

Comparative Analysis of Search Algorithms: Based on Performance and Efficiency

Name: Daniel Cyril Obon

Student ID: C2650218

Course: Computer Science (Year 2)

Assignment: Artificial Intelligence Element 2 (ICA 2)

Abstract:

This report will compare the three search algorithms, Breadth-first Search (BFS), Depth-first Search (DFS) and A-star Search (A*), in terms of performance such as time taken, accuracy and total paths taken per search, and the efficiency of each search algorithm. The purpose of this report is to determine the most optimal search algorithm among the three, i.e. BFS, DFS and A*, with the aid of program implementation in Python using Pyamaze for visualisation and the creation of mazes to be used for this report. Moreover, Matplotlib will also be used to plot the necessary graphs such as Time taken and Number of Paths taken, which is to be used for the comparison between the algorithms. Besides that, the console of each program also displays the ‘Average Time Taken’ and ‘Average Number of Paths’ on the console, which will give more reliable data due to having a sample size of 500 maze simulations used for each algorithm, which will be plotted on the appropriate graphs. Lastly, the conclusion reached by this report is that the A* search algorithm is the more optimal algorithm among the three as it provides an accurate shortest number of paths taken while also traversing a short number of paths to reach this shortest path taken. Thus, this implies that the A* search algorithm has the best performance and efficiency among the three and therefore is the optimal path finding search algorithm of the three in this report.

Introduction:

Search algorithms are computational techniques which are used to traverse or explore a specified space, such as a maze in this report, to reach a certain goal. A few examples of search algorithms being used in various fields are navigation applications such as Google Maps, AI-based games in which positioning is crucial such as puzzle solving and routing systems used by industrial logistics to find efficient transportation routes.

These algorithms are fundamental to various fields of computer science and are particularly essential in the domain of artificial intelligence (AI). As mentioned by MdRafiAkhtar and MayankAggarwal [1], the importance of search algorithms in AI is due to the involvement of developing agents with the capability of acting rationally using search algorithms. Hence, these agents usually rely on implementing certain search algorithms as a basis to accomplish a particular objective.

This report aims to compare the performance and efficiency of three search algorithms, that being Breadth-first Search (BFS), Depth-first Search (DFS), and A-star Search (A*) and determine which search algorithm among the three is the most optimized and the reason for it. BFS and DFS are classified as ‘Uninformed Seach Algorithms’, while A* is classified as an ‘Informed Search Algorithm’.

As defined by the article ‘Comparative analysis of search algorithms’ [2], an ‘Uninformed Search Algorithm’, also known as blind or brute-force search, lacks details about steps that are needed to reach a certain goal. Uninformed searches instead prioritize the paths that seem the most promising without utilizing additional information such as state transitions, hence blind exploration and not considering the optimal path exploration. On the other hand, ‘Informed Search Algorithms’, also known as Heuristic search, implements a heuristic function to estimate the proximity of a given state to the goal state when solving a problem. As such, this makes informed searches more efficient as it relies on heuristic functions as a guide to reach a specified goal and not blindly exploring a given search.

To aid in this report, Python programs with the implementation of these three algorithms will be used, in addition to packages such as Pyamaze for maze visualization and Matplotlib for graph plotting. The motivation behind evaluating this topic is to determine and prove that an informed search

algorithm is more optimal and that A* Search is the best among the three search algorithms presented. Besides that, this report will also show the behaviour of each algorithm when searching as well as some performance evaluation. Hence, promoting further advancements in this field by showing the comparison of BFS, DFS and A* search algorithms, which could aid in refining already existing algorithms and thus addressing certain limitations of these algorithms. Below is an introduction and further explanation of the three algorithms used in this research.

Breadth-first Search Algorithm (BFS)

As defined in the article ‘Search Algorithms in AI’ [1], BFS is an algorithm used to navigate a tree or graph. It starts its search in the tree’s root (or a specified node in a graph, called a ‘search key’) and explores all nodes at the current level before progressing to the next level in the tree. BFS can be executed using a queue.

Suggested in the article ‘Comparative analysis of search algorithms’ [2], there are some factors that lead to BFS being usable. These factors are, when it is required to find the shortest path of a solution, when multiple solutions might be available and at least one of them is the shortest path, and when memory is not a concern. Examples of its application is for shared network, social networking websites and for testing if a graph is bipartite.

Depth-first Search Algorithm (DFS)

As explained in ‘Search Algorithms in AI’ [1], DFS is a method which is used to traverse and search a tree or graph. It starts at the root node and delves as deeply as possible along a branch before retracing its steps and moving on to explore the next branch in the tree. This method uses a last in-first out (LIFO) strategy, therefore a stack is used for its implementation.

The article ‘Comparative analysis of search algorithms’ [2] states that a factor which make DFS suitable is when memory is limited. However, it is not reliable in obtaining a solution as there is a chance of this algorithm getting stuck in an infinite loop which occurs when a cycle exists in the graph. Moreover, it is also not ideal to retrieve the shortest path in a graph due to its limitation of exploring an entire branch before moving on, thus it could encounter a path

which also reaches the goal but is not the shortest path possible. Example applications of DFS is for identifying loops in a graph and for finding the solution of a graph that only has a single solution.

A-Star Search Algorithm (A*)

A* is an informed search algorithm also known as Heuristic search, which implements a heuristic function to estimate the proximity of the goal as mentioned previously. Examples of heuristic functions are the Manhattan distance and Euclidean distance.

The equation $f(v) = h(v) + g(v)$ is the function for an A* search algorithm, as stated in ‘Path planning with modified a star algorithm for a mobile robot’ [3]. A heuristic function is usually represented as $h(v)$ in the A* search algorithm and the length of the path from its initial state to the current state is $g(v)$. During exploration, $f(v)$ will be the next cell that is to be explored in the graph. The real-world application of A* search algorithm is for traffic navigation, video games and path planning by unmanned vehicles.

Research Methodology:

The program implemented uses Python with additional libraries such as Matplotlib for graph plotting and Pyamaze for maze visualisation and creation. For this research, it focuses on identifying the average number of paths taken (path length), the average time taken (time complexity), and the shortest path taken for each algorithm and compares them to determine the efficiency. For the average number of paths taken and the average time taken, 500 simulated runs of a maze will be run to achieve a greater sample size than just a single maze run, thus having a more reliable result. Moreover, the results for the averages will be plotted on graphs using Matplotlib to have a better visualisation of the results. Lastly, the console will also display the average time taken and average number of paths taken so that an exact value can be seen clearly.

Besides that, a demo function is also created to show the traversal of each algorithm in a maze. The reason the visualization of mazes is excluded for the 500 simulated runs is to not overload the memory of the PC, hence causing lag and a long wait time for the result generation. In this demo, it will display the shortest path taken and the total paths explored in a maze by a given algorithm. Moreover, it will also display the step-by-step exploration paths taken by the algorithm as well as highlights the shortest path found by the algorithm on the maze. The combination of the step-by-step exploitation and highlighted shortest path found ensures that the maze displayed will be very clear and easy to follow.

Below will be the explanation of each algorithm (BFS, DFS and A*) and the functions implemented within the code to achieve the results:

Breadth-first Search Algorithm (BFS)

```
9  def BFS(m):
10     start = (m.rows, m.cols)
11     frontier = [start]
12     explored = [start]
13     bfsPath = {}
14     shortestPath = {}
15     num_explored_paths = 0 # Track the number of explored paths
16
17     while len(frontier) > 0:
18         currCell = frontier.pop(0)
19         num_explored_paths += 1 # Increment for each explored path
20
21         if currCell == (5, 5):
22             break
23
24         for d in 'ESNW':
25             if m.maze_map[currCell][d] == True:
26                 if d == 'E':
27                     childCell = (currCell[0], currCell[1] + 1)
28                 elif d == 'W':
29                     childCell = (currCell[0], currCell[1] - 1)
30                 elif d == 'N':
31                     childCell = (currCell[0] - 1, currCell[1])
32                 elif d == 'S':
33                     childCell = (currCell[0] + 1, currCell[1])
34                 if childCell in explored:
35                     continue
36                 explored.append(childCell)
37                 frontier.append(childCell)
38                 bfsPath[childCell] = currCell
39
40             # Backtrack to find the shortest path
41             fwdPath = {}
42             cell = (5, 5)
43             while cell != start:
44                 fwdPath[bfsPath[cell]] = cell
45                 cell = bfsPath[cell]
46
47             # Storing the total exploration path
48             totalPath = explored
49
50             return fwdPath, totalPath
51             # return num_explored_paths, fwdPath
```

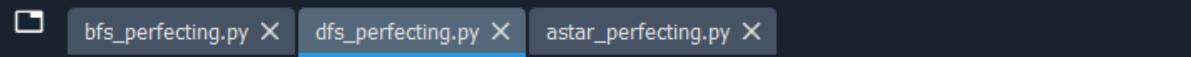
Figure 1: BFS Function Code

BFS Function Steps:

1. The function takes the maze size as an argument and sets the bottom-right corner of the maze as the starting point. It initializes the 'explored' and 'frontier' lists with the starting point and sets up other necessary variables.

2. The current cell is then set by popping the frontier list using ‘frontier.pop(0)’. Meaning it is a queue which uses a First-In-First-Out (FIFO) principle that is necessary for BFS searching.
3. If the current cell is the goal, it breaks from exploration. If it is not, it will continue to exploration.
4. During exploration using ESNW directions (east, south, north and west), it sets the ‘Child cell’ as the next cell. If the Child cell is already explored, the program will not do anything.
5. Then based on the ‘childCell’ position relative to the ‘currCell’, increases or decreases row or column according to that direction.
6. Append the visited cell (child cell) into the frontier and explored list.
7. Step 2 to 6 is repeated until the goal is reached.
8. The shortest path is then traced back.
9. Sets the total number of paths explored.
10. Returns the shortest path (fwdPath) and total paths explored (totalPath).

Depth-first Search Algorithm (BFS)



```
9  def DFS(m):
10     start = (m.rows, m.cols)
11     explored = [start]
12     frontier = [start]
13     dfsPath = {}
14     shortestPath = {}
15     num_explored_paths = 0 # Track the number of explored paths
16
17     while len(frontier) > 0:
18         currCell = frontier.pop()
19         num_explored_paths += 1 # Increment for each explored path
20
21         if currCell == (5, 5):
22             break
23
24         for d in 'ESNW':
25             if m.maze_map[currCell][d] == True:
26                 if d == 'E':
27                     childCell = (currCell[0], currCell[1] + 1)
28                 elif d == 'W':
29                     childCell = (currCell[0], currCell[1] - 1)
30                 elif d == 'S':
31                     childCell = (currCell[0] + 1, currCell[1])
32                 elif d == 'N':
33                     childCell = (currCell[0] - 1, currCell[1])
34                 if childCell in explored:
35                     continue
36                 explored.append(childCell)
37                 frontier.append(childCell)
38                 dfsPath[childCell] = currCell
39
40             # Backtrack to find the shortest path
41             fwdPath = {}
42             cell = (5, 5)
43             while cell != start:
44                 fwdPath[dfsPath[cell]] = cell
45                 cell = dfsPath[cell]
46
47             # Storing the total exploration path
48             totalPath = explored
49
50             return fwdPath, totalPath
51             # return num_explored_paths, fwdPath
```

Figure 2: DFS Function Code

DFS Function Steps:

1. The function takes the maze size as an argument and sets the bottom-right corner of the maze as the starting point. It initializes the 'explored'

and 'frontier' lists with the starting point and sets up other necessary variables.

2. The current cell is then set by popping the frontier list using 'frontier.pop()'. Meaning it is a stack which uses a Last-In-First-Out (LIFO) principle that is necessary for DFS searching.
3. If the current cell is the goal, it breaks from exploration. If it is not, it will continue to exploration.
4. During exploration using ESNW directions (east, south, north and west), it sets the 'Child cell' as the next cell. If the Child cell is already explored, the program will not do anything.
5. Then based on the 'childCell' position relative to the 'currCell', increases or decreases row or column according to that direction.
6. Append the visited cell (child cell) into the frontier and explored list.
7. Step 2 to 6 is repeated until the goal is reached.
8. The shortest path is then traced back.
9. Sets the total number of paths explored.
10. Returns the shortest path (fwdPath) and total paths explored (totalPath).

A-Star Search Algorithm (A*)

```
10     def h(cell1, cell2):
11         x1, y1 = cell1
12         x2, y2 = cell2
13         return abs(x1 - x2) + abs(y1 - y2)
14
15     def aStar(m):
16         start = (m.rows, m.cols)
17         g_score = {cell: float('inf') for cell in m.grid}
18         g_score[start] = 0
19         f_score = {cell: float('inf') for cell in m.grid}
20         f_score[start] = h(start, (1, 1))
21
22         open = PriorityQueue()
23         open.put((h(start, (1, 1)), h(start, (1, 1)), start))
24         aPath = {}
25         num_explored_paths = 0 # Track the number of explored paths
26
27         while not open.empty():
28             currCell = open.get()[2]
29             num_explored_paths += 1 # Increment for each explored path
30
31             if currCell == (5, 5):
32                 break
33
34             for d in 'ESNW':
35                 if m.maze_map[currCell][d] == True:
36                     if d == 'E':
37                         childCell = (currCell[0], currCell[1] + 1)
38                     if d == 'W':
39                         childCell = (currCell[0], currCell[1] - 1)
40                     if d == 'N':
41                         childCell = (currCell[0] - 1, currCell[1])
42                     if d == 'S':
43                         childCell = (currCell[0] + 1, currCell[1])
44
45                     temp_g_score = g_score[currCell] + 1
46                     temp_f_score = temp_g_score + h(childCell, (1, 1))
47
48                     if temp_f_score < f_score[childCell]:
49                         g_score[childCell] = temp_g_score
50                         f_score[childCell] = temp_f_score
51                         open.put((temp_f_score, h(childCell, (1, 1)), childCell))
52                         aPath[childCell] = currCell
53
54             fwdPath = {}
55             cell = (5, 5)
56             while cell != start:
57                 fwdPath[aPath[cell]] = cell
58                 cell = aPath[cell]
59
60             totalPath = list(aPath.keys()) # Storing the total exploration path
61
62             return fwdPath, totalPath
63             # return num_explored_paths, fwdPath
```

Figure 3: A* Function Code

A* Function Step:

1. A heuristic function is first defined as ‘ $h(\text{cell1}, \text{cell2})$ ’ which is used to estimate the distance between two cells. The method used for this calculation is the Manhattan Distance (sum of absolute differences in coordinates).
2. Initialize the scores and data structures by setting the starting state and initialize ‘ g_score ’ and ‘ f_score ’ dictionaries for each cell.
3. An open priority queue called ‘open’ is then initialized to store the cells explored.
4. Set the initial scores for the start cell in ‘ g_score ’ and ‘ f_score ’.
5. While the priority queue ‘open’ is not empty, get the cell with the lowest ‘ f_score ’ in the queue and increase the number of explore paths.
6. If the current cell is the goal state (5, 5), break from the exploration.
7. For loop is used with directions ESNW (east, south, north and west) to explore adjacent cells.
8. Checks the current cell ‘currCell’ has a valid path (‘True in ‘maze_map’’) for its respective direction. Then based on the ‘childCell’ position relative to the ‘currCell’, increases or decreases row or column according to that direction.
9. Calculate the temporary ‘ g ’ (distance from the start cell) and ‘ f ’ scores ($g + \text{heuristic } h$) for the child cell. Temp_g_score is used to calculate the cost from the start to child cell. Temp_f_score is used to add heuristic value (h) to the ‘ g ’ score for the child cell.
10. Update the scores and path by first comparing the new temp_f_score with the existing ‘ f ’ score of the child cell.
11. Update ‘ g ’ and ‘ f ’ scores for the child cell and it to the priority queue ‘open’ with the updated ‘ f ’ score.
12. Store the parent cell (currCell) of the child cell in the path dictionary ‘aPath’.
13. Step 4 to 12 is repeated until the goal is reached.
14. The shortest path is then traced back.
15. Sets the total number of paths explored.
16. Returns the shortest path (fwdPath) and total paths explored (totalPath).

Results and Discussions:

A 10x10 maze is used for the implementation of these algorithms with a ‘loopPercent=50’, which controls the complexity of the maze. ‘loopPercent=0’ having the least walls and ‘loopPercent=100’ having the most walls. For the results, the starting state is always at (10, 10) while the goal state is at (5, 5). The research will run 500 maze simulation, then generates the average time taken (time complexity) and average number of paths (path length) with the appropriate graphs being plotted, leading to a more reliable result due to the sample size used. Moreover, the maze demo will also display the shortest path in addition to the total paths explored in the maze display.

Maze with Starting State (1, 1) and Goal State (5, 5)

Below are the mazes (Figure 1.1, Figure 1.2 and Figure 1.3) when the starting state is (1, 1) and the goal state is (5, 5). The maze is seeded with ‘random.seed(1234)’, ensuring that the maze is the same (constant) for the comparison:

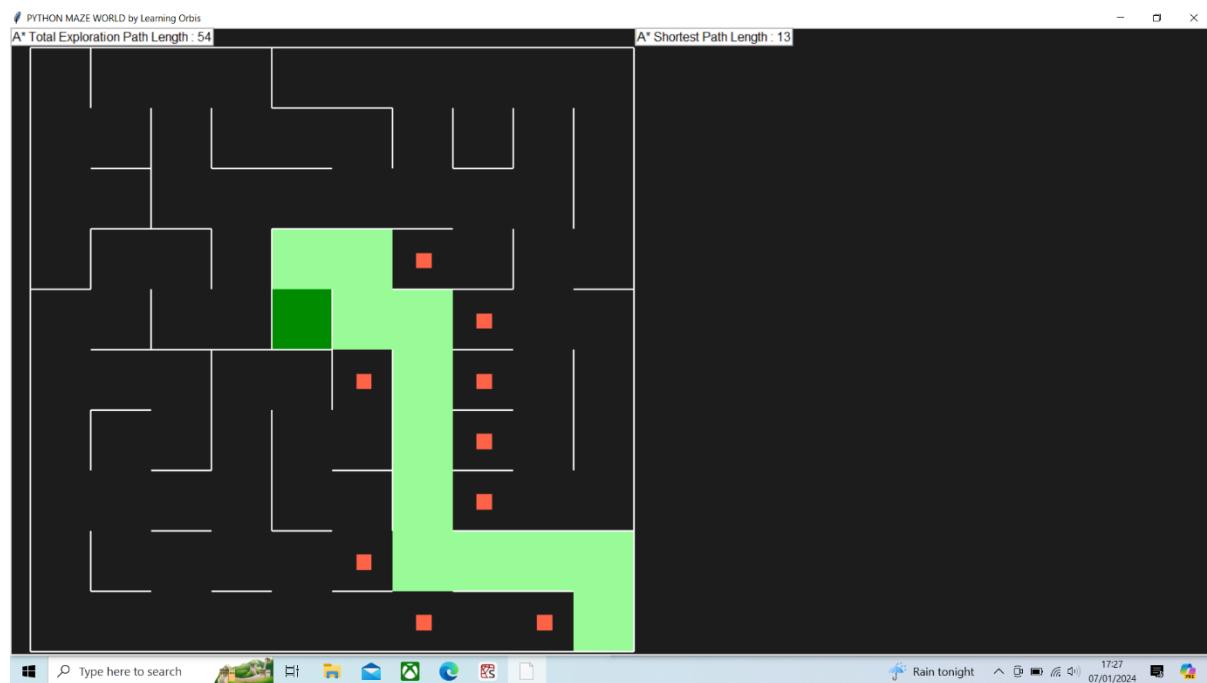


Figure 1.1: A* Maze Display

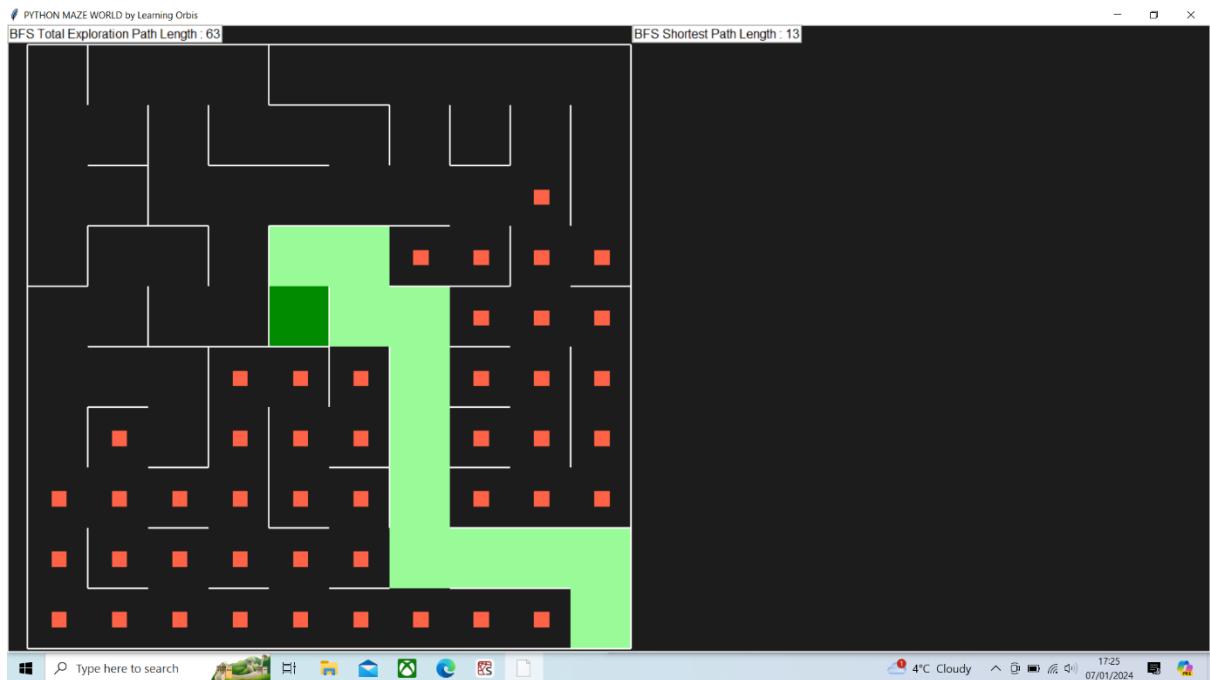


Figure 1.2: BFS Maze Display

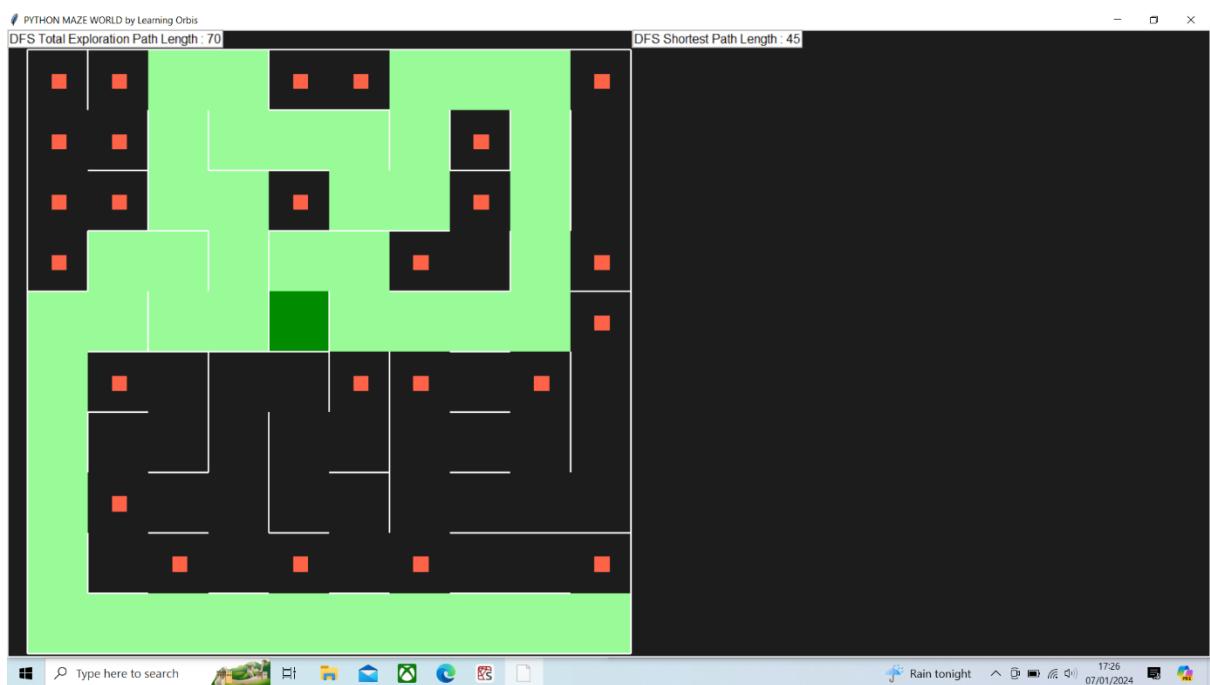


Figure 1.3: DFS Maze Display

The red dots are the paths which are explored during the search and the green path is the shortest path found by a given search algorithm. From the above mazes generated and the data provided, BFS and A* search algorithms have both found the shortest path of 13, while DFS has found the shortest path of

45. Besides that, A* also took the least number of paths to explore to reach the solution at 54 paths. BFS took 63 paths and DFS took 70 paths to reach the solution. However, even though DFS has reached the solution, it is not the shortest possible path. Whereas both BFS and A* have found the shortest path possible. Table 1 shows the results achieved from the demonstration above, using the goal state (5, 5).

Search Algorithm	Total Exploration Path Length	Shortest Path Length
BFS	63	13
DFS	70	45
A*	45	13

Table 1: Results from Maze demonstration with (5, 5) as goal state.

Time Taken:

This section will show the time complexity graph of the maze with the goal state (5, 5). Figure 2.1, Figure 2.2 and Figure 2.3 shows the graphs plotted with Matplotlib, showing the time taken for the program to run 500 maze simulations for a given algorithm.

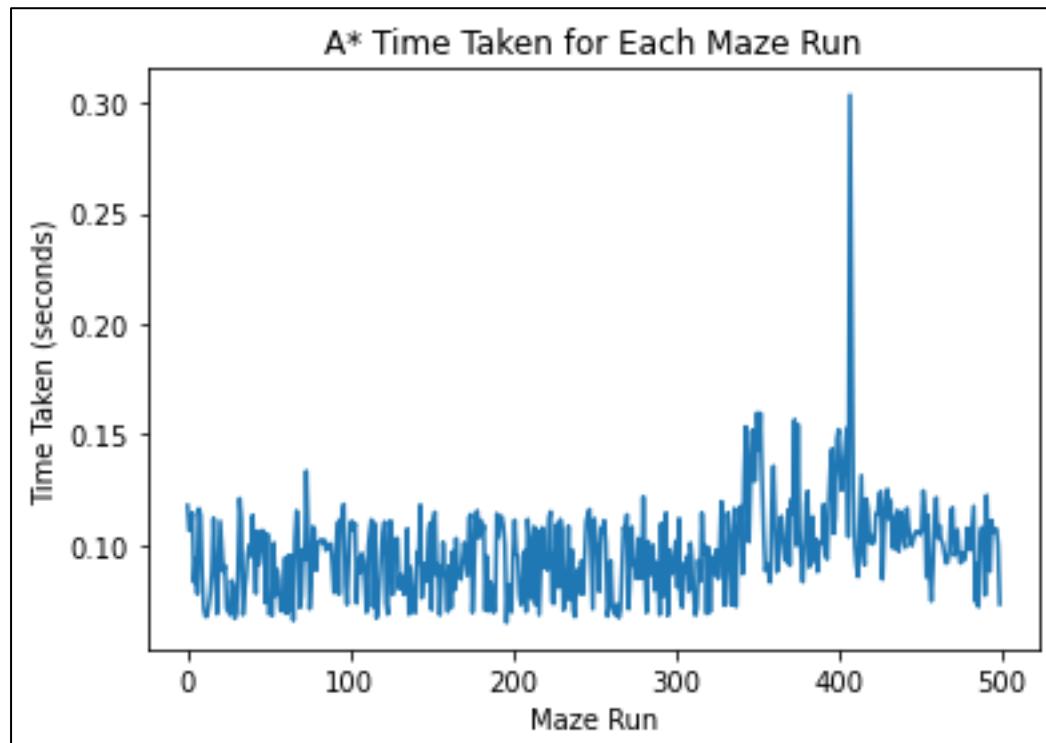


Figure 2.1: A* Time Taken Graph

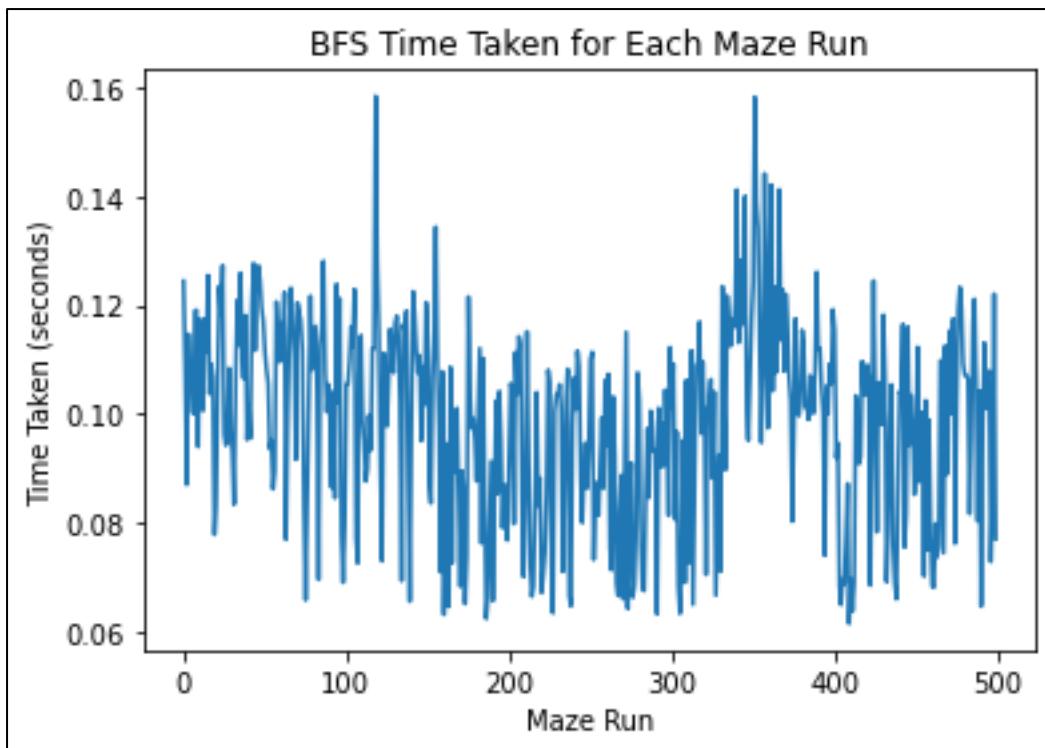


Figure 2.2: BFS Time Taken Graph

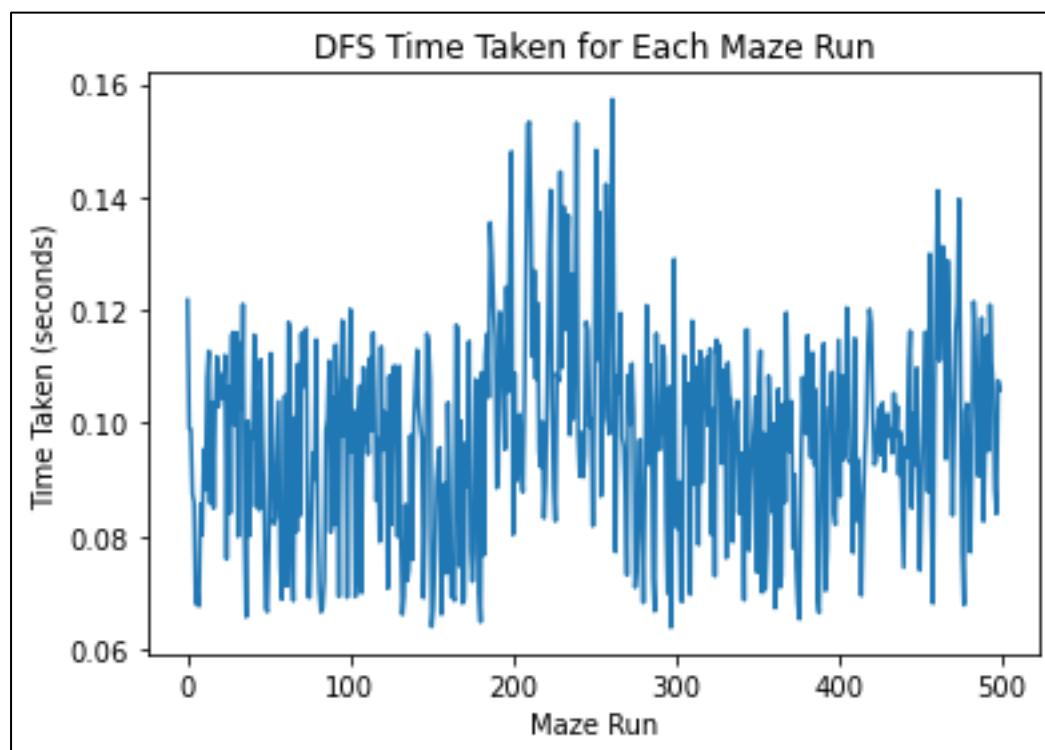


Figure 2.3: DFS Time Taken Graph

Total Number of Paths Taken:

This section will show the of the total number of paths taken graph of the maze with the goal state (5, 5). Figure 3.1, Figure 3.2 and Figure 3.3 shows the graphs plotted with Matplotlib, showing the total number of paths when the program to run 500 maze simulations for a given algorithm.

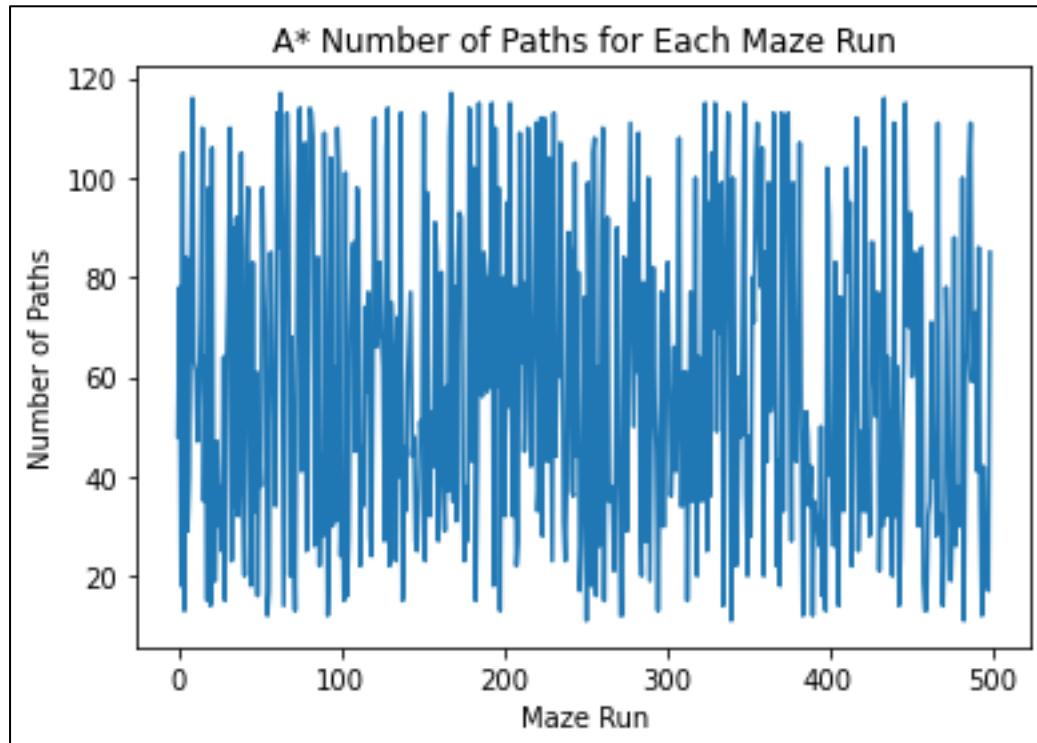


Figure 3.1: A* Number of Paths Taken Graph

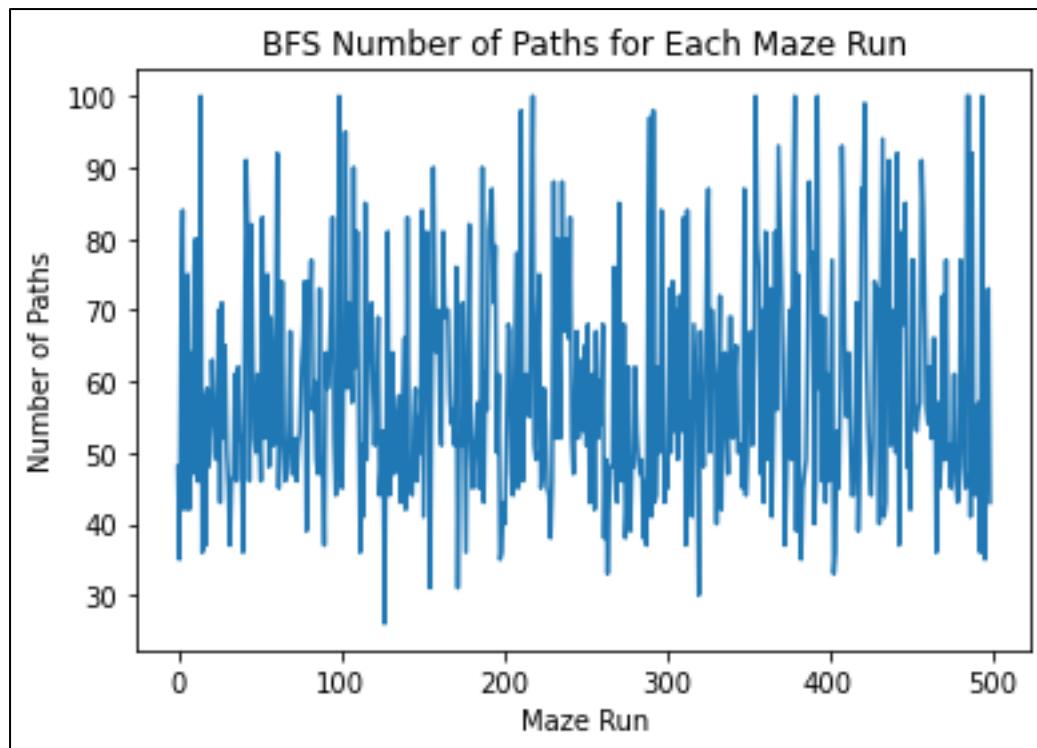


Figure 3.2: BFS Number of Paths Taken Graph

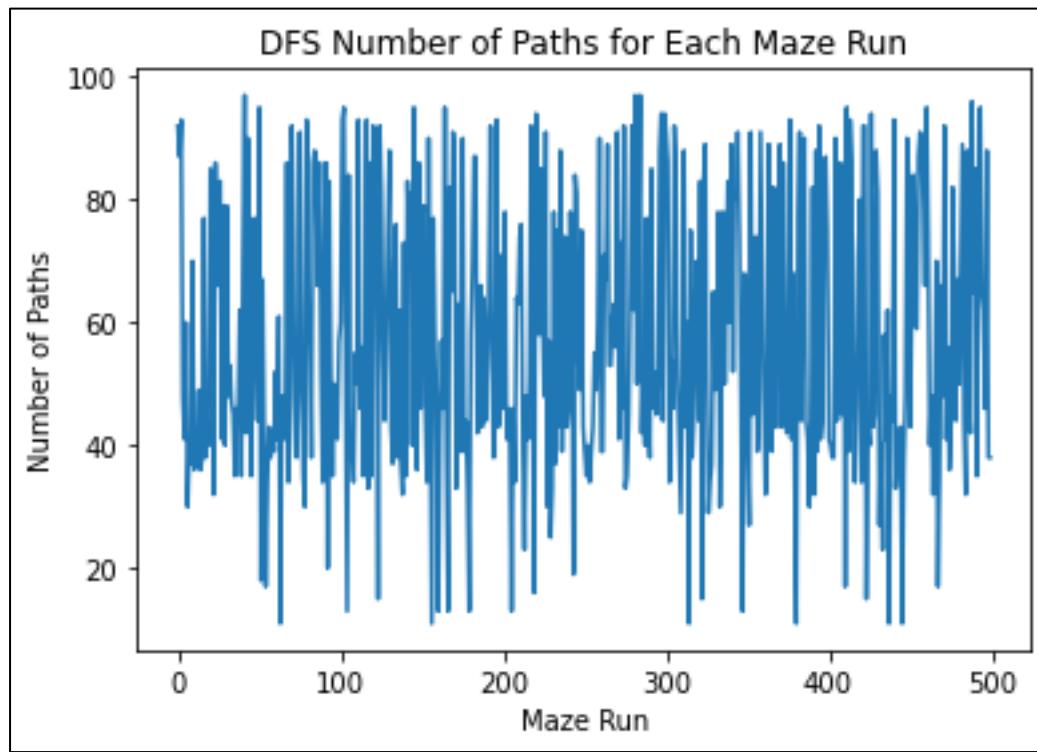


Figure 3.2: DFS Number of Paths Taken Graph

Discussion:

```

Help Variable Explorer Plots Files
Console 5/A X
Important
Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck "Mute inline plotting" under the options menu of Plots.

BFS Average time taken for 500 maze runs: 0.09874 seconds
BFS Average number of paths explored: 59.23

In [2]: runfile('C:/Users/Daniel/Desktop/Teesside Uni/Year 2/Artificial Intelligence/dfs_perfecting.py', wdir='C:/Users/Daniel/Desktop/Teesside Uni/Year 2/Artificial Intelligence')
Reloaded modules: pyamaze
DFS Average time taken for 500 maze runs: 0.09782 seconds
DFS Average number of paths explored: 57.61

In [3]: runfile('C:/Users/Daniel/Desktop/Teesside Uni/Year 2/Artificial Intelligence	astar_perfecting.py', wdir='C:/Users/Daniel/Desktop/Teesside Uni/Year 2/Artificial Intelligence')
Reloaded modules: pyamaze
A* Average time taken for 500 maze runs: 0.09741 seconds
A* Average number of paths explored: 58.91

In [4]:
IPython Console History
conda (Python 3.11.5) Completions: conda LSP: Python Line 81, Col 5 ASCII CRLF RW Mem 68%

```

Figure 4: Average Time Taken and Number of Paths for all three algorithms from 500 Simulations.

Search Algorithm	Average Time Taken (seconds)	Average Number of Paths Explored
BFS	0.09874	59.23
DFS	0.09782	57.61
A*	0.09741	58.91

Table 2: Results from 500 Simulations in a 10x10 Maze (Figure 4)

Figure 4 shows the average time taken and average number of paths that were achieved from the graphs plotted previously in the console of the program, while Table 2 is the table of these results for clarity. The reason average time taken, and average number of paths explored are used as a comparison metrics is due to them measuring the time complexity and total path length respectively.

The time complexity is used to measure the efficiency of a given algorithm when completing the task, a shorter time indicates that the algorithm is faster to execute and reach the goal. Total path length is relevant as it reflects the exploration algorithm as fewer paths explored indicates a more direct and efficient traversal through. However, this does not always indicate whether an algorithm is optimal for searching as it does not guarantee the shortest path has been found.

From the above data and results, A* is shown to take the least amount of time on average to complete the maze with 0.09741 seconds, while BFS can be seen to take the longest with 0.09874 seconds. The reason for A* being the fastest among the three could be attributed to the fact that it has a heuristic function implemented, which helps and informs it of the position of the goal beforehand to plan its path. However, both BFS and DFS are uninformed searches and therefore rely on brute force to solve the maze without any aid. BFS also takes longer than DFS, as BFS systematically explores all nodes at each given depth level before moving on to the next whereas DFS explores as far as possible before moving on to the next branch, thus it may be possible for DFS to find the goal faster.

For the average number of paths explored, DFS takes the least amount of exploration to reach the goal on average with 57.61 paths while BFS takes the most amount of exploration with 59.23 paths, with A* being in the middle with 58.91 paths. Even though DFS takes the least amount of exploration to reach the goal, it does not guarantee it finding the shortest path possible in a maze, unlike BFS and A* due to its limitation of exploring only a few branches in total compared to BFS which explores all branches and A* having a heuristic approach. This can be seen in the previous demonstration in Figure 1.1, Figure 1.2 and Figure 1.3 where both BFS and A* achieved the same shortest path of 13, while DFS did not and only achieved the shortest path of 45.

Besides that, it can also be proposed that A* and BFS take up more memory when executing due to maintaining the explored paths, while DFS generally uses less memory as it explores deeply before backtracking. This leads to DFS being the preferred algorithm when the shortest path in a maze is not necessary, but only the lowest total number of paths needed for exploring a maze and time is not an issue.

Conclusion:

To conclude, after taking to benefits and limitations of all three algorithms, A* search algorithm is the most optimal one among the three. As it takes less time to execute and reach the solution, while only taking a reasonable amount of exploration to reach the goal. Moreover, it will always achieve the shortest path possible in each maze due to its heuristic implementation. BFS is not optimal as it takes more time and memory compared A* as it lacks any heuristic implementation to help it, which leads it having to explore more and thus taking up more memory in comparison. On the other hand, DFS may take less exploration paths and is the most memory efficient compared to A* and BFS, it does not guarantee the shortest path possible thus making it unoptimized.

Therefore, even though A* may require higher computational overhead due to the heuristic function implementation, it is more scalable than BFS and DFS due to its heuristic nature, making it suitable for even large mazes. Besides that, it always guarantees the shortest path taken, making it a compelling choice for scenarios where finding the best solution is crucial.

Future areas for research that could be done from the conclusion that A* search algorithm is the most optimized among the three is to further optimize it. From the article ‘A comparative study of A-star algorithms for search and rescue in perfect maze’ [4], modifications are made to the A* search algorithm to further improve the performance in pathfinding. These modifications aim to enhance A* by reducing the number of evaluated cells, optimizing the path planning and improving computational efficiency. The techniques implemented are 8-connectivity restriction, extension to every angle searching, rectangular symmetry reduction (RSR) and jump point search (JPS).

Personal Reflection:

Doing research about this topic has allowed me to further understand search algorithms and its technical implementation using Python. Moreover, from the articles and journals I have used in this research, it has shown me the many real-world applications of these three search algorithm. It has also shown me that even though BFS and DFS are not optimal, there are still situational and niche applications for them as mentioned in the introduction of this report. As such, it has made me realized that Artificial Intelligence (AI) has a wide spectrum of applications, which search algorithms such as BFS, DFS and A* play a pivotal role in various domains. This research has further my curiosity for AI and wanting to delve deeper into the ever changing landscape of AI and the many algorithms used, inspiring me to explore more complex problem solving methodologies and their real world implications.

References:

- [1] MdRafiAkhtar and MayankAggarwal. (Last Updated: 22 March 2023). “Search Algorithms in AI”, GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/search-algorithms-in-ai/> (Accessed: 24 December 2023)
- [2] Pathak, M.J., Patel, R.L. and Rami, S.P., 2018. Comparative analysis of search algorithms. International Journal of Computer Applications, 179(50), pp.40-43. Available at: https://www.researchgate.net/profile/Ronit-Patel-2/publication/333262471_Comp...Comparative-Analysis-of-Search-Algorithms.pdf (Accessed: 24 December 2023)
- [3] Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Florek, M., Fico, T. and Jurišica, L., 2014. Path planning with modified a star algorithm for a mobile robot. Procedia engineering, 96, pp.59-69. Available at: <https://www.sciencedirect.com/science/article/pii/S187770581403149X> (Accessed: 24 December 2023)
- [4] Liu, X. and Gong, D., 2011, April. A comparative study of A-star algorithms for search and rescue in perfect maze. In 2011 international conference on electric information and control engineering (pp. 24-27). IEEE. Available at: <https://ieeexplore.ieee.org/abstract/document/5777723> (Accessed: 24 December 2023)