Proceedings of CONF-APMM 2026 Symposium: Applied Mathematics and Quantitative Modeling
DOI: 10.54254/2753-8818/2026.PU35550

A Performance Comparison of Binary Tree, Recursive Backtracking and Aldous-Broder Maze Generation Algorithms with BFS Solving in OCaml

Yuxuan Gao

Jinling High School Hexi Campus International Department, Nanjing, China
Stockfish1D@hotmail.com

Abstract. This study aimed to compare the performance of three classical maze generation algorithms including Binary Tree, Recursive Backtracking, and Aldous-Broder in terms of both generation time and subsequent solving time using Breadth-First Search (BFS). Understanding the trade-offs between generation speed and solving efficiency is important for beginners in algorithm design and for applications such as game development and procedural content generation. All three generation algorithms and the BFS solver were implemented in OCaml. Experiments were conducted on a 250×250 grid with the start cell at the bottom-right corner and the goal at the top-left corner. Generation time per algorithm was measured as an average of 20 runs, repeated 50 times with different random seeds. Binary Tree had the fastest average generation time (0.0485 seconds), followed by Recursive Backtracking (0.1416 seconds), while Aldous-Broder was the slowest (4.2691 seconds). However, solving times showed a different pattern: Aldous-Broder produced mazes that were solved the fastest by BFS (0.0029 seconds), followed by Binary Tree (0.0089 seconds), with Recursive Backtracking yielding the longest solving time (0.0243 seconds). These findings indicate that no single algorithm dominates both metrics; the optimal choice depends on whether generation speed or solving speed is prioritized. The results support the hypothesis that Aldous-Broder's uniform spanning tree property leads to more efficiently solvable mazes despite its high generation cost.

Keywords: Maze generation, BFS, OCaml, performance comparison

1. Introduction

Mazes have been interesting to people for a very long time, not only in games but also in computer science. When talking about mazes in programming, there are two main sides: how to generate a maze, and how to solve one. These two problems are connected, but they are not the same. In a broader sense, path-finding problems appear in many areas, such as Global Positioning System (GPS) navigation and robot movement. However, this study focuses only on grid-based mazes, which are simpler and easier for beginners to understand and implement.

In the last decades, some researchers and programmers have compared different maze generation algorithms. For example, they look at how fast an algorithm runs or how random the maze looks.

DOI: 10.54254/2753-8818/2026.PU35550

But many of those studies focus on only one or two algorithms. They do not always include beginner-friendly implementations. Even closer to this work, there are studies that talk about solving mazes with classic algorithms like Breadth-First Search (BFS). BFS is known to always find the shortest path in an unweighted maze [1].

Now, narrowing down further: some papers look closely at generation algorithms like Recursive Backtracking, Binary Tree, and Aldous-Broder. These three are quite different from each other. Recursive Backtracking tends to create long, winding passages [2]. Binary Tree is fast but produces a strong diagonal bias [3]. Aldous-Broder can generate perfectly uniform mazes, but it is much slower [4, 5].

When looking for studies that are very close to this work, only a few key papers that really match can be found. Some of them compare two generation algorithms, but not three. Others test BFS on only one type of maze. A small number of papers do compare Recursive Backtracking, Binary Tree, and Aldous-Broder together, but they usually focus on theoretical time complexity instead of actual running time for a beginner programmer. Also, very few of them combine generation comparison with a solving algorithm like BFS in one simple experiment.

Therefore, the aim of this study is not to discover a new algorithm. Instead, a simple but practical question is answered: among the three generation algorithms implemented, which one produces a maze that is fastest for BFS to solve, and which generation algorithm runs fastest by itself. These questions are useful for someone who is just starting to learn algorithms and wants to see how they behave in a real program. The rest of this paper is organized as follows. Section 2 describes the methods and how the experiments are set up. Section 3 presents the results and discussion together. Section 4 concludes the paper and suggests the future prospects.

2. Methods

In this section, the three maze generation algorithms and the solving algorithm are described. After that, the experimental setup is explained including the hyperparameters and the programming environment.

2.1. Maze generation algorithms

Three different maze generation algorithms were implemented: Binary Tree, Recursive Backtracking, and Aldous-Broder. All three algorithms operate on a 2D grid of cells. Each cell initially has four walls (i.e. north, east, south, west). A maze is generated by removing walls between adjacent cells according to the rules of each algorithm.

2.1.1. Binary Tree

The main idea of the Binary Tree algorithm is very simple. For each cell in the grid, a random decision is made to remove either the north wall or the east wall [3, 6]. Because of this rule, the maze always has a bias. Cells on the north edge cannot remove the north wall, so they must remove the east wall. Similarly, cells on the east edge must remove the north wall instead. The cell at the north-east corner has no north or east wall to remove, so it is left as a dead end. In the implementation, when a cell was on the north edge, only the east wall was removed; when on the east edge, only the north wall was removed. For all other cells, a random choice between north and east was made.

DOI: 10.54254/2753-8818/2026.PU35550

The principle of Binary Tree is to create a perfect maze, meaning there is exactly one path between any two cells. However, the bias is very strong. The maze tends to have long corridors running from south-west to north-east. The advantage of Binary Tree is speed. It runs in O(rows × columns) time and is very easy to implement. The disadvantage is that the generated mazes look very similar and are not uniform. They always show a diagonal pattern, which may not be suitable for applications that require randomness. As noted in a comparative study on mobile learning applications, Binary Tree performs best for small to medium-sized mazes where speed is critical [6].

2.1.2. Recursive Backtracking

The main idea of Recursive Backtracking is to perform a random depth-first search (DFS) on the grid using recursion [2, 7]. Starting from a random cell, the algorithm marks the current cell as visited. Then it randomly chooses an unvisited neighbor, removes the wall between them, and calls itself recursively for that neighbor. If the current cell has no unvisited neighbors, the recursive call returns (backtracks). The process ends when all cells have been visited. In the implementation, a recursive function was used directly, without an explicit stack. The recursion followed the depth-first order, and backtracking happened automatically when the recursive call returned.

The principle of Recursive Backtracking is similar to how a person might walk through a maze and mark visited paths. Because it follows a depth-first strategy, the resulting maze contains long, winding passages and very few short dead ends. The advantage of this algorithm is that it produces mazes with a natural, unpredictable look. It also creates a perfect maze. The disadvantage is that recursion can cause a stack overflow for very large grids, but for the grid size used in this study (250×250), the recursion depth is acceptable in OCaml. Comparative implementations in OCaml have shown that Recursive Backtracking offers higher maze complexity at the cost of longer generation time compared to Binary Tree [7, 8].

2.1.3. Aldous-Broder

The main idea of the Aldous-Broder algorithm is to use a random walk to generate a uniform spanning tree [4, 5, 8]. A random walk means the algorithm moves randomly from the current cell to any neighboring cell, regardless of whether that neighbor has been visited before. When the random walk steps into an unvisited cell for the first time, the wall between the current cell and that unvisited cell is removed. This process continues until all cells have been visited at least once. In the implementation, the random walk was allowed to continue until all cells were visited. The current cell was updated after each step, and a visited counter was used to stop the walk.

The principle of Aldous-Broder is different from the previous two algorithms. It does not use any backtracking or directional bias. Every possible spanning tree of the grid has the same probability of being generated. This property is called uniform. The advantage of Aldous-Broder is that it produces the most unbiased mazes among the three algorithms [8]. It is very useful for applications where randomness is important. The disadvantage is its speed. Because the random walk may revisit already visited cells many times, the running time can be very long. In theory, the average running time is proportional to the number of cells times the logarithm of the number of cells, but for large grids it can be much slower than Recursive Backtracking or Binary Tree.

DOI: 10.54254/2753-8818/2026.PU35550

2.2. Solving algorithm: Breadth-First Search (BFS)

To solve the generated mazes, a traditional BFS algorithm was used. BFS is a standard graph traversal algorithm that explores all cells at the current distance before moving to the next distance. In the context of a grid maze, each cell is a node, and each open passage is an edge. BFS starts from the start cell and expands outward layer by layer until the goal cell is reached [1, 9].

The principle of BFS is to use a queue. The start cell is placed in the queue and marked as visited. Then, while the queue is not empty, the front cell is taken out. All its unvisited neighbors are examined. If a neighbor is reachable (no wall between them), it is added to the queue and marked visited. The algorithm also records the parent of each visited cell. When the goal cell is reached, the shortest path is reconstructed by following the parents backward. In the implementation, the standard Queue module from OCaml's standard library was used. The parent array was stored as a 2D array of optional coordinates. The path was reconstructed only after the goal was found, so that the solving time included both exploration and path building.

The advantage of BFS is that it always finds the shortest path in an unweighted grid [1]. This makes it a fair solver for comparing different mazes. Comparative studies in OCaml have confirmed that BFS is reliable and efficient for maze solving applications [9]. The disadvantage is that BFS may explore many unnecessary cells if the maze is large and the goal is far, but it is still efficient for grid sizes used in this study.

2.3. Experimental setup

All algorithms were implemented in OCaml (version 5.4.1). No external libraries beyond the standard OCaml library (including Random, Queue, and Sys.time) were used. The experiments were run on a computer with an Intel Core i5-8250U processor and 8 GB of RAM.

2.3.1. Hyperparameters

The following hyperparameters were kept constant for all experiments. The grid size was 250×250 cells. This size was chosen because it is large enough to show performance differences but small enough to run many tests quickly. The start cell was set to the bottom-right corner (rows-1, cols-1), and the goal cell was the top-left corner (0, 0). The number of repetitions for each algorithm was 20 times per run, and the entire measurement was repeated 50 times for each algorithm. This reduced the effect of random fluctuations. Different random seeds were used for each repetition, and the same set of seeds was used for all three generation algorithms to make the comparison fair.

2.3.2. Measured metrics

Two metrics were measured for each generation algorithm: (1) generation time, the time taken to build the maze (in seconds); and (2) solving time, the time taken by BFS to solve the generated maze (in seconds). Both times were measured using OCaml's Sys.time() function, which returns CPU time in seconds. The values were recorded as seconds for accuracy. Each algorithm was tested for 50 runs, each run being the average of 20 generations. The average generation time and average solving time were then computed from the 50 runs.

DOI: 10.54254/2753-8818/2026.PU35550

3. Results and discussion

In this section, the experimental results are presented and discussed. Table 1 shows the average generation time and average BFS solving time for each of the three maze generation algorithms. All times are in seconds, averaged over 50 independent runs. For each run, the generation time was itself the average of 20 repeated generations. The maze size was 250×250 cells. The start cell was the bottom-right corner, and the goal cell was the topleft corner. The solving algorithm was BFS in all cases.

Table 1. Generation and BFS solving times of the three maze generation algorithms

Sequence	Binary Tree Algorithm Generation	Recursive Backtracking Algorithm Generation	Aldous Broder Algorithm Generation	BFS Solving for Binary Tree Algorithm	BFS Solving for Recursive Backtracking Algorithm	BFS Solving for Aldous Broder Algorithm
1	0.0531	0.1406	3.8961	0.0094	0.0258	0.0031
2	0.0485	0.1375	3.6539	0.0094	0.0297	0.0032
3	0.0484	0.1469	4.1289	0.0110	0.0273	0.0031
4	0.0469	0.1367	4.0703	0.0101	0.0219	0.0039
5	0.0492	0.1453	4.0054	0.0094	0.0219	0.0031
...	...	...	...	...	...	...
48	0.0500	0.1391	3.8844	0.0093	0.0242	0.0023
49	0.0461	0.1406	4.3093	0.0086	0.0172	0.0024
50	0.0508	0.1383	3.7156	0.0078	0.0273	0.0023
Average	0.0485	0.1416	4.2691	0.0089	0.0243	0.0029

3.1. Description of results

Binary Tree had the fastest generation time, with an average of 0.0485 seconds. Recursive Backtracking was slower, at 0.1416 seconds, which is about three times slower than Binary Tree. Aldous-Broder was significantly slower than both, taking 4.2691 seconds on average. The generation time of Aldous-Broder was about 30 times longer than Recursive Backtracking and about 88 times longer than Binary Tree.

For solving times, the pattern was different. The maze generated by Binary Tree was solved by BFS in 0.0089 seconds on average. The maze generated by Recursive Backtracking took the longest to solve, at 0.0243 seconds. The maze generated by Aldous-Broder had the shortest solving time, at only 0.0029 seconds. Solving times varied more across algorithms than might be expected, with Aldous-Broder's solving time being about one-third of Binary Tree's and about one-eighth of Recursive Backtracking's.

3.2. Analysis and discussion

The large difference in generation times can be explained by the characteristics of each algorithm. Binary Tree only visits each cell once and makes a fixed number of random choices per cell. Its O(rows × columns) time complexity is reflected in the very low generation time, even for a 250×250 grid (62,500 cells). Recursive Backtracking also visits each cell once, but it requires random

DOI: 10.54254/2753-8818/2026.PU35550

selection among unvisited neighbors and backtracking steps. These extra operations make it slower than Binary Tree, as seen in the results.

Aldous-Broder is known to be much slower because its random walk revisits already visited cells many times before finding all unvisited cells [4, 5]. The measured generation time of 4.27 seconds matches this expectation. For a 250×250 grid, the random walk may take many times the number of cells to complete. This confirms that Aldous-Broder is not suitable for large mazes when generation speed is important. These findings are consistent with other comparative studies that rank Aldous-Broder as one of the least efficient generation algorithms [6, 10].

The solving times show an interesting and unexpected pattern. The maze generated by Aldous-Broder, despite being the slowest to generate, was the fastest to solve. One possible explanation is that Aldous-Broder produces a uniform spanning tree, which tends to have a balanced structure. In a balanced maze, the shortest path from the bottom-right to the top-left may be relatively direct, and BFS may need to explore fewer cells before reaching the goal. The very low solving time of 0.0029 seconds supports this view.

In contrast, Recursive Backtracking tends to create long, winding paths with few branches. Such a maze may force the shortest path to be longer, requiring BFS to explore more cells. This explains why Recursive Backtracking had the highest solving time (0.0243 seconds). Binary Tree mazes have a strong diagonal bias. From the bottom-right start to the top-left goal, the bias may create a relatively direct diagonal corridor, but not as direct as the uniform maze from Aldous-Broder. Hence its solving time (0.0089 seconds) lies between the other two.

Another observation is that solving times for Aldous-Broder were not only the lowest but also showed very little variation (as can be inferred from the raw data). This suggests that uniform mazes provide consistent path lengths for BFS. Recursive Backtracking mazes, on the other hand, likely produce high variation in path lengths, leading to larger average solving time. A comprehensive analysis of multiple solving algorithms similarly found that BFS performance varies significantly with maze structure [11].

For a beginner programmer, if the goal is to generate many large mazes quickly, Binary Tree is clearly the best choice. However, if the application requires fast subsequent solving (e.g., in a game where many mazes need to be solved repeatedly), the high generation cost of Aldous-Broder might be justified by its very fast solving time. Recursive Backtracking appears to be the worst choice for both generation and solving speed in this large-grid setting.

However, this study also has some limitations. First, the grid size was fixed at 250×250, and the start and goal positions were also fixed from the bottom-right corner to the top-left corner. Different sizes or different start-goal pairs might change the relative solving times. Also, only one solving algorithm (BFS) was tested. Other solvers such as A* or DFS might show different patterns. Finally, the generation time for Aldous-Broder may become prohibitively large for even bigger grids, making it impractical despite its solving advantage.

4. Conclusions

Returning to the initial observation that maze generation and solving are two interconnected but distinct problems, this study has empirically compared three generation algorithms paired with BFS solving. The main methodological contribution is the implementation of all algorithms in OCaml and the measurement of both generation and solving times under controlled hyperparameters. The results confirm that Binary Tree is fastest for generation, Aldous-Broder generates mazes that are fastest to solve for a 250×250 grid. These findings suggest that for applications requiring many generations (e.g., random maze generation for each game level), Binary Tree is preferable; for

DOI: 10.54254/2753-8818/2026.PU35550

scenarios where a single maze is generated once but solved repeatedly (e.g., puzzle design), the high generation cost of Aldous-Broder may be justified. Limitations include the fixed grid size and fixed start-goal positions, as well as the use of only one solving algorithm (BFS). Future work could extend the comparison to other grid sizes, different start-goal pairs, and additional solving algorithms such as A* or Dijkstra. A further direction is to implement and compare uniform generation algorithms like Wilson's algorithm against Aldous-Broder in OCaml.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
[2] Boldovs, I., Vinogradovs, J., & Grabusts, P. (2016). Comparison of maze generation algorithms. In *HET*
[3] Vijaya, J., Gopu, A., Vinayak, K. V. S. G., Singh, T. J., & Malhotra, K. (2024). Analysis of maze generation algorithms. In *2024 5th International Conference on Innovative Trends in Information Technology (ICITIIT)* (pp. 1-6). IEEE.
[4] Aldous, D. (1990). The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM Journal on Discrete Mathematics, 3*(4), 450–465.
[5] Broder, A. (1989). Generating random spanning trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)* (pp. 442–447).
[6] Čarapina, M., Staničić, O., Dodig, I., & Cafuta, D. (2024). A comparative study of maze generation algorithms in a game-based mobile learning application for learning basic programming concepts. *Algorithms, 17*(9), 404.
[7] Zhang, H. (2025). Comparative implementation of binary tree and recursive backtracking maze generation algorithms in OCaml. In *Proceedings of the 2nd International Conference on Engineering Management, Information Technology and Intelligence (EMITI)* (pp. 246–251). SciTePress.
[8] Wang, X. (2024). OCaml-based recursive backtracking and Aldous Broder algorithms in solving maze challenges. *AIP Conference Proceedings, 3194*(1), 050011.
[9] Wang, Q. (2024). Comparative analysis of maze solving algorithms based on OCaml. *AIP Conference Proceedings, 3194*(1), 040001.
[10] Dagaev, A. V., Sorokin, A. A., Kovalenko, R. A., & Yakovleva, E. A. (2020). Mazes creation for further study of swarm intelligence. *IOP Conference Series: Materials Science and Engineering, 919*(5), 052058.
[11] Erbil, M. E., Özkahraman, M., & Bayrakçı, H. C. (2025). Comprehensive performance analysis and evaluation of various maze solving algorithms for optimized autonomous navigation and pathfinding. *Fırat University Journal of Engineering, 37*(1), 151–166.