Introduction and Analysis of Graph Search Algorithms: Depth-First Search, Breadth-First Search, and A* Search in Maze Solving

Leo Benjamin Conoy

Student Number: B1055988

Artificial Intelligence ICA

CIS2031-N

Total Words: 3924 Words

Introduction

The purpose of this report is to delve into the workings and comparative analysis of three widely used graph search algorithms: Depth-First Search (DFS), Breadth-first search (BFS), and A* search. These algorithms have extensive applications in various fields including Artificial Intelligence (AI), where they are used for tasks such as pathfinding, game state exploration, and decision making. By implementing these algorithms in a consistent test environment, a maze solving scenario, we aim to highlight their unique characteristics and compare their performance based on parameters like path length, search length and execution time.

Overview of the algorithms

- *Depth first search (DFS)*
  DFS is a fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking (Rahat, 2023). It uses a last in first out stack to remember to get the next vertex to start a search when dead end occurs in any iteration. Although it does not guarantee the shortest path, it is usually simple and quick.

- *Breadth first search (BFS)*
  BFS explores all the neighbour nodes at the present depth before moving onto nodes at the next depth level. It uses a first in first out queue for its operations, thus ensuring that the shortest path is found, when traversing unweighted graphs. BFS is typically more memory intensive than DFS (GeeksforGeeks, 2019).

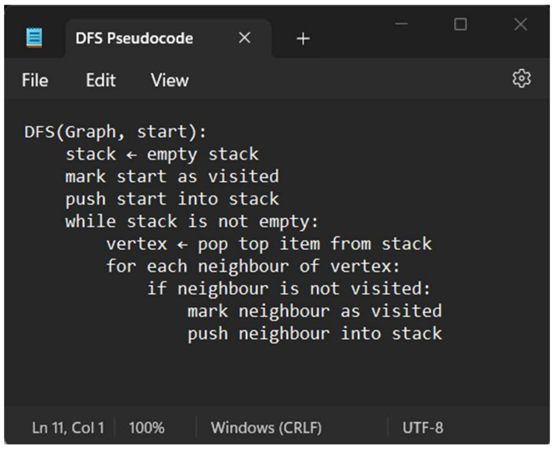
- *A* search*
  A* search is a heuristic driven algorithm and used commonly in pathfinding and graph traversal. The core idea behind a star search is to minimise the total cost while reaching the goal. It uses a best first search and finds the least cost path from a given initial node to one goal node. As a result, it is more efficient than other simple traversal methods providing the optimal solution given a suitable heuristic (Belwariar, 2018).

Background/Theory

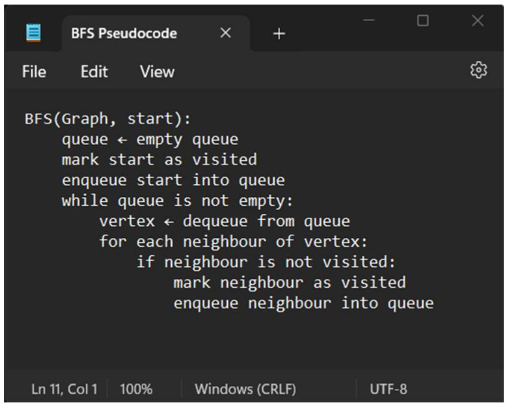
Graph theory, a fundamental topic in mathematics, studies graphs which are structures used to model relationships between objects. A graph comprises vertices (or nodes) and edges connecting them. Search algorithms use these structures to traverse and find specific paths or nodes.

Depth first search (DFS)

DFS is a popular algorithm in graph theory used to traverse or search through a graph in a depth-ward motion. The Depth-First Search method meticulously investigates each branch of the node to its furthest extent prior to initiating a backtrack. Its complexity can be defined as O(V+E), where V represents the count of vertices and E denotes the number of edges. DFS is commonly used for problems such as topological sorting, detecting cycles, and pathfinding in maze puzzles.



```
DFS(Graph, start):
    stack ← empty stack
    mark start as visited
    push start into stack
    while stack is not empty:
        vertex ← pop top item from stack
        for each neighbour of vertex:
            if neighbour is not visited:
                mark neighbour as visited
                push neighbour into stack
```



BFS Pseudocode

```
BFS(Graph, start):
    queue ← empty queue
    mark start as visited
    enqueue start into queue
    while queue is not empty:
        vertex ← dequeue from queue
        for each neighbour of vertex:
            if neighbour is not visited:
                mark neighbour as visited
                enqueue neighbour into queue
```

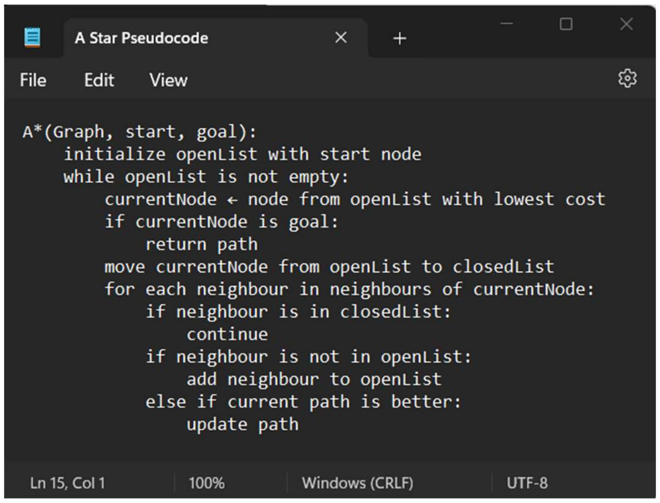
Breadth first search (BFS)

BFS is another foundational graph search algorithm that traverses the graph in a breadth-ward motion and uses a queue to remember to get back to the nodes to explore their neighbours, ensuring that vertices closer to the source are explored before those further away. Similarly to the DFS algorithm, the time complexity is also O(V+E). BFS is particularly useful for shortest path problems in unweighted graphs and cycle detection.

A* Search

A* search algorithm, unlike BFS and DFS, uses a best-first, greedy approach and includes a heuristic component. A* chooses which vertex to explore by considering the cost to reach the vertex and the cost to get from that vertex to the goal, thus finding an optimal path from source to goal. The time complexity of A* depends on the heuristic used. It is widely used in pathfinding problems, especially in gaming, robotics, and maps.

These algorithms have their specific strengths and are chosen based on the nature of the problem at hand.



A Star Pseudocode

```
A*(Graph, start, goal):
    initialize openList with start node
    while openList is not empty:
        currentNode ← node from openList with lowest cost
        if currentNode is goal:
            return path
        move currentNode from openList to closedList
        for each neighbour in neighbours of currentNode:
            if neighbour is in closedList:
                continue
            if neighbour is not in openList:
                add neighbour to openList
            else if current path is better:
                update path
```

<u>Setting up our Maze</u>

I began by setting up the maze in which I would run the code demonstrating the algorithms. The code starts by defining the size of the maze using the variables *'maze_width'* and *'maze_height'*. In this case, both the width and height of the maze are set to 25 creating a square maze with a total of 625 cells.

The next step is creating the maze object using the 'maze' class from the *'pyamaze'* library. The *'CreateMaze'* method is then called on this object, generating a random maze of the specified size.

The *'CreateMaze'* method is also passed a *'loopPercent'* argument with a value of 75. This argument specifies the percentage of extra paths, or loops, to create within the maze beyond the one correct path. Loops provide additional complexity and challenge for the agent by presenting it with more potential paths to explore.

The *'saveMaze'* argument in *'CreateMaze'* method is set to True. This means that the generated maze will be saved as a file, in this case CSV, allowing you to reuse the same maze layout in the future when desired.

Following this, an agent is created using the *'agent'* class from the *'pyamaze'* library. The agent is initialised with the *'footprints'* parameter set to True, which means that the agent will leave a visible trail or 'footprints' behind it as it moves through the maze. This can be useful for visualising the path the agent takes.

The shape and colour parameters are used to customise the appearance of the agent. In this case, the agent will appear as a red arrow.

The *'tracepath'* method is used to let the agent follow the path of the maze with a delay of one hundred milliseconds between each movement.

Finally, the *'run'* method is used to start the simulation allowing the agent to begin navigating through the maze.

This setup allows for a great deal of flexibility and customization, from the size and complexity of the maze to the appearance and behaviour of the agent.

<u>The DFS Algorithm</u>

Now the maze has been created we are now ready to begin coding our first algorithm. I began with the DFS algorithm. The DFS algorithm is broken in to two parts, the DFS setup and the Main Script.

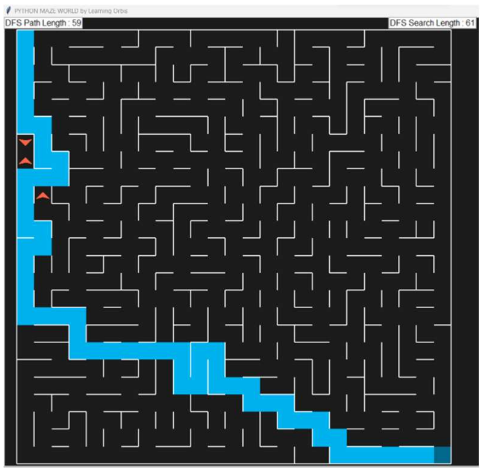


Figure 1 - Visualisation of the DFS Algorithm Search

The DFS algorithm is defined in the function '*DFS(m)'*. It starts by initialising the *'start'* position as the bottom right corner of the maze. The *'explored'* list is to keep track of the cells we have already visited, and *'frontier'* stack maintains cells we plan to visit (in DFS we use the stack data structure). *'dfsPath'* dictionary is used to maintain the path from each node to its parent, and *'dSearch'* list keeps track of the order in which nodes are visited. The algorithm then enters a loop that continues until all reachable nodes are explored. It pops the last node from the stack (according to the depth-first principle), checks all possible directions (East, West, North, South) from the current cell, and if the cell in a given direction is not a wall and hasn't been explored yet, it gets added to the *'explored'* list and 'frontier' stack, and the path is updated. This process continues until the goal (top left corner) is reached. After the loop, the shortest path from start to the goal is constructed and returned along with the search path and the full exploration path.

The main script begins by initialising a 25x25 maze and creating the maze layout from our CSV file. The DFS algorithm is then run on this maze, returning three lists containing the order of visited nodes (*'dSearch'*), the parent child relationships between nodes (*'dfsPath'*) and the shortest path to the goal (*'fwdPath'*). Three agents are initialised with the specific properties (including footprints, colour, shape, and goal location) and are made to trace the paths derived from the DFS function with different delays. Additionally, text labels are added to the maze display, showing the length of the DFS path and the length of the DFS search sequence.

When the script is run, it visualises the process of the DFS algorithm solving the maze. It shows the path taken by each agent (representing the different paths derived from the DFS function) and displays the statistics about the path lengths. The visualisation ends when all agents have reached their goal. The time it takes for the visualisation to complete will depend on the size and complexity of the maze and the delay set for each agent's movement. Please note in this code agent *'a'* traces the full search path (showing the exploration of the DFS algorithm), agent *'b'* traces the parent child path (showing the relationship between nodes), and agent *'c'* traces the shortest path from start to goal.

The BFS Algorithm

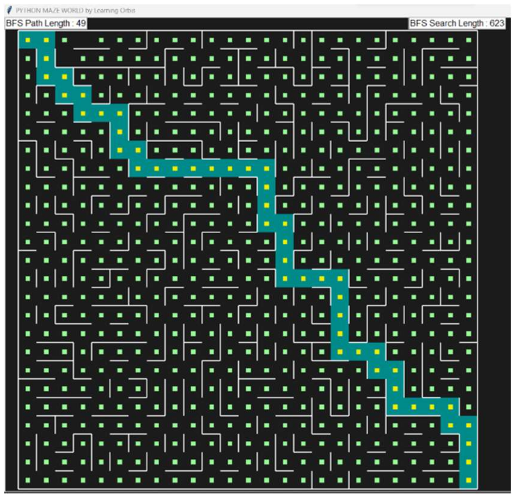


Figure 2 - Visualisation of the BFS Algorithm Search

We then move on to the BFS algorithm. This algorithm is implemented in the *'BFS(m, start=None)'* function. The algorithm begins at the start node which, if not explicitly provided, defaults to the bottom-right corner of the maze. The nodes to be explored or stored in *'frontier'*, a double-ended queue (deque), and initially contains the start node. *'bfsPath'* is a dictionary that keeps track of the path from each node to its parent, while *'explored'* is a list of nodes already visited. *'bfsSearch'* is used to record the order of visited nodes. The algorithm proceeds by dequeuing a node from *'frontier'*, then explores its neighbours in all possible directions (East, West, North, South). If a neighbour has not been visited yet (and is not a wall), it is added to *'frontier'* and marked as explored. The algorithm continues until it dequeues and processes the goal node in this case the top-left corner of the maze. Afterwards, it constructs the shortest path from the start to the goal, which, along with the complete exploration path and the order of visited nodes, is returned by the function.

The main script creates a 25x25 maze from our CSV file, then applies the BFS function to it, resulting in three lists: *'bfsSearch'* (the order of visited nodes), *'bfsPath'* (the parent-child relationships between nodes), and *'fwdPath'* (the shortest path to the goal). Three agents are created with distinct properties (including footprints, shape, colour, and goal location), and each follows one of the paths derived from the BFS algorithm. Additionally, text labels are added to the maze display, showing the length of the BFS path and the length of the BFS search sequence.

When the script is run it visualises the process of the BFS algorithm solving the maze. The path taken by each agent (corresponding to the different paths obtained from the BFS function) is shown, and statistics about the path lengths are displayed. The visualisation ends when all agents have reached their goal. The time it takes for the visualisation to complete will depend on the size and complexity of the maze and the delay set for each agent's movement. Please note in this code agent *'a'* traces the full search path (showing the exploration of the BFS algorithm), agent *'b'* traces the shortest path from start to goal, and agent *'c'* traces the parent-child path (showing the relationship between nodes).

The A* Algorithm

Finally, we move on to the A* algorithm. The A* algorithm is set up in the *'aStar(m, start=None)'* function. If no start node is provided, the function uses the bottom-right corner of the maze as the start node. The algorithm works by maintaining a priority queue of cells to be explored prioritising cells with a lower 'cost'. In the context of this algorithm the cost of a cell is calculated as *'f_score',* which is the sum of *'g_score'* (the cost from the start to the current cell) and *'h()'* (the estimated cost from the current cell to the goal, computed using a heuristic function). The heuristic function used in this script is the Manhattan distance, which is suitable for grid-based pathfinding where movement is allowed only in four directions (East, West, North, South).

The algorithm dequeues a cell from the priority queue, then explores its neighbours. For each neighbour, it calculates a tentative *'g_score'* and *'f_score'*. If the *'f_score'* for a neighbour is lower than the current *'f_score'* of the neighbour, the algorithm updates the path and the scores for the neighbour. The algorithm continues until it dequeues and processes the goal node (top-left corner of the maze). Afterwards the algorithm constructs the shortest path from the start to the goal, and this, along with the complete exploration path and the order of visited nodes, is returned by the function.

The main script creates a 25x25 maze from our CSV file, then applies the A* function to it. This results in three paths: *'searchPath'* (the order of visited nodes), *'aPath'* (the parent-child relationships between nodes), and *'fwdPath'* (the shortest path to the goal). Three agents are then created with distinct properties, and each follows one of the paths derived from the A* function. Text labels are also added to the maze display, showing the length of the A* path and the length of the A* search sequence.

When the script is run it visualises the process of the A* algorithm solving the maze. The path taken by each agent (corresponding to the different paths obtained from the A* function) is shown, and statistics about the path length are displayed. The visualisation ends when all agents have reached their goal. The time it takes for the visualisation to complete will depend on the size and complexity of the maze and the delay set for each agent's movement. In this code agent *'a'* traces the full search path (showing the exploration of the A* algorithm), agent *'b'* traces the parent-child path (showing the relationship between nodes), and agent *'c'* traces the shortest path from start to goal.

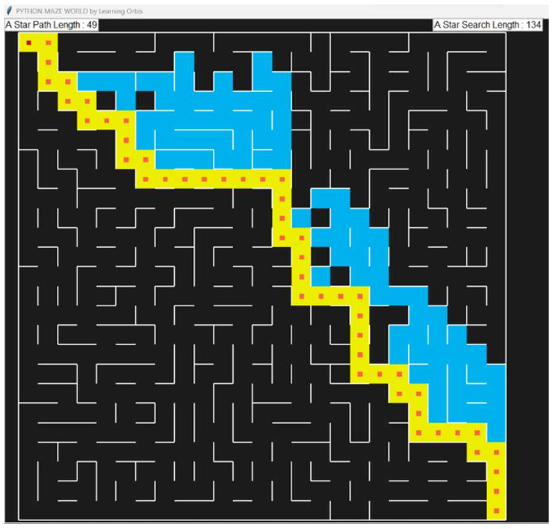

PYTHON MAZE WORLD by Learning Orbis
A Star Path Length : 49
A Star Search Length : 134

Figure 3 - Visualisation of the A Algorithm Search*

Preliminary Results

With the incorporation of text labels in our code, we are now able to analyse and compare the path lengths and search lengths of the three algorithms that were implements. Below is a summary of the results:

Algorithm	Path Length	Search Length
DFS	59	61
BFS	49	623
A*	49	134

Analysing these results reveals some interesting insights. The DFS search length is significantly smaller than that of BFS and less than half of the A* search length. However, DFS's path length is ten cells longer than both BFS and A*, making it less efficient in finding the shortest path, despite its more efficient search process.

On the other hand, BFS finds the optimal path length, though at the cost of a larger search length compared to both DFS and A*. The A* algorithm successfully finds the optimal path like BFS but uses significantly fewer search steps, suggesting it is more efficient than BFS in this scenario.

Nevertheless, in terms of speed, DFS outperforms the others due to its lower search length. To further investigate the trade-offs between efficient, path optimality and speed, I plan to perform more direct comparisons, particularly between DFS and BFS, as well as between BFS and A*.

DFS and BFS Comparison

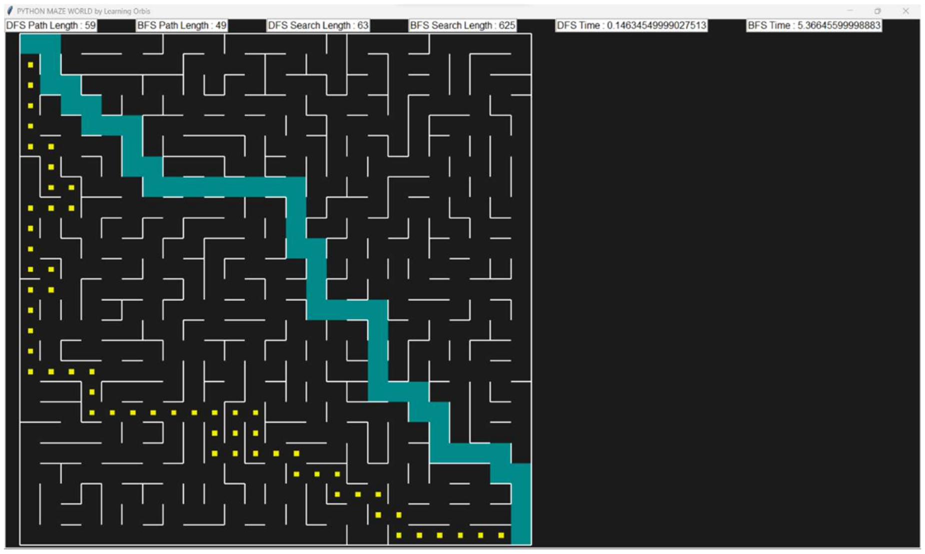
I began by examining a comparison between the DFS and BFS algorithms. I started my coding process by importing the necessary modules, including the DFS and BFS implementations, maze, and agent from the Pyamaze library for maze generation and visualisation, and the timeit function for measuring execution times.

I then created the maze instance using the *'maze'* function from the Pyamaze library. In this case, I continued to use the same maze that I had used for running all instances of the previous visualisations and loaded the maze from the same CSV file. This was loaded using the *'CreateMaze'* method of the maze instance. The script then applied both the DFS and BFS algorithms on the maze by calling the *'DFS'* and *'BFS'* functions, respectively. Both functions return the search path, the path from start to goal, and the path from goal to start.

The script displayed the path lengths (both forward and search path) found by the DFS and BFS algorithms on the maze using the text label function. Two agents are also created for visualising the paths found by the DFS and BFS algorithms. These agents are different colours for distinction. The paths found by the DFS and BFS algorithms are traced by the agents using the *'tracePath'* method of the maze instance.

The script measures and displays the execution times of the DFS and BFS algorithms by using the *'timeit'* function. And finally, the visualisation of the maze and the paths traced by the agents begin with the *'run'* method of the maze instance.

When run this code visually compares the paths generated by the DFS and BFS algorithms on the same maze, providing a clear demonstration of the difference between the two approaches. It also measures and displays the execution times of the algorithms, providing insight into their respective efficiencies. The path lengths and search lengths of each algorithm are also displayed, offering a comparison of how much of the maze was explored by each algorithm and the length of the path they found.



DFS Path Length : 59 BFS Path Length : 49 DFS Search Length : 63 BFS Search Length : 625 DFS Time : 0.14634549999027513 BFS Time : 5.36645599998883

Figure 4 - Visualisation of the DFS vs BFS search algorithms

BFS vs A* Comparison

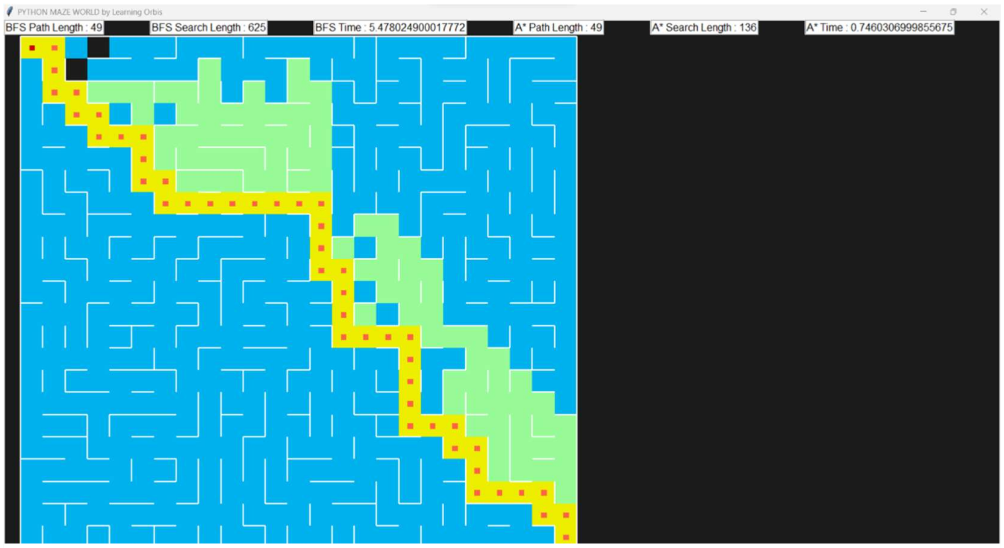
Following the previous comparison, I then moved on to a comparison between BFS and A* algorithms, I similarly began by importing the necessary modules, including the BFS and A* algorithm implementations, the Pyamaze library for creating and visualising mazes, and the timeit function for measuring execution times.

The maze instance was also created using the 'maze' function from the Pyamaze library and the layout was again loaded from the CSV file that had been previously created, using the 'CreateMaze' method of the maze instance.

The BFS algorithm was run on the maze by calling the *'BFS'* function which returns the search path, path from start to goal, and path from goal to start. The script displayed the path lengths (both forward and search path) found by the BFS algorithm on the maze using the *'textLabel'* function. The execution time for the BFS algorithm is also measured using the *'timeit'* function and displayed in a text label. Two agents are created for visualising the paths found by the BFS algorithm. These agents are different colours for distinction. The paths found by the BFS algorithm are traced by the agents using the *'tracePath'* method of the maze instance.

Like the BFS algorithm, the A* algorithm is run on the maze using the *'aStar'* function, which returns the search path, path from start to goal, and path from goal to start. The script also displays the path lengths (both forward and search path) found by the A* algorithm on the maze using the *'textLabel'* function, the execution time is also displayed using this function and measured using the *'timeit'* function. Two agents are also created for visualising the paths found by the A* algorithm and again use different colours for distinction. The paths found by the A* algorithm are traced by the agents using the *'tracePath'* method on the maze instance. Finally, the visualisation of the maze and paths traced by the agents, begins with the *'run'* method of the maze instance.

Once run, this code visually compares the paths generated by the BFS and A* algorithms on the same maze, providing a clear demonstration of the difference between the two approaches. It also measures and displays the execution times of the algorithms, providing insight into their respective efficiencies. The path lengths and search lengths of each algorithm are also displayed, offering a comparison of how much of the maze was explored by each algorithm and the length of the path they found.



BFS Path Length : 49 BFS Search Length : 625 BFS Time : 5.478024900017772 A* Path Length : 49 A* Search Length : 136 A* Time : 0.7460306999855675

Figure 5 - Visualisation of the BFS vs A search algorithms*

Comparison Results

Now that we have results from the comparisons, we can begin to compare each of the algorithms. We will look at path length, search length, and execution time.

Algorithm	Path Length	Search Length	Execution Time
DFS	59	63	0.146
BFS	49	625	5.366
A*	49	136	0.746

Path length

The path length represents the number of steps taken from the start to the goal. The BFS and A* algorithms have found the shortest path with a length of 49. In contrast, DFS has a longer path length of 59. It is a common characteristic of DFS to not always find the shortest path in a graph-like structure such as a maze.

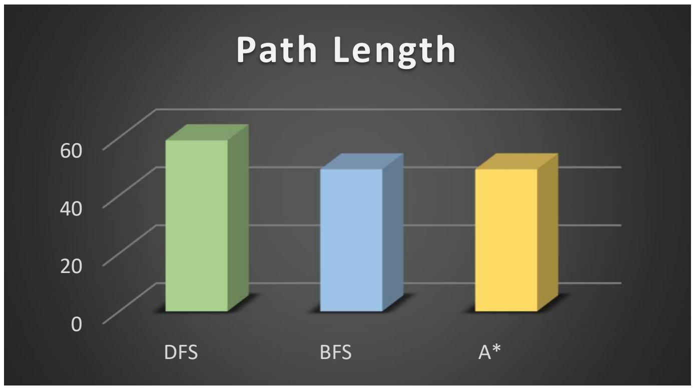


Figure 6 - Graph showing a comparison of the Path Lengths between each algorithm

Search length

Search length is indicative of how many steps the algorithm took to explore and find the goal. In this context, BFS has searched a significantly larger part of the maze (625 cells) compared to DFS (63 cells) and A* (136 cells). This is because BFS explores all neighbouring cells at the present depth before moving onto cells at the next depth level, resulting in a thorough, but time-consuming, search.

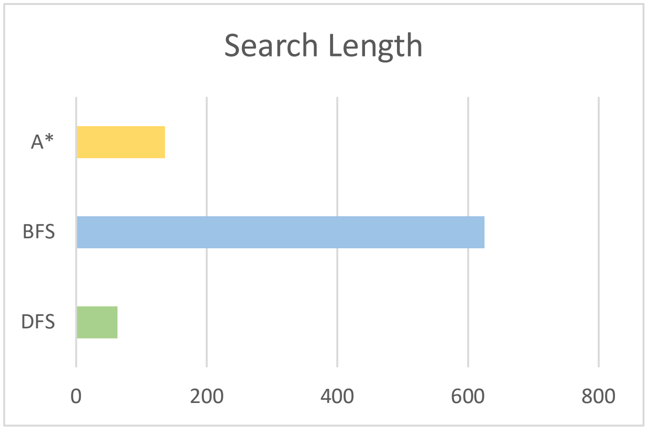


Figure 7 - Graph showing a comparison of Search Lengths between each algorithm

Execution time

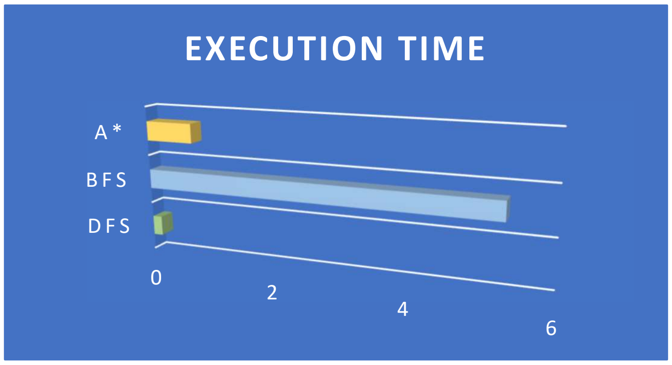


Figure 8 - Graph showing a comparison between the Execution Times of each algorithm

Execution time measures how long each algorithm took to find the goal. DFS has the shortest execution time (0.146 seconds), making it the fastest among the three. This can be attributed to DFS's aggressive exploration strategy, where it goes as deep as possible before backtracking, making it a good option if speed is of high importance. On the other hand, BFS took significantly longer (5.366 seconds), due to its exhaustive nature of searching all possible paths. A* strikes a balance with an execution time of 0.746 seconds, making it significantly faster than BFS but slower than DFS.

Trade-offs and implications

DFS can be advantageous when the depth of the tree is unknown, and the solution is expected to be far from the root. It is also faster and uses less memory in this case. However, it may not provide the optimal solution as it often does not explore the shortest path. BFS guarantees an optimal solution and can be useful when the solution is expected to be close to the root. However, it is slower, uses more memory, and might be impractical for larger, more complex mazes due to its exhaustive search nature. A* uses a heuristic to guide its search, which results in faster search times than BFS, while still providing an optimal solution. It is less memory intensive than BFS and is generally a good compromise between the thoroughness of BFS and the speed of DFS.

Conclusion

Based on the results, each algorithm has its strengths and weaknesses. Depth-First Search (DFS) demonstrated the fastest execution time, making it a suitable choice for situations where speed is prioritised over finding the shortest path. Breadth-First Search (BFS) was slower but found the optimal path, making it suitable when the shortest path is required, regardless of computation time. Finally, A* found the optimal path like BFS but in significantly less time and with a smaller search length, making it the best compromise for both speed and path optimality.

In looking at improvements for further testing, looking at the heuristic for the A* search, implementing an iterative deepening DFS (IDDFS) which is a hybrid of the DFS and BFS algorithms or a bi-directional search could provide better results, either shortening the path found or increasing the speed in which the optimal path is found.

Further testing could also be conducted to observe how the algorithms perform with different maze complexities, adjusting the size of the maze, providing different densities of walls and different start and goal positions. It could also be advantageous to measure and compare the memory usage of these algorithms as this could be a deciding factor on which algorithm is used in memory-constrained environments.

By conducting these improvements and further tests we can gain an increased understanding of these search algorithms and how they can be optimised for specific use cases.

Bibliography

Belwariar, R. (2018). *A* Search Algorithm - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/a-search-algorithm [Accessed 27 Jun. 2023].

GeeksforGeeks (2019). *Breadth First Search or BFS for a Graph - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ [Accessed 27 Jun. 2023].

Rahat, M. (2023). *Codeforces Solutions: DFS (Depth-First Search) Algorithm Code in C/C++*. [online] Codeforces Solutions. Available at: https://cfsolv.blogspot.com/2023/06/dfs-depth-first-search-algorithm-code.html [Accessed 27 Jun. 2023].