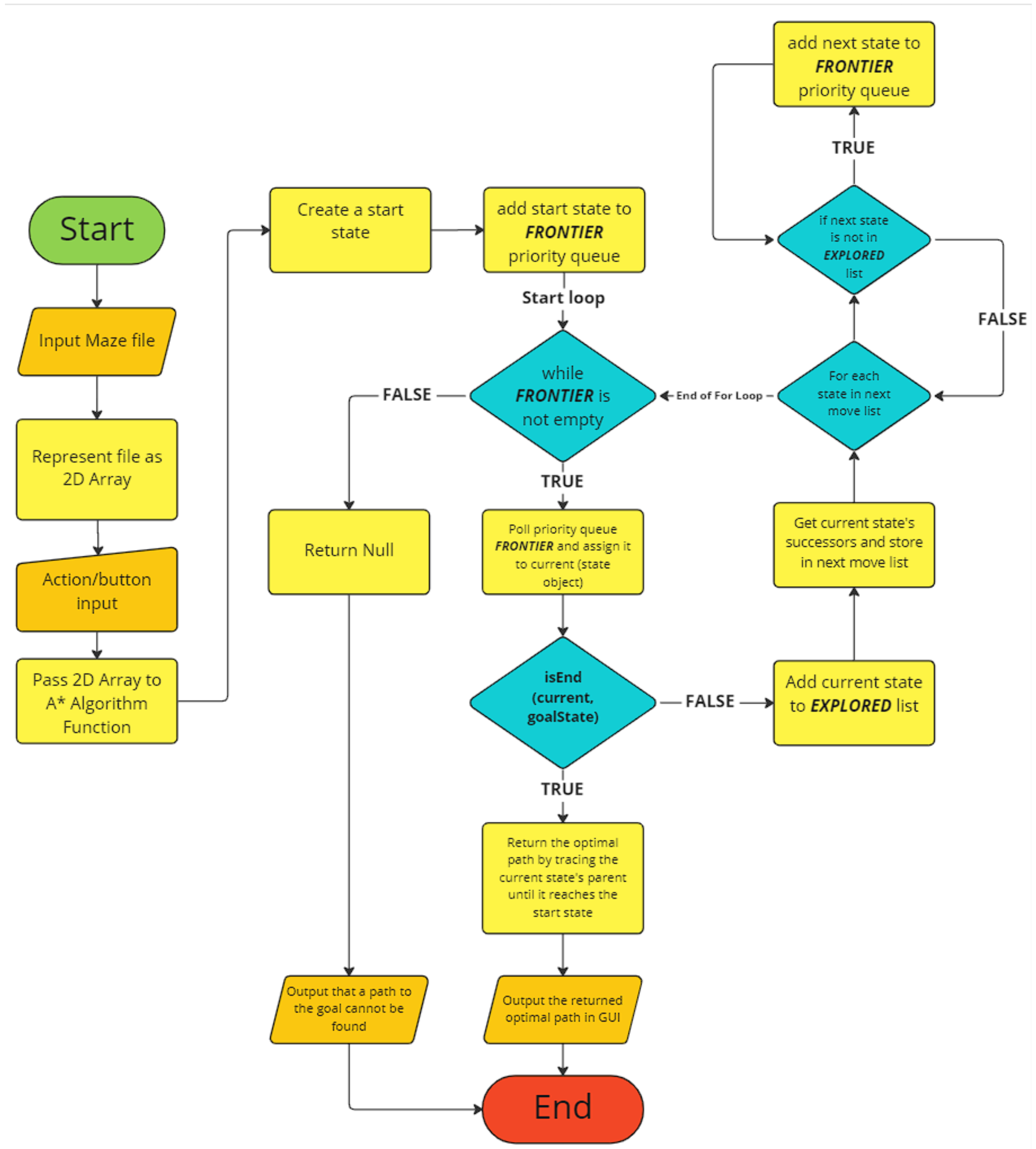
Aside from the maze and the buttons/actions, the details of the states explored are also visible on the right-hand side of the program. The details include the order of the state explored from the start state, the heuristic cost of the state, the total cost to the said state from the start, and the coordinates of the state. It will show once the **[C]** ***A*Search*** button is pressed.

A. Instructions on running the program

To run a maze,

1. Run the driver class or its GUI: ***MazeBot***.
2. Using the **[1]** **file selector**, select the text file of the maze you want to run.
3. Use the **[C]** **A* Search button**.
4. The other buttons, **[A]** **Move Bot**, **[B]** **Reset**, **[D]** **Generate Random Maze**, and **[E]** **Pick New Maze** can be then used optionally.

III. Algorithm



Start
Input Maze file
Represent file as 2D Array
Action/button input
Pass 2D Array to A* Algorithm Function
Create a start state
add start state to ***FRONTIER*** priority queue
Start loop
while ***FRONTIER*** is not empty
FALSE
TRUE
Return Null
Poll priority queue ***FRONTIER*** and assign it to current (state object)
isEnd (current, goalState)
FALSE
TRUE
Add current state to ***EXPLORED*** list
Get current state's successors and store in next move list
For each state in next move list
End of For Loop
if next state is not in ***EXPLORED*** list
TRUE
FALSE
add next state to ***FRONTIER*** priority queue
Return the optimal path by tracing the current state's parent until it reaches the start state
Output that a path to the goal cannot be found
Output the returned optimal path in GUI
End

Figure 1. Flowchart of the Program

The figure above is a flowchart that shows the flow and processes of the program from when it is run up until it is terminated.

The bot follows the standard implementation of the A* algorithm as its path-finding function. Assuming the maze is valid, the bot will always have prior knowledge of where the coordinates or locations are for the start and goal state. As this is A*, the bot is programmed to find the optimal path or the minimum number of moves to reach the goal state. To do so, the algorithm above shows the exact implementation of how the bot determines the next best state it will explore and the path it will take. Therefore, a more comprehensive explanation is needed.

There are three essential components for this A* algorithm implementation, the priority queue "*FRONTIER*", the hash set "*EXPLORED*", and a chosen, admissible heuristic function. These components help keep track of what to explore next, what has been explored, and what is the best state to explore, respectively.

Each valid state to be explored in the maze has five essential attributes that are important for the search. The attributes are as follows:

- (x, y) is the coordinate or the location of the state in the maze.
- *pathCost* is the culmination action cost of this state.
- *heuristicCost* is the heuristic cost of this state.
- *costSoFar* is defined as the formula $f(n) = pathCost + heuristicCost.$
- *Parent* is the parent or ancestor of this state.

Once the framework has been set, the algorithm starts by polling *FRONTIER* for the first state to explore, which will always be the start state. Once polled, this state will be set as the current state the bot is in. This state will first be checked if it is already the goal state by calling the function, $isEnd(c, g)$, for parameter c is the current state while parameter g is the goal state. This function simply checks if the current state of the bot is the goal state. If it is, return the path from the goal state to the start state by traversing the parent of each state, akin to a linked list. If it is not, store the current state to *EXPLORED* then proceed to find the next successor states.

When finding the next successor states, the function, $getNextStates(c, m)$ for parameter c is the current state while parameter m is the maze, is called. This function will check the four cardinal directions of the state and check if the next state is valid, meaning it is not outside the bounds of the maze or it is not a wall state. If it is valid, then the attributes of the next valid state will be updated accordingly. Once all four cardinal directions have been checked, this function will return the list of valid successor states. Once the list is retrieved, each state will be checked if it has already been explored by the bot, if not then this state will be stored in *FRONTIER*. This will then repeat while *FRONTIER* is not empty. Assuming there is a path to the goal, this algorithm will always return a valid, optimal path to the goal, else it will return a null value to indicate that no path can be found to reach the goal state.

Although this algorithm works, its efficiency and performance wholly depend on the chosen heuristic function as this serves as the search guide for the bot. Thus, an appropriate and admissible heuristic must be used, and the heuristic must be based on distance in maze navigation.

There are many types of metrics to choose from. Some of the popular distance metrics for machine learning models include Manhattan, Euclidean, Minkowski, Hamming, and Cosine distance. The group was able to narrow it down to two options, Manhattan and Euclidean. In the end, the group used the Manhattan distance because we need to calculate a grid-like path (Gohrani, 2019). Although many prefer using the Euclidean distance metric, the problem given has a high dimensionality, so it is preferable to use a heuristic with lower values (Aggarwal et al., 2001).

To confirm the claims from Aggarwal et al.(2001), tests were performed with another sample maze (mazeGen.txt). It is a bigger 16 x 16 sized maze which helps compare the difference in the number of steps taken by each heuristic with A*.

Heuristic Comparison	
Type of Heuristic	**Number of Steps**
Pure A* / No Heuristic (Constant)	82
Euclidean Distance with A*	30
Manhattan Distance with A*	23

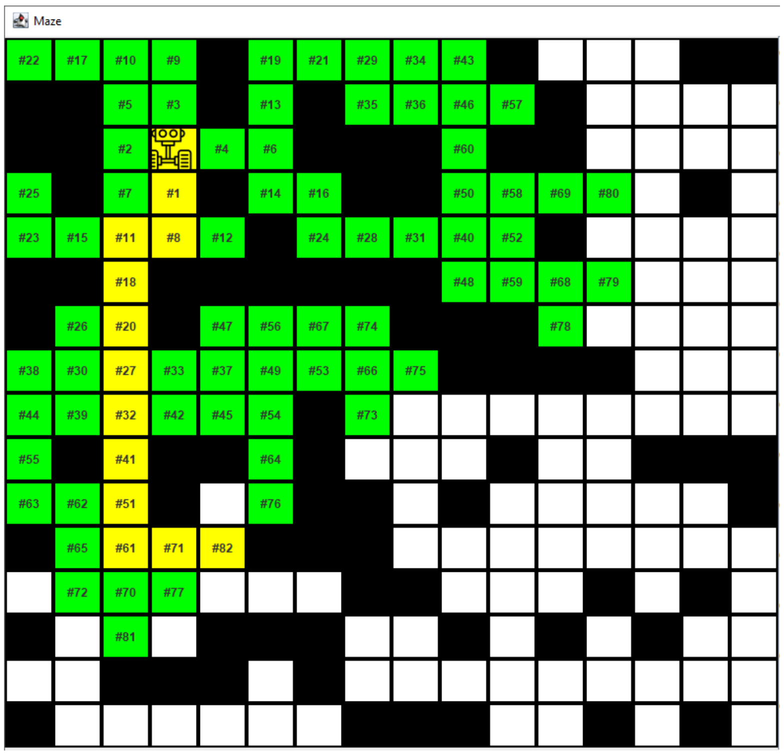


Figure 2. Pure A* / No Heuristic (Constant) ran on mazeGen.txt

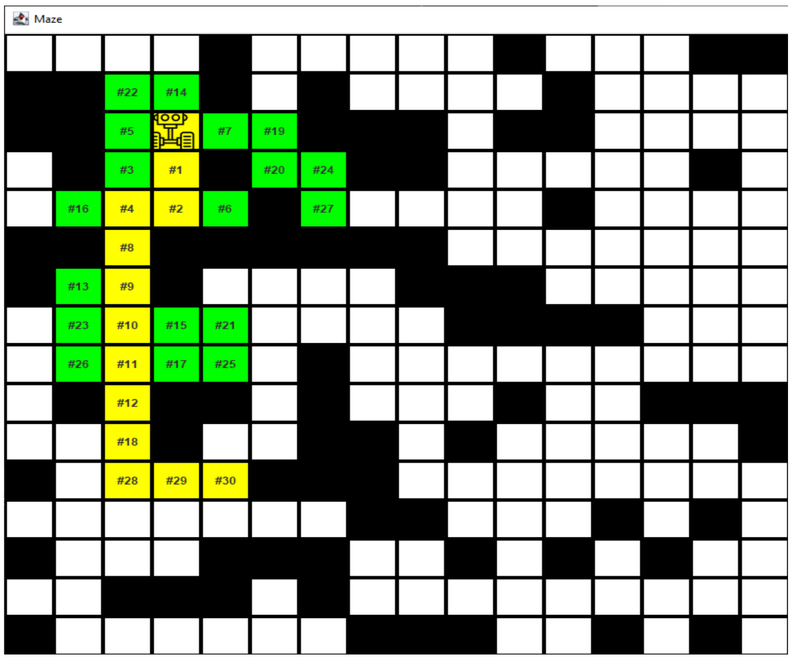


Figure 3. Euclidean Distance on mazeGen.txt

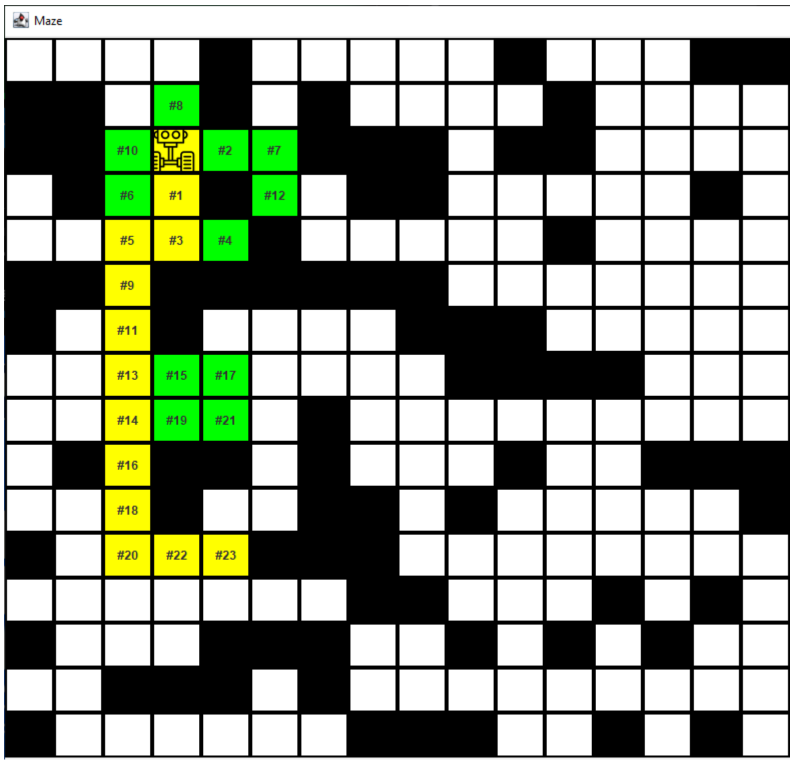


Figure 4. Manhattan Distance on mazeGen.txt

After running the different heuristics, Manhattan distance returned the lowest number of steps with just 23 steps, compared to Euclidean which returned 30 steps. Due to this, Manhattan distance will be used as the heuristic for the A* algorithm.

IV. Result and Analysis

The group ran multiple tests to determine what situations the bot performs poorly in and to determine what situations the bot can easily handle. These tests include changing the size of the maze, including more walls in the maze, or removing all walls in the maze.

The bot performs well in situations where there are a lot of walls in the maze. The bot can identify the optimal path from the start to the goal with fewer states being explored. An example of this is there is only one path to the goal which is shown in the figure below.

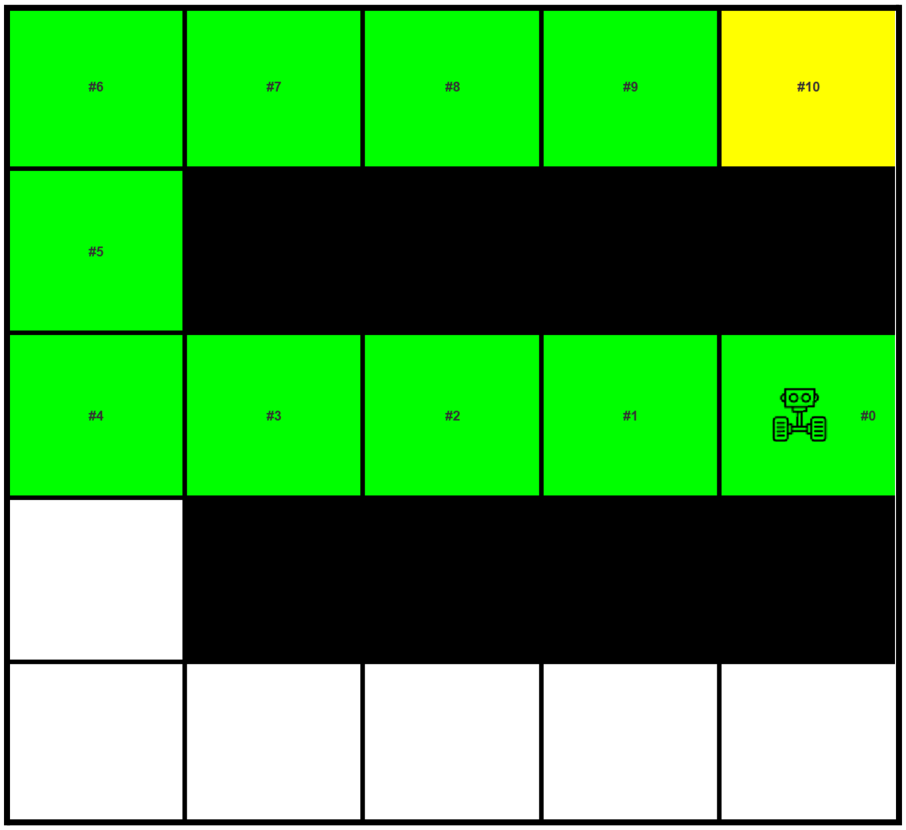


Figure 5. One Path to the Goal

The figure above shows that the only state explored by the algorithm is the optimal path itself. This is because of the walls surrounding the path which makes the bot explore only those. If the walls above the starting point were removed, the bot would have to explore those states as well. This is discussed below along with the situations where the bot performs poorly.

The bot also performs well when it comes to consistency as it handles any type of given maze, and it always returns an optimal path, or no path found. However, there are some limitations to the algorithm of the bot including situations where it performs poorly which are explained below.

While the group was doing tests on the program, the bot seemed to perform poorly when there were the least number of walls. To reach the goal, the bot must check more states in a wallless maze because of the presence of more optimal paths, unlike a maze with walls where the state search is limited. The image below shows this situation.

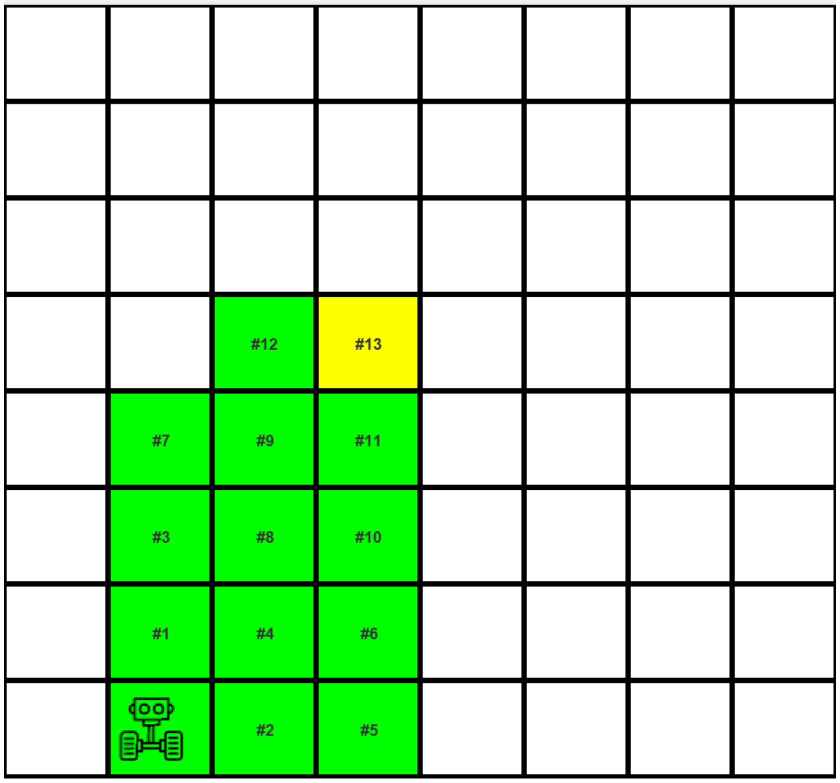


Figure 6. Maze With No Walls

As seen above, when compared to a real-life situation, the bot should have seen the goal point when there are no walls, however, this algorithm checks each state and compares it to other succeeding states on which path will be faster rather than immediately going for the goal state.

While testing this, the group also noticed that the bot will immediately find the goal state when the cost of 1 for every move is removed. This is because, when the move cost is removed, the priority cost is now only the Manhattan distance between two states and those values will decrease further when moving towards the goal state. The move cost is however helpful when there are more walls present in the maze as it helps guide the bot towards the goal state.

Another situation where the bot performs poorly is when there are two similar paths. The figure below shows this type of situation where the Manhattan distances of the states as the path progress are similar.

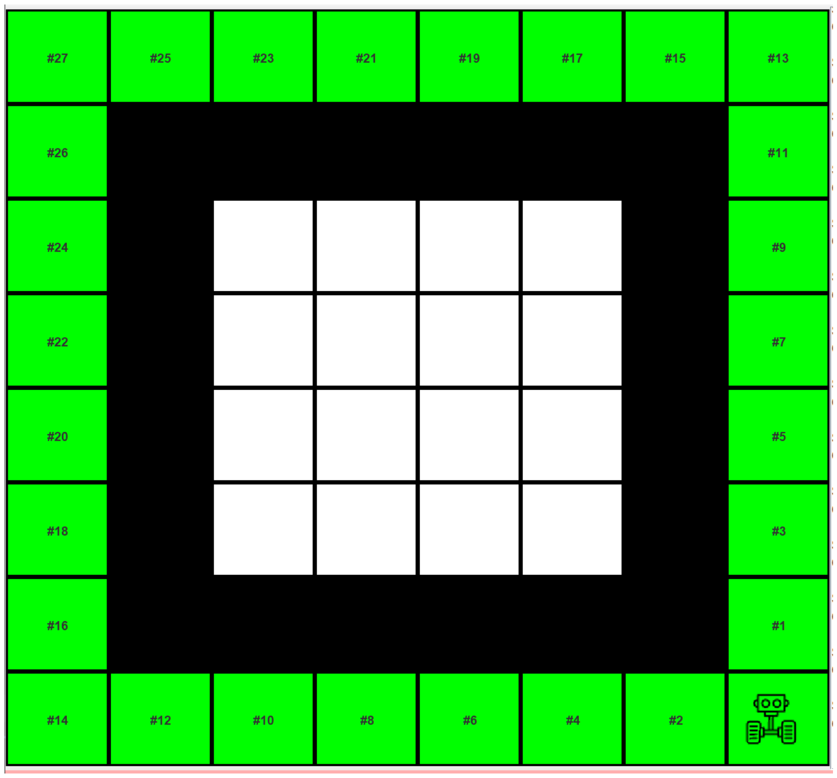


Figure 7. Similar Optimal Paths

Seen above is a situation encountered when the path is similar to another path, and either is optimal. The bot performs poorly in this situation since the bot will have to check both paths looking for the optimal one. And since both have similar costs and priorities, the bot will have to go through all those states and as more states are explored, the space complexity will be larger since when there are states explored, a new object "State" is created in the group's algorithm.

With all the mentioned situations where the bot performs well and performs poorly, answering the question if the bot is intelligent or not depends on how intelligence is defined for artificial systems. For the bot to be intelligent, it must show that it can perform a specific task that needs intelligence. For our bot, it can perform the specific task of maze navigation. As described by a Britannica article, this behavior is meant to show intelligence in problem solving wherein *"Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution."* (Copeland, 2023).

Thus, the bot is described to be "intelligent" since in any type of given maze, it can display the optimal path and the states it went through and is indicative if there is no path at all. However, with the limitations and weaknesses mentioned above, the bot is arguably "unintelligent". An example of this was mentioned when there are no walls present in the maze, where if it were to be compared to a human, the human will head directly to the goal while the bot will search for the tiles in looking for the path to the goal.

Nevertheless, its weakness and limitation cannot be a determining factor for its "unintelligentness" because intelligence does not mean to perfectly do a task but to perform the task based on the given circumstances and determining the best course of action. Since artificial intelligence is meant to imitate human intelligence, which is described as *"mental quality that consists of the abilities to learn from experience, adapt to new situations, understand and handle abstract concepts, and use knowledge to manipulate one's environment."* (Sternberg, 2022), our path finding maze bot therefore is conclusive to be artificially intelligent as it can perform its task, much like any human would in a maze path finding situation.

V. Recommendations

As analyzed previously, the main weakness of the bot is its space complexity or the number of states it explores before finding the goal. This is due to states in the 2D maze having a similar heuristic and total cost. The Manhattan distance is still the best heuristic function to use in this situation, but it cannot be denied that because of how 2D mazes are structured wherein adjacent cells are constant and spaced evenly apart, the cost of many states will tend to have a similar total cost, which will lead the bot to explore many unnecessary states. It is also worth noting that in a 2D maze, there exist multiple optimal paths, which is one of the reasons why the bot will tend to explore a lot of states before reaching the goal

In light of the bot's weakness, are potential solutions that will help reduce and optimize the number of searches the bot does. There exists a plethora of variations of the A* algorithm and heuristic functions that could potentially work better than the basic version. However, because a 2D maze is simplistic in its nature, a simple implementation is sufficient to optimize the bot's current search algorithm. A key aspect as to why the bot explores multiple paths is because the cost of each state is similar to one another, which means multiple states could contain an optimal path to the goal. Thus, an implementation wherein the bot is assured to stay in an optimal path without having to constantly check other states is implemented.

The implementation of this will be done on the priority aspect of the search. In its current version, it checks for the total cost of the state and prioritizes the lowest-cost state. In case of a tie, it will prioritize the first in line in the queue. As mentioned earlier, the total cost of states tends to be similar, this is why instead of just prioritizing the first in line, the bot should also consider the distance of both states relative to the start state. Essentially, the bot will prioritize states that are farther away from the start but still consider that the state that it is currently in is an optimal path to the goal. This implementation will help the bot stick to the farthest path and not have to check other potential paths, only changing its current path when it determines that this path will not lead to the goal. To check if this implementation helps improve the bot's search algorithm, side-by-side test comparisons are done.

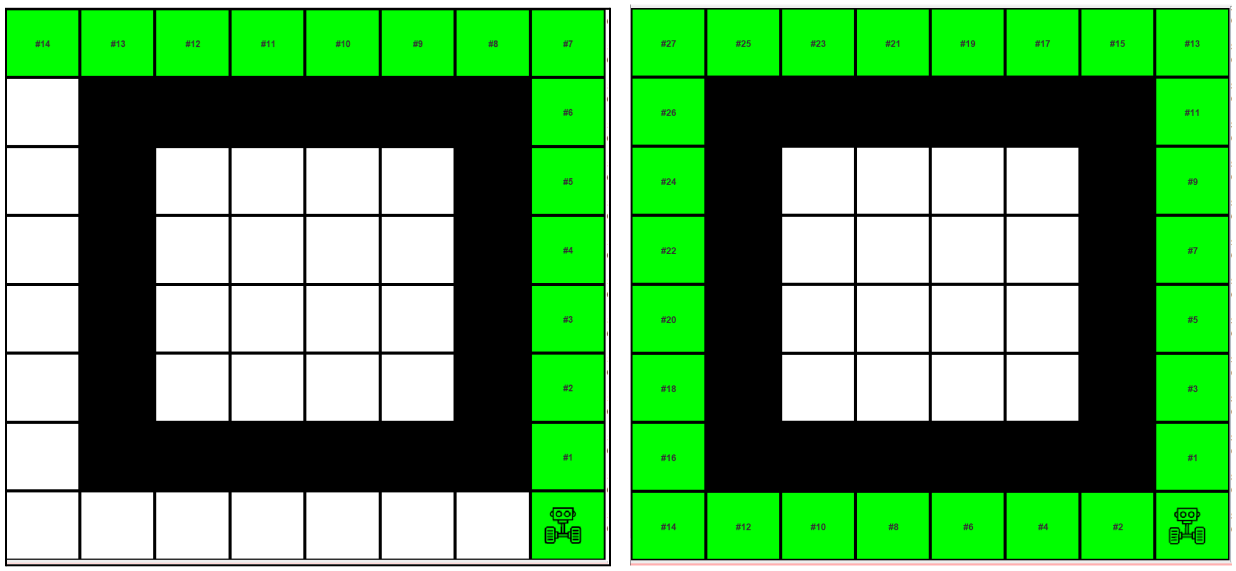


Figure 8. Side-By-Side Comparison of Similar Optimal Paths

As seen above, the bot in the left figure explored the north state and continued to explore it until it found the goal state, unlike the bot in the right figure which had to keep checking states in both directions as they both contain the optimal path.

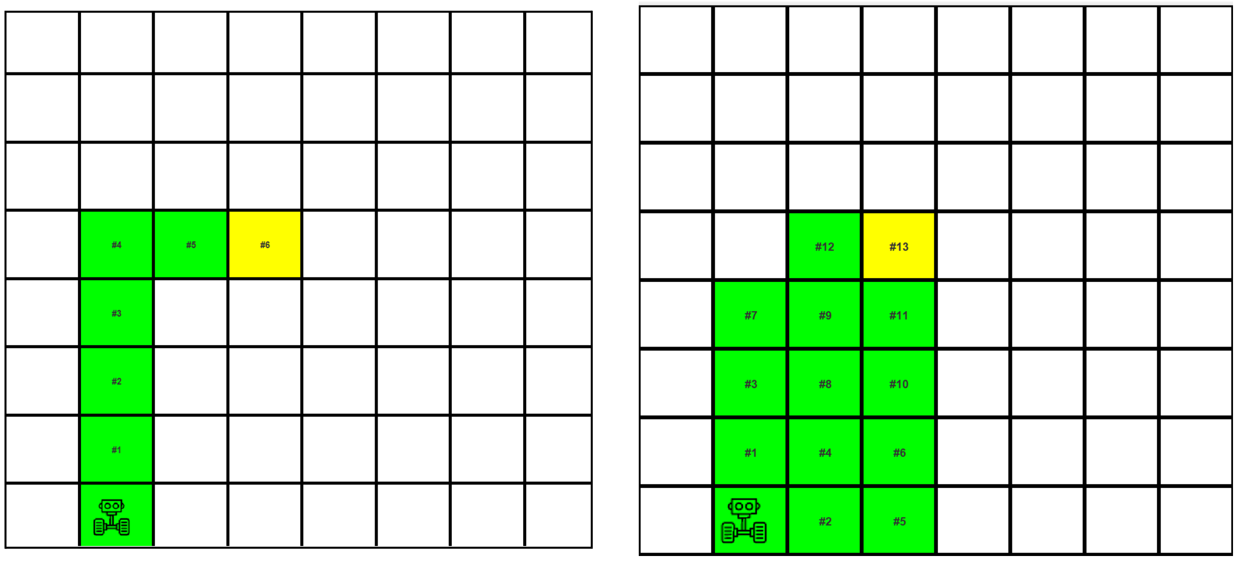


Figure 9. Side-By-Side Comparison of a Non-Wall Maze

As seen above, in a maze with no walls, there exist many optimal paths to the goal, this made the bot in the right figure explore different states and paths before reaching the goal. However, the bot in the left figure stayed on one path and explored it until it found the goal state.

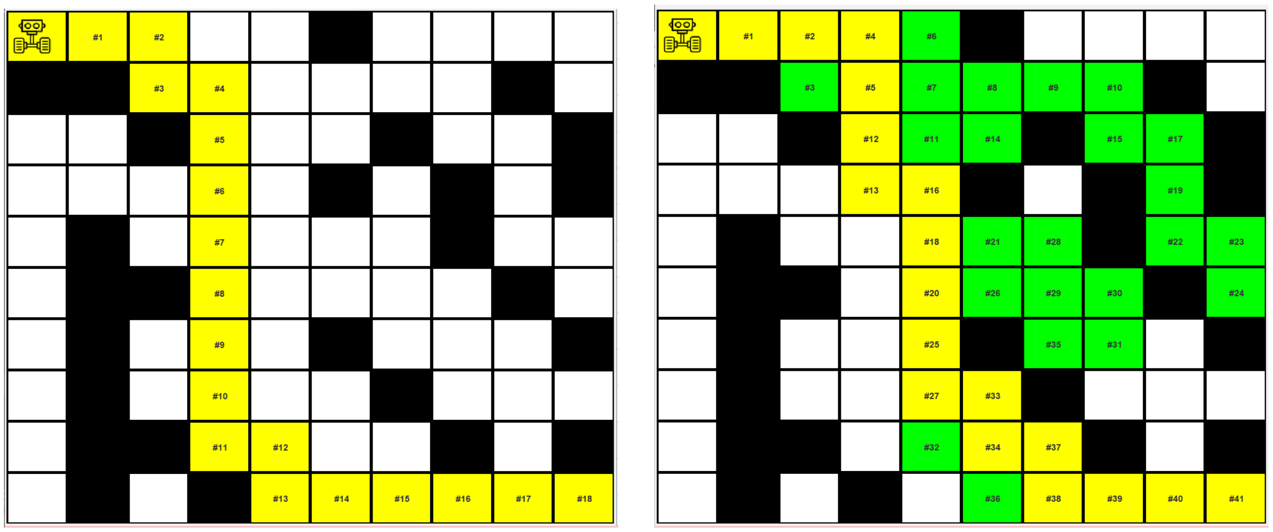


Figure 10. Side-By-Side Comparison of a Random Maze

As seen above, in a random maze, the bot on the right figure explored a lot of states before finding an optimal path to the goal. While the bot on the left figure was able to find an optimal path to the goal without having to explore unnecessary states.

While the implementation was able to improve the search of the bot, there are still limitations to this approach. One key limitation is that the current farthest path the bot is currently in is not guaranteed to lead to the goal state. This means that even if the current path is an optimal path, there might be a wall that might hinder the path which will eventually lead the bot into searching for another path.

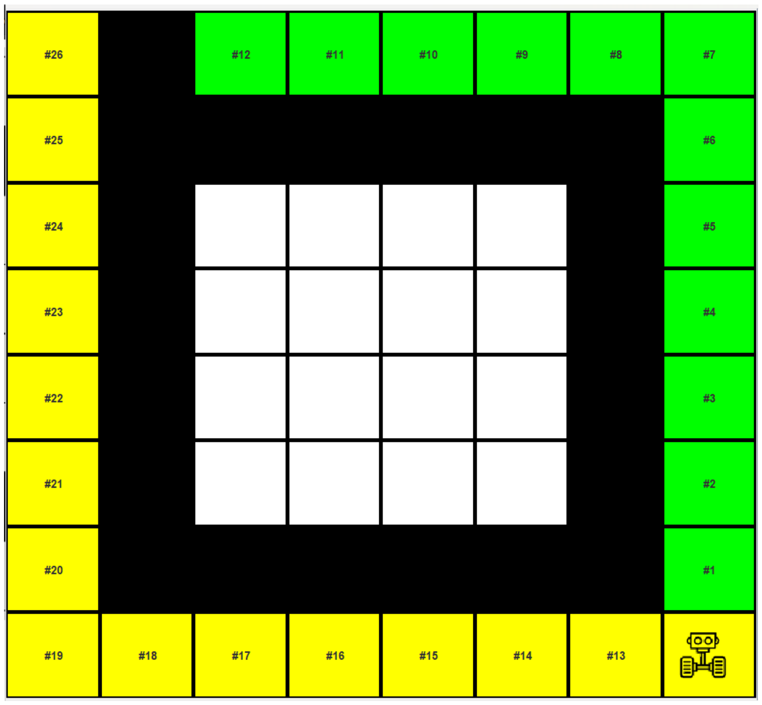


Figure 11. Limitation of the implementation

As seen above, when the bot searches the north state, it knows that it is an optimal path and continues to explore it. However, it did not anticipate that there would be a wall that would block this path which made it explore the west state and only then was it able to find the optimal path to the goal. In a standard maze with walls that block certain optimal paths, the bot will, unfortunately, must explore these paths, not knowing that this path will not lead to the goal. Nevertheless, the implementation in general will improve the search of the bot and make it less likely to explore unnecessary states.

There is no doubt that the implementation still has room for improvement and other implementations could be done to further improve the search. Two implementations in mind are wall detection and state/path pruning. The former could be done such that it detects a path blocked by a wall before having to explore the entire path while the latter prunes redundant and unnecessary states so that fewer states and paths are explored. These implementations could improve the search by quite a big margin and help optimize the algorithm even further, especially when done on bigger-sized mazes.

VI. References

Aggarwal, C., Hinneburg, A., & Keim, D. (2001). On the Surprising Behavior of Distance Metrics in High Dimensional Space. Retrieved from https://bib.dbvis.de/uploadedFiles/155.pdf.

Copeland, B. (2023, February 16). artificial intelligence. Encyclopedia Britannica. https://www.britannica.com/technology/artificial-intelligence

Gohrani, K. (2019). Different Types of Distance Metrics used in Machine Learning. Retrieved from https://medium.com/@kunal_gohrani/different-types-of-distance-metrics-used-in-machine-learning-e9928c5e26c7

Ravikiran, S. (2023, February 15). *A* algorithm in artificial intelligence you must know in 2023: Simplilearn*. Simplilearn.com. Retrieved February 18, 2023, from https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#:~:text=A*%20Search%20Algorithm%20is%20a,path%20algorithm%20(Dijkstra%27s%20Algorithm).

Sternberg, R. J. (2022, April 11). human intelligence. Encyclopedia Britannica. https://www.britannica.com/science/human-intelligence-psychology

Szabo, F. E. (2015). M. *The Linear Algebra Survival Guide*, 219–233. https://doi.org/10.1016/b978-0-12-409520-5.50020-5

VII. Contributions

Contribution Table

Names	Contributions
Balderosa, Ernest	● Assisted in making the flowchart ● Comparison of different heuristics ● Prove why the best heuristic to use for the project is Manhattan Distance
Caasi, Samantha Nicole	● Contributed to the back-end and front-end development of the code ● Defined the program and its functionalities in the report
Marcellana, John Patrick	● Debugging the code ● Introduction of the report ● Assisted in making the flowchart of the program ● Results and Analysis of the report

Noche, Zach Matthew	● Programmed and defined the algorithm used by the bot ● Contributed to both the back end and the front-end code development ● Recommendation of the report