Comparison of Pathfiding Algorithms

Lovro Hauptman, Aleš Volčini
The Upper Secondary School of Electrical and Computer Engineering and Technical Gymnasium Ljubljana, Slovenia

***Abstract* – This paper compares five pathfinding algorithms: Dijkstra's Algorithm, A*, Depth-First Search (DFS), Breadth-First Search (BFS), and the Wall Follower. We evaluate their performance based on their ability to find the shortest path, execution speed, memory usage, and the number of maze cells explored. By testing these algorithms on a maze and analyzing them, this paper aims to highlight their respective strengths and weaknesses, providing insights into their efficiency and effectiveness.**

1 INTRODUCTION

Pathfinding algorithms are crucial in areas like navigation systems, robotics, and video games for determining the best route from one point to another. Although many studies have explored these algorithms, their performance can still vary significantly depending on the application.

This paper focuses on comparing four notable pathfinding algorithms—Dijkstra's, A*, Depth-First Search (DFS), and Breadth-First Search (BFS)—along with a simpler algorithm known as The Wall Follower. The Wall Follower, which simply follows the left or right wall of a maze, is included to provide a contrast with more complex algorithms.

The study involves developing a maze-solving application with a frontend built using React.js and a backend using Go (Golang). Testing is conducted across four different machines to evaluate each algorithm's execution time, memory usage, and exploration efficiency.

Through this comparison, the paper aims to offer insights into the performance of each algorithm, helping to identify the most effective approach for various types of pathfinding problems.

2 THEORETICAL BACKGROUND

In this section, we will briefly present the theoretical background necessary for understanding and comparing the selected pathfinding algorithms.

2.1 Big O Notation

Big O notation describes the time complexity of an algorithm, indicating how its execution time grows with the size of the input, denoted by n. It provides a way to express the upper bound on the number of operations performed by an algorithm. Common time complexities include:

- $O(1)$ – Constant time
- $O(\log n)$ – Logarithmic time
- $O(n)$– Linear time
- $O(nlo\,g\,n)$– Log-linear time
- $O(n2)$– Quadratic time

2.2 Graph Theory

Graph theory is a branch of mathematics focused on studying graphs. A graph consists of a set of nodes (vertices) connected by edges (links). Nodes can represent various structures such as locations on a map, people in a social network, or cells in a maze. Graphs can be directed or undirected, depending on whether the edges have a direction. They can also be connected, meaning there is a path between every pair of nodes, or disconnected, meaning some pairs of nodes have no path between them. Additionally, graphs can be weighted, with edges having values representing distances or capacities, or unweighted.

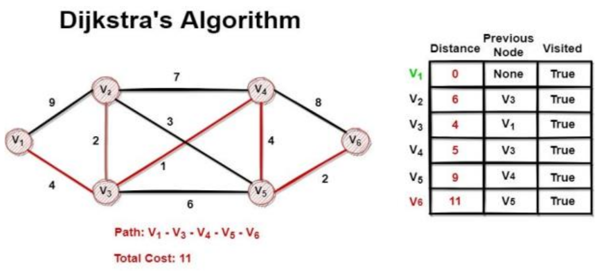
2.3 The Algorithms

In this section, we will explain how each of the algorithms work.

2.3.1 Dijkstra's Algorithm

Dijkstra's algorithm, named after Edsger W. Dijkstra, is used to find the shortest paths between nodes in a graph. The algorithm systematically explores the neighbors of a given node (in the case of a maze, this would be the starting point) and determines the shortest path to each other node in the graph. It begins with a graph G, represented as a set of nodes V and edges E. Initially, each node is assigned a tentative distance value: the distance from the start node is set to 0, while all others are set to Infinity, or in our case with Golang to the maximum value of an unsigned 32-bit int.

For each node, the algorithm examines all its neighbors and calculates their tentative distance via the current node. If the calculated distance is smaller than the current value, it updates the distance. The algorithm selects the unvisited node with the smallest tentative distance as the next node to visit and repeats the process until all nodes are visited or the target node is found.



	Distance	Previous Node	Visited
V_1	0	None	True
V_2	6	V3	True
V_3	4	V_1	True
V_4	5	V3	True
V_5	9	V4	True
V_6	11	V5	True

Figure 1: Showcase of Dijkstra’s Algorithm [1]

2.1.2 A*

The A* algorithm is a heuristic search algorithm that finds the shortest path between a starting node and the end node in a graph. It combines features of **uniform-cost search** and **greedy best-first search**.

- **Uniform-Cost Search:** This strategy explores nodes based on the lowest total cost from the start node, without considering any heuristic information. When applied in a maze it works like Dijkstra’s algorithm.
- **Greedy Best-First Search:** This strategy expands nodes based on the lowest estimated cost to reach the goal, as given by a heuristic function. It focuses solely on the estimated distance to the goal, which can make it faster but not necessarily optimal, as it doesn’t account for the cost to reach the node itself.

The A* algorithm integrates these strategies by assigning each node two values: the actual cost $g(n)$ from the start node to node n and the estimated cost $h(n)$ (heuristic function) from node n to the goal. The total cost of a node is a combination of the two:

$$f(n) = g(n) + h(n)$$

Heuristics are used to estimate the remaining cost from a node to the goal, guiding the search process. Common heuristic functions include:

- Euclidian Distance: Measures the straight-line distance between two points. It’s calculated as:

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Manhattan Distance: Measures distance based on a grid-like path, where movement is restricted to horizontal and vertical directions. It’s calculated as:

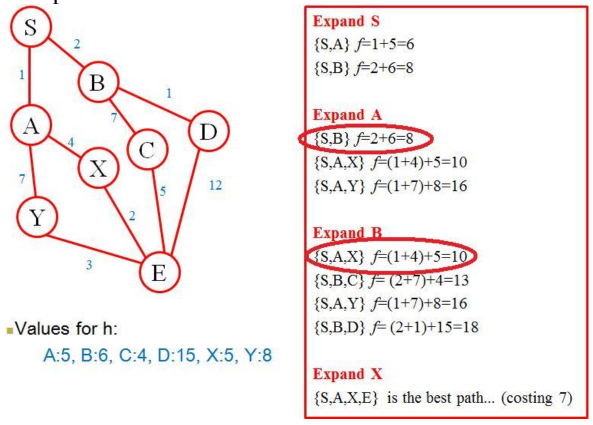
$$h(n) = |x_2 - x_1| + |y_2 - y_1|$$

- Canberra Distance: This heuristic is particularly sensitive to differences when the values of the coordinates are small. It is calculated as:

$$h(n) = \sum_{i=1}^{d} \frac{|x_i - y_i|}{|x_i| + |y_i|}$$

Since our problem involves navigating a maze, where choosing to move in one direction over another can lead to entirely different paths and outcomes, we opted to use the Canberra Distance for our implementation of the A* algorithm. This heuristic accounts for proportional differences in distances, making it particularly effective in environments where each movement decision can drastically alter the route and final position within the maze.

The A* algorithm maintains an open list (for nodes discovered but not yet expanded) and a closed list (for expanded nodes). At each step, it selects the node with the lowest $f(n)$ value from the open list, expands it, and moves it to the closed list. This process continues until the goal node is reached or no nodes remain in the open list.



Expand S
{S,A} f=1+5=6
{S,B} f=2+6=8

Expand A
{S,B} f=2+6=8
{S,A,X} f=(1+4)+5=10
{S,A,Y} f=(1+7)+8=16

Expand B
{S,A,X} f=(1+4)+5=10
{S,B,C} f= (2+7)+4=13
{S,A,Y} f=(1+7)+8=16
{S,B,D} f= (2+1)+15=18

Expand X
{S,A,X,E} is the best path... (costing 7)

Figure 2: Showcase of A* [2]

2.1.3 Breadth-Frist Search - BFS

BFS is a pathfinding algorithm that systematically explores all nodes at the present depth level before moving on to nodes at the next depth level. It starts from the initial node and explores all neighboring nodes before moving further out. BFS uses a queue to keep track of nodes that need to be explored, ensuring nodes are examined in the correct order.

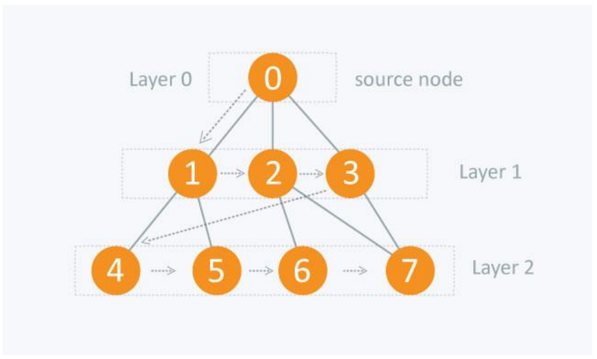


Figure 3: Showcase of BFS [3]

2.1.4 Depth-First Search - DFS

DFS is a pathfinding algorithm that explores as far down a branch as possible before backtracking. It checks all possible connections from a node in each direction before moving on to the next node. DFS can be implemented using a stack or recursively. It is crucial that the graph does not contain cycles, as the algorithm could otherwise enter an infinite loop.

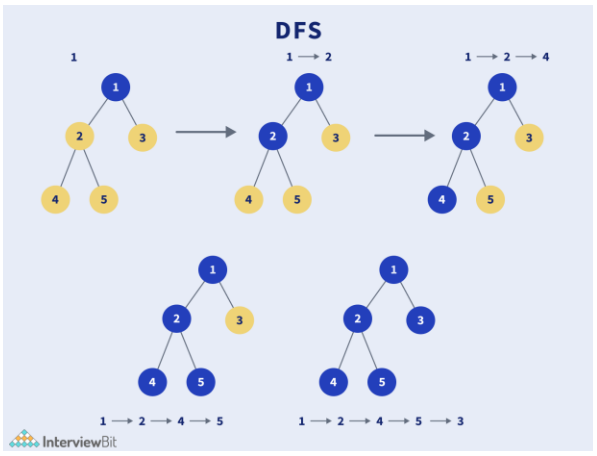


Figure 4: Showcase of DFS [4]

2.1.5 The Wall Follower

The Wall Follower Algorithm is a simple maze-solving strategy based on the idea that if one always keeps a hand on a wall (either left or right), they will eventually navigate through the maze to the exit. The algorithm begins by choosing a side to follow (left or right) and continues moving through the maze. At intersections, it consistently favors the chosen side. This process is repeated until the maze is solved.

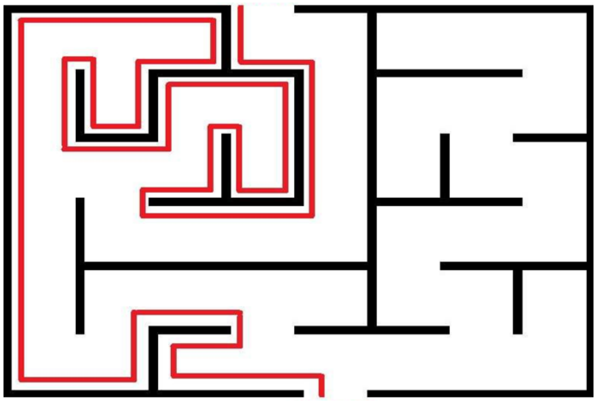


Figure 5: Showcase of Wall Follower [5]

3 IMPLEMENTATION

This section provides an overview of the implementation process for the algorithms, including the decision making behind the choice of technologies, and the integration of the frontend and backend components. We will discuss the maze generation, visualization methods, and the specific reasons for selecting the technologies used to build and optimize the system.

All code used is publicly available in these GitHub repositories:

- https://github.com/hundap/pathfinding_algorithms_visualizer
- https://github.com/hundap/pathfinding_algorithms_test_runner

3.1 Generating and Visualizing Mazes

The maze generation and visualization process involves both frontend and backend components to efficiently create and display maze data.

The frontend of our application is built with React.js, which allows us to dynamically visualize the algorithms' progress through the maze. React's state management allows us to update the algorithms' work through the maze step-by-step. The frontend interacts with a Go-based API to request maze data and display results to the user.

The backend is built with Go, chosen for its speed and concurrency capabilities. Go's efficient performance and ability to handle multiple threads simultaneously through goroutines make it well-suited for processing maze generation and solving algorithms. This concurrency allows the application to perform multiple tasks in parallel, significantly improving execution times and overall efficiency.

Upon receiving the maze data from the API, the frontend updates the visual representation of the maze. This includes mapping the grid data to visual elements and displaying the start and end nodes for each algorithm.

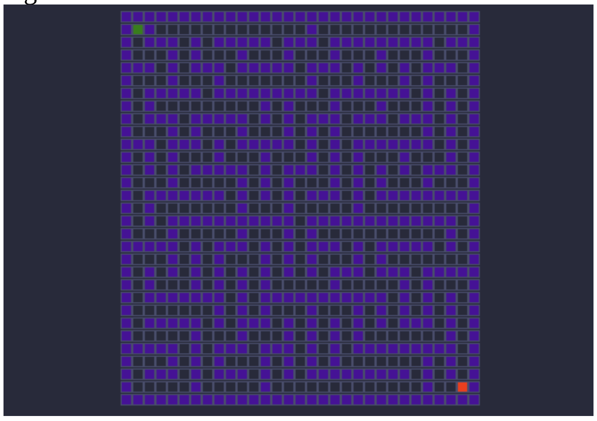


Figure 6: Closeup of a Maze Grid after API Response

3.1.1 The Problem of Generating Mazes

Maze generation poses unique challenges, especially when aiming to create mazes that are both complex and solvable. In our implementation, we used a recursive backtracker algorithm, which is a variant of

Depth-First Search (DFS) tailored for maze generation. This method efficiently creates mazes by recursively carving paths through a grid, ensuring that the resulting maze has a single, continuous path between any two points.

To introduce variability and complexity, we added randomness to the algorithm. Specifically, after generating the initial maze layout, we optionally remove a percentage of the internal walls. This process results in mazes with multiple possible paths, allowing us to test the algorithms' ability to find the most optimal paths. In addition, there is a 20 percent chance for the algorithm to backtrack and start carving a new path, which helps us avoid having a maze with a single path stretching through every node in the maze.

Additionally, we restricted our maze dimensions to odd numbers. This choice is crucial because even-numbered mazes would lead to two central columns and rows, complicating the maze generation. By using odd dimensions, we ensure a single central cell, simplifying the design and making the maze easier to manage.

3.2 Algorithm Implementations

All algorithms were implemented according to the theoretical descriptions provided earlier. They were implemented in Go due to its performance advantages and efficient concurrency features.

Dijkstra's Algorithm and A* both leverage a priority queue to manage and efficiently track nodes based on their distance or heuristic values. The other algorithms rely on Golang's built-in queue, stack data structures to manage nodes while solving the maze.

An interesting aspect of the implementation is the Wall Follower algorithm. While its logic is simple - just follow the wall - it presented unique challenges. Specifically, accurately tracking the direction in the maze was tricky. For example, turning left can mean different movements depending on your perspective, making the implementation more challenging than anticipated.

So again, the Frontend makes an API request to fetch the solutions and metrics, and the API sends the data back. The data is then interpreted by the animation functions and animated on the Frontend.

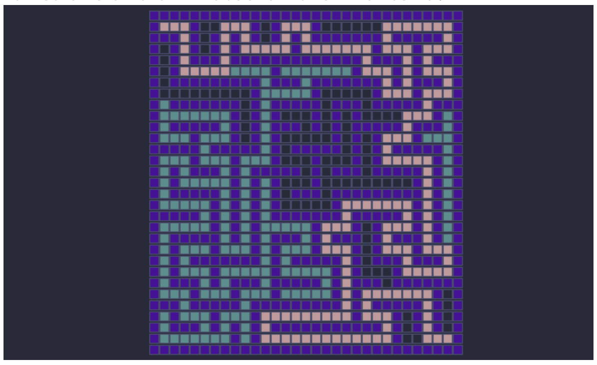


Figure 7: Closeup of a Solved Maze Grid

4 COMPARISON

In this section, we analyze the performance of the algorithms both theoretically and empirically. We compare their theoretical time complexities and compare and present the results from our testing across four different machines.

4.1 Theoretical Time Complexity

The theoretical time complexities for the algorithms are as follows:

- Dijkstra's Algorithm: $O(E + V \log V)$, where V is the number of vertices (nodes) and E is the number of edges.
- A*: $O(E)$ is the worst case, but typically $O(V \log V)$ with an efficient priority queue, where V is the number of vertices (nodes) and E is the number of edges. The performance of A* heavily depends on the heuristic used.
- BFS: $O(V + E)$, where V is the number of vertices (nodes) and E is the number of edges
- DFS: $O(V + E)$, where V is the number of vertices (nodes) and E is the number of edges
- Wall Follower: Wall Follower's time complexity is less straightforward to quantify since its dependent on the maze's layout and the chosen strategy (either going left or right). It will solve some mazes incredibly quickly and others incredibly slowly.

Simply going by these theoretical values, A* would be considered the fastest.

4.2 Empirical Testing

Each algorithm was tested on four different machines to measure their performance across varying hardware configurations:

- **Ubuntu ARM (Oracle Cloud Free Tier):** ARM CPU with 4 cores and 24 GB of RAM, released in 2021.
- **GitHub Codespace Ubuntu x86:** AMD Epyc 7763 with 4 cores and 16 GB of RAM, released in 2021.
- **Termux (Android) ARM:** Snapdragon 8 Gen 2 with 8 cores and 12 GB of RAM, released in 2022
- **Ubuntu WSL on Windows 11:** Ryzen 7 7700X with 8 cores and 32 GB of RAM, released in 2022.

Noteworthy is that the Oracle Cloud and GitHub Codespace machines are part of shared cloud environments, meaning they do not fully utilize their CPUs. For example, while the AMD Epyc 7763 CPU has 64 cores, only 4 were allocated for this testing.

The algorithms were run on mazes of sizes ranging from 25x25 until they either crashed, predominantly due to lack of RAM, or were manually stopped. For each maze size, each algorithm was executed 10 times on mazes with a single solution and 10 times on mazes with multiple solutions. From these runs, we collected the following data metrics:

- Execution time, measured in milliseconds
- RAM Usage, measured in megabytes
- Path Length
- Delta Path Length (for mazes with multiple solutions, the difference in path length compared to Dijkstra's Algorithm)
- Visited Percentage

From these multiple runs, averages were calculated to provide a comprehensive assessment of each algorithm's performance across different sizes.

4.3 Analysis

4.3.1 Pathfinding Accuracy

In our testing, Dijkstra's Algorithm and Breadth-First Search (BFS) consistently found the shortest path in all tested scenarios. A*, while theoretically guaranteed to find the shortest path, depends heavily on the effectiveness of its heuristic and the tie-breaking mechanism used when multiple nodes have the same cost. Additionally, errors stemming from floating-point arithmetic can impact the precision of the calculations, sometimes leading to suboptimal paths. In our experiments, A* found the most optimal path approximately 46.07% of the time. When A* did not find the absolute shortest path, it still generally provided a path that was more optimal compared to Depth-First Search (DFS) and Wall Follower, with an average path length delta of 0.61%, compared to the 1712.18% and 403.11% of DFS and Wall Follower respectively.

4.3.2 Visited Percentage

The visited percentage, representing the proportion of nodes explored by each algorithm, varies significantly:

Dijkstra's Algorithm and BFS typically have the highest percentage, averaging around 96.96% for mazes with a single solution and 99.85% for mazes with multiple solutions. This is because they ensure that the shortest path is found.

A* shows a lower visited percentage for smaller mazes (up to sizes of approximately 1451x1451), but this also goes up to around the same percentage as Dijkstra's algorithm and BFS to 96.07% for mazes with single solutions and 96.74% for mazes with multiple solutions respectively.

DFS generally has a lower visited percentage compared to the others at around 50.92% for single solution mazes and 14.20% for multi solution mazes. However, this is somewhat biased as the maze was generated by a variation of DFS, specifically the recursive backtracker, which may be influencing the result.

The Wall Follower algorithm exhibits the most variability. Its visited percentage can vary significantly, sometimes exceeding even 100%. This occurs because the Wall Follower can revisit nodes multiple times. Its average percentage is 99.30% for single solution mazes and 36.28% for multi solution mazes, but the overall percentages range from 1.18% to 139.35%.

4.3.3 Execution Time and Memory Usage

We compared memory usage (in Kilobytes) and execution time (in milliseconds) per 1000 nodes across all machines. The Ryzen 7 7700X performed best, followed by the Snapdragon 8 Gen 2, with the other two machines alternating between third and fourth place. Because the Ryzen 7 7700X was the fastest, we will focus on data from it in our analysis.

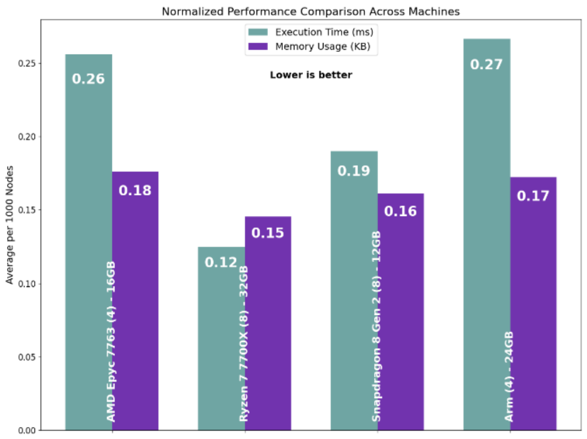


Figure 10: Performance Comparison Graph

In execution time DFS was the fastest, outperforming A*, the slowest, by 93.48%. Wall Follower and BFS were in the middle, running 81.46% and 74.23% faster than A*, respectively. Dijkstra's Algorithm, while still slower, outperformed A* by 35.97%. Although A* was theoretically expected to be the fastest based on its time complexity, our actual testing results show otherwise.

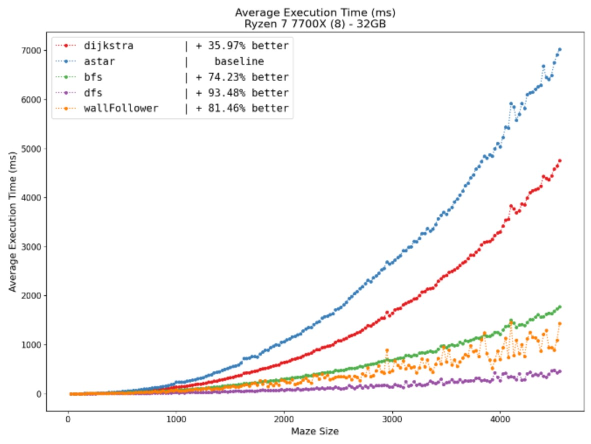


Figure 11: Algorithms Execution Time Comparison Graph

Memory measurements fluctuated due to Go's garbage collector. However, average values indicate that DFS used the least memory, outperforming Dijkstra's Algorithm by 43.67%. Wall Follower was second, beating Dijkstra by 21.82%, followed by A* with a 10.36% reduction, and BFS with a 1.85% reduction.

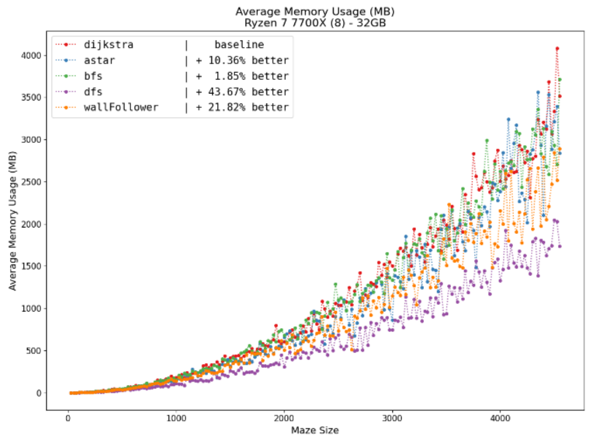


Figure 12: Algorithms Memory Usage Comparison Graph

This order of least to most memory-hungry algorithms corresponds to the order of algorithms that visit the fewest to the most nodes on average, particularly in mazes with multiple solutions.

5 CONCLUSIONS

Our comparison revealed that for single-solution mazes, DFS was the most efficient in execution time, memory usage, and visited percentage. For mazes with multiple solutions, A* emerged as the best, consistently delivering paths very close to the optimal, with room for improvement through better heuristics. A* is also adaptable to weighted graphs, making it versatile for more complex problems. The Wall Follower, though simple, performed surprisingly well, showing that even basic algorithms can be effective. Notably, Dijkstra's and BFS proved reliable and effective, making them strong choices for applications where finding the shortest path is crucial.

References

[1] https://miro.medium.com/v2/resize:fit:720/format:webp/0*3FMvRMEr3Sv6NkWt.jpeg

[2] https://miro.medium.com/v2/resize:fit:720/format:webp/0*LSXBwjDHVA3csu7C.jpg

[3] https://hes3.s3.amazonaws.com/media/uploads/fdec3c2.jpg

[4] https://www.interviewbit.com/blog/wp-content/uploads/2021/12/DFS-Algorithm-768x595.png

[5] https://www.researchgate.net/publication/315969093/figure/fig7/AS:668320751685645@1536351492897/Left-Wall-Follower-solvable-maze-7.jpg

[6] Amit Patel, "Amit's A* Pages," electronic file available at https://theory.stanford.edu/~amitp/GameProgramming/, accessed on March 2024.

[7] Jamis Buck, "Buckblog: Maze Generation: Growing Tree Algorithm," electronic file available at https://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm, accessed on March 2024.

[8] Wikipedia, "Wikipedia," electronic file available at https://www.wikipedia.org/, accessed on March 2024.

[9] Meta Platforms, Inc., "Getting Started – React," electronic file available at https://legacy.reactjs.org/docs/getting-started.html, accessed on March 2024.

[10] Go.dev, "Go Documentation," electronic file available at https://go.dev/doc/, accessed on June 2024.

[11] Stack Overflow, "Free Algorithms Book," electronic file available at https://books.goalkicker.com/AlgorithmsBook/, accessed on March 2024.

[12] Daniel Monzonís Laparra, *Pathfinding Algorithms in Graphs and Applications*, Barcelona, 2019.