Proceedings of CONF-APMM 2026 Symposium: Applied Mathematics and Quantitative Modeling
DOI: 10.54254/2753-8818/2026.PU35704

A Comparative Study of BFS and Left-Hand Wall-Following Algorithms for Maze Solving in OCaml

Jiajie Yuan

School of Computer and Information Technology, Nanjing Tech University, Nanjing, China
202521139037@njtech.edu.cn

Abstract. This paper aims to investigate and solve maze-solving problems by implementing the Breadth-First Search (BFS) algorithm and the Left-Hand Wall-Following algorithm, providing references for maze navigation of small embedded robots. In this study, the OCaml programming language is adopted. This paper implements both the BFS and Left-Hand Wall-Following algorithms to solve three mazes of different sizes (30×28, 30×17, 30×16). The total moving steps and the shortest valid paths are recorded. The time and space complexity of the two algorithms are calculated, and comparative tables are generated for analysis. Experiments show that the Left-Hand Wall-Following algorithm has a constant space complexity of O(1) with stable memory consumption. However, its generated path length can reach 2.5 to 10 times the theoretical shortest path, and it is only applicable to simple mazes. In contrast, BFS can stably output globally optimal paths and adapt to all types of complex mazes, yet its memory overhead grows linearly with maze size. The two algorithms fit distinct application scenarios and can be selected flexibly according to equipment memory constraints and requirements for optimal paths.

Keywords: Maze solving, BFS algorithm, Left-hand wall-following algorithm

1. Introduction

Graph traversal and pathfinding algorithms are fundamental to artificial intelligence and computer algorithm research [1]. As a typical grid-based pathfinding problem, maze solving has always been a common experimental method for testing the performance of traversal algorithms.

In 1959, Edward F.-Moore adopted BFS to guide Shannon's mechanical mouse to solve mazes and first formally presented the BFS algorithm [2]. In a standard maze-solving task, a program needs to find a feasible path from a given start point to an end point in a 2D grid that contains obstacles. However, in practice, finding just any path is often not enough. The shortest path is usually required to reduce movement costs and improve computational efficiency. Currently, Depth-First Search (DFS) and Breadth-First Search (BFS) are the two main methods in this field. In artificial intelligence, BFS and DFS have distinct practical applications. BFS is widely adopted in robot navigation, AGV scheduling, autonomous driving local path planning, NPC shortest pathfinding, sliding puzzle minimal-step solving, social network layered relation mining, and image connected region segmentation, as it can obtain the optimal path with the least steps under equal action cost [3, 4]. In contrast, DFS is suitable for backtracking tasks including Sudoku and N-

© 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.PU35704

Queens solving, game tree traversal, unknown environment deep exploration for robots, decision tree reasoning, syntax parsing in natural language processing, maze generation, and sparse graph cycle detection, featuring low memory consumption and full solution enumeration capability. DFS uses a "go as far as possible" strategy, following one branch until it reaches a dead end. As a result, the path it finds is often winding and long, and it cannot guarantee the shortest path in unweighted grids [5]. In contrast, BFS uses a first-in-first-out queue structure to expand nodes layer by layer. It explores all nodes at the current level before moving to the next level [6]. Therefore, the first time BFS reaches the destination in an unweighted maze, the path it finds is guaranteed to be the shortest. This is the main reason why BFS was chosen for this project [7].

The core goal of this paper is to build a stable BFS framework that can solve all given maze models, accurately output the shortest valid path for each test maze, and avoid runtime issues such as path errors or program freezes. The logic behind maze solving has strong cross-domain scalability. Its pathfinding core can be applied to real-world scenarios such as autonomous mobile robot navigation, indoor positioning systems, autonomous driving route planning, and logistics warehouse robot scheduling. This provides some practical reference value for industrial intelligent systems.

2. Methods

2.1. Maze setting

This work develops a grid maze modeling scheme that splits a 2D plane into separate grid cells. Each cell is marked as an open passage or an impassable obstacle, making up a rectangular maze sized W × H, with W standing for width and H for height. A unique coordinate pair (x, y) is allocated to every cell to locate its position accurately.

The horizontal axis x stretches across the maze's width, taking values from 0 up to (but not including) w. The vertical axis y tracks the maze's height, limited to the range [0, h).

All structural data of each maze is saved in a standard triple structure containing three core parts: maze width w, maze height h, and a binary coded string raw_wall that logs wall layouts. A full maze record includes four pieces of information: width parameter, height parameter, wall encoding string and reference shortest path.

Most classic maze storage techniques rely on 2D arrays, while this paper puts forward a more compact storage approach. All wall states of grid cells are condensed into one single string. The transform rule idx = x × h + y converts 2D coordinates into a 1D string offset. All data loading and writing processes use this linear index, which noticeably speeds up the program's data lookup performance.

Letters a through p are used to encode wall surroundings for each grid cell. Every single letter matches a 4-bit binary value, which separately records whether walls exist on the cell's west, south, east and north sides.

As an example, letter p means walls block all four directions around the cell. On the contrary, letter a means there are no surrounding walls, so movement to all four adjacent cells is allowed.

When initializing maze data, the first coordinate in the reference path list is set as the maze entry point, and the last coordinate acts as the exit.

This grid modeling system consumes little memory and supports expandable encoding formats. The consistent coordinate rule also eases boundary judgment logic during program execution and cuts down runtime calculation costs.

DOI: 10.54254/2753-8818/2026.PU35704

2.2. Breadth-First Search

When applying the Breadth-First Search (BFS) algorithm, every grid cell inside the maze carries the same movement cost. This search strategy leverages a first-in-first-out (FIFO) queue to realize layer-by-layer outward node expansion. Starting from the designated starting cell, the algorithm iteratively traverses adjacent grid cells layer by layer. Within unweighted maze environments, the first path reaching the destination is guaranteed to be the optimal solution. This algorithm is highly applicable to optimal pathfinding scenarios such as robot autonomous navigation and grid maze solving. Combined with the a-p character compression encoding mechanism for maze wall information, the proposed method realizes standardized maze data input and supports batch verification of algorithm accuracy on multiple large-scale maze groups. The BFS algorithm first cleans and models the raw maze string by reserving valid wall mask characters and converting two-dimensional coordinates to one-dimensional indices for fast querying, then initializes a FIFO queue, a visited matrix and a parent coordinate matrix with the starting cell marked and enqueued, subsequently expands layer by layer by continuously checking and enqueuing valid unvisited adjacent cells while recording their parent coordinates, and finally terminates the traversal to backtrack a complete path from the destination to the start if the target is reached or returns an empty list if no valid path exists after all cells are traversed. In unweighted grid-based mazes, BFS features completeness and optimality. Provided a connected path exists, the algorithm can always locate it with the minimum-step route. Its time complexity is O(W×H), since each cell is processed at most once. Owing to memory consumption from the visited-mark array, parent-node array and traversal queue, its space complexity is also O(W×H) [8]. The queue may store numerous grid cells in the worst-case scenario. Despite relatively high-memory overhead, BFS serves as the benchmark algorithm for shortest-path-finding in grid mazes given sufficient storage space.

2.3. Left-hand rule

Both the Left-Hand Wall-Following Algorithm and the BFS algorithm are coded in OCaml [9, 10]. The time complexity of the left-hand wall-following algorithm is O(W multiplied by H), where W stands for the width of the maze and H stands for the height of the maze [11].

This algorithm runs based on simple greedy iterative logic, and the upper limit of its maximum moving steps is four times the total number of cells in the maze. Each step only completes wall detection and coordinate update, so the overall running time of the algorithm has a strict linear relationship with the size of the maze. In terms of space complexity, the algorithm reaches constant space complexity O1 [7, 11].

Different from other search algorithms, it does not need to create large two-dimensional arrays or memory-heavy storage structures. It only records the current coordinates, moving direction, step counter and fixed-length path buffer. No extra memory will be consumed even if the maze becomes larger. In contrast, BFS relies on global visited node storage and queue maintenance, which generates obvious memory pressure when dealing with large maze grids and robot navigation tasks [12].

DOI: 10.54254/2753-8818/2026.PU35704

3. Results and discussion

3.1. Results

This program realizes both the left-hand wall-following algorithm and Breadth First Search. Its code structure is clear and supports complete comparative experiments. The core advantages and disadvantages of the two methods are summarized as follows shown in Table 1.

Breadth First Search can always find the shortest optimal path, yet it takes up more memory; the left-hand wall-following algorithm uses very few resources and is lightweight, but it cannot guarantee to find the optimal path. This comparison table contrasts the core characteristics of two maze pathfinding algorithms, namely the left-hand wall-following algorithm and Breadth First Search (BFS), across seven dimensions. The table concludes that although the two algorithms share the same upper bound of time complexity, they feature entirely different trade-offs. The left-hand wall-following algorithm is lightweight, simple to implement and memory-efficient, yet it yields suboptimal paths and is limited in applicable scenarios. In contrast, BFS boasts strong versatility and guarantees the shortest path, while it comes with higher implementation difficulty and greater memory consumption. Three groups of large mazes with dimensions of 30 by 28, 30 by 17 and 30 by 16 were tested in this experiment, and the experimental results are shown in the corresponding Table 2.

Table 1. The comparison of left-hand wall-following algorithm and Breadth First Search

Comparison Item	Path Optimality	Time Complexity	Space Complexity	Implementation Difficulty	Maze Adaptability	Resource Requirement
Left-Hand Wall Following	Non-optimal, lots of detours	O(W×H)	O (1) (Constant)	Very easy	Only for simply connected mazes	Ultra-low
BFS	Always the globally shortest path	O(W×H)	O(W×H) (Linear)	Moderate	All connected maze types	High

Table 2. The experimental results of different mazes

Maze Number	Maze Size	Steps of Left-hand Wall-following Algorithm	Theoretical Shortest Path Steps	Steps of BFS Algorithm	Does Left-hand Algorithm Produce Shortest Path	Does BFS Produce Shortest Path	Do the Two Algorithms Get Identical Result
Maze 1	30x28	1161	117	117	No	Yes	No
Maze 2	30x17	188	68	68	No	Yes	No
Maze 3	30x16	623	69	68	No	Yes	No

The experimental results verify the optimality, limitation and consistency of the two algorithms. The BFS produces step counts identical to the theoretical shortest-path values across all three tests, which confirms its global optimality. In contrast, the left-hand wall-following algorithm generates far more steps with redundant detours in every trial, and its step count is almost ten times the minimum value in the 30×28 maze with the most obvious detour, indicating that it cannot obtain optimal paths. Moreover, the results of the two algorithms differ in all experiments, further proving that the wall-following algorithm fails to find the shortest path.

DOI: 10.54254/2753-8818/2026.PU35704

3.2. Discussion

This work codes two maze search schemes including the left-hand wall-following algorithm and Breadth-First Search (BFS) using OCaml and carries out comparative tests on both pathfinding approaches [10]. Test data highlight major gaps between the two algorithms across three key metrics: path optimality, adaptability to maze layouts and memory resource demand.

The left-hand wall-following method is a local responsive navigation rule that only judges directions from nearby wall layouts, with no access to full-map global data [13]. Since it moves alongside maze walls passively, it cannot generate the shortest possible routes. In the tests, paths produced by this algorithm contain lots of unnecessary backtracking and repeated grid visits; their total lengths range from 2.5 to 10 times the theoretical minimal path length.

Moreover, this algorithm can only find complete valid paths for simply connected mazes whose walls link to the outer border. If a maze contains standalone inner rings or separated wall blocks, the algorithm will loop endlessly and fail to output workable routes [13].

On the other hand, BFS scans grid nodes level by level with full global map data as reference. Under unweighted grid settings, the first time the algorithm reaches the target cell always yields the globally shortest path. Its steady, repeatable test performance makes it a widely used benchmark for grid path planning tasks. It works reliably on all kinds of maze topologies, including loops, dead ends and branching passages, which greatly expands its applicable scenarios [14].

A core contrast lies in their memory usage patterns. The left-hand wall-following algorithm runs at O(1) constant space complexity, so its memory load never shifts no matter how large the maze grows. Even on huge, intricate maze maps, its memory footprint stays consistent. Its performance loss only appears as extra redundant movements, rather than memory overflow failures.

In contrast, BFS needs to constantly store visited grid tags, parent node logs and search queues, so its memory cost rises proportionally with the total number of grid cells [14]. Recursive BFS implementations face stack size limits from the OCaml runtime system; large mazes will quickly trigger stack overflow and shut down the program directly. To remove such runtime bottlenecks and boost overall stability, this research adopts an iterative queue-based BFS design, which avoids stack overflow risks and eases excessive memory consumption.

4. Conclusion

In the introduction, this study primarily aims to construct a reliable pathfinding framework centered on the BFS algorithm. The proposed system is well compatible with diverse maze test scenarios, accurately computing the shortest feasible path for each maze instance while effectively avoiding operational failures such as path deviation and program freeze.

To comprehensively compare the practical performance of typical maze exploration strategies, this work implements both the left-hand wall-following method and the BFS algorithm using OCaml. Comparative evaluations are conducted on maze datasets with three different size levels and structural complexities.

Experimental results clearly indicate inherent performance trade-offs between the two strategies. The wall-following approach maintains constant O(1) space complexity, with its memory consumption remaining stable regardless of maze expansion. However, it commonly produces highly redundant trajectories, where the resulting path length can be 2.5 to 10 times the actual optimal solution. Moreover, the algorithm fails to handle mazes containing standalone inner loops, restricting its valid application only to simple, small-scale maze environments.

DOI: 10.54254/2753-8818/2026.PU35704

In contrast, BFS is able to consistently find the globally optimal path for any fully connected maze topology. Despite its high solution accuracy, BFS requires progressively more memory resources as the number of maze grids increases, presenting a linear memory growth trend.

Two key shortcomings exist in the current research. First, all tests are completed under static maze conditions, so the current algorithm lacks adaptability to dynamic maze environments with changing obstacle layouts. Second, the recursive BFS implementation is limited by the inherent stack size constraints of the OCaml runtime, making it susceptible to stack overflow when processing large-complexity mazes.

In future research, a hybrid pathfinding strategy will be designed to combine the low-memory characteristic of wall-following algorithms and the optimal-path advantage of BFS. Meanwhile, iterative queue-based BFS will completely replace the recursive version to improve program robustness. Additionally, dynamically variable obstacle settings will be introduced to enrich experimental diversity and achieve more comprehensive test validation.

References

[1] Dwivedy, S., & Gupta, S. K. (2016). A comparative study of BFS and DFS for autonomous exploration of unknown environments. *International Journal of Robotics and Automation, 31*(1), 68–76.

[2] Moore, E. F. (1959). The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching Circuits* (pp. 285–292). Harvard University Press.

[3] Patle, B. K., Babu, L. G., Pandey, A., Parhi, D. R., & Jagadeesh, A. (2019). A review: On path planning strategies for navigation of mobile robot. *Defence Technology, 15*, 582–606.

[4] Bhattacharya, S., & Ghosh, A. (2018). Breadth-first search and its diversified real-world applications. *International Journal of Computer Applications, 180*(47), 1–7.

[5] Skiena, S. S. (2008). *The algorithm design manual* (2nd ed.). Springer.

[6] Elkari, B., Oubah, L., & Sekkat, H. (2024). Exploring maze navigation: A comparative study of DFS, BFS, and A* search algorithms. *Statistics, Optimization and Information Computing, 12*(3), 761–781.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

[8] Koenig, S., & Likhachev, M. (2004). Lifelong planning A*. *Artificial Intelligence, 155*(1–2), 93–146.

[9] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Sivaramakrishnan, K. C., & Vouillon, J. (2024). *The OCaml system release 5.2: Documentation and user's manual* (Doctoral dissertation, Inria).

[10] Wei, X., & Li, M. (2024). Comparative analysis of maze solving algorithms based on OCaml. In *AIP Conference Proceedings (Vol. 3194, No. 1, Article 040001)*. AIP Publishing.

[11] Del Rosario, J. (2013). Modelling and characterization of a maze-solving mobile robot using wall follower algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(7), 771–776.

[12] Saman, A. B. S., & Abdramane, I. (2013). Solving a reconfigurable maze using hybrid wall follower algorithm. *International Journal of Computer Applications, 82*(3), 10–14.

[13] Mazumdar, S., & Roy, S. (2020). Performance comparison robot path finding uses flood fill-wall follower algorithm. *International Journal of Robotics and Automation, 5*(1), 22–28.

[14] Rahman, M. (2025). A comparative study of maze-solving algorithms: Performance, complexity, and practical applications in AI and robotics. *Journal of Artificial Intelligence Research, 12*(2), 89–102.