

Major Course Output #1: MazeBot

Cruzada, Carla

Escalona, Miguel

Francisco, Piolo

Loyola, Rainier

MCO Group #5

CSINTSY

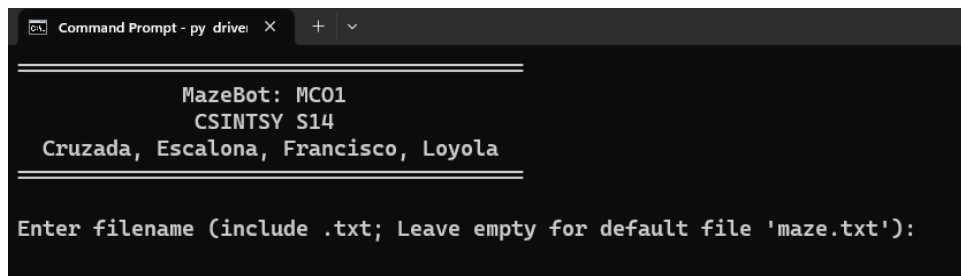
Sir Thomas Tiam-Lee

I. Introduction

A* Search algorithm guides the bot into reaching the goal. It must use a viable path to reach the goal while considering the existence of the walls. This algorithm selects states to be explored first based on their priority, which can be obtained by combining the cost from the initial state to a current state and the estimated cost from the current state to the goal state. An approximation heuristic, specifically the Manhattan distance, is used to compute the estimated cost from the current state to the goal state because it takes into account the limitations of the bot's movement to only move one step up, down, left, and right.

II. Program

- A. The application requires Python 3 in order to run.
- B. Go to the **MazeBot** directory.
- C. Start the application by entering the command: **py driver.py** on a terminal.
- D. Enter the filename of the desired maze file. Leave empty if the default maze.txt will be used as the maze file.



```
Command Prompt - py drive X + v
MazeBot: MC01
CSINTSY S14
Cruzada, Escalona, Francisco, Loyola
Enter filename (include .txt; Leave empty for default file 'maze.txt'):
```

E. The user will then be asked to enable or disable certain search features:

```
Command Prompt - py drive: X + v

=====
MazeBot: MC01
CSINTSY S14
Cruzada, Escalona, Francisco, Loyola
=====

Enter filename (include .txt; Leave empty for default file 'maze.txt'):
Enable rapid search (Y/N): n
Enable manual progression (Y/N):
```

- Fast Search: Enable to skip in-search progress animations and to enable search time recording for performance checking.
- Manual Progression (Fast Search must be disabled): Enable to see the step by step search of the algorithm.

F. As the program runs, press Enter when prompted to do so to proceed to the next best step that the algorithm sees fit. Be warned that when Manual Progression is enabled, there will be screen blinking when the search process is being animated.

```
C:\Windows\System32\cmd.e X + v

=====
MazeBot: MC01
CSINTSY S14
Cruzada, Escalona, Francisco, Loyola
=====

A* Searching...

NOTICE: THIS DOES NOT SHOW THE OPTIMAL PATH YET
AS THE UI ITERATES THROUGH THE EXPLORED LIST.

    0  1  2  3  4
    -  -  -  -  -
0 | .  .  .  .  G
1 | .  #  #  #  #
2 | .  F  E  #  SB
3 | .  #  E  #  E
4 | .  #  E  E  E

Bot's Frontier (1): (1, 2)
Bot's Explored (7): (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2)

Press Enter to continue...|
```

- G. Once the search completes, the application will do either two things:
- Show a replayable final path animation which shows how the bot is recommended to traverse the maze map.
 - Notify the user that the bot did not find a path on the given maze map.
- H. After a successful path search, the final path traversal map will be displayed along with the coordinates of the tiles that should be traversed to get from the Start tile to the Goal tile. From here, you could either type the characters S/s to save the output as a .txt file or X/x to exit the program or press Enter to run another maze file.

```

C:\Windows\System32\cmd.e  X  +  v

MazeBot: MC01
CSINTSY S14
Cruzada, Escalona, Francisco, Loyola
=====

Recommended Path Direction Grid:
  0  1  2  3  4
  -  -  -  -  -
0 | ↑ → → → →
1 | ↑ # # # #
2 | ← ← ↑ # -
3 | . # ↑ # ↓
4 | . # ← ← ↓

Total Move Count: 15
Recommended Path Coordinate: (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2) (1, 2)
                             (0, 2) (0, 1) (0, 0) (1, 0) (2, 0) (3, 0) (4, 0)
Recommended Path Directions: - ↓ ↓ ← ← ↑ ↑ ←
                             ← ↑ ↑ → → → →
Total Frontier States (1): (0, 3)
Total Explored States (15): (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2) (1, 2)
                             (0, 2) (0, 1) (0, 0) (1, 0) (2, 0) (3, 0) (4, 0)
Time taken: 0.01400136947631836s

Menu:
S/s - Save Output
X/x - Exit without Saving Output
Enter - Repeat Runtime

Enter Selection: |

```

(Fast Search Enabled)

```
C:\Windows\System32\cmd.e  X  +  v

=====
MazeBot: MC01
CSINTSY S14
Cruzada, Escalona, Francisco, Loyola
=====

Recommended Path Direction Grid:
      0  1  2  3  4
      -  -  -  -  -
0 |  ↑  →  →  →  →
1 |  ↑  #  #  #  #
2 |  ←  ←  ↑  #  -
3 |  .  #  ↑  #  ↓
4 |  .  #  ←  ←  ↓

Total Move Count: 15
Recommended Path Coordinate: (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2) (1, 2)
                             (0, 2) (0, 1) (0, 0) (1, 0) (2, 0) (3, 0) (4, 0)
Recommended Path Directions: - ↓ ↓ ← ← ↑ ↑ ←
                             ← ↑ ↑ → → → →

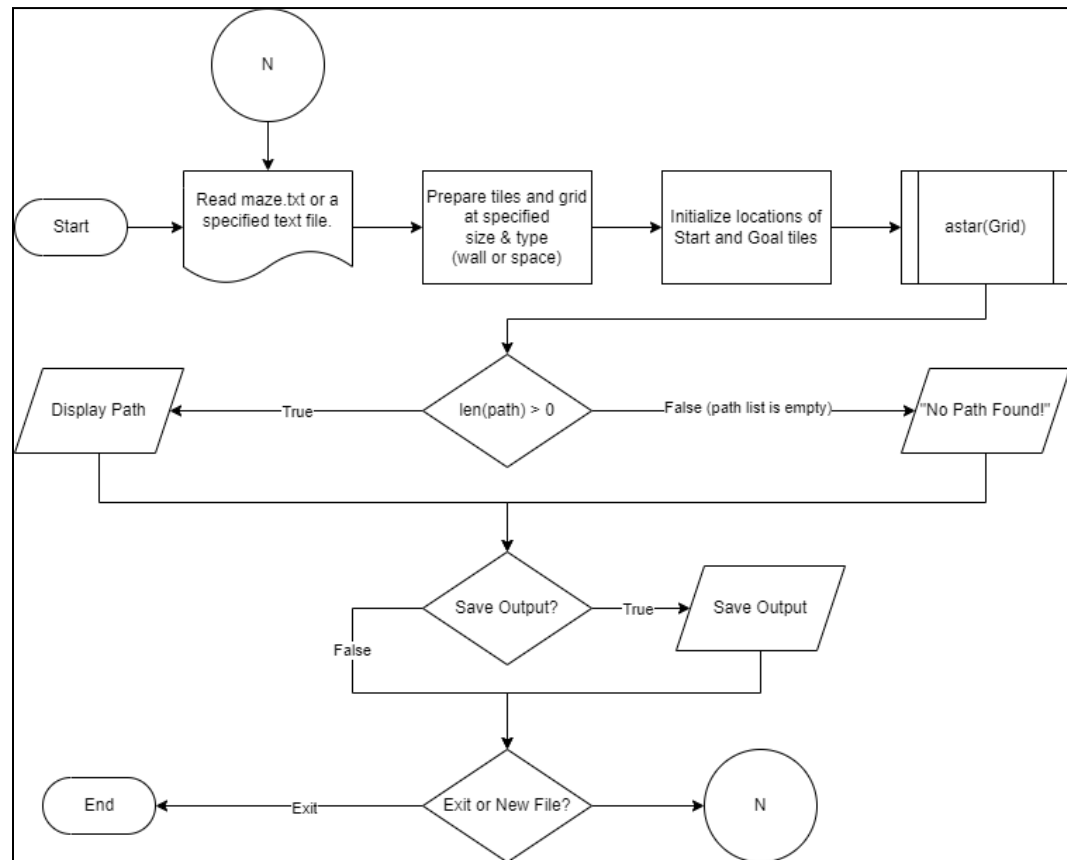
Total Frontier States (1): (0, 3)
Total Explored States (15): (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2) (1, 2)
                             (0, 2) (0, 1) (0, 0) (1, 0) (2, 0) (3, 0) (4, 0)

Menu:
S/s - Save Output
X/x - Exit without Saving Output
Enter - Repeat Runtime

Enter Selection: |
```

Manual Progression, No Fast Search

III. Algorithm



Main Program Flow

```

1 def astar(grid, rapid_search=bool=False, cont=bool=False):
2     frontier = []
3     explored = []
4
5     start_tile = find_start(grid)
6     end_tile = find_end(grid)
7
8     frontier.append(start_tile) #Add start tile as initial frontier state
9
10    #Loop until no frontier unexplored
11    while len(frontier) > 0:
12        current_tile = frontier[0]
13        current_index = 0
14
15        #Find lowest cost
16        current_tile, current_index = find_lowest_cost(frontier)
17
18        #Update lists
19        frontier.pop(current_index)
20        explored.append(current_tile)
21
22        # If the goal is found
23        if isGoal():
24            final_path = []
25            current = current_tile
26            while current is not None:
27                final_path.append(current)
28                current = current.parent
29            return final_path[::-1]
30
31        # Get the actions
32        actions = grid.get_actions(current_tile)
33
34        # Iterate through the action list
35        for action_tile in actions:
36            is_explored = False
37            is_frontier = False
38
39            # Minimize exploration of nodes in the explored list
40            for explored_tile in explored:
41                if action_tile == explored_tile:
42                    is_explored = True
43                    break
44
45            if is_explored: #Skip if explored
46                continue
47
48            #Compute costs of action tile:
49            #Distance from S = Distance from S of current_tile + 1
50            #Distance to G = Manhattan Distance from G to current tile
51            #Priority = Distance from S and Distance to G
52            action_tile.costs = find_costs(action_tile, current_tile, end_tile)
53
54            # Check if the action tiles is present in the frontier list
55            for frontier_tile in frontier:
56                if action_tile == frontier_tile and action_tile.priority < frontier_tile.priority:
57                    frontier_tile.priority = a
58
59            #Remove duplicates for faster search on next run of while loop
60            frontier = remove_duplicates(frontier)
61            explored = remove_duplicates(explored)
62
63    return [] #No viable paths found (frontier list exhausted)

```

A* Algorithm Pseudocode as Implemented

When the driver program is executed, the specified maze file will be parsed into the equivalent grid after which the bot is then injected into the grid or maze by calling a function that locates the character 'S' in the maze and its new location is now labeled as 'SB.' Then, the bot starts searching a path towards the goal state through the use of A* search. *Frontier* and *Explored* are represented as

lists in the program which are then updated at every iteration of the search depending on the bot's current location.

The coordinates are expressed in such a way that the *x coordinate* represents the column number and the *y coordinate* is the row number. In the A* function, the tile objects of the *start tile* and the *end tile* are obtained by calling another function. Then, the initial state *S* is appended to the frontier. A *while* loop with a condition of $\text{len}(\text{frontier}) > 0$ (i.e., the loop won't end until there is a frontier state not explored) is used to conduct the A* search.

Since there are no other states that can lead state *S* to have a lower priority, it is popped from the *Frontier* and appended into *Explored*. At this point, state *S* is initialized as *current_tile* and goes through a conditional statement that checks whether the *current_tile* is the same as the *end tile* (i.e., the goal state) that was obtained earlier.

If the *current_tile* is not yet equal with the *end_tile*, a function with *current_tile* as the argument is called to its surrounding valid tile objects. Only tiles that are not a wall or '#' are appended to the list of actions, that movements that are up, down, left, right in directions are considered, and that such movements won't go beyond grid/maze limits. Then, for each tile in the list of actions, it passes through a conditional statement that checks whether that tile matches a tile in the explored list. If it is in the explored list already, then the iteration for that action tile is skipped to minimize the exploration of nodes. If the action tile is not yet in the explored list, then priority is computed for that tile. The distance of the starting tile to the action tile is just incremented by 1 since movement is limited to go one step up, down, left, or right. The heuristic is also computed by getting the manhattan distance of the action tile to the goal tile. Adding these two distances will obtain the priority value of the action tile.

The next step is to check whether the action tile is present in the *Frontier*. If it is already present in the *Frontier*, then the priority of the two tiles (action tile and frontier tile) must be compared. If the priority of the action tile is less than the priority of the one in the frontier list, then the priority of that frontier tile is updated with the value of the action tile's priority. If the action tile is not yet present in the *Frontier*, then it is simply appended to it.

The succeeding states will have different tiles in the *Frontier* for each iteration, which brings us back to the rules at the start of the *while* loop. The first element from the *Frontier* will be the *current_tile* and *current_index* is set to 0. Then, the *Frontier* is enumerated in consideration of the next tiles and their respective indices. The priority of the *next_tile* is compared against the priority of the current tile. If *next_tile's* priority is less than the *current_tile's* priority, the *current_tile* and *current_index* will be updated to the *next_tile's* priority and index as it offers a better overall cost. This will allow the Bot to traverse the tile with a lesser priority. Then the rest of the code in the while loop mentioned earlier

starting at the comment “Update lists” (lines 19-20) from the pseudocode above executes in the perspective of the intermediate states.

Once the *current_tile* is equal with the *end_tile*, then another *while* loop is executed to find the path that the Bot will traverse to reach the goal tile. A variable *current* holds the value of *current_tile*. While *current* is not None, *current* is appended to the list of final path and its value is updated to the parent of that tile (i.e the previous tile). Once this loop is done (such that no parents are found for *current*), the list of final path will be returned in reverse since its elements were appended in such a way that it begins at the goal tile towards the start tile. If a path leading to the goal tile is not found, A* search will just return an empty list signifying that no viable path is found after exhausting all frontier states to explore.

Do note that the pseudocode on lines 60-61 is just a measure at Python level to make sure there will be no duplicates on both frontier and explored lists which could help improve performance in search and replay functions.

IV. Results and Analysis

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

The bot can handle cases where the only traversal movement for the given grid is only one move up, left, or right. The algorithm can search for the optimal path when the goal state is reachable and not blocked by walls. When the bot reaches the goal, the program shows the recommended path (coordinates) the bot should take, the specific movements from start to goal, and the time taken to reach the goal.

The bot can also handle cases when the goal state is unreachable, where the only traversal movement for the given grid is only one move up, left, or right. When there is no path found, the program does not crash and outputs the message “There is no recommended path found” and asks the user to save the output or exit. In this scenario, the A* algorithm returns an empty path list.

- B. 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.

There was no identified situation that the bot cannot handle nor perform poorly as fixes for those problems were already implemented.

V. Recommendations

- A. 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.

In our first implementation, the main weakness of the bot was its performance. Searching in mazes of larger sizes took a while due to the fact that there were duplicates of coordinates in both the frontier and explored states. This occurs when the configuration of a maze has few to no walls, or if its size is greater than or equal to six (6). To address this, we added an additional line of code to remove the duplicate coordinates in the frontier and explored states from a Python level.

```
frontier = list(dict.fromkeys(frontier)) #Remove duplicates from frontier  
explored = list(dict.fromkeys(explored)) #Removed duplicates from explored
```

Line of code in astar.py to remove duplicates from frontier and explored

In the case of the User Interface, a possible way to address the blinking issue (which risks possible seizure triggers) is to implement a better GUI interface that is not Command line based.

VI. References

- GeeksforGeeks. (2022, May 30). *A Search Algorithm*.
<https://www.geeksforgeeks.org/a-search-algorithm/>
- Sebastian Lague. (2014, December 16). *A* Pathfinding (E01: algorithm explanation)* [Video]. YouTube.
<https://www.youtube.com/watch?v=-L-WgKMFuHE>

VII. Contributions

A. Cruzada

- Contributed reference utility functions.
- Ran every test case against the MazeBot in our test case document and reported findings to the group.
- Contributed in writing the introduction, program, algorithm, and recommendations sections of the MazeBot report.

B. Escalona

- Contributed on skeletal code.
- Added GUI integration.
- Fixed bugs that were discovered from running test cases and for improving performance.

C. Francisco

- Contributed in testing the code for any bugs and reported the findings to the group.
- Contributed to the writing of the results and analysis, and the recommendations sections of the MazeBot Report.

D. Loyola

- Contributed on the code implementation of the A* algorithm.
- Contributed in testing the code for any bugs.
- Fixed bugs that were discovered from the test cases.