De La Salle University

Term 2, A.Y. 2022-2023

Major Course Output I: Maze Bot

Mr. Thomas James Tiam-Lee, PhD

In partial fulfillment of requirements for CSINTSY (S16)

Introduction to Intelligent Systems

By:

Francis Benedict Martinez

Robert Joachim Encinas

Armina Jawali

Hans Martin Rejano

March 06, 2023

Major Course Output I: Maze Bot

Francis Martinez, Armina Jawali, Martin Rejano, Robert Encinas

DLSU Course CSINTSY (S16)

Term 2, A.Y. 2022-2023, March 6, 2023

---

1 Introduction

This paper discusses the design, development, and evaluation of a rudimentary "intelligent" maze bot. Its intelligence will be deliberated through a series of examinations.

Included in the bot's behavior is the ability to traverse through a square $\mathbf{n} \times \mathbf{n}$ maze configuration as represented through a text file of symbols. The bot may only move one space at a time and along the four cardinal directions. Symbols in the file are defined as:

Symbol	Description
.	Open corridor
#	Wall
S	Starting location
G	Target location / end goal

Table 1. Maze representation symbol legend

The bot is expected to load a maze configuration through a text-file similar to Figure 1 and then must find a way to the goal, show the amount of states visited, and the actual path it took. If it cannot find a goal, it must show that it was unsuccessful in its journey.

```
4
S.##
#.#G
#.#.
#...
```

Figure 1. Example of an Input Maze file

1.1 Overview of the Algorithm (A-Star)

In order for the bot to explore the maze and reach its destination, a search algorithm must be employed. The developers of the bot decided on the A-star algorithm.

A-star or A* is a search algorithm that attempts to find the optimal path from the starting node to an end node by adding a heuristic function to augment the actual cost by estimating a remaining cost to reach the goal (Heineman, 2016).

To accomplish this, the algorithm keeps track of:

- The cost to reach each node from a starting point and;
- Total cost or the sum of the cost to reach the node and result of the heuristic function

For this bot, two heuristic functions were defined based on the Manhattan and Euclidean distance between the current state and the goal state which will be added together to form the heuristic. This will be discussed more in Section 3.

2 Program

In this section, one can see the step by step instructions on how to run the program and use its features.

2.1 Prerequisites

Please install the following languages, libraries, and dependencies:

- Python 3.10.7
- Tkinter module v8.6
- Threading module

2.2 Installing the Program

Download the project's repository found on the link below:
https://github.com/nemurinoU/CSINTSY-MCO1-MazeBot.git

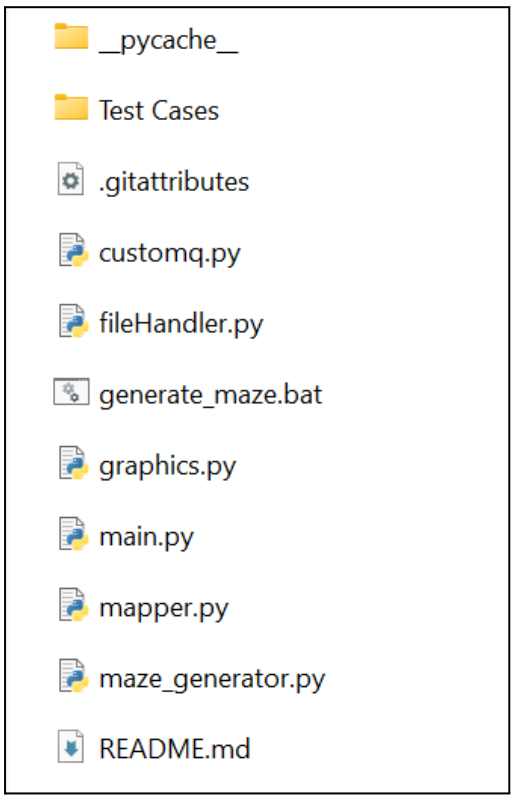

Figure 2. Working Project Directory

Upon cloning the project, you will be greeted with this parent directory and the ff. files:

`Test Cases` : holds text files of maze configs

`customq.py` : personal implementation of a priority queue

`fileHandler.py` : parses maze configs

`generate_maze.bat, maze_generator.py` : makes new test cases

`graphics.py` : module that handles the graphical interface of the maze

`main.py` : main script

`mapper.py` : module that simplifies the maze

For further information, read the documentation found within each file.

2.3 Running the Program

2.3.1 Preparing the Maze

Copy and paste a new configuration file to the "Test Cases" folder and rename it to maze.txt.

2.3.2 Starting the script

To start the program on Windows machines, one may simply double click the `launch.bat` file found in the *app* folder. It will try to run the main script for the program based on common PATH variables/aliases for Python.

Otherwise, you may manually run the script through your terminal with your system's alias for Python:

```
python3 main.py
```

If you are stuck on this step, please consider looking up how to configure PATH variables.

2.3.3 Initial Screen

The user will be greeted by a window showing the maze in a graphical user interface. Another window will pop up asking the user to press “OK” in order to start the search.

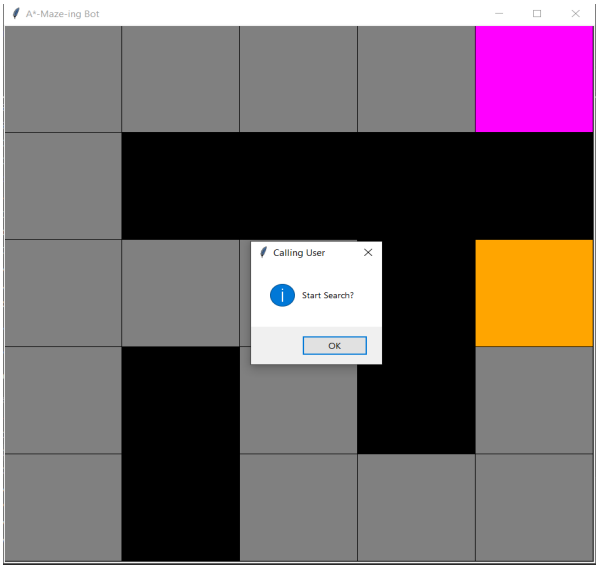

Figure 3. Maze GUI and Prompt

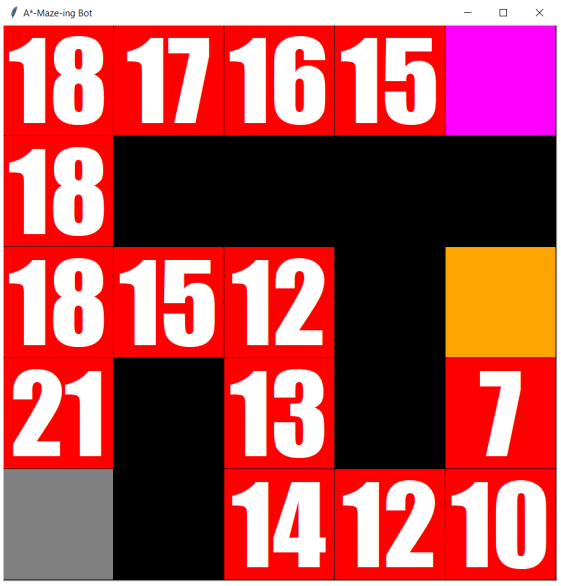

Figure 4. Search Animation

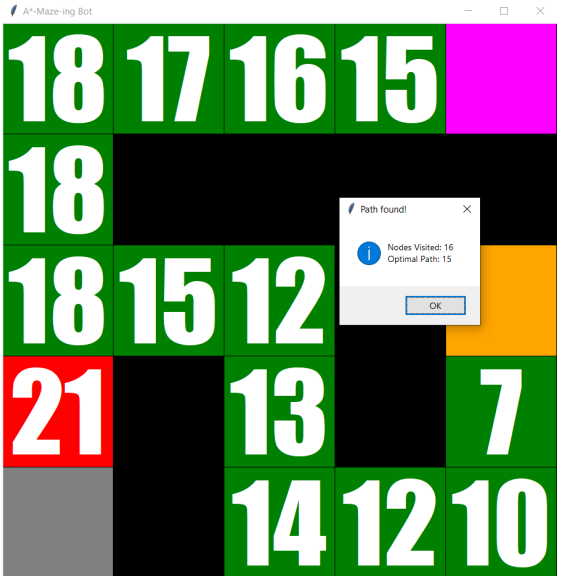

Figure 5. Completed Search

Upon initiating the search, the bot’s starting point is represented in ORANGE and will promptly explore the maze to find the goal colored MAGENTA. It will show an animation of explored RED states first and will begin tracing the “optimal” path in GREEN. Note the text inside represents the TOTAL COST for that state.

3 Algorithm

In this section, you can see the pseudocode of how the bot works, as well as the algorithms and rules used by it.

3.1 Pseudocode of Main Algorithm

To aid the bot in traversal, the inputted maze is modeled as a graph whereas each node contains information about a tile’s euclidean coordinates, total cost, and heuristic cost. This model is implemented in a priority queue data structure.

A-star Algorithm

```
1   nodes ← PriorityQueue ()
2   visited ←HashTable()
3   g_cost ← HashTable() {tile : inf for all tiles
    except start_tile : 0}
4   f_cost ← HashTable() {tile : inf for all tiles
    except start: heuristic (start)}

5   push (f_cost[key=start],
          heuristic (start),
          start) into nodes
6
7   while node is not empty or goal reached:
8         current_tile ← node.pop ()[2]
9         for cardinal direction 'NESW':
10              child ← next explorable tile
11              temp_g ← g_cost (current_cost) + 1
12              temp_f ← temp_g + heuristic (child)
13
14              if temp_f < f_cost[key=child]:
15                    g_cost[key=child] ← temp_g
16                    f_cost[key=child] ← temp_f
17                    push (f_cost[key=child],
                            heuristic (child),
                            child)
18                    visited (child)←current_tile
19
20  actual ← HashTable()
21  head ← goal
22  success ← True
23  actual[key=goal] ← goal
24
25  while head is not at start:
26        actual[key=visited[key=head]] ← head
27        head ← visited[key=head]
28        if goal cannot be reached:
29              success ← False
30              break
31
32  return visited, actual, start, goal, success, f_cost
```

Figure 6. Pseudocode for A-star algorithm

Lines 1-4: declaration of preliminary variables

- `nodes` contains all the coordinate data of the maze
- `visited` is the set of nodes expanded by the bot (ex. visited(start) expanded next node)
- `g_cost` contains the actual cost of all nodes
- `f_cost` contains all the total cost of all nodes
- `actual` contains the set of coordinates traversed to be the "optimal" path
- `success` tells us if there is a way to reach the goal

Lines 7-18: main traversal block of the bot

This block describes how the bot decides to visit each tile. The function works by having a priority queue (var node) of the nodes to be traversed. Once the current node has been selected the function then checks North, East, South, and West for other explorable nodes adjacent to the current node. This explorable node is then assigned to the variable child and then the current actual cost and the heuristic plus actual cost are stored in temporary variables temp_g and temp_f respectively.

If this temporary cost is less than the f_cost associated with the child node then temp_f and temp_g are stored into their respective dictionaries that store the actual g_cost and f_cost. This new child node is then pushed into the priority queue as a valid node to explore and the tile that was explored is placed into the visited list.

Lines 25-32: Tracing the Path Taken by the Bot

This section of the bot's code traces the path taken by the bot to reach the goal state in reverse. If at any point this traversal of the found path ends before reaching the goal the loop terminates and returns that there is no path from the start state to the end state.

3.2 Algorithm and Rules

Among the various search algorithms, the developers chose A-star for its ability to choose the cheapest "optimal" path based on a defined heuristic.

A* is an extension of the Uniform Cost Search/Djikstra's algorithm as it essentially does the same thing but ultimately expands less nodes by prioritizing nodes that are

estimated to be cheaper. Thus, aptly named an “informed search algorithm”.

The heuristic **H** used is the sum of two admissible heuristics (Patel, 2011): Euclidean Distance and Manhattan Distance. Defined as:

$$H(s) = |tile_x - goal_x| + |tile_y - goal_y| + \sqrt{(goal_x - tile_x)^2 + (goal_y - tile_y)^2}$$

Due to how the maze only allows movement in the four cardinal directions and how the Manhattan Distance is calculated, the proposed heuristic is said to be inadmissible. The scenario below proves this:

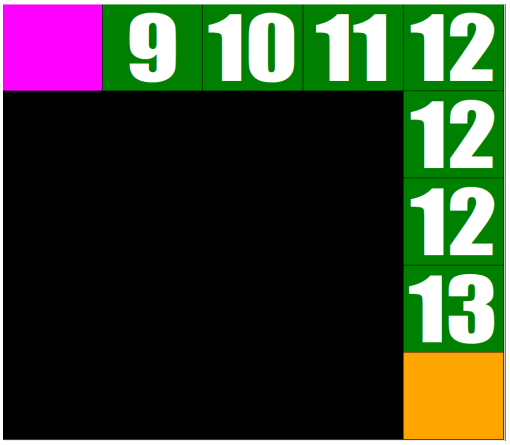

Figure 7. Maze with equal Manhattan Distance and Actual Cost

In this case, the Manhattan Distance from the starting point (4,4) to the goal state (0,0) the Manhattan Distance would return 8 and the actual cost would also be 8. Thus when the Euclidean Distance is added to the Manhattan Distance heuristic the resulting value would be above the actual cost.

Due to the fact that the heuristic function would have cases in which the heuristic overestimates the actual cost, we cannot guarantee that the path that A* returns would be the optimal path (Nilsson, 1986).

Despite the path returned by A* being sub-optimal, the bot still yields a correct path from the starting point to the goal state. In that regard, we can say that the bot meets the requirements.

Additionally, despite its inadmissibility, the developers have decided to use it instead of an admissible one like Manhattan Distance only or Euclidean distance only for its ability to prune more states while still returning a decently optimized path (see figures below).

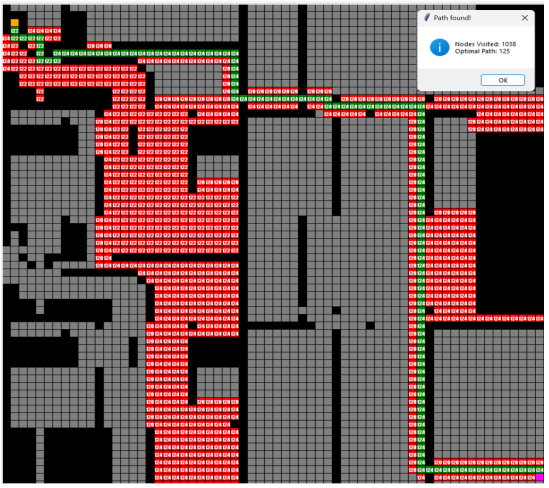

Figure 8. Output using Manhattan Distance

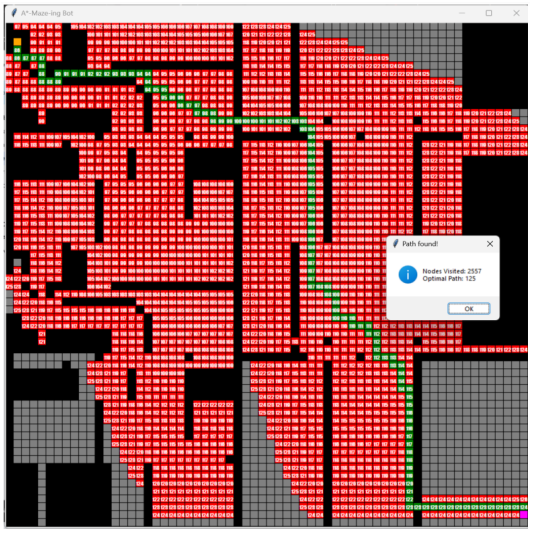

Figure 9. Output using Euclidean Distance

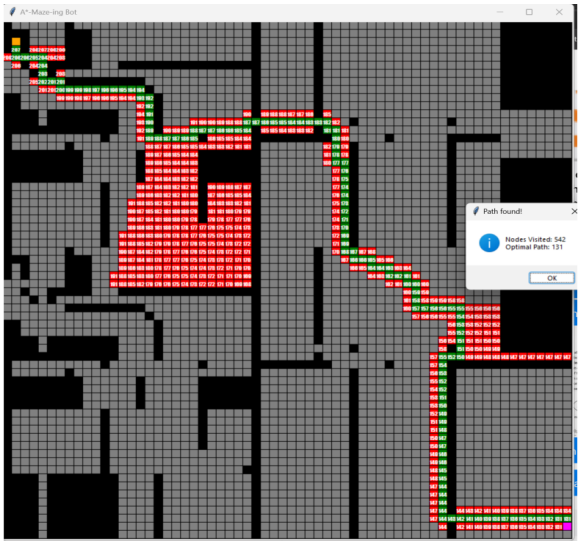

Figure 10. Output using Manhattan and Euclidean Distance

Using a 64 x 64 test case, the developers tried to show that combining the Manhattan and Euclidean distance, the bot will be able to provide a decent path - not necessarily the most optimal path with lesser nodes visited compared to using only either Manhattan Distance or Euclidean Distance as a heuristic.

As shown in Figure 8, the bot explored 1038 nodes while providing a path that only goes through 125 nodes using only the Manhattan Distance.

Moreover, when only the Euclidean Distance is used as the heuristic, the bot was able to explore 2557 nodes, which is much higher than the Manhattan Distance, but provided the same number of nodes for the optimal path as shown in Figure 9.

However, by combining both Manhattan and Euclidean distance, the bot was able to visit less nodes compared to the previous two outputs with only 542 nodes visited but the actual path it returned from the start state towards the goal is only suboptimal as shown in Figure 10.

In cases like these, the heuristic function that the developers have used can still be considered regardless if it is inadmissible because what the bot is after is to generate an actual path from the start state to the goal state.

3.3 Helper Functions

3.3.1 fileHandler.py

This module provides a simple and efficient way of loading maze files and displaying their contents in text form. The first function, print_text_grid, takes two parameters: grid, which is a two-dimensional matrix containing the maze, and n, which is the dimension of one side of the matrix. This function prints the inputted grid file in text through the Command Line Interface (CLI).

The second function, load_file, takes one parameter: FILE_PATH, which is the path of the file containing the maze. This function reads the file and converts its contents into a two-dimensional matrix, which it then returns along with the dimension of one side of the matrix.

3.3.2 mapper.py

This module provides functions for mapping a maze represented as a grid to two different types of data structures: a direction hashmap and a Euclidean space hashmap.

The direction_map function generates a dictionary that contains the coordinates of every non-wall tile in the maze as its keys. Each value of this dictionary is a dictionary itself, where the keys are the directions north, south, east, and west, and the values indicate whether the tile adjacent to the current tile in that direction is free or not (i.e., not a wall).

The euclid_map function generates a tuple of start and goal coordinates as well as a list

containing the coordinates of every non-wall tile in the maze.

Both functions take in a grid (represented as a 2D list of characters) and its dimension n as arguments. The isEdge function is used as a helper function to determine if a tile is adjacent to the edge of the grid.

3.3.3 graphics.py

This module provides a convenient way to create a GUI representation of a maze using the Tkinter canvas.

First, it defines a Graphics class that is used to create a GUI representation of a maze. The class has two attributes, tileSize and canvas, which represent the size of each tile in the maze and the canvas on which the maze is drawn, respectively. The makeTiles() method is called to draw all the tiles of a given grid. The method takes in two parameters, n and grid, which represent the dimension of the grid and the matrix containing the grid, respectively. The method uses a dictionary to map the characters in the grid to their corresponding colors and then calls the drawTile() method to draw each tile.

The drawTile() method takes in three parameters, row, col, and color, which represent the index of the row, index of the column, and the color of the tile, respectively. The method uses the tileSize and the row and column indices to calculate the coordinates of the top-left and bottom-right corners of the tile. It then creates a rectangle object on the canvas using the create_rectangle() method of the canvas object, with the calculated coordinates and the given color.

4 Results and Analysis

Based on the test cases that were fed to the MazeBot, it can be seen that the bot is capable of finding a path from the starting point to the goal state whenever a feasible path exists. This highlights the bot's ability to navigate through the maze and reach the intended destination.

Test Case 1: a 12x12 Maze where the entire maze was traversed

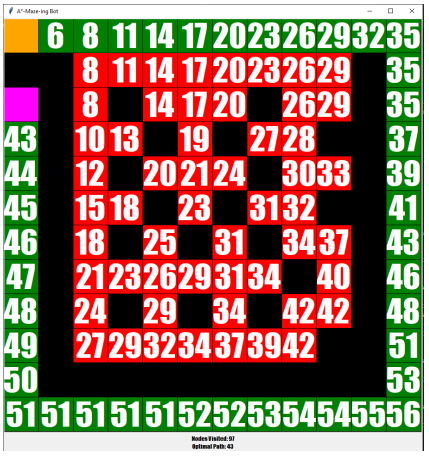

Figure 11. Entire maze is traversed

Despite being able to solve the problem, it must be noted that for mazes that have a large amount of dead ends that “seem” to go closer to the end state, the bot performs very poorly and will traverse nearly all tiles to find the solution or in worst cases the entirety of the maze.

To better understand why the bot most likely “fails” for these test cases, we must understand the limitations of the heuristic functions used by the AI. The two component heuristic functions use the Manhattan distance and Euclidean distance of the tile to the end tile, while ignoring walls.

This property where it ignores walls would explain well why the bot performs poorly when needing to consider dead ends and

paths where a path to the goal goes away from the end tile first.

As mentioned before in the 12x12 test case, the bot thinks that by instantly going down at the first possible chance it can get to the goal state quicker, but in reality the solution is to go far first before going back around to the end tile.

Test Case 2: a 20x20 Maze with lesser dead ends

The best case scenario for the bot is actually the inverse of the worst case. This is to say that the bot performs best when there are no dead ends that would make it "seem" to the bot that it is going closer to the end tile

This is evidenced by the 20x20 test case, a test case that is completely open and without dead ends.

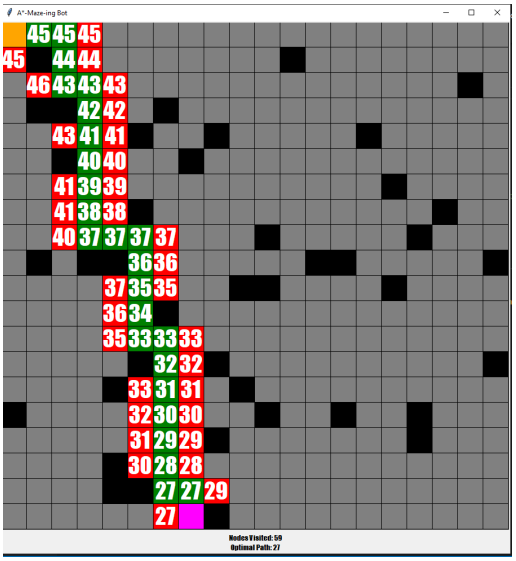

Figure 12. Maze with less dead ends

These observations support the idea proposed by Rachet Freak (2017) that states that the Manhattan Distance is the optimal heuristic for a grid where only the 4 cardinal directions can be traversed and there are no obstacles in the way.

Test Case 3: a 64x64 maze that will show how the bot diverges from the optimal path because of the obstacles

However, the developed bot highlights further the importance of the lack of obstacles in the grid. This is best illustrated by the 64x64 test case as can be seen below.

Figure 13. Bot diverges from actual path

It can be observed that the bot diverges from the solution path when there were dead ends that made it "seem" that the bot was getting closer to the goal but in reality the bot would get stuck in that dead end as seen in the cyan circle.

The performance issue of the bot in mazes with a large number of dead ends is a result of the maze's topology, which affects the maze's connectivity and the distribution of dead ends. When there are a large number of dead ends, the maze becomes more disconnected, meaning there are fewer paths that lead to the goal state.

The bot, in this case, can easily get stuck in these dead ends as the heuristic does not take into account walls that can obstruct the path to the goal state, which causes for more states to be visited before finding the correct path.

Additionally, this phenomenon results in a prolonged exploration of dead ends before finding the correct path *(See: Figure 9)*.

Furthermore, the bot exhibits suboptimal performance in mazes where the optimal path requires initial traversal away from the end goal before subsequently turning back towards it.

Test Case 4: 5x5 Maze that will not reach the goal state

The bot can also handle a test case where the goal state will never be reached. However, it will explore all the nodes until it will say that the goal cannot be reached.

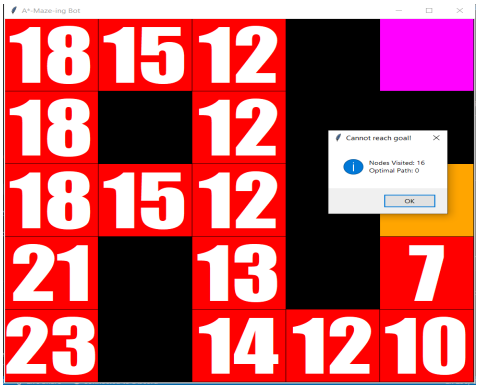

Figure 14. Goal cannot be reached

In the figure above, it can be seen that the goal state is blocked by walls. During these scenarios, the bot explores all the nodes of the maze it possibly can where it then notifies the user that reaching the goal is impossible.

5 Conclusion

Despite the heuristic function being inadmissible and not yielding the optimal path, the created bot still meets the requirements for the given task. The developed bot through a series of tests may be considered *INTELLIGENT* enough through its ability to explore, make decisions at each intersection, and reach the end of a configured maze on its own.

Ultimately, the developers feel that the ability of the bot to prune more states to reach the goal faster despite the suboptimal nature of the path returned is a worthwhile tradeoff. This is reasonable and necessary to enhance the bot's performance in real-world scenarios where sub-optimality is good enough.

6 Recommendations

Due to these limitations of the bot's heuristic function, as well as other issues identified during testing, the following recommendations are made:

- The bot's performance is severely impacted when faced with a maze that contains obstacles between the start and the goal state. In such instances, the bot tends to traverse the majority of the maze before arriving at a solution (See: Figure 9 and 11).

  Development of a novel heuristic that accounts for the presence of obstacles or dead ends along the path such as an "obstacle-aware heuristic" may discourage the bot from pursuing paths that are likely to result in a dead end.

- In cases where a path to the goal state is impossible to find, the bot explores all the nodes before it can tell that it cannot reach the goal state.

  Instead of searching for a path from the starting point to the goal state, an alternative search algorithm could be employed which involves bi-directional search, where the starting point and goal state are explored simultaneously and meet at a common node to confirm the presence of a path. This method may decrease the number of explored nodes, leading to a more efficient search and

quicker confirmation of an existence of a path to the goal state.

- Due to the heuristic being inadmissible, the actual path returned by the bot is no longer guaranteed to be optimal.

  Utilization of an admissible heuristic alleviates that. However, due to the reasons laid out above about why the inadmissible heuristic was chosen, the new heuristic must still prune roughly the same amount of states that the inadmissible heuristic does to be able to maintain the benefits that this current heuristic provides.

7 References

Computerphile. (2017). A* (A Star) search algorithm. YouTube. https://www.youtube.com/watch?v=ySN5Wnu88nE

Freak, R. (2017). Best heuristic for A*. https://cs.stackexchange.com/questions/83618/best-heuristic-for-a

Heineman, G. T. (2016). Algorithms in a nutshell. O'Reilly.

Lague, S. (2014). A* pathfinding (E01: algorithm explanation). YouTube. https://www.youtube.com/watch?v=-L-WgKMFuhE

Nilsson, Nils. (1986). Principles of Artificial Intelligence. Morgan Kaufmann Publishers. https://books.google.com.ph/books?id=F3VFAAAAYAAJ

Patel, Amit. (2011). Pathfinding. Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Song, J. (2019). A comparison of pathfinding algorithms. YouTube. https://www.youtube.com/watch?v=GC-nBgi9r0U

Tech With Tim. (2019). Python path finding tutorial - breadth first search algorithm. YouTube. https://www.youtube.com/watch?v=hettiSrJjM

The Python Standard Library. Python documentation. (n.d.). https://docs.python.org/3/library/index.html

TkDocs. (n.d.). Modern TK best practices. https://tkdocs.com/

8 Contributions of Each Member

Every individual involved in this project made *significant* contributions to its completion.

On a rating of 0-100%:

Member	**Contribution**
Martinez, Francis Benedict	Code (100%) Paper (100%) Testing (98%)
Encinas, Robert Joachim	Code (98%) Paper (100%) Testing (100%)
Jawali, Armina	Code (95%) Paper (100%) Testing (100%)
Rejano, Hans Martin	Code (95%) Paper (100%) Testing (100%)

Table 2. Contributions of each member