COS30019- Introduction to Artificial Intelligence

Assignment 1 – Tree Based Search Report

HAI NAM NGO

STUDENT ID: 103488515

Page Count: 14 Pages (Excluding Cover pages, TOC, References)

Table of Contents

INSTURCTIONS

Step 1: Make sure that the laptop has downloaded python and pygame.

To download pygame, this command line should be used:

```
PS C:\Users\ngoha> pip install pygame
```

Step 2: Open the folder that has my "search.py" source code and the test case (txt file) that will be used (They need to be in the same folder) with the command format: **"cd [the path file]".**

```
PS C:\Users\ngoha> cd 'C:\Users\ngoha\OneDrive\Desktop\COS30019-Introduction to Artificial Intelligence\Asm1'
```

Step 3: Run the command line with the following format:

python search.py [filename.txt] [the shorten name of the search algorithm]

```
C:\Users\ngoha\OneDrive\Desktop\COS30019-Introduction to Artificial Intelligence\Asm1> python search.py .\RobotNav-test.txt BFS
```

There are 6 options for the search algorithm available in the program:

1. BFS - Breath First Search
2. DFS - Depth First Search
3. DLS- Depth Limited Search
4. AS- A* Search
5. GBFS- Greedy Best First Search
6. IDA- Iterative Deepening A* Search

Step 4: If you want to try other algorithms with the same test cases, you can use the GUI provided in the program (details will be explained later in the report).

Step 5: If you want to test other test cases, please quit the program, and use the command line to open the new test case, instructions are given in **Step 2** and **Step 3**.

INTRODUCTION

Robotics aims to enable robots to find the most effective routes from beginning to end while avoiding obstacles to improve autonomous navigation across a variety of environments. This is also known as the description for **The Robot Navigation Problem** and is essential for applications in automated vehicles and robotics.

In general, an agent (robot in this case) must take through an environment to get from a starting point (start node) to a destination (goal node). Usually, this environment is shown as a graph, with nodes representing potential robot locations and edges representing viable routes between these locations. The key objectives of solving this problem are to guarantee that the robot avoids any obstacles that are represented in the graph as impassable nodes or edges, and that the path is optimal or nearly optimal in terms of cost or distance.

Basic Graph and Tree Concepts

Graph: A graph is made up of nodes and the lines that join node pairs. Nodes in robot navigation refer to locations, and lines indicate paths between these locations.

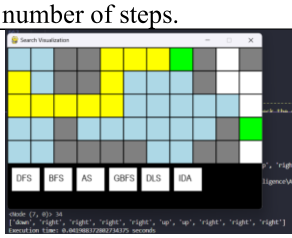
Tree: This is a unique kind of graph. There are no cycles in trees, they are made up of a root node and child nodes, each of which may have additional child nodes.

This program will be more like a graph search, and the report will analyze the problem based on the collected data from the actual implementation. Other terminology will be explained when it is used in the report.

SEARCH ALGORITHMS

Breadth First Search (BFS)

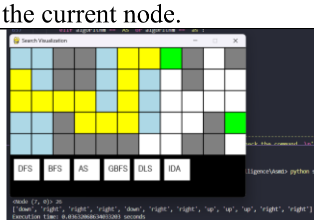
According to the research (*Uninformed Search Algorithms - Javatpoint*, n.d.), An algorithm for graph traversal called Breadth First Search (BFS) investigates every node in a graph at the depth it is currently at before going on to the next depth level. It visits each of its neighbors after beginning at a given node and progressing to the next level of neighbors. BFS is frequently utilized in graph-based pathfinding, connected components, and shortest path algorithms.

Advantage	Disadvantage
• BFS will offer a solution if one exists. • If there are multiple solutions to a given problem, BFS will select the simplest solution with the fewest number of steps.	• It requires a large amount of memory because each level of the tree must be saved in memory before proceeding to the next level. • BFS requires a significant amount of time if the solution is located far from the initial state.
 *Figure 1 BFS algorithm is used in the case of the sample map (Execution time: 0.0419 seconds)*	For example, in this sample map (provided by Assignment 1), the BFS algorithm has successfully found the shortest path to the nearest goal state. Optimal solution, however, we can see that it needs to explore nearly every node in the map (34/40), to find the path successfully. This proves that BFS needs lots of data related to explored node in, which is not as fast as other methods.

Depth First Search (DFS)

Depth-first search begins at the root node (initial state) and travels as far as possible along each branch before backtracking. The process of the DFS is nearly the same when comparing to BFS.

The difference between DFS and BFS: DFS uses LIFO queue (stack- Last in First Out) for its implementation, while BFS uses FIFO queue (First in First Out).

Advantage	Disadvantage
• DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.	• DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.
 *Figure 2 DFS algorithm is used in the case of the sample map (Execution time: 0.0363 seconds)*	In this sample, the execution time is faster than BFS (from 0.0419 to 0.0363 seconds), which means DFS might have ability to search for the goal faster than BFS if it searches in the correct path. Moreover, the explored node has also decreased, from 34/40 in BFS, down to 26/40 in DFS. However, the path is longer than the path found by using BFS, proves that this might not be the optimal algorithm.

Depth Limited Search (DLS- custom search 1)

DLS algorithm is like DFS but has a specified limit. DLS can overcome the disadvantage of the infinite path in DFS. In this algorithm, the node at the depth limit will be treated as if it has no more child nodes and continue to search for the node in the next branch.

Advantage	Disadvantage
• Memory efficient	• Have possibility of unable to find a solution if the goal node is deeper than the depth limit. • Might not be optimal if there is more than one solution path.

Figure 3 DLS with depth limit 1000

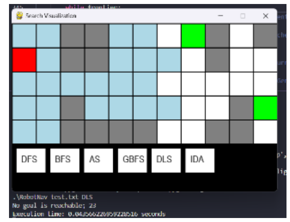


Figure 4 DLS with depth limit 7

In this sample, a solution path is found, and it is the same solution as the DFS above. However, Figure 3 is the case when the depth limit is set to 1000, if it is set to a low number like in the Figure 4, for example, the depth limit is 7, the algorithm cannot find the path, even though there is a possible solution available, because the goal state is deeper than the depth limit.

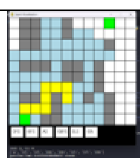
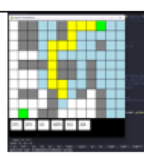
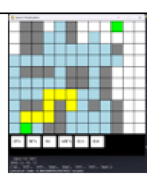
Greedy Best First Search (GBFS)

According to (*Informed Search Algorithms in AI - Javatpoint*, 2011), GBFS always chooses the most efficient node only at that time, and it will connect the efficient node list together to make up a path. It is the combination between DFS and BFS. It makes use of heuristics and searches. GBFS allows us to benefit from both algorithms. At each step, it can select the most promising node, and expands the node that is closest to the goal node, and the closest cost is estimated using a heuristic function and the cost to act.

f(n)= g(n), where g(n) is the cost to travel from the start node to the current node.

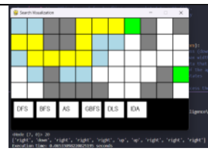
Advantage	Disadvantage
• This algorithm is more efficient than BFS and DFS.	• Might cause infinite loop like DFS • Might not optimal

Since the sample map is easy to solve it is difficult to see the difference between BFS, DFS and GBFS, so a more complicated map (map3.txt in the test cases) has been created to analyze the GBFS algorithm.

 *Figure 5 BFS with the map3.txt*	For this map, BFS needs to explore 59 nodes, to come up with the most optimal path to the goal (1,9), which is closer comparing to the other goal.
 *Figure 6 DFS with the map3.txt*	DFS needs to explore 61 nodes, and it shows the path to the further goal, not even search for the goal that is closer to the initial state.
 *Figure 7 GBFS with the map3.txt*	GBFS might come up with the best option with the least nodes that need to explore, only 57 nodes, even though it is not that huge different from the length of explored list of the BFS and DFS, it still proves that GBFS is more efficient than both DFS and BFS. Sometimes, GBFS might not be able to show the most optimal path, it is not optimal because of the natural of GBFS itself, this case is just because that path is the only option to reach the goal.

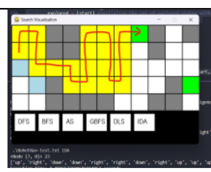
A Star Search (AS)

It employs the heuristic function h(n) and the cost to reach node n from the initial state g(n). It combines UCS features with greedy best-first search to efficiently solve the problem. The A* search algorithm uses the heuristic function to find the shortest path through the search space. This search algorithm expands fewer search trees and delivers optimal results faster.

Advantage	Disadvantage
• can solve extremely complex problems.	• Complexity issues • Have high chance (but not always) give the best path
 *Figure 8 AS with the sample map*	Compared to other search algorithm above, AS come up with the optimal solution with the least node that need to be explored. This proves that AS is the best algorithm so far, although the execution time is quite long compared to other algorithm (0.065 seconds)

Iterative Deepening A Star Search (IDA- custom search 2)

IDA is a path-finding algorithm that combines the heuristic-driven search of AS with the iterative deepening approach of depth-first search to find the shortest path in a weighted graph. It functions similarly to AS, using a heuristic to estimate the cost to the goal, thereby determining which nodes to explore and at what depth.

Advantage	Disadvantage
• use less memory than AS	• As depth increases, there is a significant amount of redundancy and computational overhead because of each iteration's re-exploration.
 *Figure 9 IDA with the sample map*	In this sample, IDA allows the agent to travel through the map, it can still reach the goal even though it is not that optimal.

Comparison between algorithms

Ranking (Personal Idea)	Algorithm	Time Complexity	Space Complexity	Complete	Optimal
Top2	BFS	$O(b^d)$	$O(b^d)$	Complete (finite nodes)	Optimal
Top3	DFS	$O(b^m)$	$O(b^m)$	Complete (finite nodes)	No (might require more step to reach the goal)
Top5	DLS	$O(b^l)$	$O(b^l)$	Complete (if the goal state is above the depth limit)	Not optimal even when l>d.
Top4	GBFS	$O(b^m)$	$O(b^m)$	Incomplete	Not optimal
Top1	AS	$O(b^d)$	$O(b^d)$	Complete (finite nodes)	Optimal (If the heuristic function is valid)
Top6	IDA	$O(b^d)$	$O(b)$	Incomplete (if heuristic function is not consistent)	Not optimal (if heuristic function is not consistent)

b: maximum child node can have; d: the depth; m: maximum depth; l: depth limit

IMPLEMENTATION

Breadth First Search (BFS)

```
Class BFS inherits Search

    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize explored with initial_state
        Initialize frontier with initial_state
        Initialize bfsPath as an empty dictionary

        While frontier is not empty
            current_state = Remove the oldest element from frontier (FIFO)

            If current_state is a goal state
                Set self.explored to explored
                Return generate_output(explored, bfsPath)

            Define directions as list of tuples [(0, -1), (-1, 0), (0, 1), (1, 0)]  # UP, LEFT, DOWN, RIGHT

            neighbors = find_neighbor(current_state, directions)
            For each neighbor in neighbors
                If neighbor is in my_map.path_list and not in explored
                    Add neighbor to explored
                    Add neighbor to the end of frontier (FIFO)
                    Set bfsPath[neighbor] to current_state  # Map the neighbor back to current state

        Set self.explored to explored
        Return the number of explored nodes
```

Figure 10 Pseudocode for BFS implementation

Depth First Search (DFS)

```
Class DFS inherits Search
    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize explored with initial_state
        Initialize frontier with initial_state
        Initialize dfsPath as an empty dictionary

        While frontier is not empty
            current_state = Remove the last element from frontier (LIFO)
            If current_state is a goal state
                Set self.explored to explored
                Return generate_output(explored, dfsPath)

            Define directions as list of tuples [(1, 0), (0, 1), (-1, 0), (0, -1)]  # RIGHT, DOWN, LEFT, UP
            neighbors = find_neighbor(current_state, directions)
            For each neighbor in neighbors
                If neighbor is in my_map.path_list and not in explored
                    Add neighbor to explored
                    Add neighbor to frontier (LIFO)
                    Set dfsPath[neighbor] to current_state  # Map the neighbor back to current state

        Set self.explored to explored
        Return the number of explored nodes
```

Figure 11 Pseudocode for DFS implementation

Depth Limited Search (DLS- custom search 1)

```
Class DLS inherits Search

    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize explored as a list with initial_state
        Initialize frontier as a stack with (initial_state, 0)  # Node with its depth
        Initialize dlsPath as an empty dictionary
        Set depth_limit to 7  # Define a maximum depth limit

        While frontier is not empty
            current_state, depth = Remove the last element from frontier (LIFO)

            If depth is greater than depth_limit
                Continue to the next iteration of the loop

            If check_goal_state(current_state) is true
                Set self.explored to explored
                Return generate_output(explored, dlsPath)  # Generate and return output if goal is found

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # RIGHT, DOWN, LEFT, UP
            neighbors = find_neighbor(current_state, directions)  # Get valid neighbors

            For each neighbor in neighbors
                If neighbor is in my_map.path_list and not in explored
                    Add neighbor to explored
                    Add (neighbor, depth + 1) to frontier  # Push neighbor with incremented depth
                    Set dlsPath[neighbor] to current_state  # Map neighbor back to current state

        Set self.explored to explored
        Return the number of explored nodes  # Return count if no goal is found
```

Figure 12 Pseudocode for DLS implementation

<table>
<tr>
<td>Greedy Best First Search (GBFS)</td>
<td>
<pre><code>Class GBFS inherits Search

    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize unvisited with all cells in my_map.path_list set to infinity
        Set unvisited[initial_state] to 0
        Initialize explored with initial_state
        Initialize gbfsPath as an empty dictionary

        While unvisited is not empty
            current_state = Get the state with the minimum value in unvisited

            If check_goal_state(current_state) is true
                Set self.explored to explored
                Return generate_output(explored, gbfsPath)

            directions = [(0, -1), (-1, 0), (0, 1), (1, 0)] # UP, LEFT, DOWN, RIGHT
            neighbors = find_neighbor(current_state, directions)

            For each neighbor in neighbors
                If neighbor is in my_map.path_list and not in explored
                    tempDist = unvisited[current_state] + 1
                    If tempDist < unvisited[neighbor]
                        Add neighbor to explored
                        Set unvisited[neighbor] to tempDist
                        Set gbfsPath[neighbor] to current_state

            Remove current_state from unvisited

        Set self.explored to explored
        Return the number of explored nodes</code></pre>
Figure 13 Pseudocode for GBFS implementation
</td>
</tr>
<tr>
<td>A Star Search (AS)</td>
<td>
<pre><code>Class AS inherits Search
    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize explored with initial_state
        Initialize aPath as an empty dictionary
        Initialize priority queue `open` and dictionaries g_score and f_score with infinite values

        For each goal in goal_states
            Set g_score[start] to 0
            Set f_score[start] to manhattan distance from start to goal
            Add start to `open` with (f_score[start], manhattan distance, start)

            While open is not empty
                current_state = get the node from open with the lowest f_score

                If current_state is a goal state
                    Set self.explored to explored
                    Return generate_output(explored, aPath)

                directions = [(0, -1), (-1, 0), (0, 1), (1, 0)] # UP, LEFT, DOWN, RIGHT
                neighbors = find_neighbor(current_state, directions)

                For each neighbor in neighbors
                    If neighbor is valid and not in explored
                        Add neighbor to explored

                        Calculate temp_g_score and temp_f_score
                        If temp_f_score < f_score[neighbor]
                            Update g_score[neighbor] and f_score[neighbor]
                            Add neighbor to open with its new f_score
                            Update aPath mapping neighbor to current_state

                If no valid neighbor found and open is empty
                    Break
        Set self.explored to explored
        Return the length of explored nodes</code></pre>
Figure 14 Pseudocode for AS implementation
</td>
</tr>
<tr>
<td>Iterative Deepening A Star Search (IDA-custom search 2)</td>
<td>
<pre><code>Class IDA inherits Search

    Method __init__(my_map, initial_state, goal_states)
        Call parent class constructor with my_map, initial_state, goal_states

    Method search()
        Initialize path_cost as 1
        Initialize explored with initial_state
        Initialize idaPath as an empty dictionary
        Initialize start as initial_state

        For each goal in goal_states
            Set initial threshold using the Manhattan distance from start to goal
            Repeat
                result, new_threshold = search_recursively(start, goal, path_cost, threshold, explored, idaPath)
                If result is 'FOUND'
                    Set self.explored to explored
                    Return generate_output(explored, idaPath)
                If result is 'NOT_FOUND'
                    If new_threshold is infinite
                        Break
                Update threshold to new_threshold

        Set self.explored to explored
        Return the number of explored nodes

    Method search_recursively(current_state, goal, path_cost, threshold, explored, idaPath)
        Calculate f_score as Manhattan distance from current_state to goal
        If f_score exceeds threshold
            Return 'NOT_FOUND', f_score

        If current_state is the goal
            Return 'FOUND', f_score

        Define directions for movement (UP, LEFT, DOWN, RIGHT)
        neighbors = find_neighbor(current_state, directions)

        Initialize valid_neighbor as False
        For each neighbor in neighbors
            If neighbor is in my_map.path_list and not in explored
                Append neighbor to explored
                valid_neighbor = True
                Recursively search from neighbor with increased path cost
                result, new_threshold = search_recursively(neighbor, goal, path_cost + 1, threshold, explored, idaPath)
                If result is 'FOUND'
                    Update idaPath with neighbor pointing to current_state
                    Return 'FOUND', new_threshold

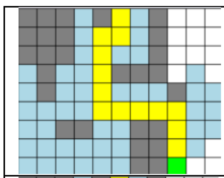
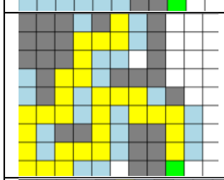
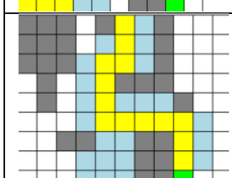
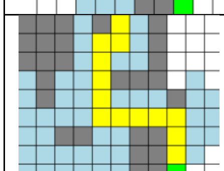
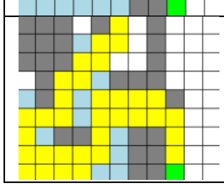
        If no valid neighbors are found
            Return 'NOT_FOUND', infinity # Use a very large number as infinite

        Return 'NOT_FOUND', infinity # No goal found at this depth</code></pre>
Figure 15 Pseudocode for IDA implementation
</td>
</tr>
</table>

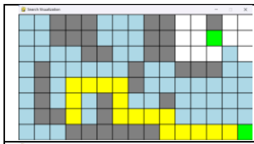
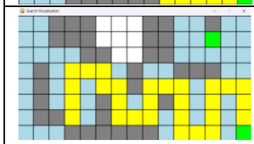
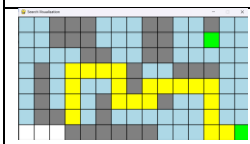
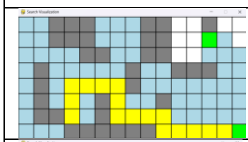
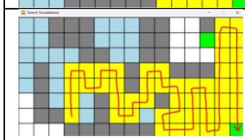
Algorithm	Approach	Implementation
BFS	Explores nodes in layers, ensuring that all nodes at the current depth are explored before moving to the next depth.	Uses a queue to manage the frontier, ensuring nodes are explored in the order they were discovered.
DFS	Explores as deep as possible along each branch before backtracking.	Uses a stack to manage the frontier, allowing the algorithm to dive deep into the graph by processing the most recently discovered node first.
DLS	Like DFS but with a predetermined limit on the depth of the search, preventing it from going infinitely deep.	Uses a stack with an additional depth parameter for each node to control how deep the search can go.
GBFS	Uses a heuristic to prioritize nodes that are estimated to be closest to the goal, ignoring the cost from the start node.	Like AS, it uses a priority queue but only considers the heuristic value (h(n)), making it less optimal but often faster.
AS	Uses a heuristic to estimate the cost from the current node to the goal, combined with the cost from the start node to the current node	Employs a priority queue based on the f-score of nodes, ensuring that the path with the lowest estimated total cost is explored first.
IDA	Combines the depth-first exploration of DLS with the heuristic evaluation of AS.	Repeatedly performs depth-first searches, increasing the depth threshold based on the f-score until the goal is found.

TESTING

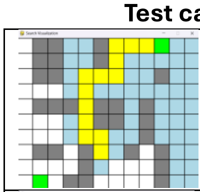
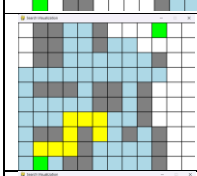
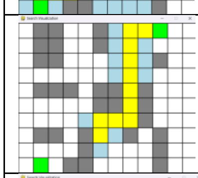
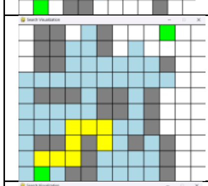
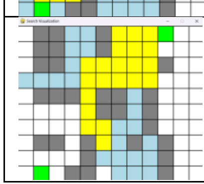
Test case 1: map1.txt

	**.\map1.txt BFS** <Node (8, 8)> 58 ['down', 'left', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'down', 'down', 'down'] Execution time: 0.04405331611633301 seconds
	**.\map1.txt DFS (DLS also have the same output)** <Node (8, 8)> 51 ['down', 'left', 'left', 'down', 'down', 'left', 'down', 'down', 'left', 'left', 'down', 'down', 'down', 'right', 'right', 'up', 'right', 'right', 'up', 'up', 'up', 'right', 'right', 'down', 'right', 'right', 'down', 'down', 'down'] Execution time: 0.003448486328125 seconds
	**.\map1.txt AS** <Node (8, 8)> 36 ['down', 'down', 'left', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'down', 'down', 'down'] Execution time: 0.0025932788848876953 seconds
	**.\map1.txt GBFS** <Node (8, 8)> 58 ['down', 'left', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'down', 'down', 'down'] Execution time: 0.0022513866424560547 seconds
	**.\map1.txt IDA** <Node (8, 8)> 44 ['down', 'left', 'down', 'left', 'down', 'left', 'down', 'down', 'left', 'left', 'down', 'down', 'down', 'right', 'up', 'right', 'down', 'right', 'up', 'right', 'up', 'up', 'up', 'right', 'down', 'right', 'up', 'right', 'down', 'right', 'down', 'down', 'down'] Execution time: 0.0019855499267578125 seconds

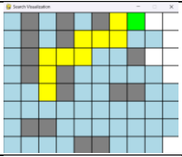
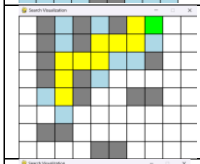
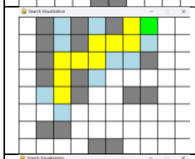
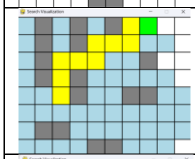
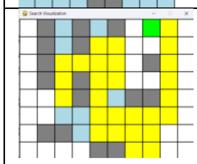
Test case 2: map2.txt

	**.\map2.txt BFS** <Node (14, 7)> 73 ['up', 'up', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'right', 'right', 'right'] Execution time: 0.0010006427764892578 seconds
	**.\map2.txt DFS (DLS also have the same output)** <Node (14, 7)> 77 ['up', 'up', 'up', 'right', 'right', 'down', 'right', 'down', 'down', 'right', 'right', 'up', 'up', 'right', 'right', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'down'] Execution time: 0.0010089874267578125 seconds
	**.\map2.txt AS** <Node (14, 7)> 82 ['up', 'up', 'up', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'up', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'right', 'right'] Execution time: 0.003039121627807617 seconds
	**.\map2.txt GBFS** <Node (14, 7)> 76 ['up', 'up', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'right', 'right', 'right'] Execution time: 0.0023317337036132812 seconds
	**.\map2.txt IDA** <Node (14, 7)> 73 ['up', 'up', 'up', 'right', 'down', 'right', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'right', 'up', 'up', 'right', 'down', 'right', 'down', 'down', 'left', 'down', 'right', 'right', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down'] Execution time: 0.0021967887878417969 seconds

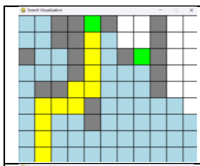
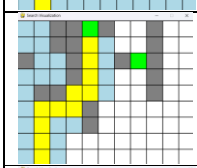
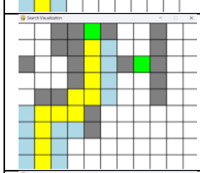
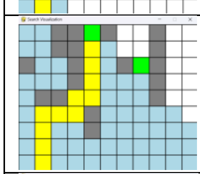
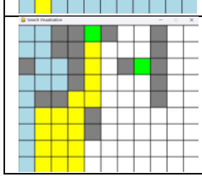
Test case 3: map3.txt

	**.\map3.txt DFS (DLS also have the same output)** <Node (9, 0)> 61 ['up', 'left', 'up', 'up', 'up', 'up', 'right', 'right', 'up', 'up', 'right', 'right', 'right'] Execution time: 0.0010082721710205078 seconds
	**.\map3.txt BFS** <Node (1, 9)> 59 ['up', 'left', 'left', 'down', 'down', 'left', 'left', 'down'] Execution time: 0.00269317626953125 seconds
	**.\map3.txt AS** <Node (9, 0)> 22 ['up', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'right'] Execution time: 0.0010092258453369141 seconds
	**.\map2.txt GBFS** <Node (14, 7)> 76 ['up', 'up', 'right', 'right', 'right', 'down', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'right', 'right', 'right'] Execution time: 0.0023317337036132812 seconds
	**.\map3.txt IDA** <Node (9, 0)> 44 ['up', 'left', 'up', 'up', 'up', 'up', 'right', 'down', 'right', 'up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'right'] Execution time: 0.0010058879852294922 seconds

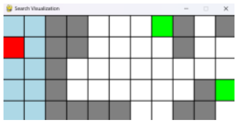
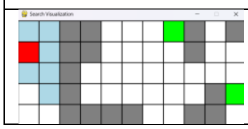
Test case 4: map4.txt

	**.\map4.txt BFS** <Node (7, 0)> 57 ['up', 'up', 'right', 'right', 'up', 'right', 'right', 'up', 'right'] Execution time: 0.023509502410888672 seconds
	**.\map4.txt DFS (DLS also have the same output)** <Node (7, 0)> 19 ['up', 'up', 'right', 'right', 'up', 'right', 'right', 'up', 'right'] Execution time: 0.0012240409851074219 seconds
	**.\map4.txt AS** <Node (7, 0)> 19 ['up', 'up', 'right', 'right', 'up', 'right', 'right', 'up', 'right'] Execution time: 0.001680135726928711 seconds
	**.\map4.txt GBFS** <Node (7, 0)> 57 ['up', 'up', 'right', 'right', 'up', 'right', 'right', 'up', 'right'] Execution time: 0.0012960433959960938 seconds
	**.\map4.txt IDA** <Node (7, 0)> 35 ['up', 'up', 'right', 'right', 'up', 'right', 'down', 'down', 'left', 'down', 'down', 'down', 'right', 'up', 'right', 'down', 'down', 'right', 'up', 'up', 'right', 'up', 'up', 'up', 'up', 'up', 'left'] Execution time: 0.0022673606872558594 seconds

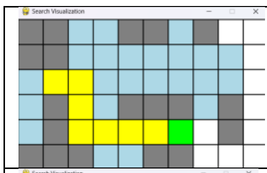
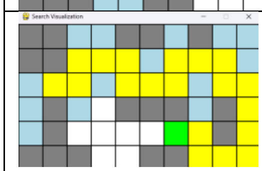
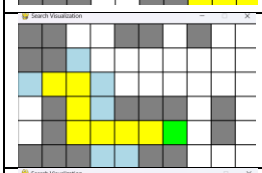
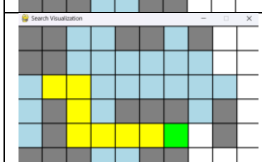
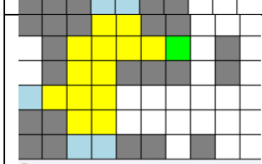
Test case 5: map5.txt

	**.\map5.txt BFS** <Node (4, 0)> 65 ['up', 'up', 'up', 'right', 'right', 'up', 'right', 'up', 'up', 'up', 'up'] Execution time: 0.002506732940673828 seconds
	**.\map5.txt DFS (DLS also have the same output)** <Node (4, 0)> 34 ['up', 'up', 'up', 'right', 'right', 'up', 'right', 'up', 'up', 'up', 'up'] Execution time: 0.0019555091857910156 seconds
	**.\map5.txt AS** <Node (4, 0)> 24 ['up', 'up', 'up', 'right', 'right', 'up', 'right', 'up', 'up', 'up', 'up'] Execution time: 0.001996755599975586 seconds
	**.\map5.txt GBFS** <Node (4, 0)> 65 ['up', 'up', 'up', 'right', 'right', 'up', 'right', 'up', 'up', 'up', 'up'] Execution time: 0.004678249359130859 seconds
	**.\map5.txt IDA** <Node (4, 0)> 32 ['up', 'up', 'up', 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'up'] Execution time: 0.00225830078125 seconds

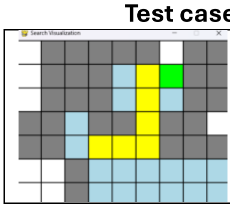
Test case 6: map6.txt

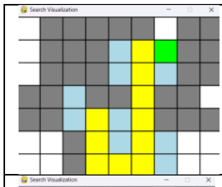
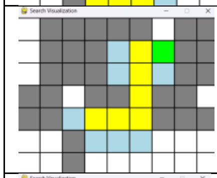
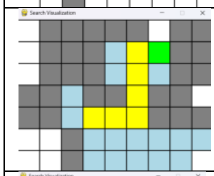
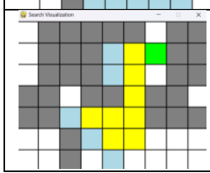
	**.\map6.txt BFS** No goal is reachable; 10 Execution time: 0.0010693073272705078 seconds
	.\map6.txt DFS (DLS also have the same output) No goal is reachable; 10 Execution time: 0.001466512680053711 seconds
	.\map6.txt AS No goal is reachable; 10 Execution time: 0.001992464065551758 seconds
	.\map6.txt GBFS No goal is reachable; 10 Execution time: 0.002001523971557617 seconds
	**.\map6.txt IDA** No goal is reachable; 7 Execution time: 0.0009114742279052734 seconds

Test case 7: map7.txt

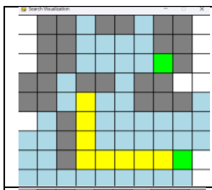
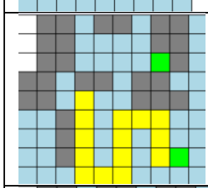
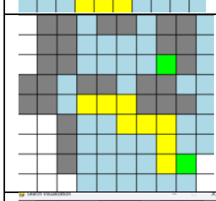
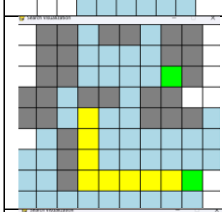
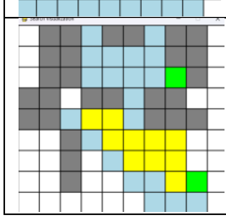
	**.\map7.txt BFS** <Node (6, 4)> 31 ['right', 'down', 'down', 'right', 'right', 'right', 'right'] Execution time: 0.027990341186523438 seconds
	**.\map7.txt DFS (DLS also have the same output)** <Node (6, 4)> 34 ['right', 'up', 'right', 'right', 'down', 'right', 'right', 'up', 'right', 'right', 'down', 'right', 'down', 'down', 'down', 'left', 'left', 'up', 'left'] Execution time: 0.0 seconds
	**.\map7.txt AS** <Node (6, 4)> 14 ['right', 'down', 'down', 'right', 'right', 'right', 'right'] Execution time: 0.001001119613647461 seconds
	**.\map7.txt GBFS** <Node (6, 4)> 30 ['right', 'down', 'down', 'right', 'right', 'right', 'right'] Execution time: 0.002357006072998047 seconds
	**.\map7.txt IDA** <Node (6, 4)> 17 ['right', 'up', 'right', 'down', 'down', 'left', 'down', 'right', 'down', 'right', 'up', 'right', 'right'] Execution time: 0.0015130043029785156 seconds

Test case 8: map8.txt

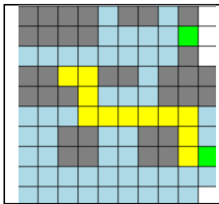
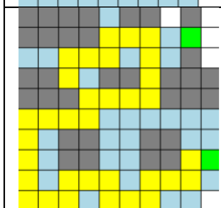
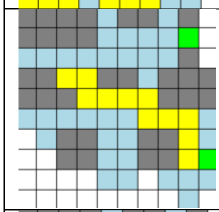
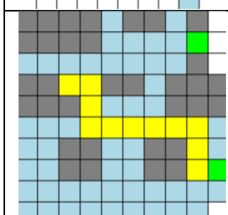
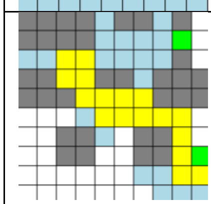
	**.\map8.txt BFS** <Node (6, 1)> 24 ['right', 'right', 'up', 'up', 'up', 'right'] Execution time: 0.026655912399291992 seconds

	**.\map8.txt DFS (DLS also have the same output)** <Node (6, 1)> 20 ['down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'right'] Execution time: 0.0009949207305908203 seconds
	**.\map8.txt AS** <Node (6, 1)> 14 ['right', 'right', 'up', 'up', 'up', 'right'] Execution time: 0.0010156631469726562 seconds
	**.\map8.txt GBFS** <Node (6, 1)> 23 ['right', 'right', 'up', 'up', 'up', 'right'] Execution time: 0.0017108917236328125 seconds
	**.\map8.txt IDA** <Node (6, 1)> 14 ['right', 'down', 'right', 'up', 'up', 'up', 'up', 'right'] Execution time: 0.0009102821350097656 seconds

Test case 9: map9.txt

	**.\map9.txt BFS** <Node (8, 7)> 51 ['down', 'down', 'down', 'right', 'right', 'right', 'right', 'right'] Execution time: 0.030934810638427734 seconds
	**.\map9.txt DFS (DLS also have the same output)** <Node (8, 7)> 59 ['down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'right', 'right', 'down', 'down', 'right'] Execution time: 0.0010154247283935547 seconds
	**.\map9.txt AS** <Node (8, 7)> 48 ['right', 'right', 'down', 'right', 'right', 'down', 'down', 'right'] Execution time: 0.0024745464324951172 seconds
	**.\map9.txt GBFS** <Node (8, 7)> 51 ['down', 'down', 'down', 'right', 'right', 'right', 'right', 'right'] Execution time: 0.0010273456573486328 seconds
	**.\map9.txt IDA** <Node (8, 7)> 31 ['right', 'down', 'right', 'down', 'right', 'up', 'right', 'down', 'down', 'right'] Execution time: 0.0012373924255371094 seconds

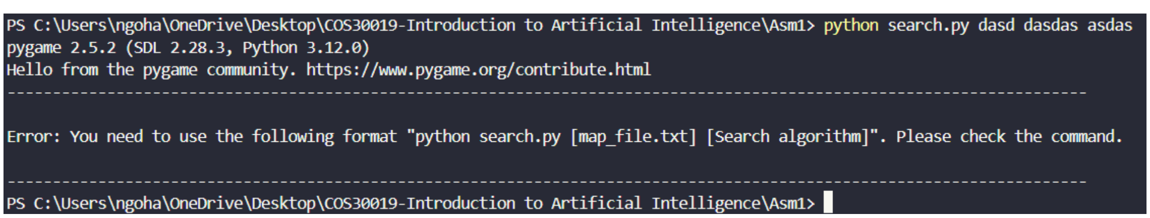
Test case 10: map10.txt

	**.\map10.txt BFS** <Node (9, 7)> 62 ['right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'right'] Execution time: 0.03209662437438965 seconds
	**.\map10.txt DFS (DLS also have the same output)** <Node (9, 7)> 61 ['up', 'right', 'right', 'up', 'right', 'right', 'down', 'down', 'down', 'left', 'left', 'left', 'down', 'left', 'left', 'left', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'right', 'right', 'down', 'right', 'right', 'up', 'right', 'right', 'up', 'right'] Execution time: 0.0019443035125732422 seconds
	**.\map10.txt AS** <Node (9, 7)> 46 ['right', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'down', 'down', 'right'] Execution time: 0.0010890960693359375 seconds
	**.\map10.txt GBFS** <Node (9, 7)> 62 ['right', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'right'] Execution time: 0.0020020008087158203 seconds
	**.\map10.txt IDA** <Node (9, 7)> 38 ['up', 'right', 'down', 'down', 'right', 'down', 'right', 'up', 'right', 'down', 'right', 'right', 'down', 'down', 'down', 'right', 'up'] Execution time: 0.001901865005493164 seconds

FEATURES/ BUGS/ MISSING

Error Handling Feature: To make sure that the program can run smoothly without errors that come from command line and text file. Some error checking has been created to check if the command line has the correct format or not, and to see if the text file is a valid one or not. If it is not satisfactory with the condition, guidance will be given for the user to check the error.

Situation 1: The user uses the wrong command format



```
PS C:\Users\ngoha\OneDrive\Desktop\COS30019-Introduction to Artificial Intelligence\Asm1> python search.py dasd dasdas asdas
pygame 2.5.2 (SDL 2.28.3, Python 3.12.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
------------------------------------------------------------------------------------------------------------

Error: You need to use the following format "python search.py [map_file.txt] [Search algorithm]". Please check the command.

------------------------------------------------------------------------------------------------------------
PS C:\Users\ngoha\OneDrive\Desktop\COS30019-Introduction to Artificial Intelligence\Asm1>
```

Figure 16 Situation 1 Demonstration

Situation 2: The user uses the wrong map file

Figure 17 Situation 2 Demonstration

Situation 3: The user uses the wrong algorithm (the algorithm which is not exist in the program)

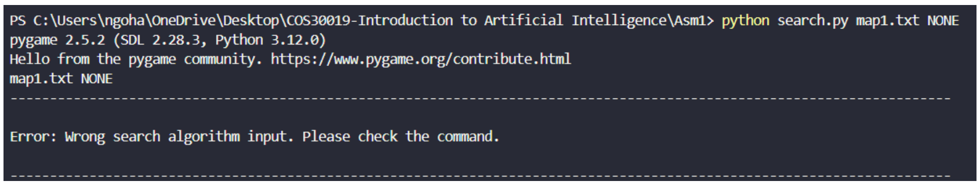


Figure 17 Situation 3 Demonstration

Situation 4: When the text file is not valid (wrong format)

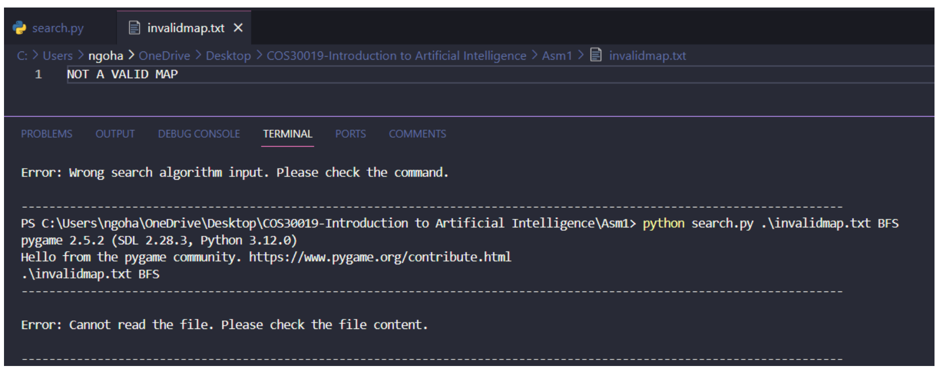


Figure 18 Situation 4 Demonstration

Execution time feature: The Python time library is used to implement this feature. An algorithm's running time from the starting state to one of the goal states can be estimated using this feature. This execution time is only an estimate and may vary each time the code runs. However, the user can still use this to compare the execution time between algorithm in the same map.

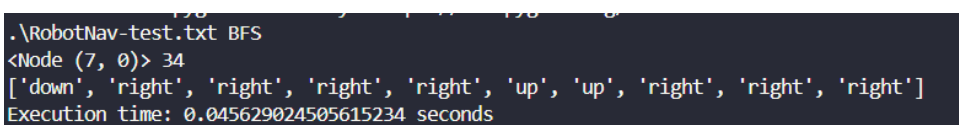


Figure 16 Execution time feature Demonstration

About custom search 2- IDA:

Even though it is successfully implemented, sometimes it also cannot find an available path even though there is one available. The reason might be that it also applies the natural of DLS, make the search is limited to specific number of nodes (that is what I think now). This issue is still in the process of fixing and analyzing, however, it does not affect the program, and is not considered as a bug.

RESEARCH

For further research, implementing a GUI for the program is the option that applied to this assignment.

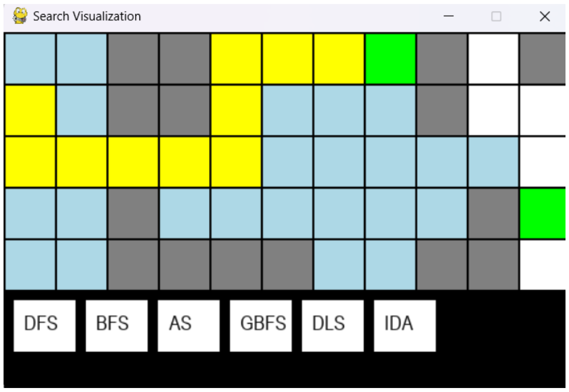


Figure 17 Appearance of the GUI

The source code for this visualization can be found in the visualization class. This function is made with the help of the pygame library. This function's logic is to print the static map first and then receive the result output from the algorithm once it has finished. The explored list (blue nodes) will then be printed. Ultimately, the completed route will be printed (with yellow nodes).

Click on any of the six buttons below the map to see a different algorithm displayed on the same map. The six buttons correspond to the six algorithms that are available in the program.

Note: The user should wait for the drawing process to finish and not press any buttons while the GUI is in this state. When a button is clicked while the GUI is drawing, the request for that button will be visualized immediately after the current action is completed, preventing the user from seeing the results of the previous drawing process.

CONCLUSION

In conclusion, after thorough analysis and implementation of the Robot Navigation Program, it's recommended to use Breadth First Search (BFS) for uninformed search and A Star Search (AS) for informed search. BFS and AS consistently provide optimal solutions, with AS often requiring the least number of explored nodes. While the program's current performance is satisfactory, there is room for improvement in addressing issues like the reliability of Iterative Deepening A Star Search (IDA) and enhancing the graphical user interface (GUI) for better quality, for example, it can actually print out all neighbor nodes at the same time, not each of them at a time, if that is successfully implemented, it will be more efficient.

ACKNOWLEDGEMENTS/ RESOURCES

Java point website: This website has full information of both informed search and uninformed search in the program. This source helps me to understand more about the algorithm implementation, their logic and the advantages and disadvantages of each algorithm.

Learning Orbit's YouTube Channel: This is a channel that guides me through the pseudo code of BFS, DFS, AS and GBFS (in 4 different videos). And, how to implement using Python. In the tutorial, he used the library "pyamaze" which is created by himself., however, I did not use that but create everything by myself.

Algorithms & Technologies' Website: This website helps me to know how to implement IDA in Python, however my way of implementing is still different from this website's code.

ChatGPT: When my first source code isn't working well, this AI helps me identify and correct errors. It also suggests alternative approaches to the problem and evaluates the quality of my source code. I did not, however, copy and paste AI's source code; instead, I used the inspiration it provided to create a solid foundation for my own source code. To ensure that the generated map is entirely random, ChatGPT also provides most of the test cases, to make sure everything is randomly created.

The URLs will be cited in the references list below.

REFERENCES (APA7)

1. *Informed Search Algorithms in AI - Javatpoint*. (2011). Www.javatpoint.com. https://www.javatpoint.com/ai-informed-search-algorithms
2. *Iterative Deepening A Star in Python*. (2019, December 14). Www.algorithms-And-Technologies.com. https://www.algorithms-and-technologies.com/iterative_deepening_a_star/python
3. Learning Orbis. (2021a, September 18). *Breadth First Search (BFS) in Python [Python Maze World- pyamaze]*. Www.youtube.com. https://www.youtube.com/watch?v=D14YK-0MtcQ&t=319s
4. Learning Orbis. (2021b, September 21). *Depth First Search (DFS) in Python [Python Maze World- pyamaze]*. Www.youtube.com. https://www.youtube.com/watch?v=sTRK9mQgYuc&t=0s
5. Learning Orbis. (2021c, October 2). *A-Star A* Search in Python [Python Maze World- pyamaze]*. Www.youtube.com. https://www.youtube.com/watch?v=W9zSr9jnoqY&t=0s
6. *Uninformed Search Algorithms - Javatpoint*. (n.d.). Www.javatpoint.com. Retrieved April 16, 2024, from https://www.javatpoint.com/ai-uninformed-search-algorithms#:~:text=Uninformed%20search%20is%20a%20class