SWINBURNE UNIVERSITY OF TECHNOLOGY

SCHOOL OF SCIENCE, COMPUTING

AND ENGINEERING TECHNOLOGIES

=====***=====

COS30019 – INTRODUCTION TO AI

RESEARCH REPORT

Assignment 1, Semester 1, 2023

Robot Navigation Problem

Name: Duy Khoa Pham

Student ID: 103515617

Tutorial 1-4 EN210 Tuesday 10:30

Facilitator: Anika Kanwal

April - 2023

Table of Contents

1. Instructions

The program contains two main versions: Command Line Operation and Graphical User Interface (GUI).

1.1 Command Line Operation

The command is executed on a Disk Operating System (DOS) command-line interface within Windows Operation System. The syntax appears as "$search < filename > < method >$", where the search is the main file "python main.py" used to call the program. The possible inputs for the $< method >$ are bfs, dfs, gbfs, astar, cus1, and cus2 which are all case-insensitive. Upon entering the input into the command line, the output displays information in the format "filename method number_of_nodes \n path" (Figure 1). This shows the number of nodes in the search tree and the pathfinding of the algorithm. In case no solution is found for the search, there will be a message for no solution found in the path.

```
(py10) D:\SWINBURNE\SECOND YEAR - SEM 1\Introduction to AI\Assignment 1\dist>search input.txt astar
input.txt astar 28
right; down; right; right; right; up; up; right; right; right;
```

Figure 1. Sample input and output of the command

1.2 Graphical User Interface

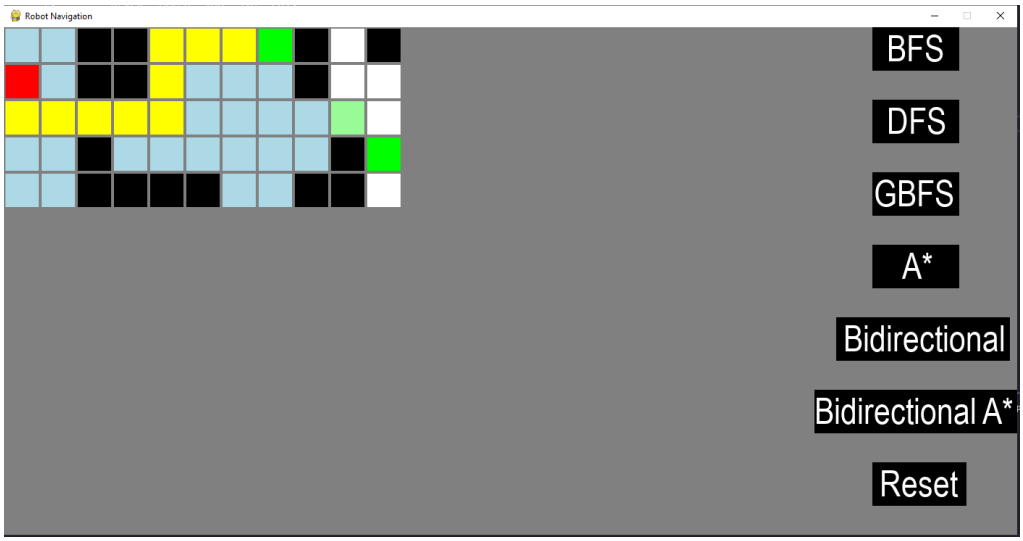

Figure 2. Breadth First Search GUI

The graphical user interface (GUI) runs simultaneously with the output of the command line when the legal input command in. The screen is divided into two sections, with the left side showing a visual representation of the maze in a form of a grid with cells inside. Following the project requirement grid, the start cell is marked in red color, goals are filled in green, and walls are filled in black. Furthermore, to make it easier to track progress, visited and queued nodes are shown in light blue and light green colors, respectively, whereas the finding path of the algorithm is marked in yellow. On the right side of the interface, there are six buttons for selecting different search algorithms,

allowing users to visualize the algorithms in action. In addition, only one algorithm can run at a time according to the command-line requirements, and to visualize data for a different algorithm, the map needs to be reset using the "Reset" button. For example, the interface of running a breadth-first search is shown below in Figure 2.

2. Introduction

2.1 Objective and outline of the report

The report will provide a comprehensive understanding of the search algorithms to solve the "Robot Navigation" problem and their function implementation. The report will cover the key features of the program and highlight any bugs or missing that were encountered during the development process. The report also shows the research initiative of the progress for creating the GUI to control and visualize the map along with its improvement for the program. Finally, the report will detail the conclusion, acknowledgment, and a list of references used in it.

2.2 Problem introduction

The "Robot Navigation" problem involves navigating an NxM grid (where $N > 1$ and $M > 1$). In the grid, there are blank cells and walls which block the direction of the robot. The goal of the problem is to find the optimal path for the robot to travel from its initial starting point to one or more predetermined goals. The problem can be represented as a uniform value graph, where all the moves have a uniform cost of one to calculate.

2.3 Glossary

The report describes several technical terms to analyze the algorithm and its implementation to solve the problem. The following list will interpret the meaning of them for the reader to gain closer insight before reading the whole report:

- Graph: A type of non-linear data structure including nodes and vertices. It is a powerful tool to abstract the relationship of objects in the environment. Thus, it is widely made used in finding path problems.
- Frontier: A set of expanded nodes that have not been explored yet. In the report, depending on the property of the algorithm, the author stores those nodes in a list called stack, queue.
- Priority Queue: An abstract data type stores a collection of items. It provides a special key to prioritize the data in the queue and sort it.
- Heuristic function: In informed search, the function is of paramount importance to get the optimization for finding the problem. In the applied algorithm, the heuristic function calculates the distance to reach the given goal from the location of the point.
- Admissible: A heuristic function is admissible if it does not overestimate the cost of reaching the goal

- A*: A Star research algorithm
- b: branching factor of the search tree
- d: minimum depth of the search tree
- m: maximum depth of the search tree

3. Search Algorithms

There are 6 distinct search algorithms, including depth-first search, breadth-first search, greedy best-first search, A*, bidirectional algorithm, and bidirectional A* algorithm for instructing the robot to discover a path toward the goal. The algorithm for each search is displayed in detail in the following subsection.

3.1 Breadth-First search (BFS)

3.1.1 Overview

BFS is a uniform search used for searching trees, graphing data structures, or traversing. The algorithm expands the shallowest unexpanded node before moving to the next depth. In other words, all the nodes in the same depth are explored before exploding to the next layer.

3.1.2 Properties

Since the shallowest unexpanded nodes are always expanded, a large number of cells in the grid will be put in the queue, and the visited cells. Thus, if b is finite, the search will be completed with the optimal path assuming $\text{cost} = 1$ in each step. The time complexity is $O(b^{d+1})$ and the space complexity for keeping every node in the memory is $O(b^d)$ (Russell and Norvig, 2010).

3.2 Depth-First search (DFS)

3.2.1 Overview

DFS is also a uniform search that has the same application as BFS. The algorithm starts at the root node and explores as deep as possible in each branch. After that, it backtracks to continue exploring the next remaining branch. In particular, the search always explores until the last layer of a branch before moving to the next one.

3.2.2 Properties

This approach makes the time complexity of the algorithm more efficient than BFS as it only needs to store nodes and keep track of one branch at one time. However, the finding path will not always be optimal because the algorithm concentrates on exploring the nodes to the end of the branch. Additionally, the search fails in infinite-depth and iterative spaces. The time complexity of it is $O(b^m)$, which is a disaster if m is much larger than d. The space complexity is $O(b * m)$ (Russell and Norvig, 2010).

3.3 Greedy Best-First Search (GBFS)

3.3.1 Overview

GBFS is an informed search used for tree search, graph data structure, or traversing. Unlike BFS and DFS, the search expands the node base on its evaluation function

$f(n) = h(n)$ where h(n) is the estimated cost of the cheapest path from node n to goal. The function constructs the heuristic to sort the node to choose which is the promising node for exploring. Therefore, the algorithm can improve the performance since it does not expand unnecessary nodes.

3.3.2 Properties

The algorithm is efficient and could give significant improvement if a good heuristic function is defined. Nonetheless, the disadvantage of the search is that it does not always complete which is a major drawback. Also, it does not always find the optimal path by not considering the cost of the current path. The time complexity is $O(b^m)$ and the space complexity is $O(b^m)$ (Russell and Norvig, 2010).

3.4 A* Algorithm

3.4.1 Overview

A* plays a pivotal role in searching and is the best-known form of BFS. It avoids expanding paths that are already expensive by using an evaluation function called $f(n)$. This is formulated as $f(n) = g(n) + h(n)$. The cost of each node is the sum of the cost so far to reach the node ($g(n)$) and the estimated cost to get from the node to the goal ($h(n)$). Afterward, the algorithm chooses the node with the lowest cost to explore.

3.4.2 Properties

This approach ensures that the search is optimistic if the heuristic function is admissible. In addition, unless there are infinitely many nodes n with $f(n) \leq f(G)$, the search will be completed. The time complexity is exponential with path length. The space complexity is all stored nodes.

3.5 Bidirectional Algorithm

3.5.1 Overview

The bidirectional algorithm is a modified version of BFS that searches from both the start node and goal nodes simultaneously. Two separate queues are used to store the path to reach the intersection from the start node and goal nodes (GeeksforGeeks, n.d.). When the intersection is defined, the algorithm will be terminated. This search could be faster than the original BFS in many cases.

3.5.2 Properties

This algorithm is a custom code 1 of the program. It is guaranteed to find an existing solution provided when both start and goal nodes meet each other and are connected by a path. It is not optimal as there would be no shortest path solution as it often finds a near-optimal solution that is faster to compute than the optimal ones. The time complexity is $O(b^{\left(\frac{d}{2}\right)})$, which is much faster than the original BFS. The space complexity is $O(b^d)$ (GeeksforGeeks, n.d.).

3.6 Bidirectional A* Algorithm

3.6.1 Overview

This algorithm is a custom code 2 of the program. It works by running two simultaneous searches: forward (from start to goal) and backward (from goal to start) like the bidirectional algorithm above. However, instead of stopping at the middle of the start and goal nodes (when the intersection is defined), the Bidirectional A* uses a more complicated termination condition. According to Pohl (1969) and Whangbo (2007), the meeting in the middle does not guarantee an optimal proposed path. Therefore, a new modified bidirectional A* algorithm is applied to deal with that problem. This will be carefully demonstrated in the implementation part of this algorithm.

3.6.2 Properties

The algorithm is completed and optimal if the heuristic function used is admissible and consistent. The time and space complexity depends deeply on how good the heuristic function is. If a suitable function is implemented, the complexity will be reduced remarkably.

3.7 Comparison

The table below is tested and run through a specific generated test case of the author. From the table, each algorithm's time and number of nodes are illustrated and compared to get a better evaluation.

Table 1. The comparison table of six algorithms

Criterion	BFS	DFS	A*	GBFS	Bidirectional	Bidirectional A*
Time (ms)	1.998	0.9955	0.9972	0.5439	1.435	0.952
Number of nodes	102	123	84	61	109	79

4. Implementation

4.1 Search map

The whole map is abstracted using the class “Cell” which is shown in Figure 3. The start, goal, wall, and blank cells are indexed 2, 3, 1, and 0 respectively (represented in Figure 4). This can be done through function to set it become wall, goal or initial position.

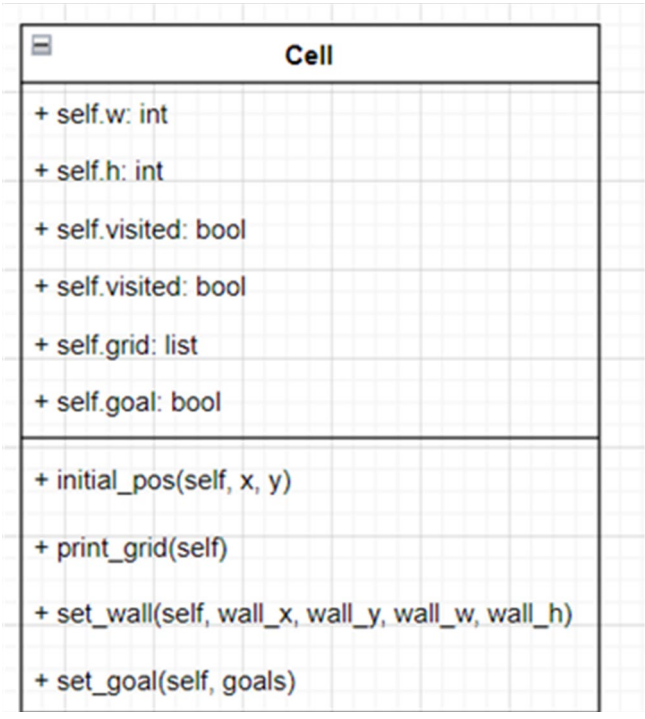

Figure 3. Cell diagram

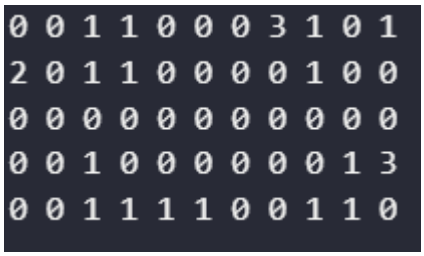

Figure 4. Printed map of RobotNav-test in terminal

4.2 Breadth-First search (BFS)

In the problem, BFS is implemented using the list "queue" in Python. The queue has a "first in first out" (FIFO) structure; therefore, it allows the function to pop the most recently added node for expansion. The "while" loop begins by adding the current node for exploring. It then takes the first node of the list and removes it from the queue. All neighbor nodes are appended at the end of the list so each layer can be visited before traveling to the next one. The image below gives a better understanding of the algorithm:

```
procedure bfs(cell):
    INSTRUCTION = {"up":(0, -1), "left":(-1, 0), "down":(0, 1), "right":(1, 0)}
    queue = []
    queue.append((initial_x, initial_y))
    father = [[None for y in range(cell.h)] for x in range(cell.w)]
    visit = [[False for y in range(cell.h)] for x in range(cell.w)]
    visit[initial_x][initial_y] = True

    while queue is not empty:
        current = queue.pop(0)
        if cell.grid[current[0]][current[1]] == 3:
            return result
        end if
        for instruction in INSTRUCTION:
            new_x = current[0] + INSTRUCTION[instruction][0]
            new_y = current[1] + INSTRUCTION[instruction][1]
            if 0 <= new_x < cell.w and 0 <= new_y < cell.h and cell.grid[new_x][new_y] != 1 and not visit[new_x][new_y]:
                queue.append((new_x, new_y))
                visit[new_x][new_y] = True
                father[new_x][new_y] = instruction
            end if
        end for
    end while
    return "No solution found."
end procedure
```

Figure 5. BFS Pseudocode

4.3 Depth-First search (DFS)

The DFS follows a nearly similar flow to BFS but uses a stack instead of a queue (Figure 6). The order of added node is different when this algorithm use "last in first out" (LIFO). This means that the last nodes in the stack are added and iterate backward.

```
procedure dfs(cell):
    INSTRUCTION = {"right":(1, 0), "down":(0, 1), "left":(-1, 0), "up":(0, -1)}
    stack = []
    stack.append((initial_x, initial_y))
    father = create 2D array with dimensions cell.w x cell.h, initialized to None
    visit = create 2D array with dimensions cell.w x cell.h, initialized to False

    while stack is not empty:
        current = stack.pop()
        visit[current[0]][current[1]] = True
        if cell.grid[current[0]][current[1]] == 3:
            return result
        end if
        for instruction in INSTRUCTION:
            new_x = current[0] + INSTRUCTION[instruction][0]
            new_y = current[1] + INSTRUCTION[instruction][1]
            if 0 <= new_x < cell.w and 0 <= new_y < cell.h and cell.grid[new_x][new_y] != 1 and not visit[new_x][new_y]:
                stack.append((new_x, new_y))
                father[new_x][new_y] = instruction
            end if
        end for
    end while
    return "No solution found."
end procedure
```

Figure 6. DFS Pseudocode

4.4 Greedy Best-First Search (GBFS)

The frontier of GBFS is Priority Queue allowing it to decrease sort based on the priority cost. The heuristic function calculates the distance between the node and the goal cells. The algorithm chooses the node with the closest distance to the goal cell to expand.

```
procedure greedy_best_first_search(cell):
    INSTRUCTION = {"up":(0, -1), "left":(-1, 0), "down":(0, 1), "right":(1, 0)}
    frontier = PriorityQueue()
    frontier.put((0, -1, (initial_x, initial_y)))
    father = 2D array with null values for each cell
    visit = 2D array with false values for each cell

    while frontier is not empty:
        current_tuple = get the tuple with the smallest heuristic value from frontier
        current = get the third element of the tuple
        visit[current[0]][current[1]] = true
        if cell.grid[current[0]][current[1]] is equal to 3:
            return result
        end if
        for index, instruction in enumerate(INSTRUCTION):
            new_x = current[0] + INSTRUCTION[instruction][0]
            new_y = current[1] + INSTRUCTION[instruction][1]
            if new_x is between 0 and cell.w and new_y is between 0 and cell.h and cell.grid[new_x][new_y] is not equal to 1 and visit[new_x][new_y] is false:
                frontier.put((heuristic((new_x, new_y), goals), index, (new_x, new_y)))
                visit[new_x][new_y] = true
                father[new_x][new_y] = instruction
            end if
        end for
    end while
    return "No solution found."
end procedure
```

Figure 7. GBFS Pseudocode

4.5 A* Algorithm

The A* is nearly like GBFS except for the evaluation function. The function calculates the sum of the Manhattan distance to goal nodes and the cost so far from the origin to that node. The flow of the code is shown in Figure 8:

```
procedure astar(cell):
    count_node = 1
    INSTRUCTION = {"up":(0, -1), "left":(-1, 0), "down":(0, 1), "right":(1, 0)}
    frontier = PriorityQueue()
    frontier.put((0,count_node,(initial_x, initial_y)))
    father = 2D array with null values for each cell
    visit = 2D array with false values for each cell
    cost_so_far = {}
    cost_so_far[(initial_x, initial_y)] = 0

    while frontier is not empty do
        current_tuple = frontier.get()
        current = current_tuple[2]
        visit[current[0]][current[1]] = True
        if cell.grid[current[0]][current[1]] equals 3 then
            return result
        end if
        for instruction in INSTRUCTION do
            new_x = current[0] + INSTRUCTION[instruction][0]
            new_y = current[1] + INSTRUCTION[instruction][1]
            if 0 <= new_x < cell.w and 0 <= new_y < cell.h and cell.grid[new_x][new_y] not equals 1 and not visit[new_x][new_y] then
                new_cost = cost_so_far[current] + 1
                if (new_x, new_y) not in cost_so_far or new_cost < cost_so_far[(new_x, new_y)] then
                    cost_so_far[(new_x, new_y)] = new_cost
                    priority = new_cost + heuristic((new_x, new_y), goals)
                    count_node = count_node + 1
                    frontier.put((priority, count_node, (new_x, new_y)))
                    visit[new_x][new_y] = True
                    father[new_x][new_y] = instruction
                end if
            end if
        end for
    end while
    return "No solution found."
end procedure
```

Figure 8. A Pseudocode*

4.6 Bidirectional Algorithm

The application is mostly similar to the original BFS. Nevertheless, there are two queues to search forward and backward and the intersection variable for terminating condition.

```
procedure bidirectional(cell, draw_gui = None):
    start_father = 2D array with null values for each cell
    goal_father = 2D array with null values for each cell
    start_queue = []
    start_visit = create_2d_array(cell.w, cell.h, False)
    goal_queue = []
    for goal in goals:
        goal_queue.append(goal)
    goal_visit = 2D array with false values for each cell
    for goal in goals:
        goal_visit[goal[0]][goal[1]] = True
    intersection = False
    while not intersection and len(start_queue) != 0 and len(goal_queue) != 0:
        start_current = start_queue.pop(0)
        goal_current = goal_queue.pop(0)
        if goal_visit[start_current[0]][start_current[1]]:
            intersection = True
            if draw_gui is not None:
                return print_bi_path(start_father, goal_father, (initial_x, initial_y), start_current, grid)
            else:
                return print_bi_path(start_father, goal_father, (initial_x, initial_y), start_current)
            end if
        end if
        if start_visit[goal_current[0]][goal_current[1]]:
            intersection = True
            if draw_gui is not None:
                return print_bi_path(start_father, goal_father, (initial_x, initial_y), goal_current, grid)
            else:
                return print_bi_path(start_father, goal_father, (initial_x, initial_y), goal_current)
            end if
        end if
```

```
    for start_instruction in INSTRUCTION:
        new_start_x = start_current[0] + INSTRUCTION[start_instruction][0]
        new_start_y = start_current[1] + INSTRUCTION[start_instruction][1]
        if 0 <= new_start_x < cell.w and 0 <= new_start_y < cell.h and cell.grid[new_start_x][new_start_y] != 1 and not start_visit[new_start_x][new_start_y]:
            start_queue.append((new_start_x, new_start_y))
            start_visit[new_start_x][new_start_y] = True
            start_father[new_start_x][new_start_y] = start_instruction
        end if
    end for
    for instruction in GOAL_INSTRUCTION:
        new_goal_x = goal_current[0] + GOAL_INSTRUCTION[instruction][0]
        new_goal_y = goal_current[1] + GOAL_INSTRUCTION[instruction][1]
        if 0 <= new_goal_x < cell.w and 0 <= new_goal_y < cell.h and cell.grid[new_goal_x][new_goal_y] != 1 and not goal_visit[new_goal_x][new_goal_y]:
            goal_queue.append((new_goal_x, new_goal_y))
            goal_visit[new_goal_x][new_goal_y] = True
            goal_father[new_goal_x][new_goal_y] = instruction
        end if
    end for
end while
return "No solution found."
```

Figure 9. Bidirectional Pseudocode

4.7 Bidirectional A* Algorithm

The algorithm is implemented following the bidirectional flow. The main difference is the code uses an evaluation function to choose the priority node and the stopping condition. As the bidirectional algorithm and A* algorithm show in detail above, in this section, the author only discusses the stopping condition implementation. The variable "min_value" is created to be infinite. This variable is updated to the minimum of its, path cost of start and path cost of goal when the start-current node is in the goal-visited two-dimension array. The terminated condition is when the variable is less than or equal to the maximum of the start evaluation function and goal evaluation function. This condition ensures the algorithm terminates when it is unlikely that a shorter path can be found. By comparing the maximum of two evaluation functions with the threshold "min_value", the search can decide whether it should continue searching on that branch or not.

5. Features/Bugs/Missing

5.1 Features

Some highlighted features of the program are:

- GUI interface can be used to visualize the progress of the start node reaching the goal. With different colors, the queue and visited nodes are both shown to support the user in having a better understanding of how the algorithm runs.
- Three informed search and three uniform search algorithms are implemented and pass the assignment's requirement sample test and more than 20 other test cases are generated.
- The command line operates the main file, input file, and search method.
- The stopping condition of bidirectional A*

5.2 Bugs

When implementing this program, bugs are inevitable despite the effort of the author to deal with them. For the bidirectional algorithm, when calculating, the frontier is pop continuously through all available goals. Therefore, in some cases, the finding path is not the shortest one, and following the instruction orders of the requirements is not

optimal. For example, in Figure 10 for the case of 3 goals, the bidirectional algorithm finds a different path compared with the bidirectional A* algorithm shown in Figure 11 (which is the correct one according to the requirement).

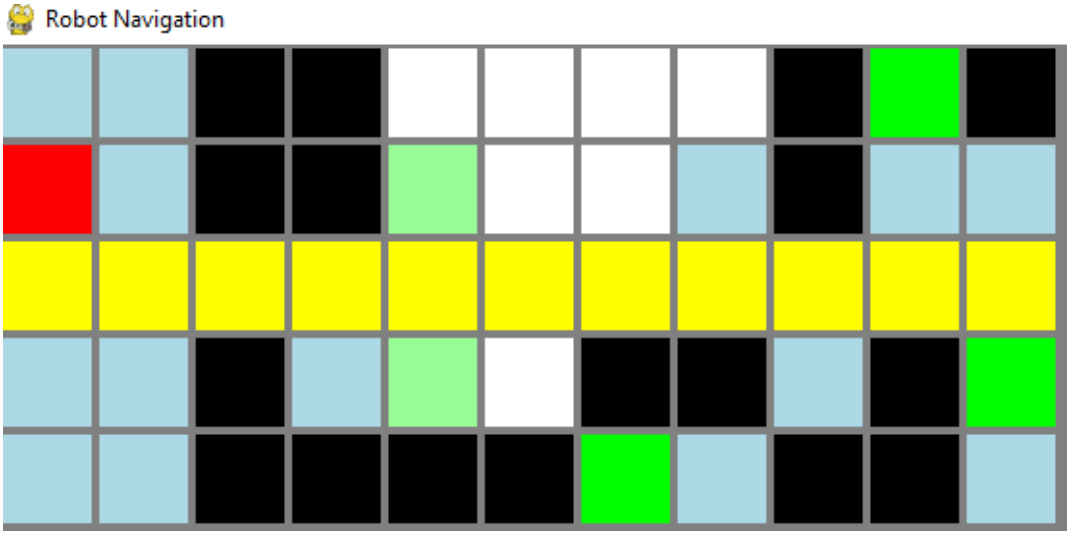

Figure 10. Bidirectional GUI for specific test case

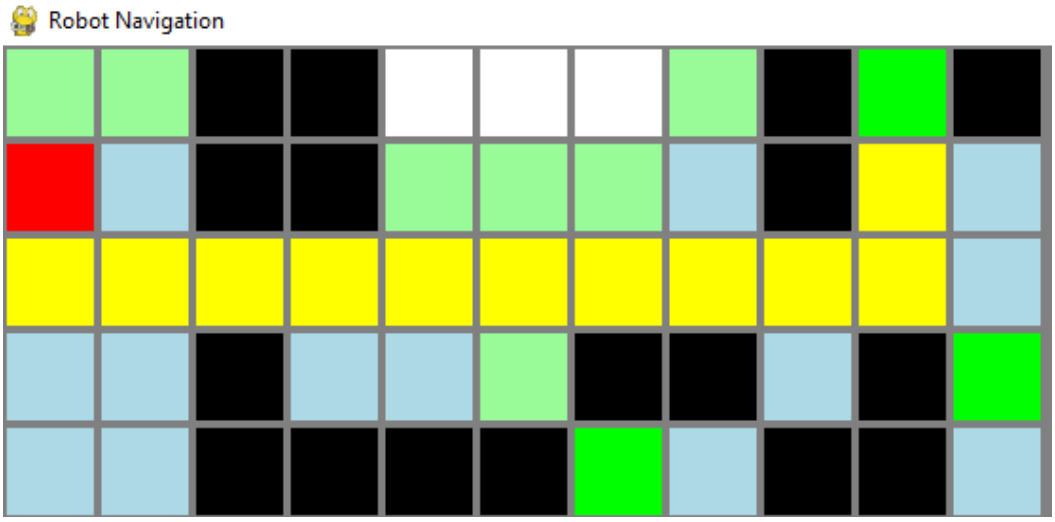

Figure 11. Bidirectional A GUI for specific test case*

5.3 Missing

The program GUI cannot visualize large grid dimensions which are not so flexible. Also, it misses the error handling for the input in case users want to test the robustness of the program while reading the input file and command. In addition, the application has not covered the map for the user to detect manually.

6. Research

To gain a closer insight into an algorithm and debug it, a GUI can be a useful tool. The GUI can provide a visual representation of the algorithm's behavior, allowing the user to see what is happening at each step of the process. This can help to identify any errors or issues that may be occurring in many test cases. To create the GUI, a pygame library

of Python is called to draw the screen consisting of every item. Additionally, the author abstracts each cell to the class CellGUI (Figure 12). This has the following fields:

- self.x: Horizontal coordinate of the cell in the grid
- self.y: Vertical coordinate of the cell in the grid
- self.width: The width of each square cell
- self.color: The drawing color of each cell. Functions change the color of the cell depending on its feature: visited, queue, wall, goal, start, and path.

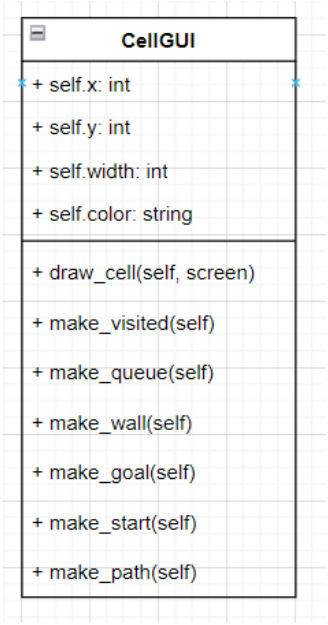

Figure 12. CellGUI

7. Conclusion

To sum up, the report has covered a comprehensive overview of 6 search algorithms used to solve the "Robot Navigation" problem. It has provided the key features of the program with several bugs and missing features. Furthermore, the report has illustrated the research initiative towards creating the GUI to visualize the map and better tracking as well as testing for each algorithm. Among all applied algorithms, A* search demonstrates its merits in terms of speed, efficiency, and optimization. To enhance the performance, multiple threads should be used in the program. This would act as a precursor to reducing the processing time significantly since multiple-core processors could work at the same time. Therefore, when the environment is much bigger, this idea could give more optimal results since the program can explore the different branches of the search tree simultaneously.

8. Acknowledgements/Resources

- The author wants to express deep gratitude toward the lecture and tutorial of the Introduction to AI unit which widens the knowledge regarding search algorithms.

- The report is inspired by Russell, S.J. and Norvig, P through useful material about search algorithms provided in the "Artificial Intelligence: A Modern Approach" 3rd edition book.
- PyGame library: The main library for building the GUI application and helping the user to interact with the screen through a physical click.

9. References

Russell, S., & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd ed.). Pearson.

Pohl, I. (1969). *Bi-directional and heuristic search in path problems* (No. SLAC-104). Stanford Linear Accelerator Center, Calif.

GeeksforGeeks. (n.d.). *Bidirectional Search.* https://www.geeksforgeeks.org/bidirectional-search/

Whangbo, TK. (2007). *Efficient Modified Bidirectional A * Algorithm for Optimal Route-Finding*. https://doi.org/10.1007/978-3-540-73325-6_34