Proceedings of CONF-APMM 2026 Symposium: Applied Mathematics and Quantitative Modeling
DOI: 10.54254/2753-8818/2026.PU36012

Implementation and Comparison of Maze Solving Algorithms in OCaml

Qipeng Han

College of Medical and Biological Information Engineering, Northeastern University, Shenyang, China
20237384@mails.neu.edu.cn

Abstract. With the continuous development and application needs of robots, games and intelligent navigation, maze-solving algorithms have also begun to be researched in relation to traditional graph-search algorithms. The three maze-solving algorithms and their corresponding results in this paper are DFS, BFS and A*, and they are all based on graph-search methods for 2D-grid mazes. A* is based on the Manhattan distance. Experiments are carried out on the three types of mazes (sparse, dense and complex) and the three scales (10x10, 20x20 and 50x50), and the main indicators are execution time, path length, visited nodes and scalability. BFS and A* both return the shortest path; however, A* has visited fewer nodes than BFS. For example, 419 vs. 1480 in the 50x50 maze; DFS can find the maze in the shortest time but has a longer path. As OCaml has pattern matching and recursion, it can be used to implement the above search algorithm.

Keywords: Maze solving, pathfinding, OCaml, search algorithms

1. Introduction

The old problem of maze solving in computer science and artificial intelligence has been widely applied to robot navigation, autonomous systems, games, path planning, etc. The first purpose of a maze-solving algorithm is to find a path from the starting position to the end and at the same time keep the computational effort and search scope relatively small. Maze solving has had some practical value and theoretical benefits, so many types of graph-search and path-finding algorithms have been tested in various areas for a long time.

Many search algorithms have been put forward to solve the problem of maze-solving, such as Depth-First Search (DFS), Breadth-First Search (BFS), Dijkstra's algorithm and A* search. DFS is relatively simple to code and generally uses a smaller amount of memory, partly because of its recursive nature, it is relatively small. However, it will not be the shortest way either. BFS can always find the shortest path in an unweighted graph, but its memory consumption will be large when the search space is extensive. Dijkstra's algorithm and A* search are somewhat more sophisticated ways to increase the speed of exploration by adding path costs and heuristic information, so they do not need to expand nodes level by level. Previously, it has been shown in some studies that heuristic-driven search is generally more efficient than basic graph traversal algorithms for large, complicated mazes, and thus, it may accelerate the speed of search and path

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2753-8818/2026.PU36012

planning [1, 2]. Recently, some studies have also been carried out to solve the problem of complex networks by combining heuristic search strategies with swarm intelligence and optimisation techniques [3].

In addition to algorithm Design, the selection of a programming paradigm can also impact how to build and maintain the search algorithm. The old way of solving maze-searching problems is generally imperative programming, but functional programming has gained more attention recently and is often used due to recursion and immutable data structures [4]. Compared with imperative programming, functional programming can be relatively simple to implement for recursive search, and it may also improve the clarity and organization of the code.

Of all the functional programming languages, OCaml is very suitable for the construction of algorithms due to its robust static type system, pattern matching, algebraic data types and a fast compiler [5, 6]. Therefore, OCaml is relatively easy to use for writing recursive algorithms and graph problems. Recently, some studies have begun to apply OCaml for maze generation and pathfinding, and it has been found that functional programming can also be used in search-based problems [7, 8]. However, most of the existing papers have been studying single algorithms one by one and have conducted only minor performance comparisons. There is no common test set for some classical search algorithms in the same OCaml framework that has been widely studied; that is to say, the recursive implementation strategy and the general performance or correctness of the resulting path have not been well verified.

To solve the above problem, a maze-solving framework will be built in OCaml, and then the three general search algorithms: DFS, BFS, and A* search will be tested. Mazes of different sizes and complexities will be employed in the experiment to test the algorithm, and based on some performance indicators such as running time, path length, number of visited nodes and scalability, comparisons will be made. The paper also studies how functional programming methods are applied in the design of recursive algorithms, and evaluates how suitable OCaml is for pathfinding and graph-search problems; it is noted that OCaml is not one of the first languages people think of for this purpose.

2. Methodology

2.1. Maze representation

Maze solving is often considered to be a graph-search problem. Each of the mazes shown here is a two-dimensional grid, each walkable cell is a node, and the connection between adjacent cells is an edge. The objective is to find a valid path from the start node to the goal node.

This kind of representation is widely used in the research of pathfinding because it is possible to compare different search algorithms in the same framework [9, 10]. All of the algorithms were written in OCaml. The reason for this choice of language is its good handling of recursion and the availability of pattern matching and functional programming functions for graph traversal and search problems [5, 11].

2.2. Maze generation

Based on the above studies, it has been found that both maze density and obstacle distribution affect the behaviour of search algorithms [12, 13]. Therefore, this work produced three types of mazes with different structural characteristics, which are named Sparse Maze, Dense Maze and Complex Maze.

DOI: 10.54254/2753-8818/2026.PU36012

The sparse maze has relatively few obstacles and offers several other paths from the start node to the end node. Dense mazes, on the other hand, had a relatively high proportion of blocked cells, and in practice, the number of paths that could still be traversed was limited. More difficult mazes with many branches and dead ends have been added to increase the difficulty of the maze; therefore, there is a relatively high probability that the algorithm will waste time exploring unnecessarily.

To investigate how different sizes of mazes can affect the operation efficiency of the algorithm, the three grid sizes chosen in this paper are 10x10, 20x20 and 50x50. For every pair of maze category and maze size, five different maze instances were created. As shown in Table 1, in total there were 45 such experiments. Generally speaking, the above experiment can be divided into two sections based on the level of maze complexity and size of the search space.

Table 1. Experimental setup

Maze Type	Maze Size	Number of Instances
Sparse	10×10	5
Sparse	20×20	5
Sparse	50×50	5
Dense	10×10	5
Dense	20×20	5
Dense	50×50	5
Complex	10×10	5
Complex	20×20	5
Complex	50×50	5

2.3. DFS algorithm

Depth-First Search (DFS) is a type of graph traversal that goes as deep as it can in a particular search path and then backs up to explore other paths. Recursion was used in OCaml for the DFS implementation in this paper to conduct a search using recursion and pattern matching. In order to have a stack-like search, recursion is employed, and a list of visited nodes is used to avoid repeated traversal of the same location. The algorithm will stop when it reaches the goal node or has no more reachable paths. DFS is relatively simple to implement and is in line with the functional programming model due to its recursive nature. BFS is generally lighter on memory than DFS because it only needs to store one path during the search at any given time. DFS is based on the principle of path discovery, not path optimality, so it cannot ensure that the shortest path will be obtained; especially in large or highly branched maze environments, this is a problem.

2.4. BFS algorithm

BFS is Breadth-First Search; it explores all the neighbouring nodes at the current depth and then moves to the next level in the search tree. A Queue is used for the algorithm, and therefore the nodes will be processed in the order of discovery. It is the same as DFS; therefore, a list of visited nodes has been stored to avoid revisiting them. BFS is based on the distance of nodes from the start node, so it can find the shortest path in an unweighted maze. Therefore, BFS is often used when the main goal is to find a short path. However, the algorithm often needs to be run on a large amount of memory because many frontier nodes must be kept in memory at the same time. As the size and

DOI: 10.54254/2753-8818/2026.PU36012

complexity of the maze increase, so will the required memory and processing time; thus, it will be relatively slow for searching.

2.5. A* algorithm

A* search is a way to find the path that takes advantage of cost-based and informed search. A* has a function that takes into account both the path cost to the current location and an estimate of how far it is from the destination to select the better candidate node among those generated by DFS and BFS.

$$f(n) = g(n) + h(n) \tag{1}$$

Given that the movement in the maze is limited to the four directions, a Manhattan Distance heuristic will be used here. The heuristic function is given by:

$$h(n) = |xn - xg| + |yn - yg| \tag{2}$$

At runtime, the node with the smallest evaluation value is chosen to be expanded in order to restrict the search scope and focus the search on a region that may contain the optimum solution. A* is faster than BFS in terms of searching for nodes and will run fewer times to find the shortest path. Therefore, A* is often used as an efficient algorithm to solve the problem of mazes, especially when the search space is large or the structure of the maze is complex.

2.6. Evaluation metrics

In order to evaluate the actual performance of the search algorithm, this paper will use four indices: Execution Time, Path Length, Visited Nodes and Scalability.

Execution Time is employed to estimate the computational expense of solving a maze, and it generally indicates how efficient an algorithm is. Path Length is the number of nodes in the final solution path, so it has been used to evaluate the optimality of the path. Visited Nodes, on the other hand, is the total number of nodes explored in a search run, and this can show the efficiency of the search. Scalability is to determine how the performance of an algorithm changes with an increase in the size of the maze, and thus it can be observed that how well each method handles large and complex search spaces. Together, these indicators can show how good a solution is and how efficient it is in terms of computation; it is not enough to consider only one factor.

3. Results and discussion

3.1. Overall performance comparison

Table 2 shows the mean experimental results of all maze configurations.

Table 2. General experimental results

Maze Size	Algorithm	Exec. Time (ms)	Path Length	Visited Nodes
10×10	DFS	0.004	24	28
10×10	BFS	0.023	22	60
10×10	A*	0.014	22	25

DOI: 10.54254/2753-8818/2026.PU36012

Table 2. (continued)

20×20	DFS	0.018	76	83
20×20	BFS	0.136	68	247
20×20	A*	0.124	68	132
50×50	DFS	0.089	252	297
50×50	BFS	1.021	234	1480
50×50	A*	0.422	234	419

There are some differences among the three algorithms. DFS is fast and visits a small number of nodes. The Paths obtained by DFS were generally longer than those obtained by BFS and A*.

BFS is always the shortest path, but it has to expand many more nodes. A* is the same as BFS in path quality and has fewer explored nodes. The above are in line with the previous studies on maze-solving and pathfinding algorithms [1, 14].

3.2. Execution time analysis

Figure 1 is the execution time of the three algorithms with different maze sizes. The length of the execution time increased with the size of the maze. DFS had the shortest execution time in all the experiments. BFS had the longest execution time, especially in the 50x50 maze. A* is between DFS and BFS.

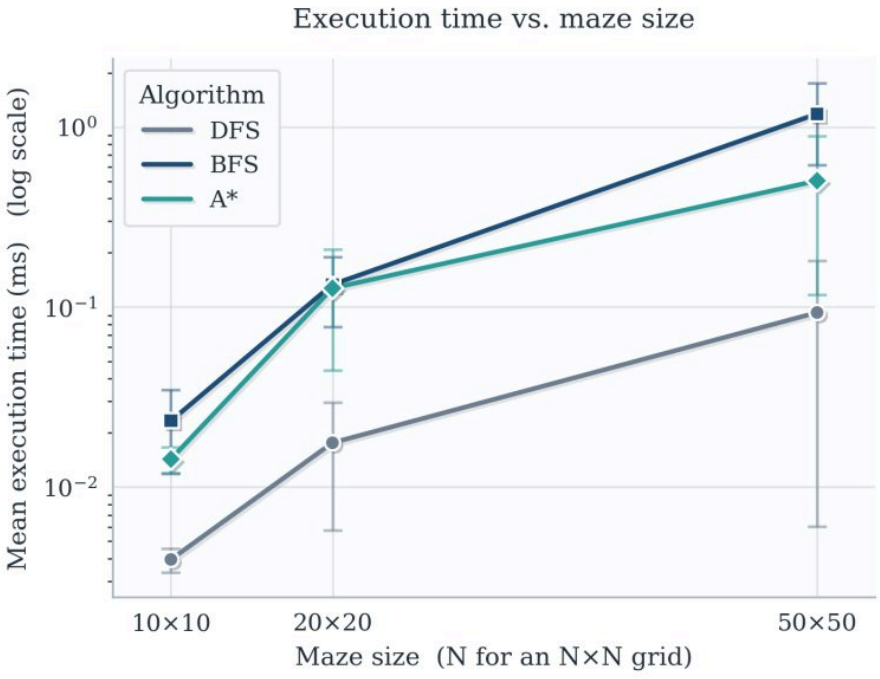

Figure 1. Execution time of the three algorithms vs. maze size (log scale) (picture credit: original)

The reason for this can be explained by the search method of each algorithm. DFS has only one branch and needs to back up when it cannot proceed further. BFS will list all the reachable nodes level by level and thus has a relatively high time complexity. A* is directed by a heuristic function to guide the search, so it can avoid some of the unnecessary exploration that BFS has to do.

DOI: 10.54254/2753-8818/2026.PU36012

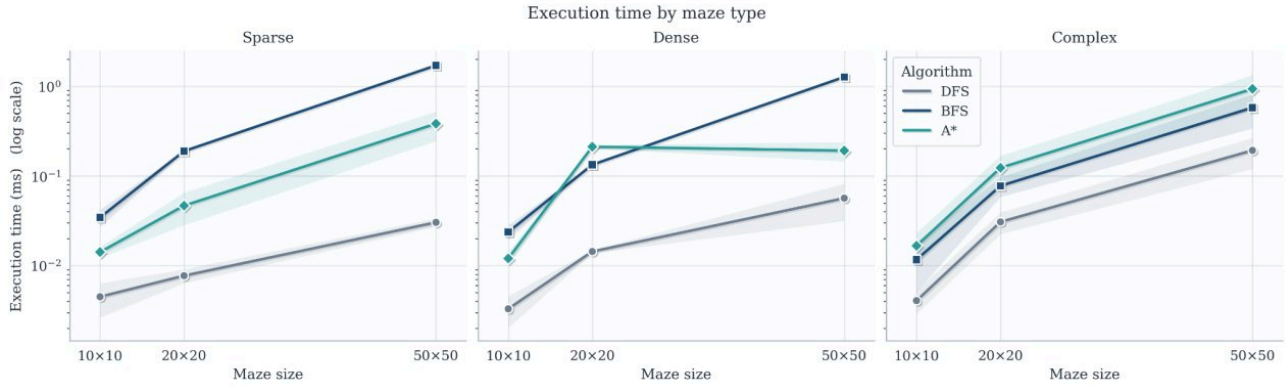

Figure 2. Execution time of the three algorithms by maze type (picture credit: original)

Figure 2 breaks down the execution time by maze type. The same trend can be seen in the categories of sparse, dense and complex; as maze complexity increases, the advantage of A* becomes more obvious.

3.3. Path length analysis

Figure 3 is the average path length of the three algorithms, and it is not very surprising. BFS and A* had the same path length for all mazes. Both ways have a mean path length of 234 nodes for the 50x50 maze, and they are roughly equal. DFS has a longer path length and is about 252 nodes long on average; it is relatively long.

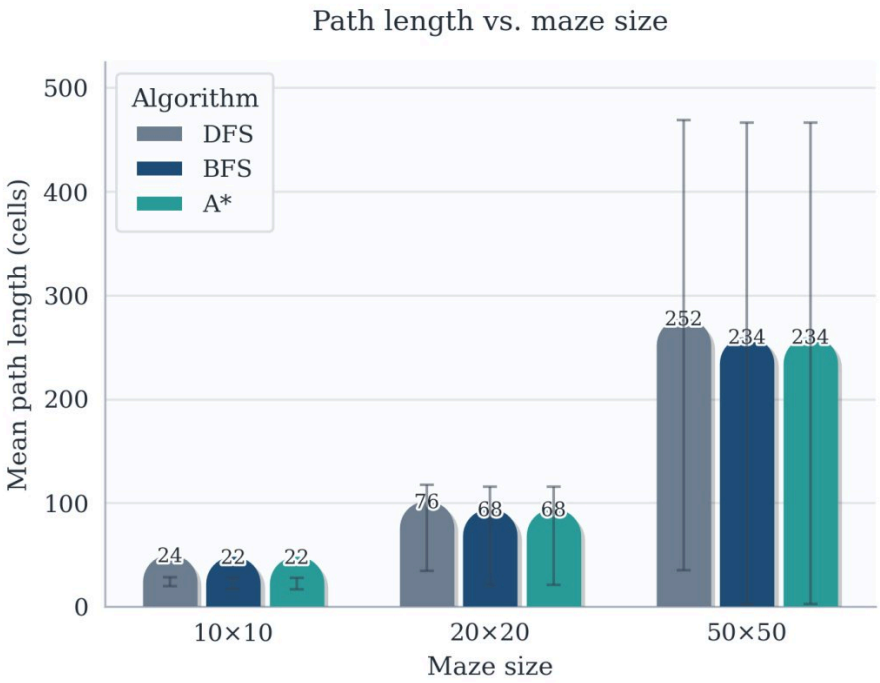

Figure 3. Average path length as a function of maze size (picture credit: original)

The above results are in line with what this study expected from the algorithm. BFS has the property of finding the shortest path in an unweighted graph by design, and A* can still be optimal if

DOI: 10.54254/2753-8818/2026.PU36012

it has an admissible heuristic [10, 14]. DFS is to find a possible or reasonable path that works; it does not need to be the shortest one. Therefore, the experiments show that there are longer paths even when the mazes are the same size and structure.

3.4. Efficiency of search

Figure 4 is the number of nodes visited by the three algorithms. BFS explored many more nodes than the others. The average number of nodes visited by BFS in the 50x50 maze was around 1480, and it was DFS that visited fewer nodes. A* has been visiting more nodes than DFS and BFS so far.

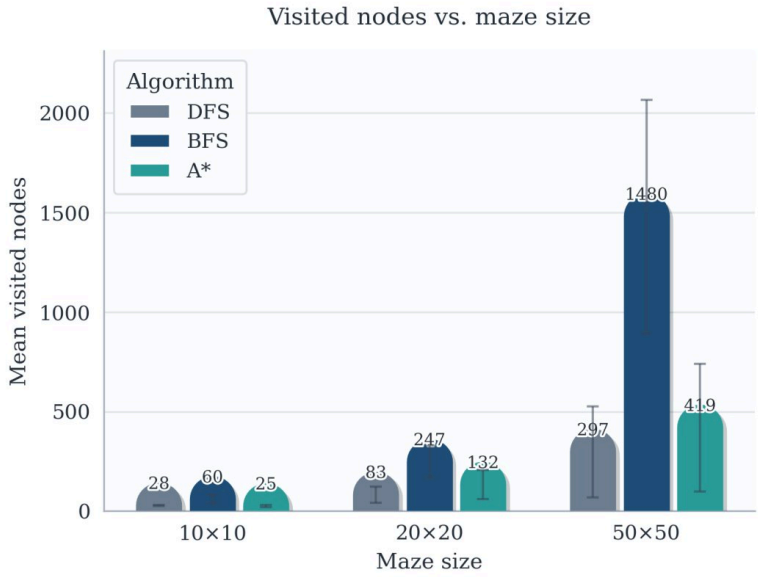

Figure 4. Number of visited nodes as a function of maze size (picture credit: original)

The reason is how each search works. BFS will visit all reachable nodes in an organised manner before it reaches the goal. DFS will not expand too many nodes; if it finds a valid path, it will stop at that time. Early stopping may be too early, and the solution will not be ideal. A* is a combination of the two; it can use a heuristic function to guide the search in the direction of the goal rather than exploring randomly or searching exhaustively from the beginning.

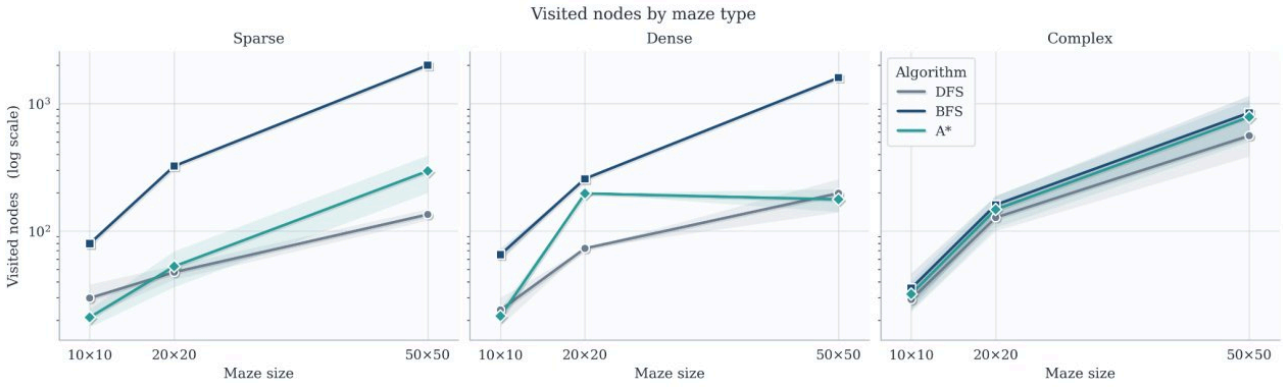

Figure 5. Number of visited nodes by maze type (log scale) (picture credit: original)

DOI: 10.54254/2753-8818/2026.PU36012

Figure 5 is then shown to break down the visited node counts by maze type. The gap between BFS and A* is relatively large in a large number of mazes. Therefore, it can be seen that the heuristic guidance will be more beneficial as the size of the search space increases and the structure becomes more complex.

3.5. Scalability analysis

Figure 6 is shown below, and the number of visited nodes changes with the size of the maze area. All the algorithms need more computing power when the maze is large. However, the rates of growth were different. BFS had the most rapid growth in the number of expanded nodes, and DFS was relatively slow.

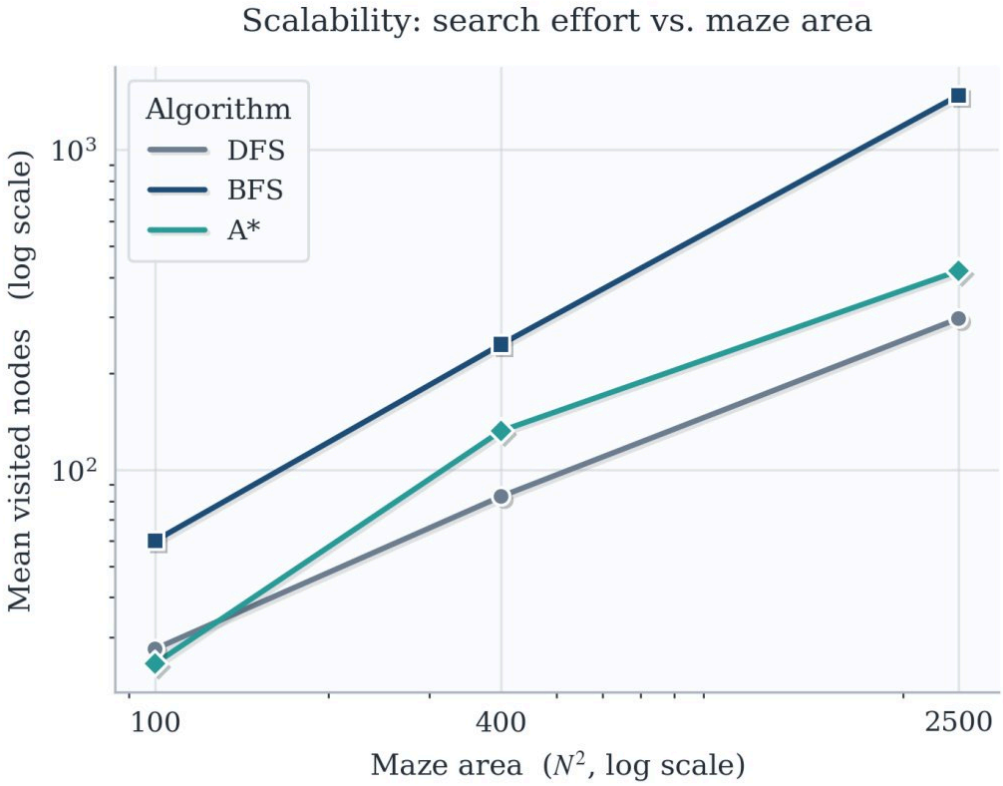

Figure 6. Mean visited nodes vs. maze area (N2, log scale) (picture credit: original)

A* had a relatively small scale. Although the number of explored nodes increased with the size of the maze, the rate of growth was much slower than that for BFS. As shown in the above results, A* is relatively fast for large maze environments and needs to be searched promptly.

3.6. Heuristic effectiveness of A*

Figure 7 compares the number of nodes visited by A* with that of BFS. Based on the above results, A* has only explored 41 per cent of the nodes that BFS has explored in a 10x10 maze; it is 54 per cent for a 20x20 maze and 28 per cent for a 50x50 maze.

DOI: 10.54254/2753-8818/2026.PU36012

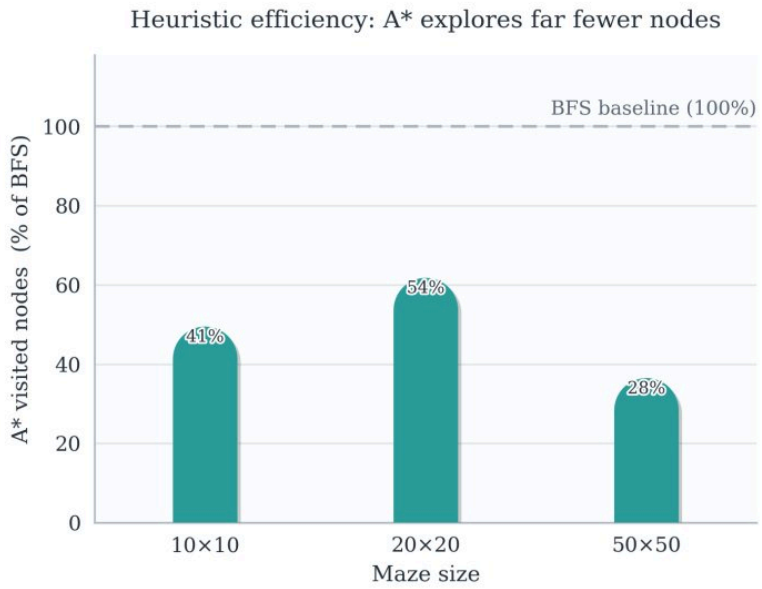

Figure 7. Efficiency of A* search vs. BFS (percentage of visited nodes) (picture credit: original)

As the size of the maze increases, so does the decrease. Therefore, the heuristic information will not be explored unnecessarily and still obtain the same solution quality. Based on the theory of A* proposed by Hart et al. [14], it is expected that the heuristic function will be admissible and consistent for faster search.

3.7. Algorithm comparison

Table 3 presents a summary of the qualitative characteristics of the three search algorithms in this paper.

Table 3. Features of the three search algorithms

Algorithm	Search Strategy	Shortest Path Guarantee	Main Data Structure	Advantages	Limitations
DFS	Depth-first (LIFO)	No	Stack / recursion	Low memory; simple recursive design; quick to reach a goal	Path usually non-optimal; may explore deep dead ends
BFS	Breadth-first (FIFO)	Yes (unweighted)	Queue	Guarantees shortest path; complete	Expands many nodes; high memory on large mazes
A*	Best-first, f = g + h (Manhattan)	Yes (admissible h)	Priority queue (heap)	Optimal; expands far fewer nodes than BFS	Per-node heuristic and priority-queue overhead

DFS follows one path in the maze until it cannot continue; then, it backtracks and tries another. This way does not require much memory, and in OCaml it can be written quite naturally using recursive functions, although it can be relatively high in terms of call stack. Still, DFS itself cannot guarantee that it will find the shortest path.

DOI: 10.54254/2753-8818/2026.PU36012

BFS adds the search level by level, starting from the start node. Therefore, BFS will always find the shortest path in an unweighted graph. BFS is that it needs to visit a large number of nodes before finding the goal, especially in a large search space.

A* is the sum of the actual cost accumulated so far and the heuristic guidance; thus, it can direct exploration more efficiently. The Manhattan Distance heuristic was employed in this paper to estimate how far the current state is from the goal state. The above way can guide A* to a better location and still guarantee the shortest path.

The features in Table 3 are in good agreement with the experimental results presented in Table 2 above. BFS and A* produced the same path length for all mazes, so both could find the shortest path. Meanwhile, DFS had longer paths; it is because the search stops as soon as a valid path is found, and does not continue to find the optimal one.

There were also variations in the extent. DFS visited the fewest nodes but did not have an optimal path. A* examined more nodes than DFS, but it was still far from BFS. In the 50x50 case, the average number of nodes expanded by BFS over each run was about 1480, and that of A* was only about 419. Thus, A* could better balance the search cost and the quality of the path.

According to the experimental results above, DFS is relatively easy to implement and has a small memory footprint; BFS can be employed to find the shortest path, and A* is suitable for solving large-scale maze-solving problems.

3.8. Discussion on OCaml implementation

The Algorithm works and, based on the experiments carried out, it is considered a suitable language for maze-solving. Recursion and Depth-First Search are more in line with functional programming; they feel quite natural. Pattern matching can be used to simplify the code for handling nodes and checking neighbors; it is sometimes slightly longer. It is organised more neatly with OCaml modules than this paper had expected in the beginning.

Although imperative languages are often used for pathfinding problems, it can be seen that OCaml is also suitable for writing an efficient graph-search algorithm. Generally speaking, OCaml is a suitable language for the teaching and research of maze-solving and pathfinding.

4. Conclusion

The three popular pathfinding algorithms are DFS, BFS and A*; they will be presented and compared in this paper through a common maze-solving framework in OCaml. Over 45 experiments have been conducted in the three kinds of mazes (sparse, dense, and complex) and the three sizes (10×10, 20×20, and 50×50), and the methods are evaluated in three ways: execution time, path length, and the number of visited nodes. Generally speaking, the results show that DFS runs the fastest, but it is not always the shortest path. BFS will find the shortest path, but it will have to explore more nodes. A* is still the optimal shortest-path algorithm and can also reduce the amount of search work significantly, so it is generally used in practice. A certain problem is that only the three classic algorithms are employed, and the mazes are static and unweighted. Weighted graphs, changing environments and swarm optimisation are not mentioned, but they will be topics for future studies.

References

[1] Rohan, S., Shandilya, S., Rawat, S., & Choudhary, A. (2025). Comparative analysis of BFS, Dijkstra, and A* pathfinding algorithms in 2D maze: A time-based approach. In *2025 International Conference on Intelligent and*

DOI: 10.54254/2753-8818/2026.PU36012

Secure Engineering Solutions (CISES) (pp. 1205–1208). IEEE. https://doi.org/10.1109/CISES66934.2025.11265425

[2] AlKahtani, R., Alhabdan, A., Alosami, M., Alshammari, W., Abdo, A. A., & Hamdi, L. (2025). Comparative analysis between BFS and DFS-shortest path algorithms. In *2025 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)* (pp. 1–5). IEEE. https://doi.org/10.1109/eStream66938.2025.11016860

[3] Espejon, K. T., Jayoma, J. M., & Empang, K. P. A. (2024). Hybrid A* and inverse ant algorithm for adaptive pathfinding in constrained node networks. In *2024 IEEE 16th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM)* (pp. 1–6). IEEE. https://doi.org/10.1109/HNICEM64917.2024.11258860

[4] Giegerich, R., & Hughes, J. (2021). Functional programming in the real world. In *Dagstuhl Seminar 9420: Functional programming*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[5] Hickey, J. (2008). Introduction to Objective Caml. California Institute of Technology. http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf

[6] Doligez, A., et al. (2020). Objective Caml for multicore architectures. *arXiv*. https://doi.org/10.48550/arXiv.2006.05862

[7] Wang, Q. (2024). Comparative analysis of maze solving algorithms based on OCaml. *AIP Conference Proceedings, 3194*(1), 040001. https://doi.org/10.1063/5.0222821

[8] Wang, X. (2024). OCaml-based recursive backtracking and Aldous Broder algorithms in solving maze challenges. *AIP Conference Proceedings, 3194*(1), 050011. https://doi.org/10.1063/5.0226607

[9] Wu, C.-M., Liaw, D.-C., & Lee, H.-T. (2018). A method for finding the routes of mazes. In *2018 International Automatic Control Conference (CACS)* (pp. 1–4). IEEE. https://doi.org/10.1109/CACS.2018.8606753

[10] Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Pearson Education.

[11] Minsky, Y., Madhavapeddy, A., & Hickey, J. (2013). *Real world OCaml: Functional programming for the masses*. O'Reilly Media.

[12] Gabrovšek, P. (2019). Analysis of maze generation algorithms. *IPSI Transactions on Internet Research, 15*(1), 23–30.

[13] Vijaya, J., Gopu, A., Vinayak, K., et al. (2024). Analysis of maze generation algorithms. In *2024 5th International Conference on Innovative Trends in Information Technology (ICITIIT)* (pp. 1–6). IEEE.

[14] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136