1. INTRODUCTION

1.1. Robot Navigation

Robot navigation Problem is a goal-based agent scenario, where the Robot must navigate throughout cells in a two-dimensional grid to find the solution from the starting point to one or more goals without moving outside the grid boundaries. Because of the requirements, the robot, in this case, can only move up, left, down, and right, and will follow this order when all options are equal. This reports aim to summarize my understanding of this problem, and present some of the algorithms used in solving it, as well as describe how I implemented it with a graphical user interface (GUI) and additional features

1.2. Terminology

Terminology	Definition
State	The robot's position is represented in a 2D tuple (x, y), with x being the column and y being the row.
State space	All the available states
Successors/Neighbors	All the possible movements of the robot at a specific state.
Graph	A set of nodes (also called vertices) connected by edges; the edges can either have a direction or not, representing the possible transitions and relationships between the nodes.
Tree	Tree is a special type of graph with no cycles, and a tree must have a root node. Other nodes in the tree are spanned out from the root, and every node has exactly one parent.
Node	Node is a point in graphs or trees, representing a unique state.
Action	The process of moving from one node to another node or from the current state to another state.
Frontier	A list to store the neighbors, but it has not been explored yet.
Cost	The cost of moving from one node to another node.
Heuristic function	A heuristic function is a method for estimating the cost

Terminology	Definition
	from a given node to the goal in search algorithms.
Admissible	A heuristic is admissible when it never overestimates the true cost to reach the goals.
Informed search	Search algorithms that use heuristic functions.
Uninformed search	Search algorithms that do not use heuristic functions.
Complete	The path is guaranteed to be found if it exists.
Optimal	The shortest path is guaranteed to be found if it exists.

2. INSTRUCTIONS

2.1. Command-line Interface (CLI) instructions

To run the searching algorithms in the CLI, follow the given format:

Example:

```
python search.py test/RobotNav-test.text gbfs
```

2.2. Graphical User Interface (GUI) instructions

The GUI is available at https://maze-searching-visualizer.vercel.app/. There are four main pages in this GUI, including the Dashboard, Guidelines, Report, and ReadMe. Each of them will be responsible for specific purposes, aiming to enhance the experience of the users and alleviate the efforts needed for the lecturer to grade this assignment.

a. Guidelines Page

When users access the GUI, the **Guidelines** page is displayed first. This page provides essential information about the website, along with clear instructions on how to use its features as shown in Figure 1.
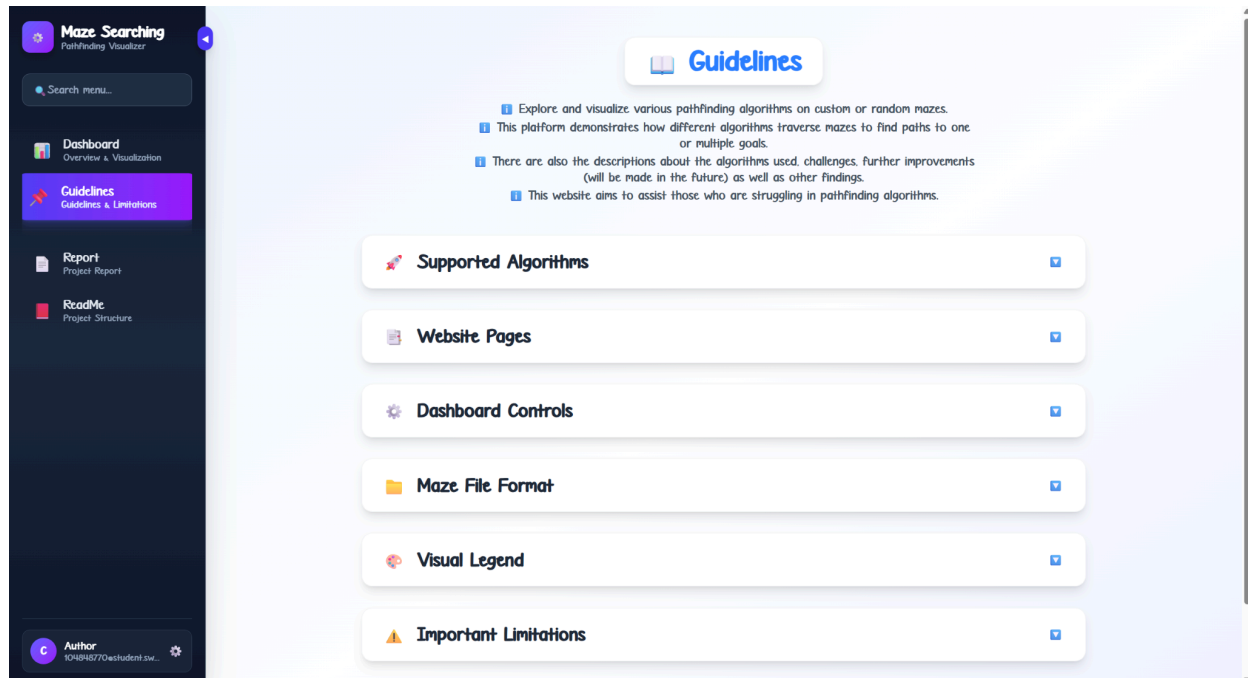
Figure 1: Guidelines Page

b. Dashboard Page

The visual representations of the maze searching algorithms are shown in the Dashboard Pages. The component on the rightmost part of the page allows the users to configure the maze (including uploading the maze .txt file, changing the maze dimension, switching between the search algorithms, and setting the speed for the visualization). The maze component is drawn in the middle of the page, for visualization, while Figure 3 defines the symbols used in the maze and their definitions.
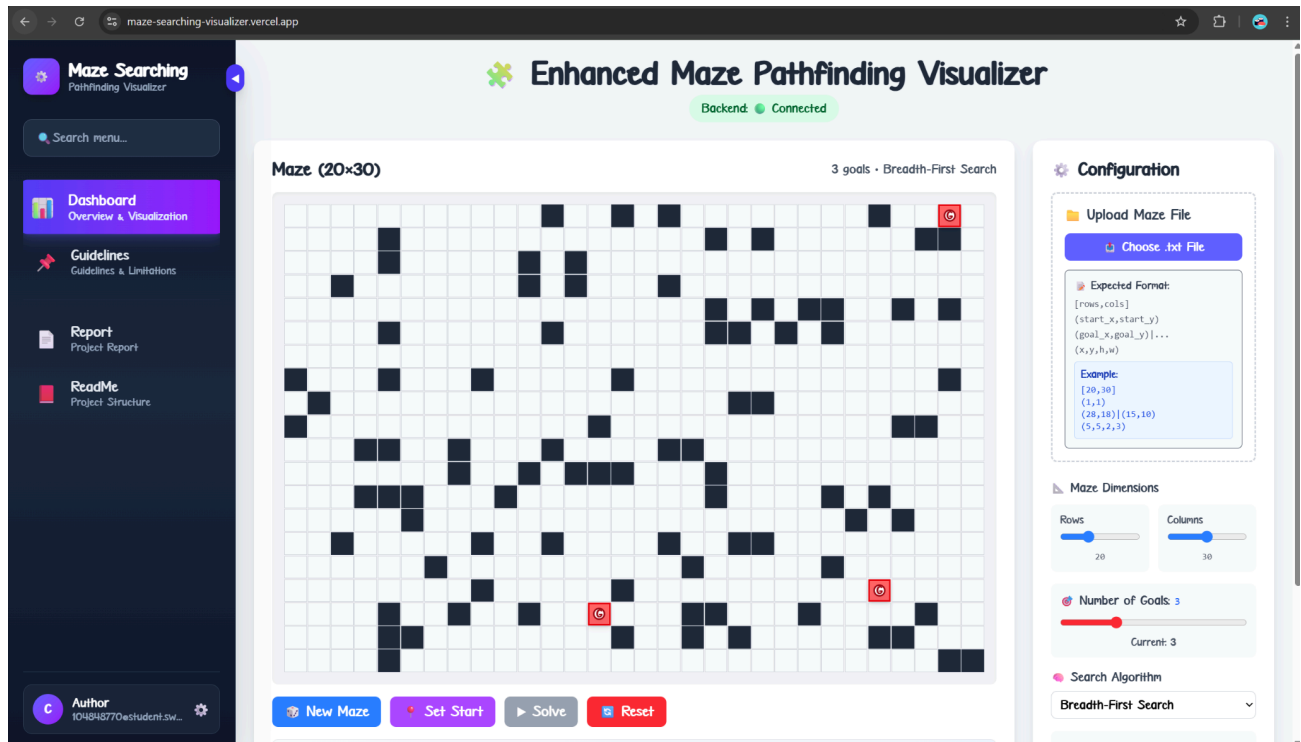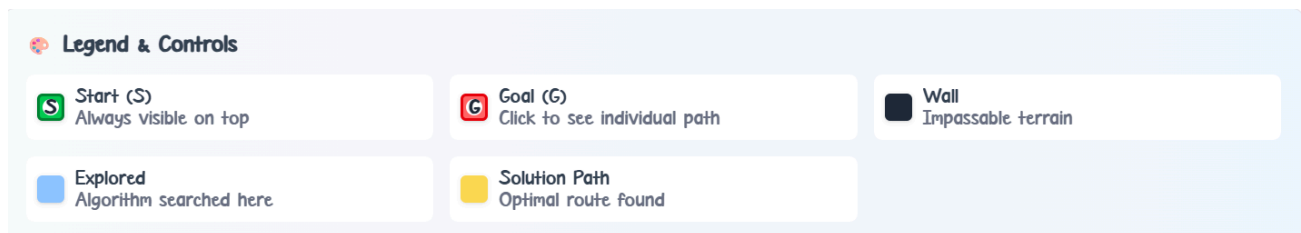
Figure 2: Dashboard Page

Figure 3: Maze symbol explanations

c. Report Page

The report page will show this report, so if the users want to know more about the work, they can read it. Moreover, it also assists the lecturer(s) in marking the scores because they can read the report directly on the GUI without needing to download it. However, the .pdf version is still attached to this page and will be submitted via Canvas.

d. ReadMe Page

This page provides all the necessary information about the project structure, including a short description about the project, the algorithms, technology used, the code structures, the implementation guide, the usage guide, as well as providing the link to GitHub, Live Demonstration, and the downloading of the codes (Figure 4).

Pathfinding Visualizer

Interactive maze solving with multiple algorithms

- 🔱 View on GitHub
- ⧉ Live Demo
- ⤓ Download ZIP

Repository	Language	Version
COS30019	JavaScript/React/Python	v1.0.0

📋 Project Description

A comprehensive web application for visualizing pathfinding algorithms in maze environments. Users can upload custom mazes or generate random ones, then watch as different algorithms find paths from start to goal positions.

This project demonstrates the behavior and efficiency of various pathfinding algorithms including BFS, DFS, Greedy Best-First, A*, Backtracking ,and iterative deepening variants. Perfect for educational purposes and algorithm comparison.

✨ Key Features

- Interactive maze visualization
- Custom maze file upload
- Multiple goal support
- 8 different pathfinding algorithms
- Real-time algorithm animation
- Responsive design

🔧 Technology Stack

- **React** – Frontend framework
- **Lucide React** – Icon library
- **Tailwind CSS** – Styling framework
- **JavaScript** – Programming language

🚀 Quick Start

Clone the repository:

```
git clone https://github.com/Trchien123/COS30019.git
```

Install dependencies:

```
FRONTEND

cd COS30019
cd frontend
npm install
```

```
BACKEND

cd COS30019
cd backend
npm install
```

Start development server:

```
FRONTEND

npm run dev
```

```
BACKEND

uvicorn server:app --reload --port 5000
```

Build for production:

```
npm run build
```
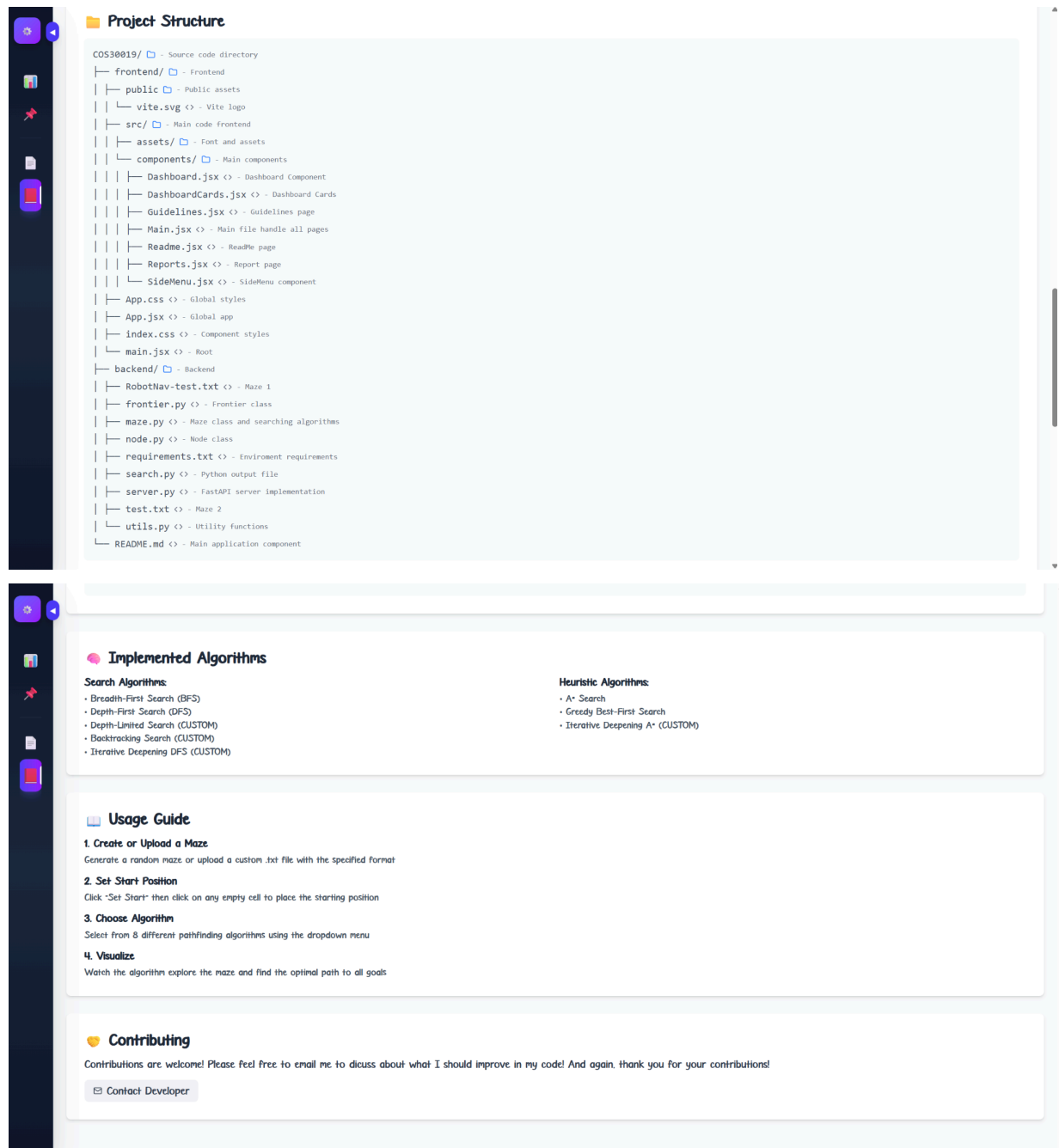
Figure 4: ReadMe Page

3. SEARCH ALGORITHMS

8 algorithms are being implemented, and the ideas primarily came from the textbook (Artificial Intelligence: A Modern Approach, 4th Global ed. by Stuart Russell and Peter Norvig). The next subsections will discuss the algorithms and measure their performance based on these 4 criteria:

- Completeness: Whether the solution can be found if it exists or not?
- Optimality: Whether the algorithm finds the shortest path or not?
- Time complexity: How much time does the algorithm take to find the solution?

- Space complexity: How much memory is needed to find the solution?

The other terminology used in this section is:
- b: branching factor (how many children each node can have)
- d: depth of the shallowest solution
- m: maximum depth of the state space

3.1. Breadth-First Search

Breadth-first search works by exploring the root node first, then all the neighbors of the root node are expanded, then their neighbors, and so on. In other words, the agent in this case will expand the shallowest nodes first by using a First In First Out (FIFO) queue for the frontier. Compared with the first criterion, this algorithm is complete, which means that it will always find the solution when one exists. However, this algorithm is not an optimal one when the path costs between the nodes are not the same. Therefore, technically, this algorithm is optimal when all actions have the same cost. In terms of the third criterion, given that there is a tree, where the first level is the root node, the second level contains b nodes, and each node in the second level contains b nodes, the time complexity when the goal is at depth d is $O(b^d)$. However, when the algorithm is designed to check for the goal before expanding, not to check for the goal before adding to the frontier, the time complexity is $O(b^{d+1})$ because the algorithm may generate and store all nodes up to depth $d + 1$ before finding the goal. As for space complexity, this search algorithm is applied as a graph search, where the explored set will be stored in an explored set ($O(b^{d-1})$ nodes) and $O(b^d)$ nodes in the frontier. Therefore, the space complexity is $O(b^d)$.

As shown in Figure 5, it can be noticed that one of the limitations of this algorithm is that if the goal is far from the starting point, it will explore almost all the nodes (including the redundant and not close to the goal at all). However, it found the optimal path in this case, and in other cases when the path cost is the same between every node.

Figure 5: Breadth-first Search test

3.2. Depth-First Search

Opposite to Breadth-first Search, Depth-first Search always expands the deepest node in the current frontier first. Therefore, instead of using the FIFO queue like Breadth-first Search, this type of algorithm will use the Last In First Out (LIFO) queue. In other words, the node, which is the most recently added to the frontier, is chosen for expansion.

Regarding the first criterion - completeness, the chosen between using the graph-search or tree-search version. For the graph-search version with finite state spaces, this search algorithm is complete because it will expand the deepest path and go back to expand the others until there are no available paths. However, for a tree-graph version with cyclic state spaces, this algorithm can get stuck in an infinite cyclic loop, preventing it from reaching the goal. Moreover, when there are infinite state spaces, both the depth-first search versions are complete. Based on its properties, this search algorithm is not optimal (as shown in Figure 6 when compared to Breadth-first Search). Because Depth-first Search explores to the deepest node, its time complexity is $O(b^m)$, which is much larger than Breadth-first Search (m can be a lot larger than d). Despite all the disadvantages of DFS, there is one major benefit when compared to BFS, which is the space complexity. For the graph-search version, there is no difference, but for the tree-search, the space complexity is only $O(bm)$, which is a lot better than BFS.

Figure 6 below shows that Depth-first Search may not find the optimal path, but it is sometimes a lot better when we consider the number of explored nodes (the sum of blue and yellow).

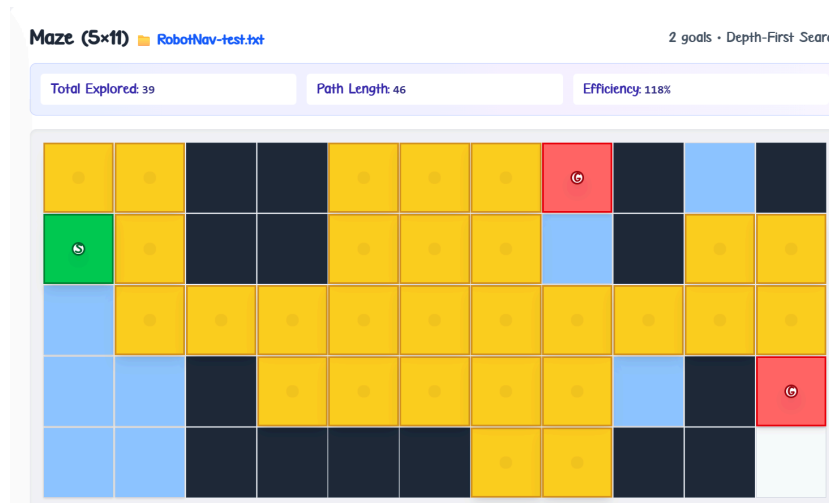*Figure 6: Depth-first Search test*

3.3. Backtracking Search (Custom 1)

Backtracking is another version of Depth-First Search, and it even uses less memory. This is because this type of search will generate one neighbor instead of all possible neighbors. During the searching process, each partially expanded node remembers which neighbor to generate. Thus, the space complexity is only b and not bxm.

In terms of the measuring criteria, this search type behaves the same as Depth-first Search, and it is only different in the space complexity. We can see that the solutions for Depth-first Search and Backtracking are the same.
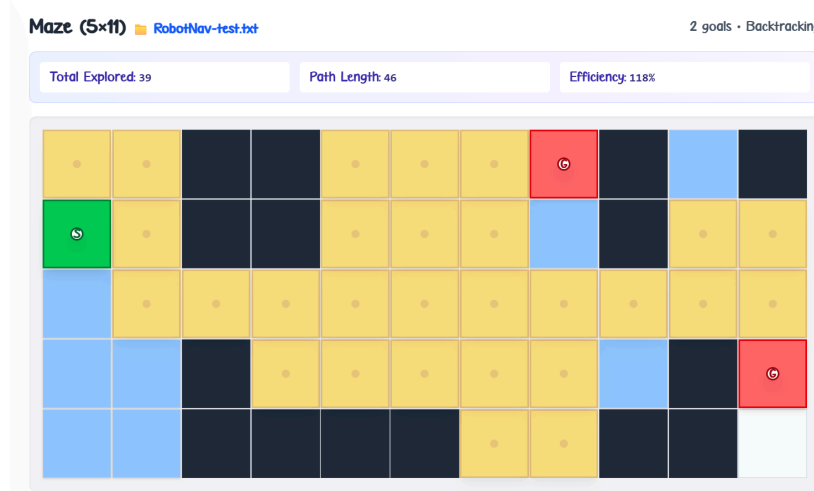
Figure 7: Backtrackingt Search test

3.4. Depth-Limited Search (Custom 2)

Depth-limited Search is developed to alleviate the failure of Depth-first Search in infinite state spaces. As mentioned before, when the state spaces are infinite, Depth-first Search will explore a path forever without finding the deepest path. However, with Depth-limited Search, a limited l, with $l < d$, is added into the algorithm as the threshold, so the algorithm will backtrack when it reaches the threshold. Moreover, Depth-limited is sometimes not optimal when $l > d$ is chosen. Depth-first search can also be considered as a special case of Depth-limited search where $l = \infty$.

The performance of Depth-Limited Search is similar to Depth-First Search in terms of time complexity and optimality. However, their completeness and space complexity differ: Depth-Limited Search is only complete if the depth limit is greater than or equal to the goal depth, while DFS is not complete in infinite-depth or cyclic spaces, while the space complexity of Depth-Limited is $O(bl)$.
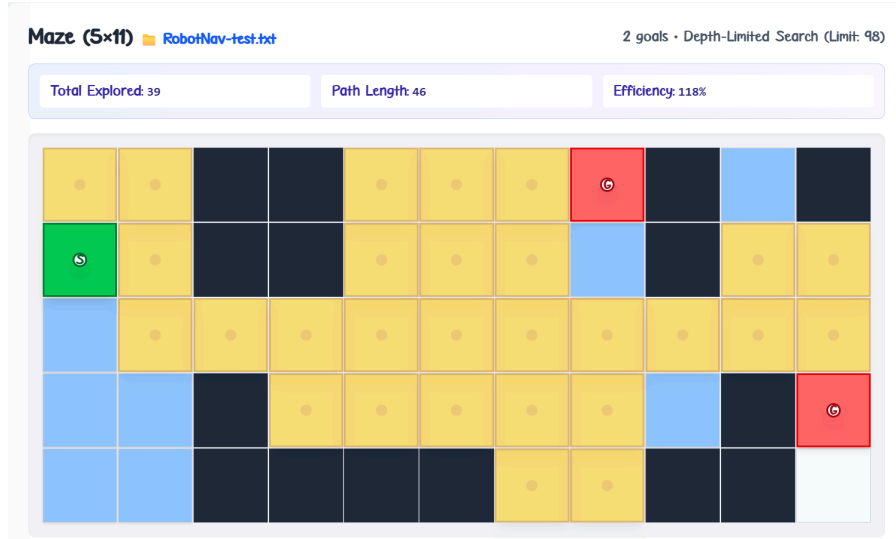
Figure 8: Depth-Limited Search

3.5. Iterative Deepening Depth-First Search (Custom 3)

Iterative Deepening Depth-first Search is a combination of the Depth-first Search and Breadth-first Search. This searching algorithm finds the best depth limit for Depth-first Search by gradually increasing the depth limit (0, 1, 2,...), and it does this until it reaches the depth of the shallowest solution d.

Because it is the combined version of DFS and BFS, its memory requirements are $O(bd)$ (the same as Depth-First Search), while it is complete when the branching factor is finite and optimal with the path cost is equal between 2 nodes. However, the time complexity of it is very high $O(b^d)$, consuming a lot of time to find out the answers for a large maze with multiple goals. Therefore, in the GUI, it is recommended to use a smaller maze for this algorithm with only 1 or 2 goals.

Figure 9: Iterative Deepening Depth-first Search test

3.6. Greedy Best-First Search

Unlike other uninformed searches described above, Greedy Best-first Search is an informed (heuristic) search strategy. In order to do that, it uses a heuristic function $h(n)$, which is the cheapest estimated cost from the current state to the goal. The Robot, in this case, will evaluate all the $h(n)$ of the current state in the frontier to choose the state with the shortest cost and continue. This search strategy relies on the usage of a Priority Queue, where the nodes in the frontier are sorted by $f(n) = h(n)$.
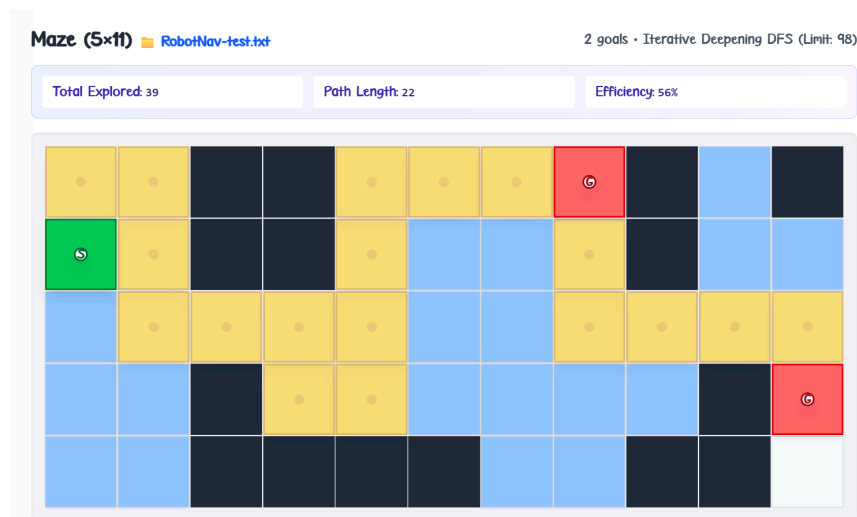
This searching strategy is not complete in infinite state spaces, and it sometimes does not fid the optimal path. The worst-case time and space complexity of Greedy Best-first Search is $O(b^m)$.

Figure 7 shows that Greedy Best-first Search explored fewer steps compared to all the uninformed searches above.
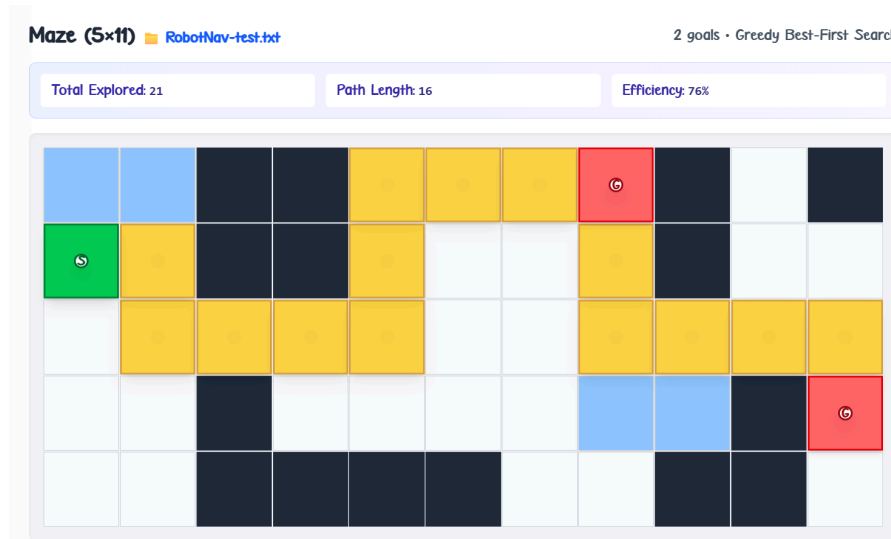
Figure 10: Greedy Best-first Search test

3.7. A* Search

One of the most common informed search algorithms is A* Search, which can be considered as an enhanced version of Greedy Best-first Search. Instead of using a cost function $f(n) = h(n)$, it uses $f(n) = h(n) + g(n)$, where the $g(n)$ is the path cost from the initial state to the current state and $h(n)$ is the cheapest estimated cost from the current state to the goal.

A* search is complete, and it is most of the time an optimal search algorithm. One special and most important property of A* is that it is admissible, meaning that it will never estimate the actual cost from the start to the goal. The time complexity of A* is $O(b^d)$, where b is the branching factor (average number of children per node) and d is the depth of the solution, while the space complexity of A* is also $O(b^d)$.

Figure 8 shows that A* explored more nodes than Greedy Best-first Search and Breadth-first Search, but it found the optimal solution.
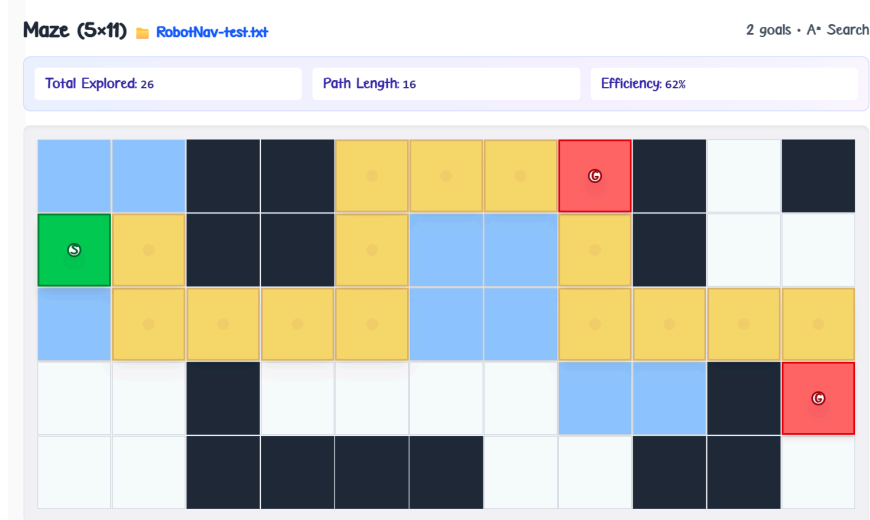
Figure 11: A* Search test

3.8. Iterative Deepening A* (Custom 4)

Iterative Deepening A* works quite similarly to Iterative Deepening Depth-first Search, and the only difference is that it uses the threshold as the cost function of A*. Moreover, IDA* provides the benefits of A* without the need to store all the visited nodes in the explored set, but with the cost of allowing visiting some states multiple times.

The time complexity of IDA* is generally $O(b^d)$, similar to A*, but with a smaller memory footprint. Specifically, its space complexity is typically $O(bd)$, where b is the branching factor and d is the depth of the solution. However, in real cases, IDA* takes a lot more time to run than A*. Therefore, in the GUI, it is recommended to set a grid with a small size and not too many goals.
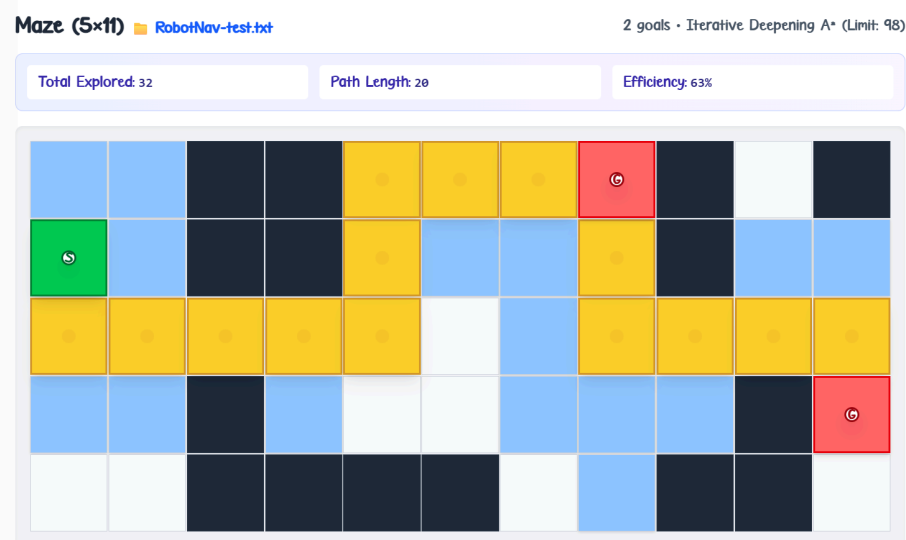
Figure 12: Iterative Deepening A* Search test

4. IMPLEMENTATION

4.1. UML Diagram

Figure 10 shows the UML class diagram of the implementation, and in this assignment, the classes are not really well-designed, so it looks quite complicated. This is currently one of the limitations of my work, and I will reimplement and fix this one in future work.
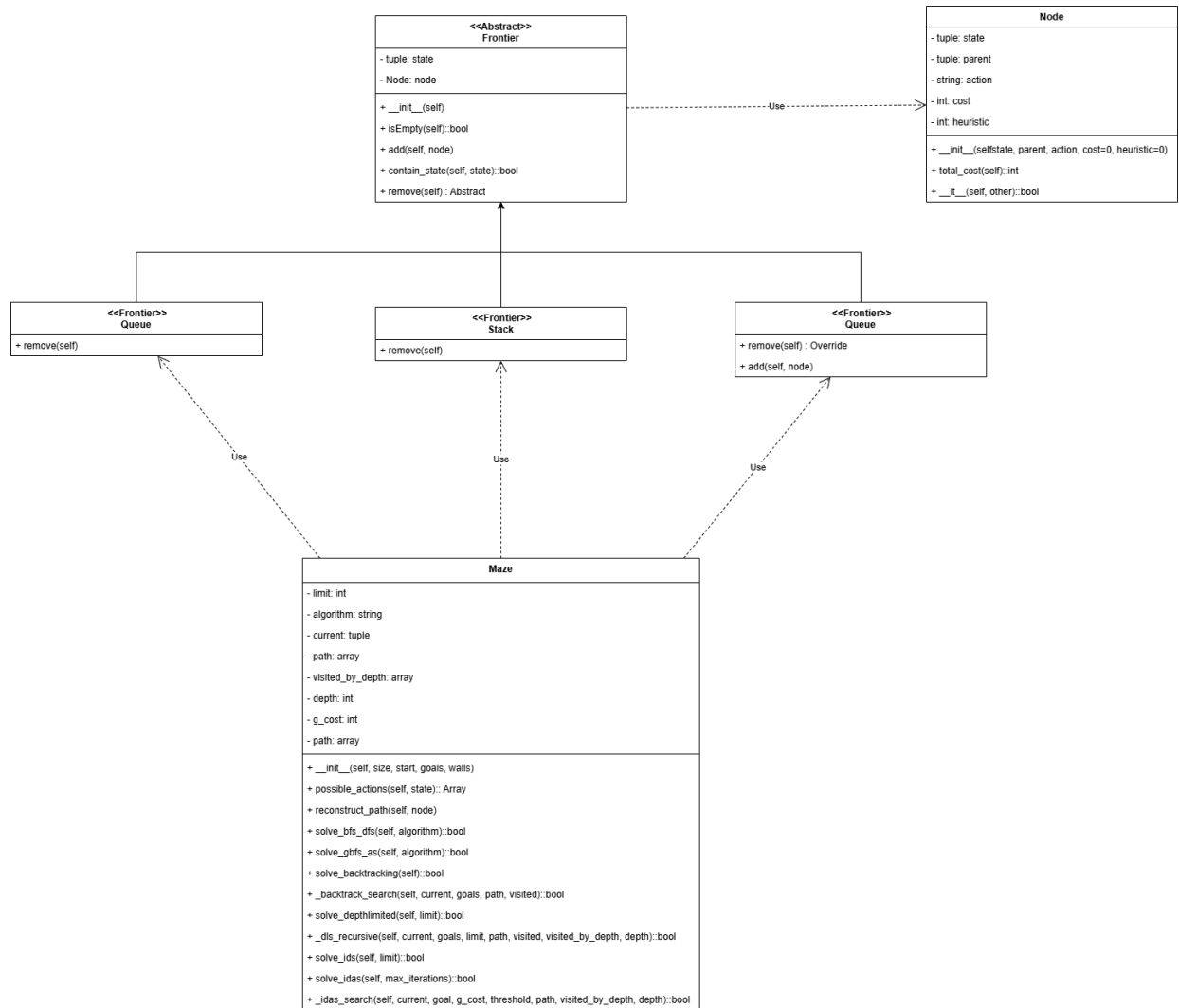
4.2. BFS and DFS pseudocode

```
Function solve_bfs_dfs(algorithm):
    Start timer
    Initialize all tracking variables (solutions, explored sets, counters)

    If algorithm is 'bfs':
        Use Queue as Frontier
    Else:
```

```
        Use Stack as Frontier

    Set current_start to initial start position
    Set remaining_goals to the list of goal positions
    Initialize list for found_goals

    While there are still goals to find:
        Create a new frontier (Queue or Stack)
        Add the start node to the frontier
        Reset explored set and counters for this goal

        goal_found = False

        While frontier is not empty:
            Remove node from frontier
            Mark node as explored
            Add node state to tracking lists

            If node state is one of the remaining goals:
                Mark goal as found and remove from remaining_goals
                Update current_start to this goal (for next search)
                Reconstruct and save the path
                Update single and multiple solution tracking
                goal_found = True
                Break

            For each (action, next_state) in possible actions:
                If next_state not in explored and not in frontier:
                    Create child node with next_state
                    Add child to the frontier

        If no goal was found in this loop:
            Stop timer
            Print "No Goals Found"
            Return False

    Stop timer
    Print tracking info (nodes explored, solutions, path lengths)
    Return True
```

4.3. GBFS and A* pseudocode

```
Function solve_gbfs_as(algorithm):
    Start timer
    Initialize all tracking variables (solutions, explored sets, counters)

    Set remaining_goals to list of all goal positions
    Set current_start to start position
    Set found_goals to empty list

    While remaining_goals is not empty:
```

```
        Reset explored set, counters, and node tracking
        Initialize empty priority frontier

            closest_goal ← goal in remaining_goals with smallest Manhattan distance from
current_start

        Create start_node with:
            - state = current_start
            - cost = 0
            - heuristic = distance to closest_goal
        Add start_node to frontier

        goal_found ← False

        While frontier is not empty:
            node ← remove node with lowest priority from frontier

            If node.state is already explored:
                Skip to next loop iteration

            Mark node.state as explored
            Track node.state in metrics

            If node.state == closest_goal:
                Mark goal as found
                Remove closest_goal from remaining_goals
                Update current_start to current_goal

                Reconstruct and save path
                Track explored and solution data
                goal_found ← True
                Break

            For each (action, next_state) in possible actions from node.state:
                If next_state is not in frontier and not explored:
                    heuristic ← Manhattan distance from next_state to closest_goal
                    cost ← node.cost + 1  if algorithm == 'as' else 0
                    Create child node with next_state, parent=node, action, cost, heuristic
                    Add child to frontier

    If goal_found is still False:
        Stop timer
        Print "No Goals Found"
        Return False

  Stop timer
  Return True
```

4.4. Backtracking Search pseudocode

```
'''--------------- solve_backtracking -------------------'''
Function solve_backtracking():
    Start timer
    Initialize tracking variables for:
        - solutions
        - explored nodes
        - path lengths

    Set current_start to the initial starting position
    Set remaining_goals to the list of all goals

    While there are still goals to find:
        path ← empty list
        _current_explored ← empty list

        found_goal ← None

        If _backtrack_search(current_start, remaining_goals, path, empty visited set):
            found_goal ← last node in path
            Remove found_goal from remaining_goals

            complete_path ← [current_start] + path
            Add complete_path to solution tracking
            Add _current_explored to explored node tracking
            Update counters and lengths

            Set current_start = found_goal
        Else:
            Stop timer
            Print "No Goals Found"
            Return False

    Stop timer
    Return True

'''--------------- _backtrack_search -------------------'''
Function _backtrack_search(current, goals, path, visited):
    Add current to _current_explored
    Add current to visited

    If current is in goals:
        Return True

    For each (action, next_state) in possible actions from current:
        If next_state is within bounds AND not a wall AND not visited:
            Add next_state to path
            If _backtrack_search(next_state, goals, path, visited):
                Return True
```

```
            Remove next_state from path (backtrack)

    Return False
```

4.5. Depth-Limited Search pseudocode

```
'''--------------- solve_depthlimited ------------------'''
Function solve_depthlimited(limit):
    Start timer
    Initialize all tracking variables (solutions, explored nodes, depth info)

    current_start ← start position
    remaining_goals ← list of all goal positions

    While remaining_goals is not empty:
        found ← False

        For each goal in remaining_goals:
            path ← empty list
            _current_explored ← empty list
            visited ← empty set
            visited_by_depth ← empty map

            (result, found_goal) ← _dls_recursive(
                current = current_start,
                goals = remaining_goals,
                limit = limit,
                path = path,
                visited = visited,
                visited_by_depth = visited_by_depth,
                depth = 0
            )

            If result == "found":
                complete_path ← [current_start] + path
                Save complete_path and tracking data
                current_start ← found_goal
                Remove found_goal from remaining_goals
                found ← True
                Break from inner loop

        If not found:
            Stop timer
            Print "No reachable goal found within depth limit!"
            Return False

    Stop timer
    Return True
```

```
'''-------------- _dls_recursive ----------------'''
Function _dls_recursive(current, goals, limit, path, visited, visited_by_depth, depth):
    Add current to _current_explored
    Mark current as visited

    Record current in visited_by_depth[depth]

    If current is in goals:
        Return ("found", current)

    If limit <= 0:
        Return ("cutoff", None)

    cutoff_occurred ← False

    For each (action, next_state) in possible_actions(current):
        If next_state is within bounds AND not a wall AND not visited:
            Add next_state to path
            (result, found_goal) ← _dls_recursive(
                current = next_state,
                goals = goals,
                limit = limit - 1,
                path = path,
                visited = visited,
                visited_by_depth = visited_by_depth,
                depth = depth + 1
            )

            If result == "found":
                Return ("found", found_goal)
            Else if result == "cutoff":
                cutoff_occurred ← True

            Remove next_state from path (backtrack)

    Return ("cutoff", None) if cutoff_occurred else ("failure", None)
```

4.7. Iterative Deepening DFS pseudocode

```
Function solve_ids(limit):
    Start timer
    Initialize all tracking variables (solutions, explored nodes, etc.)
    Set current_start ← initial start
    Set remaining_goals ← all goal positions

    While remaining_goals is not empty:
        found ← False
        goal_explored ← empty list
        visited_by_depth_combined ← empty dictionary

        For depth from 1 to limit:
```

```
        Reset path, visited, visited_by_depth
        Run _dls_recursive() with current depth

        Append current_explored nodes to goal_explored
        Merge visited_by_depth into visited_by_depth_combined

        If a goal was found:
            Save path and tracking info
            Update current_start to found goal
            Remove goal from remaining_goals
            found ← True
            Break inner loop

    If not found after full depth range:
        Stop timer
        Print "Goal Not Found or Max Depth Reached"
        Return False


    Stop timer
    Return True
```

4.8. Iterative Deepening A* pseudocode

```
'''--------------- solve_idas -----------------'''
Function solve_idas(limit):
    Start timer
    Initialize all tracking variables

    current_start ← start position
    remaining_goals ← list of goals

    While remaining_goals not empty:
        current_goal ← pop first goal
        threshold ← Manhattan distance from current_start to current_goal
        found ← False
        visited_by_depth_combined ← {}

        Repeat up to `limit` iterations:
            Initialize empty path with current_start
            Reset _current_explored and visited_by_depth

            result ← idas_search(current_start, current_goal, g=0, threshold, path)

            Track explored nodes and visited_by_depth

            If result == "found":
                Record path, update solution and stats
                current_start ← current_goal
                found ← True
                Break loop
            Else if result is a number:
                threshold ← result   # Increase limit for next iteration
```

```
        Else:
            Break loop

    If not found:
        Print "Goal Not Found or Max Iterations Reached"
        Return False

    Stop timer
    Merge and de-duplicate visited_by_depth
    Print final tracking information
    Return True


'''--------------- idas_search ------------------'''
Function idas_search(current, goal, g_cost, threshold, path, visited_by_depth, depth):
    Add current to _current_explored and visited_by_depth[depth]

    f_cost ← g_cost + Manhattan distance from current to goal
    If f_cost > threshold:
        Return f_cost
    If current == goal:
        Return "found"

    minimum ← ∞

    For each (action, next_state) in possible_actions(current):
        If next_state is valid and not in path:
            Append next_state to path
            result ← idas_search(next_state, goal, g_cost+1, threshold, path)
            If result == "found":
                Return "found"
            Else if result < minimum:
                minimum ← result
            Remove next_state from path

    Return minimum
```

5. TESTING

For the purpose of testing the algorithm, 1000 mazes are automatically created and tested. Each algorithm will be tested with 125 cases, the test.py will record mazes that have solutions and do not have, then all the results are stored in the results folder.

6. FEATURES / BUGS / MISSING

6.1. Assignment Requirements

All the fundamental requirements of the assignment are correctly implemented, including:

- Implementing the correct usage of the Command-Line Interface.
- The nodes are expanded in the correct order (Up - Left - Down - Right) when all conditions are equal.
- Depth-first Search and Best-first Search work perfectly.

- Greedy Best-first Search and A* Search work extremely well and correctly.
- All of my custom algorithms work well in the backend.
- 10 manual test cases covering different scenarios have been verified, while 800 carefully designed cases have been tested correctly (with results shown in Section 5).
- All the codes are well-designed with helpful comments.

6.2. Finding multiple goals

All the search strategies are designed to find all the available goals. To achieve this, a goal array is created, and when a goal inside that goal array is found, it is removed from the array. Then the array is checked whether it is empty or not, and the algorithm only returns when there is no remaining goal in the array. (For an exact implementation for each algorithm, please go to my GitHub linked attacked in the ReadMe page in https://maze-searching-visualizer.vercel.app/)

6.3. GUI

Another special feature about my work is the GUI, which contains all the necessary information about all the algorithms, the instructions, the source code, a video demonstration, as well as the report. One major problem I encountered was the connection between backend and frontend, specifically in parsing data from the frontend to the backend and from the backend back to the frontend. In order to visualize the path correctly, I have to maintain various variables (especially for the iterative versions), making it harder for the data parsing. Moreover, the GUI also allows the users to analyse every individual goal, increasing the complexity of the implementations.

6.4. Limitations

Even though all the algorithms work well, the code structures are quite complicated because each algorithm has not been designed as a separate class. Therefore, the UML Diagram representing the code is quite complex and not optimal. Moreover, the Iterative Algorithms take a lot of time to find the solutions, so I will try my best to further improve them in the future.

7. RESEARCH

Besides doing the CLI, I also implemented the GUI to enhance the experience of the users. The GUI offers all the necessary information, including the instructions, visualizations, and so on (see Section 2).

One more special feature that I implemented is that the algorithm can find multiple goals, and the users can see each path in the GUI. In order to implement this, the Robot will navigate until it finds the first goal, then that goal will be set as the start to find the next goal. For uninformed search, the goal order is predefined, while the goals are sorted by the shortest for informed search. Even though the algorithms can find all goals, they do not guarantee that they will find the shortest path to travel to all goals, like the traveling salesman problem. For this specific problem, I will try to implement it in the future.

8. CONCLUSION

This report has provided a comprehensive overview of the robot navigation problem, to which eight search algorithms—Breadth-First Search, Depth-First Search, Backtracking, Depth-Limited Search, Iterative Deepening Depth-First Search, Greedy Best-First Search, A* Search, and Iterative Deepening A*—have been implemented to find solutions to maze navigation problems. Completeness, optimality, time complexity, and space complexity have been analyzed for each of the algorithms, and their merits and demerits are mentioned. The union of a command-line interface (CLI) and an easy-to-use graphical user interface (GUI) at https://maze-searching-visualizer.vercel.app/ enhances user-friendliness and visualization, making it easier for users to interact with algorithms, configure customized mazes, and inspect outcomes effectively. GUI elements like comprehensive documentation, multi-target pathfinding, and test case outcomes reveal a robust approach towards meeting assignment specifications and streamlining user experience.

Despite successful deployment, there are still challenges to be addressed, such as the complex code structure and heavy computational expense of iterative algorithms for big mazes. Improving code modularity, enhancing iterative algorithm efficiency, and investigating new high-level multi-objective pathfinding techniques, including Traveling Salesman Problem-inspired ones, will be future tasks. This project highlights the importance of balancing usability with algorithmic efficiency, providing a foundation for future innovation in robotic search and navigation algorithm applications.

9. ACKNOWLEDGEMENTS