Fırat Üniversitesi Müh. Bil. Dergisi
37(1), 151-166, 2025

Araştırma Makalesi
https://doi.org/10.35234/fumbd.1518386

Comprehensive Performance Analysis and Evaluation of Various Maze Solving Algorithms for Optimized Autonomous Navigation and Pathfinding

Mustafa Emre ERBİL[1], Merdan ÖZKAHRAMAN[2*], Hilmi Cenk BAYRAKÇI[3]
[1,2,3] Mechatronics Engineering, Faculty of Technology, Isparta Univesity of Applied Sciences, Isparta, Türkiye
[1] mail@mustafaemreerbil.com, [*2] merdanozkahraman@isparta.edu.tr, [3] cenkbayrakci@isparta.edu.tr

(Geliş/Received: 19/07/2024; Kabul/Accepted: 11/11/2024)

Abstract: Recent advancements in technology have led to the widespread use of maze-solving algorithms in various applications, such as autonomous robots, GPS-based navigation systems, smart traffic management systems, and healthcare services. This study provides a comprehensive comparative analysis of the performance of several maze-solving algorithms, including A*, Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra, Flood Fill, Random Mouse, and Recursive Backtracker. The algorithms were evaluated based on key performance metrics such as solution speed, memory usage, and CPU consumption. The results indicate that while the DFS algorithm demonstrates the fastest solution time with minimal memory usage, it has higher CPU consumption. In contrast, the Random Mouse algorithm is the least efficient, showing the highest memory and CPU usage along with the longest solution time. The A* algorithm, although efficient in finding the shortest path, showed moderate performance in both memory and CPU usage. These findings offer valuable insights into the strengths and weaknesses of each algorithm, providing guidance for future improvements and applications in real-world scenarios. This study aims to be a valuable resource for researchers and engineers focused on enhancing the efficiency of maze-solving algorithms in various technological domains.

Keywords: Autonomous systems, maze solving algorithms, navigation technologies, pathfinding optimization, performance analysis.

Çeşitli Labirent Çözme Algoritmalarının Optimum Otonom Navigasyon ve Yol Bulma için Kapsamlı Performans Analizi ve Değerlendirilmesi

Öz: Teknolojideki son gelişmeler, otonom robotlar, GPS tabanlı navigasyon sistemleri, akıllı trafik yönetim sistemleri ve sağlık hizmetleri gibi çeşitli uygulamalarda labirent çözme algoritmalarının yaygın olarak kullanılmasına yol açmıştır. Bu çalışma, A*, Genişlik Öncelikli Arama (BFS), Derinlik Öncelikli Arama (DFS), Dijkstra, Flood Fill, Random Mouse ve Recursive Backtracker dahil olmak üzere çeşitli labirent çözme algoritmalarının performansının kapsamlı karşılaştırmalı bir analizini sunmaktadır. Algoritmalar, çözüm hızı, bellek kullanımı ve CPU tüketimi gibi ana performans metrikleri temelinde değerlendirilmiştir. Sonuçlar, DFS algoritmasının minimal bellek kullanımı ile en hızlı çözüm süresini gösterirken, daha yüksek CPU tüketimine sahip olduğunu göstermektedir. Buna karşılık, Random Mouse algoritması en verimsiz olup, en yüksek bellek ve CPU kullanımının yanı sıra en uzun çözüm süresini göstermektedir. A* algoritması, en kısa yolu bulmada verimli olmasına rağmen, bellek ve CPU kullanımında orta düzeyde performans göstermiştir. Bu bulgular, her bir algoritmanın güçlü ve zayıf yönlerine ilişkin değerli bilgiler sunmakta ve gerçek dünya uygulamalarında gelecekteki iyileştirmeler ve uygulamalar için rehberlik sağlamaktadır. Bu çalışma, labirent çözme algoritmalarının verimliliğini artırmayı hedefleyen araştırmacılar ve mühendisler için değerli bir kaynak olmayı amaçlamaktadır.

Anahtar kelimeler: Otonom sistemler, labirent çözme algoritmaları, navigasyon teknolojileri, yol bulma optimizasyonu, performans analizi.

1. Introduction

The rapid advancement of technology has led to the widespread use of maze-solving algorithms. These algorithms are critically important in various fields such as autonomous robots, agricultural drones, GPS-based navigation systems for exploration and search-and-rescue operations, smart traffic management systems for optimizing traffic flow, and healthcare services for patient transportation and material distribution [1-3]. This growing importance necessitates a thorough examination and enhancement of these algorithm's performance. Autonomous robots must navigate complex paths and obstacles to increase efficiency in industrial environments such as storage facilities and factories [4]. Maze-solving algorithms play a crucial role in enhancing these

[*] Sorumlu yazar: merdanozkahraman@isparta.edu.tr. Yazarların ORCID Numarası: [1] 0009-0003-9394-8588, [2] 0000-0002-3501-6497, [3] 0000-0001-5064-7310

capabilities. Agricultural drones utilize these algorithms to effectively navigate large farming areas and optimize agricultural activities [5, 6]. GPS-based navigation systems require these algorithms to find the fastest and safest routes in challenging and hazardous environments during exploration and search-and-rescue operations. Smart traffic management systems use optimized pathfinding solutions to regulate urban traffic flow and reduce congestion [7, 8]. In healthcare services, systems that transport patients or distribute materials in hospitals benefit from these algorithms to increase efficiency and enable quick response in emergencies.

In this study, the performance of A* (A-Star), Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra, Flood Fill, Random Mouse, and Recursive Backtracker algorithms are comparatively analyzed on the same maze. Each algorithm's performance metrics, including solution speed, memory usage, and CPU consumption, were evaluated and the results were analyzed comparatively. Additionally, the scenarios in which each algorithm is more efficient and the conditions under which their performance declines were examined in detail. Evaluating the performance of maze-solving algorithms is crucial for understanding their effectiveness in real-world applications. This assessment reveals the strengths and weaknesses of the algorithms, providing valuable insights for future development efforts. This study aims to determine which algorithm is more suitable under specific conditions by evaluating the performance of different algorithms in various applications.

The objectives of this study are as follows:

- To compare the solution speeds, memory usage, and CPU consumption of different maze-solving algorithms.
- To analyze how these algorithms perform in various application scenarios.
- To identify the strengths and weaknesses of the algorithms and determine the conditions in which they are more suitable.
- To provide recommendations for improving the efficiency of these algorithms in real-world applications based on the results obtained.

This study aims to be a valuable resource for researchers and engineers focused on enhancing the efficiency of maze-solving algorithms, particularly in autonomous systems and navigation technologies. The findings will guide the improvement of these algorithms and the development of new approaches.

2. State of the Art

The A* algorithm is a widely used method for solving pathfinding and graph traversal problems. It was developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael at the Stanford Research Institute. The A* algorithm combines the benefits of Dijkstra's algorithm and the Greedy Best-First Search algorithm, leveraging both approaches' strengths. This algorithm operates using a heuristic function, which is a form of guided search, to find the path from the starting point to the goal with the lowest cost [9]. A* is preferred in many applications because it is both precise and efficient. By employing the heuristic function, it accelerates the search process by evaluating only the nodes that are close to the goal. This significantly enhances the algorithm's performance and makes it effective in numerous real-world applications [10, 11]. The heuristic function speeds up the search process. The use of a properly selected heuristic guarantees the shortest path. The algorithm can quickly adapt to changing environmental conditions and recalculate paths as necessary.

The Breadth-First Search (BFS) algorithm is designed to traverse a graph or tree data structure layer by layer, aiming to locate a node with a specific property. The BFS algorithm was developed in 1959 by E. F. Moore in the context of finding the shortest path through a maze. It was independently rediscovered in 1961 by C. Y. Lee for pathfinding applications on circuit boards [12-14].

The Depth-First Search (DFS) algorithm is designed to explore all nodes in a graph or tree data structure by starting at a root node and diving as deep as possible before backtracking to discover all nodes. The foundations of DFS were established in the 19th century by French mathematician Charles Trémaux through his work on maze problems. Later, in the 1970s, John Hopcroft and Robert Tarjan formalized the algorithm and introduced various optimizations [15, 16].

The Dijkstra algorithm is a widely used algorithm for finding the shortest path between nodes in a graph [17, 18]. It was developed by the Dutch computer scientist Edsger W. Dijkstra in 1956 and published in 1959 [19]. The algorithm is designed to find the shortest paths from a starting node to all other nodes in a graph, and it can be applied to both directed and undirected graphs with non-negative edge weights. The Flood Fill algorithm is a graph or image processing algorithm used to identify contiguous regions of the same color from a starting point and fill these areas with a new color. The origins of the algorithm trace back to computer graphics and it is commonly known in paint programs as the "bucket fill" tool. The algorithm's initial uses date back to the 1970s, and it holds significant importance in the field of computer graphics [20, 21].

The Recursive Backtracker algorithm is a widely used method for maze solving and maze generation. The origins of the algorithm are generally rooted in the development of backtracking methods, with various contributions from scientists since the mid-20th century. However, the specific developer or exact date of the Recursive Backtracker algorithm's inception is not clearly defined. This algorithm is extensively utilized in computer graphics and robotics applications [22, 23].

The Random Mouse algorithm is a simple algorithm that attempts to solve a maze by following random paths. This algorithm aims to reach the target by moving in completely random directions without a specific exploration strategy. Although the origins and the inventor of the algorithm are unclear, it is known to be used for educational and research purposes in computer science and robotics [24-26]. Due to its simplicity and ease of implementation, the Random Mouse algorithm is employed for educational and research purposes. Because it does not rely on a specific strategy, it serves as a baseline for comparing the performance of heuristic and systematic algorithms. Additionally, it is used as a reference model for non-heuristic problems.

The table presents a comparative analysis of various maze-solving algorithms, highlighting their performance metrics across different test environments, as shown in Table 1. Each algorithm was tested on a simple 2D grid, a complex 2D maze, and a three-dimensional navigation scenario. The performance metrics evaluated include average solution time (in milliseconds) and memory usage (in megabytes).

Table 1. Comparative Performance Analysis of Maze Solving Algorithms Across Different Test Environments.

Tested Algorithms	**Test Platform**	**Performance**		**Reference**
		Average Solution Time	**Memory Usage**	
A*	Static 2D Grid (50x50)	36.806 seconds	Not provided	[27]
BFS	Hopper (40,000 cores) BFS time changes parallel to the number of cores	2.67 to 7.18 seconds	Not provided	[28]
DFS	Binary trees, tested on symmetric and asymmetric structures	Parallel time: 0.047 seconds, Sequential time: 0.094 seconds	Not provided	[29]
Dijkstra	Erdős-Rényi graphs with edge weights ranging from 1 to 1000, tested with 10 million nodes	25 ms	Not provided	[30]
Flood Fill	625 pixel	Solution time: 13 ms on the largest test	Not provided	[21]
Recursive Backtracker	16x16 and 32x32 2D grid	16x16: 35ms. 32x32: 65ms	16x16: 1.2 MB 32x32: 2.8 MB	[31]

As seen in Table 1, the A* algorithm demonstrates an average solution time of 36.806 seconds on a static 2D grid (50x50). The BFS algorithm, tested on the Hopper platform with 40,000 cores, reports an average solution time ranging between 2.67 and 7.18 seconds, with performance scaling in parallel to the number of cores. The DFS algorithm, tested on both symmetric and asymmetric binary trees, shows a parallel processing time of 0.047 seconds, while the sequential processing time reaches 0.094 seconds. The Dijkstra algorithm applied to Erdős-Rényi graphs (with edge weights ranging from 1 to 1000 and 10 million nodes), shows an average solution time of 25 milliseconds. For the Flood Fill algorithm, tested on a 16x16 grid, the solution time varied based on grid size, achieving 13 milliseconds on the largest test. The Recursive Backtracker algorithm reports solution times of 35 ms for 16x16 grids, and 65 ms for 32x32 grids, with memory usage ranging from 1.2 MB to 2.8 MB.

3. Method

In this study, a grid-based maze with dimensions of 51x51, generated using a Depth-First Search (DFS) algorithm, was utilized. This maze is represented as a binary matrix where each cell is either a path (0) or a wall

(1). The starting point of the maze is set at (0,0) and the ending point at (50,50). A maze of this size provides sufficient complexity and difficulty, making it ideal for testing the performance of various algorithms.

The primary reason for using such a maze is that it offers a structure that is both sufficiently complex and orderly for evaluating algorithm performance. Randomly generated mazes are effective in simulating real-world scenarios as they allow for testing across different sizes and levels of complexity. This enables a more realistic assessment of the algorithms' solution speed, memory usage, and CPU consumption. Additionally, random mazes allow for the observation of how different algorithms perform under various conditions [32].

To generate the maze, a DFS-based algorithm and the generate_maze function, written in Python, were used. The algorithm produces a random maze using a specific seed value (seed=42). This is crucial for ensuring the repeatability of the experiments; the same seed value will generate the same maze in each run. The algorithm starts from the initial point, visits all cells, and creates paths by removing the walls between randomly selected neighboring cells [33, 34]. The generated maze was visualized using the plot_solution function, as shown in Figure 1.

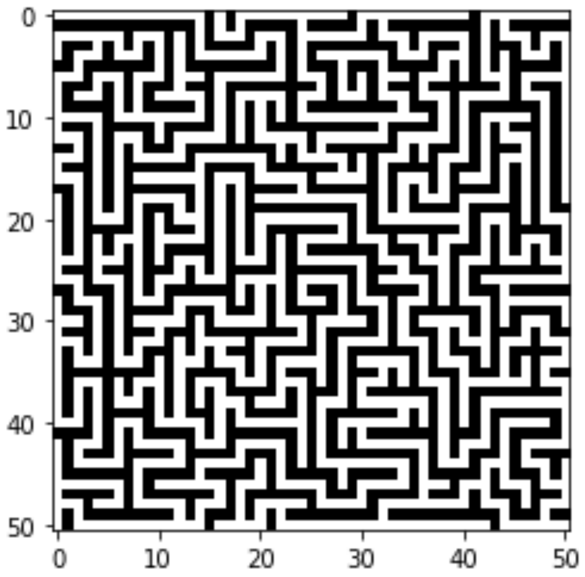

Figure 1. The maze created for the experimental study.

The maze generation process follows these steps:

- The starting point (0,0) is selected, and a stack is used to keep track of unprocessed cells.
- From the current cell in the stack, a random neighbor is chosen from the four main directions (right, left, up, down).
- If the chosen neighboring cell has not been visited yet, the wall between them is removed, and this cell is added to the stack.
- This process continues until all cells have been visited.

The flowchart of these operations is shown in Figure 2.

This method not only creates complex and challenging mazes but also provides an ideal environment for testing the performance of algorithms. The uniqueness and repeatability of the generated mazes ensure consistency in the comparative analysis of the algorithms [34, 35].

The Depth-First Search (DFS) algorithm was chosen for the maze generation process because it is simple, efficient, and easy to implement. DFS uses a deep search strategy to systematically visit all cells of the maze and create paths through random selections. It is efficient in terms of memory usage and, due to its recursive nature, can generate complex maze structures. Additionally, the use of DFS ensures that the algorithm effectively visits all cells starting from a specific point, guaranteeing that the maze is unique and complex each time. These characteristics make DFS an ideal algorithm for maze generation [32, 36].

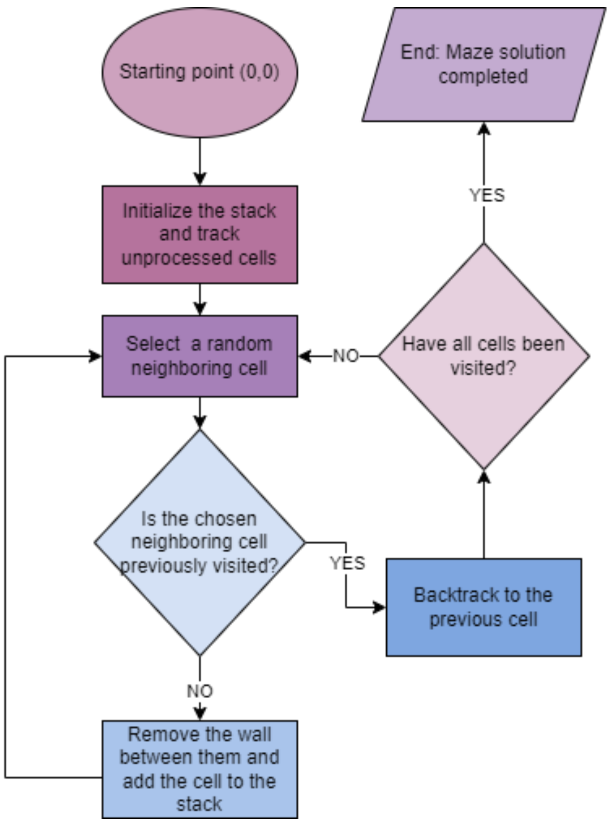
Starting point (0,0)
Initialize the stack and track unprocessed cells
Select a random neighboring cell
Is the chosen neighboring cell previously visited?
YES
Backtrack to the previous cell
NO
Remove the wall between them and add the cell to the stack
Have all cells been visited?
NO
YES
End: Maze solution completed

Figure 2. Flowchart of maze solve.

3.1. A* (A-Star) algorithm

This method not only creates complex The A* algorithm finds the shortest path between nodes in a graph by using both cost (g) and heuristic (h) functions. The algorithm maintains two lists: an open list (nodes yet to be evaluated) and a closed list (nodes already evaluated). It selects the node with the lowest f(n) = g(n) + h(n) value from the open list and evaluates it. When the goal node is reached, the algorithm has found the shortest path [18]. Basic functions on mathematical calculations of the A* Algorithm:

- *g(n)* : The actual cost from the start node to node n.
- *h(n)* : The estimated cost from node n to the goal node (heuristic function).
- *f(n)* : The total estimated cost of reaching the goal through node nnn. It is the sum of the two components as shown in Equation 1.

$$f(n) = g(n) + h(n) \quad (1)$$

where *g(n)*, represents the actual cost to reach a particular node, which is the sum of the transition costs at each step. *h(n)*, represents the estimated cost from node nnn to the goal, typically determined using different heuristic functions depending on the problem-solving strategy [37].

The heuristic function *h(n)*, A argely determines the efficiency of the A* algorithm. This function is used to estimate the shortest path from a node to the goal. An ideal heuristic function should be:

- Consistent: $h(n) \leq c(n,m) + h(m)$
- Admissible: $h(n) = 0$ for the goal node and should not overestimate the actual cost for all other nodes.

Manhattan distance, the sum of the horizontal and vertical steps between two points as shown in Equation 2 [37].

$$h(n) = |x1 - x2| + |y1 - y2| \quad (2)$$

Euclidean distance, the straight-line distance between two points as shown in Equation 3.

$$H(n)= (x1-x2)^2 + (y1-y2)^2 \quad (3)$$

In the best case, if the heuristic function provides a perfect estimate, the time complexity is $O(b^d)$ where b is the branching factor at each node and d is the depth of the solution. In the worst case, if the heuristic function is weak, the time complexity is $O(|E|)$, where E represents the number of edges in the graph.

The space complexity is $O(|V|)$, where V represents the number of nodes in the graph. This accounts for the total number of nodes stored in the open and closed lists. Step-by-step process of an algorithm:

- Initialization: Add the start node to the open list.
- Selecting the Node with the Lowest
- Evaluation: If the goal node is reached, return success and the path. Otherwise, move node n to the closed list and evaluate its neighbors.
- Evaluating Neighbors: For each neighbor node, calculate the g and f values:
- Updating Lists: If the neighbor node is not in the open list, add it to the open list. If a lower-cost path is found in the open list, update the values.
- Repeated Evaluation: Return to step 2 and continue until the open list is empty.

The A* (A-Star) Algorithm was implemented using the astar function in Python to solve the maze. The maze solved using the A* (A-Star) algorithm is shown in Figure 3.

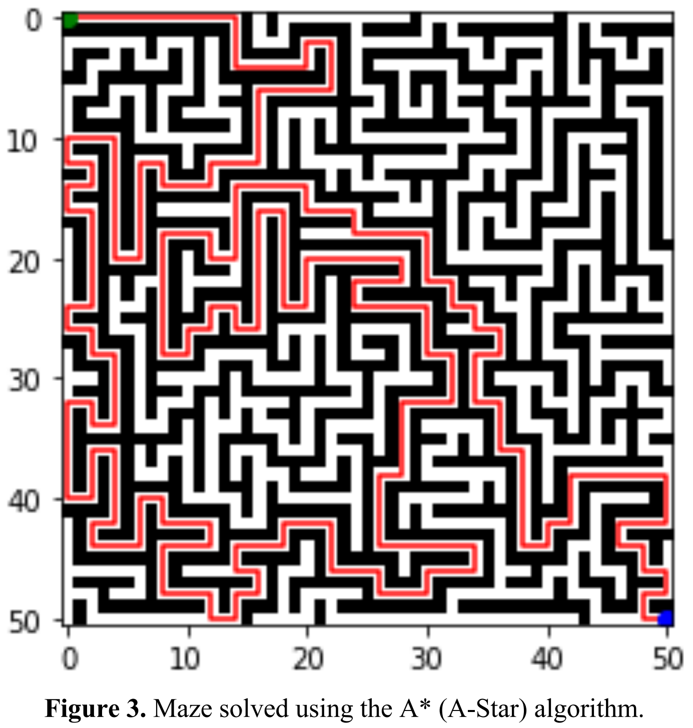

Figure 3. Maze solved using the A* (A-Star) algorithm.

3.2. BFS algorithm

The Breadth-First Search (BFS) algorithm is a fundamental and widely used algorithm for graph traversal and pathfinding problems. BFS starts at the root node and explores all nodes at the present depth level before moving on to nodes at the next depth level. This process is typically implemented using a queue data structure. The steps of the BFS algorithm are as follows:

- The start node is added to the queue and marked as visited.
- The node at the front of the queue is removed, and its neighboring nodes are added to the queue in order.
- This process continues until the goal node is found or all nodes have been visited.

The BFS algorithm is used to find the shortest path from a specific node to all other nodes by exploring nodes layer by layer. The mathematical calculations and complexity analysis of BFS are as follows:

Mathematical calculations and complexity analysis of the BFS algorithm, The time complexity of the BFS algorithm is expressed as O(V + E), where, V is the number of vertices in the graph. E is the number of edges in the graph. This time complexity arises because each vertex and each edge is visited at most once. BFS explores each vertex and edge in the graph only once. The space complexity of the BFS algorithm is O(V). This is because BFS stores and visits each vertex in the queue at most once. Therefore, the maximum number of vertices stored in memory is equal to the number of vertices in the graph. Step-by-step process of the a algorithm:

- Initialization: The starting node (s) is added to the queue and marked as visited.
- Loop: The node at the front of the queue (u) is removed. All neighbors (v) of node u are explored. If a neighboring node has not been visited, it is added to the queue and marked as visited. This process continues until the goal node is found or all nodes have been visited.

Reasons for preferring BFS, guarantees reaching the goal node if it exists. In unweighted graphs, BFS ensures finding the shortest path [14, 36, 38]. It is easy to implement and understand, using a queue that operates on the FIFO (First In, First Out) principle.

The effectiveness of the BFS algorithm stands out, especially in problems requiring the exploration of large areas and in scenarios where the shortest path needs to be found. Additionally, it has many applications in analyzing cycles and connections within graph structures. The maze solved using the BFS algorithm is shown in Figure 4.

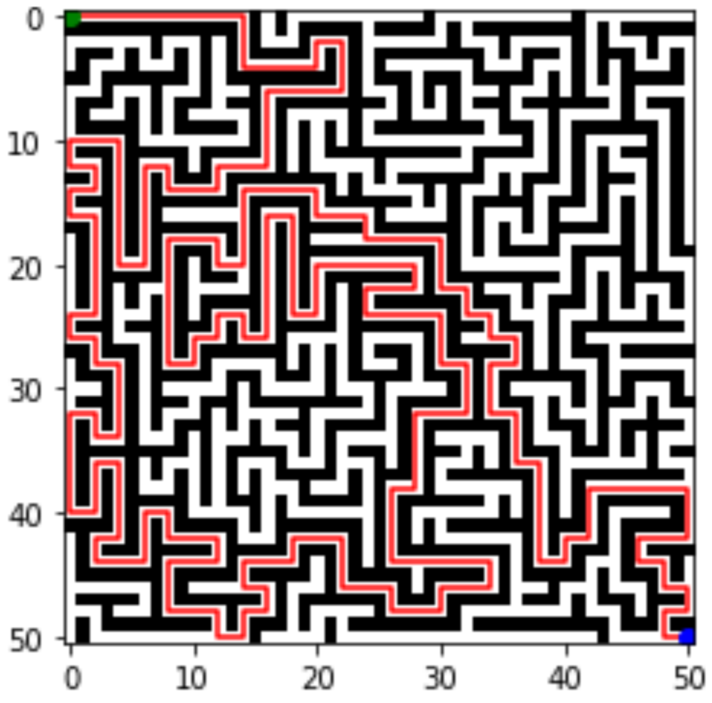

Figure 4. Maze solved using the BFS algorithm.

3.3. DFS algorithm:

The DFS (Depth-First Search) algorithm is used to visit all nodes in a graph or tree structure using a depth-first search strategy. The algorithm starts at the initial node and explores one neighboring node at a time. This process continues until there are no more neighbors to explore, at which point the algorithm backtracks to discover other nodes [33, 36]. DFS is typically implemented using a stack data structure. Here are the general steps of DFS:

- The starting node is added to the stack and marked as visited.
- The node at the top of the stack is removed, and its neighbors are visited.
- Each visited node is added to the stack, and the process continues in this manner.
- This process is repeated until all nodes have been visited.

Mathematical calculations of DFS, The time complexity of the DFS algorithm is expressed as *O(V + E)*, where, *V* represents the number of vertices. *E* represents the number of edges. This arises because each vertex and edge is visited at most once. The DFS algorithm processes nodes and edges only once, ensuring linear time complexity. The space complexity is *O(V)* because each vertex is added to the stack at most once. This means the maximum size of the stack will be equal to the number of vertices in the graph. DFS can also be implemented using the call stack due to its recursive nature, in which case the space complexity remains *O(V)*. Step-by-step process of the a algorithm:

- Adding the start node to the stack and marking it as visited
- Removing the node at the top of the stack and visiting its neighbors
- Repeating the process until all nodes are visited

Advantages and applications of the DFS algorithm, DFS guarantees visiting all nodes in a connected graph. This is ideal for situations where all nodes need to be explored. DFS can quickly find a path or solution by performing a deep search. This is particularly advantageous when the target node is deep within the graph. The algorithm is easy to implement and understand. It can be implemented simply using recursion or a stack. The DFS algorithm is preferred in many applications, such as pathfinding on maps, network traffic analysis, and movement planning in AI games [38]. The maze solved using the DFS algorithm is shown in Figure 5.

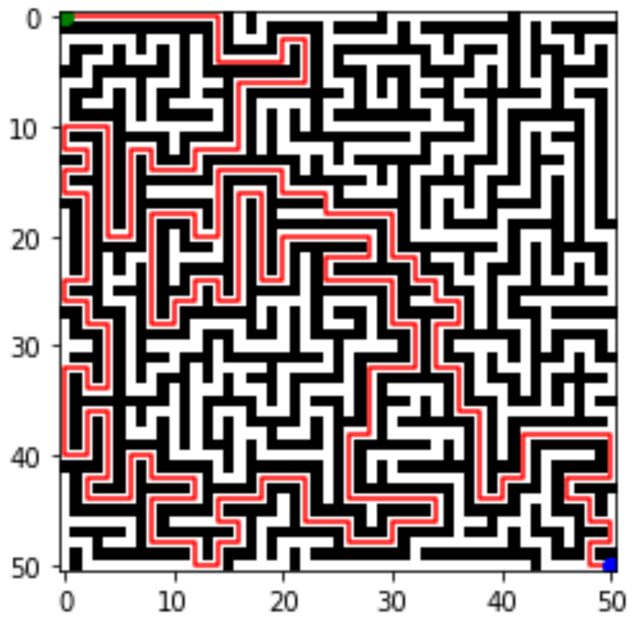

Figure 5. Maze solved using the DFS algorithm.

3.4. Dijkstra algorithm

The Dijkstra algorithm works similarly to the A* algorithm but does not use a heuristic function. The algorithm starts at the initial node and, at each step, selects the node with the shortest distance, then updates the distances to neighboring nodes through this node. This process continues until all nodes have been processed [17, 39]. The basic steps of the algorithm are as follows:

- Set the distance to the start node to zero and the distance to all other nodes to infinity.
- Add all nodes to the list of unprocessed nodes.
- Select the unprocessed node with the shortest distance.
- Update the distances to neighboring nodes through this node.
- Mark this node as processed and return to step 3.
- Repeat this process until all nodes have been processed.

Mathematical calculations of Dijkstra Algorithm, the time complexity of the Dijkstra algorithm is generally expressed as *O*(V^2), where, *V* represents the number of vertices. This time complexity is applicable when a simple priority queue data structure is used [24]. When more efficient data structures, such as the Fibonacci heap, are used, the time complexity as shown in Equation 4.

$$O(V \backslash log\ V + E) \tag{4}$$

Here, V represents the number of vertices. E represents the number of edges. The Fibonacci heap allows for more efficient management of the priority queue, so each node processing operation is performed in *O(log V)* time. This significantly reduces the overall time complexity [19]. Step-by-step process of the algorithm:

- Setting the distance to the start node to zero and the distance to all other nodes to infinity
- Adding all nodes to the list of unprocessed nodes
- Selecting the unprocessed node with the shortest distance
- Updating the distances to neighboring nodes through this node
- Marking this node as processed and returning to step 3
- Repeating the process until all nodes have been processed.

The Dijkstra algorithm guarantees the shortest path when the correct data structures are chosen. This feature enhances the algorithm's accuracy and reliability. It works efficiently in graphs with non-negative weights. In graphs without negative edge weights, it performs effectively in finding the shortest path. The Dijkstra algorithm has a wide range of applications, including pathfinding, network routing, and geographic information systems. In these areas, finding the shortest path ensures efficient resource utilization and enhances system performance [18]. The maze solved using the Dijkstra algorithm is shown in Figure 6.

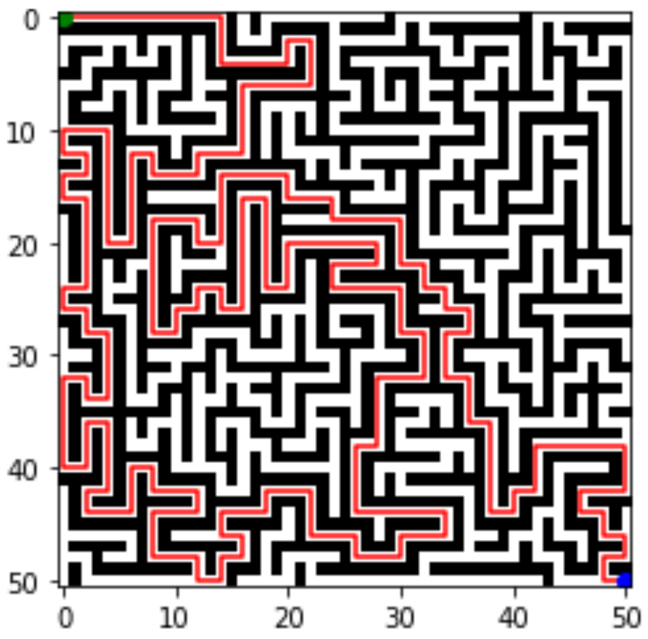

Figure 6. Maze solved using the dijkstra algorithm.

3.5. Flood Fill algorithm

The Flood Fill algorithm operates by using either four-directional or eight-directional connectivity. It starts by changing the color of the pixel at the initial point and then checks its neighboring pixels. It changes the color of neighboring pixels that are the same color, continuing this process until all connected pixels are filled with the new color [24]. The algorithm is typically implemented using stack or queue data structures. Here are the general steps of the algorithm:

- The starting pixel is added to the stack.
- The pixel at the top of the stack is removed, and its color is changed.
- The neighboring pixels are checked, and those of the same color are added to the stack.
- This process is repeated until the stack is empty.

The Flood Fill algorithm is used to fill a specific color or value starting from a given pixel in a graph or image. The algorithm visits each neighboring pixel, continuing this process until all relevant pixels are filled. Mathematical calculations of Flood Fill algorithm, The time complexity of the Flood Fill algorithm is *O(N)* , where

N represents the total number of pixels to be filled. The algorithm visits and processes each pixel at most once. The space complexity of the Flood Fill algorithm is also *O(N)*. This is because the algorithm uses stack (for DFS) or queue (for BFS) data structures, which can store all pixels in the worst-case scenario. The Flood Fill algorithm is preferred for many applications because it is easy to implement and understand. It provides fast and effective results in graphic and image processing applications [40]. It can work with different connectivity types (four or eight directions). Step-by-step process of the algorithm:

- Initial Pixel and Color Change: Assume the starting pixel is (x, y) and its color is c. The new color will be c'.
- Checking Neighboring Pixels: The algorithm checks neighboring pixels using either four-directional (up, down, left, right) or eight-directional (up, down, left, right, upper left, upper right, lower left, lower right) connectivity.

The Flood Fill algorithm starts from a specific pixel and fills all relevant pixels with the new color. The time and space complexities determine the efficiency of the algorithm, making it an ideal solution for many applications. The maze solved using the Flood Fill algorithm is shown in Figure 7.

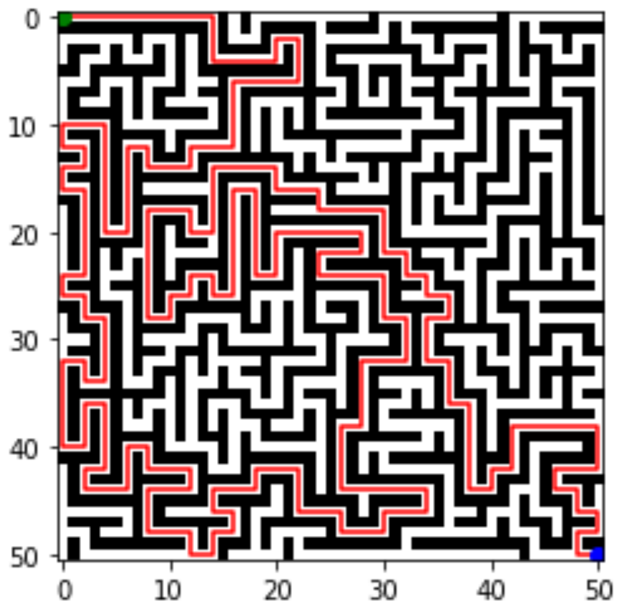

Figure 7. Maze solved using the Flood Fill algorithm.

3.6. Recursive Backtracker algorithm

The Recursive Backtracker algorithm is based on the depth-first search (DFS) method. It starts at the initial cell, randomly selects a neighboring cell, and moves to that cell. If the cell has not been visited before, it marks the cell as visited and continues this process until a dead-end is encountered. When a dead-end is reached, the algorithm backtracks to the previous cell and tries to visit other neighboring cells. This process continues until all cells have been visited [26]. Steps of the Algorithm:

- Start at the initial cell: Add the initial cell to the stack and mark it as visited.
- Randomly select a neighboring cell and move: From the current cell in the stack, randomly select a neighboring cell and move to that neighbor.
- Mark the cell as visited if it hasn't been visited before: If the new cell has not been visited, mark it as visited and add it to the stack.
- Backtrack if a dead-end is encountered: When a dead-end is reached, remove the cell from the stack, backtrack to the previous cell, and try to visit other neighboring cells.
- Repeat this process until all cells have been visited: Continue this process until all cells have been visited [22, 41].

Mathematical calculations of Recursive Backtracker Algorithm, The time complexity of the Recursive Backtracker algorithm is expressed as *O(V + E),* where *V* represents the number of vertices and *E* represents the number of edges. This complexity arises because each vertex and edge is visited at most once. The space complexity of the algorithm is *O(V)* because each vertex is added to the stack at most once. Being based on DFS, it processes each vertex using a stack data structure, which is stored in memory. Step-by-step process of the a algorithm:

- Cell Selection and Movement: Starting from the initial cell, the algorithm randomly selects a neighboring cell and moves to it.
- Visiting and Backtracking: Each visited cell is marked as visited. When a dead-end is encountered, the algorithm backtracks.

The Recursive Backtracker algorithm is preferred for many applications because, It is easy to implement and understand. It uses a stack data structure to perform depth-first search. It provides quick results, especially in maze generation and solving tasks. It explores various paths through random selections. It can work with different types of connectivity and can be adapted to various applications. It can operate with both four-directional and eight-directional connections. The maze solved using the Recursive Backtracker algorithm is shown in Figure 8.

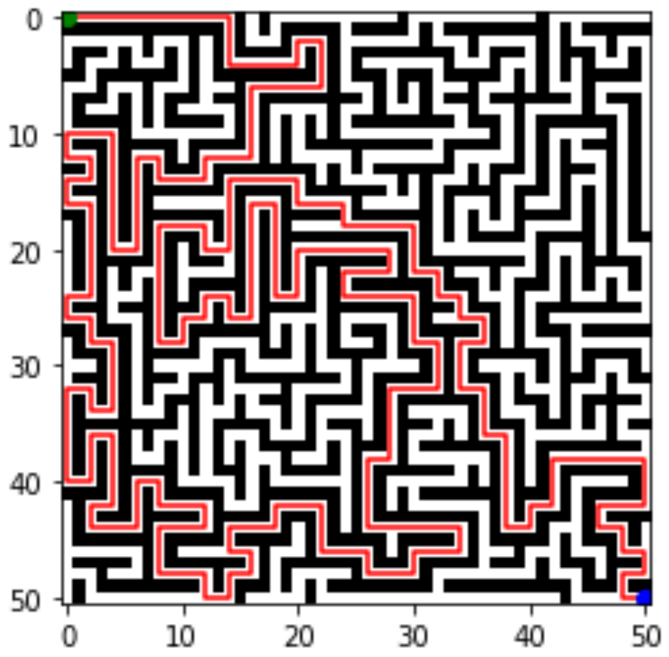

Figure 8. Maze solved using the Recursive Backtracker algorithm.

3.7. Random Mouse algorithm

The Random Mouse algorithm uses a completely random strategy without following any heuristic or systematic path. This algorithm starts from the initial point and moves by randomly selecting a direction at each step. If it reaches a dead-end or returns to a previously visited node, it randomly selects another direction and continues [25, 42]. This process continues until the target node is reached. Steps of the Algorithm:

- Start from the initial node: The algorithm starts from the initial node and marks this node as visited.
- Select a random direction and move: The algorithm randomly selects a neighboring node from the current node and moves to that node.
- Repeat the steps until the target node is reached: The algorithm continues moving by selecting random directions until it reaches the target node.

Mathematical calculations of Random Mouse Algorithm, The time complexity of the Random Mouse algorithm is not defined by a specific measure since it moves randomly. Theoretically, in the worst-case scenario, the time complexity could be *O(∞)* because the algorithm can take an infinite number of steps. However, in practice, the time complexity usually varies depending on the size and structure of the maze. The space complexity of the algorithm is also not defined by a specific measure. Since the algorithm moves randomly, the number of visited nodes and the path length are unpredictable. In the worst case, the algorithm may visit the entire maze, resulting in a space complexity of *O(V),* where *V* represents the number of nodes. Step-by-step process of the a algorithm:

- Random Movement and Visiting: The Random Mouse algorithm selects a random direction and moves at each step.
- Dead-End and Backtracking: When the algorithm encounters a dead-end or returns to a previously visited node, it continues to select a random direction.

Advantages of the algorithm, the algorithm is extremely simple and easy to understand. It works by selecting random directions, requiring no complex calculations. It is easy to implement and does not require any data structures or advanced algorithm knowledge. Disadvantages of the algorithm, since it moves randomly, it may take unnecessary steps and can be inefficient, especially in large and complex mazes. The algorithm's time to reach the goal is unpredictable and can sometimes result in an infinite loop. The maze solved using the Random Mouse algorithm is shown in Figure 9.

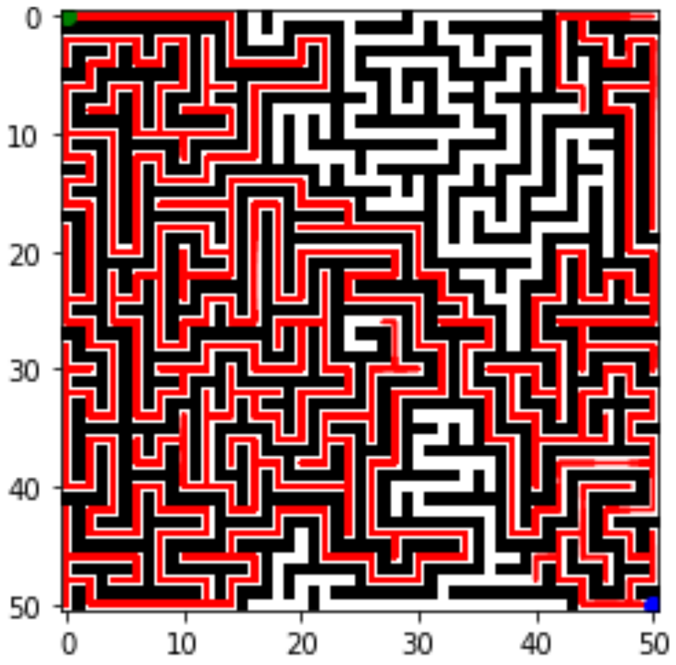

Figure 9. Maze solved using the Random Mouse algorithm.

4. Results

The performance analysis was conducted on a laptop with an Intel i7 12650H processor, 32GB DDR5 RAM, and an Nvidia Geforce RTX 4060 graphics card. The performance of each maze-solving algorithm was analyzed under the same conditions based on the following criteria:

- Solution Speed: The time it takes for the algorithm to solve the maze from the start point to the endpoint.
- Memory Usage: The amount of memory used by the algorithm during the solution process.
- CPU Consumption: The amount of CPU power used by the algorithm during the solution process.

These criteria are important for evaluating the efficiency and resource usage of the algorithm. Each algorithm was tested using the following steps:

- Algorithm Implementation: Each algorithm was implemented using the same code written in the Python programming language.
- Maze Solving: The maze created (Figure 2) had the top-left corner as the starting point and the bottom-right corner as the endpoint. The starting point was marked in green, and the endpoint in blue. The algorithm was then run to find the solution path from the green point to the blue point. The solved maze was visualized using the plot_solution function, with the path taken by the algorithm shown as a red line.
- Performance Measurement: The solution time, memory usage, and CPU consumption were measured and analyzed.

A* (A-Star) algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 164.63 KB of memory, took 0.007539 seconds, and CPU usage was 41.08%.

BFS Algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 104.69 KB of memory, took 0.005499 seconds, and CPU usage was 36.41%.

DFS algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 61.28 KB of memory, took 0.004622 seconds, and CPU usage was 43.30%.

Dijkstra algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 102.85 KB of memory, took 0.007037 seconds, and CPU usage was 44.06%.

Flood Fill algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 88.18 KB of memory, took 0.005994 seconds, and CPU usage was 33.42%.

Recursive Backtracker algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 75.59 KB of memory, took 0.005400 seconds, and CPU usage was 39.18%.

Random Mouse algorithm, memory usage and solution time were measured using the profile_and_visualize function in Python. To accurately measure CPU usage, the algorithm was run multiple times, and the average values were calculated. These measurements were performed using the tracemalloc, time.perf_counter, cProfile, and pstats libraries. During the solution, the algorithm used 23063.39 KB of memory, took 3.292647 seconds, and CPU usage was 89.90%. All test results are presented in Table 2.

Table 2. Comparison of algorithms based on conducted tests.

Tested Algorithm	Memory Usage (KB)	Time Elapsed (s)	CPU Usage (%)
A*	164.63	0.007539	41.08
BFS	104.69	0.005499	36.41
DFS	61.28	0.004622	43.30
Dijkstra	102.85	0.007037	44.06
Flood Fill	88.18	0.005994	33.42
Recursive Backtracker	75.59	0.005400	39.18
Random Mouse	23063.39	3.292647	89.90

Based on the obtained data, the memory usage, solution time, and CPU usage of the algorithms were compared. The results (Figure 10, Figure 11, and Figure 12) are presented below. The Random Mouse algorithm was excluded from comparison graphs because, in the tests, it performed significantly worse than others, with memory usage (KB), time elapsed (s), and CPU usage (%) recorded at 23063.39, 3.292647, and 89.90, respectively. These results were far below the performance of the other algorithms tested.

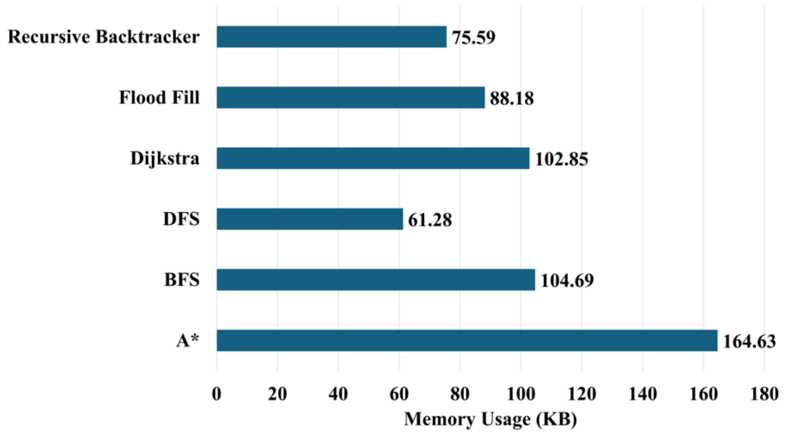

Figure 10. Memory usage of algorithms.

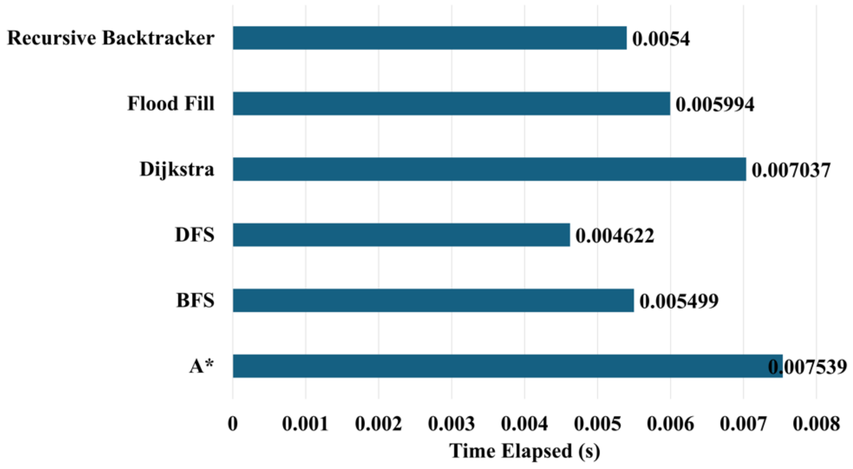

Figure 11. Solution time of algorithms.

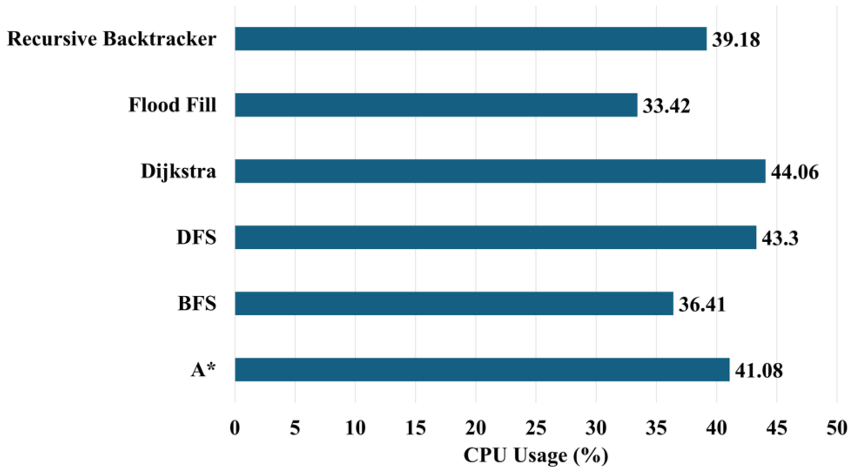

Figure 12. Cpu usage of algorithms.

5. Conclusion

The comparative analysis of maze-solving algorithms has revealed the unique strengths and weaknesses of each approach. The A* algorithm effectively finds the shortest path, utilizing a heuristic function, but shows higher memory usage and longer solution times compared to some alternatives. Its CPU usage remains moderate, making it a balanced option for certain applications. The BFS algorithm, while not the fastest, exhibits low CPU usage and performs well in terms of solution time, ranking third overall. DFS, on the other hand, stands out for its minimal memory consumption and the fastest solution time, though it incurs higher CPU usage. Dijkstra's algorithm shares similarities with A*, offering comparable performance in terms of memory and solution time but showing higher CPU consumption. Despite its lack of heuristic functions, it remains a viable option, especially in complex scenarios. The Flood Fill algorithm, known for filling all empty spaces, demonstrates low CPU usage, moderate solution time, and slightly higher memory usage, making it suitable for scenarios where CPU efficiency is critical. The Recursive Backtracker algorithm provides a good balance between speed and memory usage, making it the second most efficient in both aspects. Lastly, the Random Mouse algorithm, which relies on random movements, significantly underperforms in all key metrics: memory usage, CPU consumption, and solution time, proving to be the least efficient approach overall. Its inefficiency further emphasizes the effectiveness of the more systematic algorithms evaluated in this study. This comparative evaluation offers valuable insights for selecting appropriate algorithms based on performance requirements, paving the way for optimized autonomous navigation and pathfinding in real-world applications..

Future work could explore optimizations for these algorithms in dynamic and real-time environments, where the maze or path changes over time, requiring algorithms to adapt on the fly. Additionally, hybrid approaches that combine the strengths of multiple algorithms, such as the precision of A* with the speed of DFS, could be developed for more specialized applications. Investigating hardware acceleration and parallelization techniques, particularly for algorithms like BFS and A*, may also lead to improvements in performance for large-scale or more complex pathfinding tasks. These directions will contribute to further advancements in autonomous navigation systems and other real-world applications of pathfinding algorithms.

References

[1] Lumelsky VJ. A comparative study on the path length performance of maze-searching and robot motion planning algorithms. IEEE Trans Robot Autom 1991; 7(1): 57-66.

[2] Alamri S, Alshehri S, Alshehri W, Alamri H, Alaklabi A, Alhmiedat T. Autonomous maze solving robotics: Algorithms and systems. Int J Mech Eng Robot Res 2021; 10(12): 668-675.

[3] Kaur NKS. A review of various maze solving algorithms based on graph theory. Int J Sci Res Dev 2019; 6(12): 431-434.

[4] Covaci R, Harja G, Nascu I. Autonomous Maze Solving Robot. In: IEEE Int Conf Autom Qual Test Robot; 21 May 2020. pp. 1-4.

[5] Husain Z, Al Zaabi A, Hildmann H, Saffre F, Ruta D, Isakovic AF. Search and rescue in a maze-like environment with ant and dijkstra algorithms. Drones 2022; 6(10): 273.

[6] Budiman JS, Laurensia M, Arthaya BM. Arthaya. Maze mapping based modified depth first search algorithm simulator for agricultural environment. In: IEEE Int Conf Mechatron Robot Syst Eng; 17 November 2021. pp. 1-6.

[7] Ali SI, Duraisamy Y, Rasheed BH, Abas SM, Masood TD. Navigating Network Traffic: An Exploration of Maze Algorithm Applications in Machine Learning. Int Innov Res J Eng Technol; 2 April 2024; 9(3).

[8] Olivier T. A grid-based maze approach to humanitarian logistics. PhD, North-West University, South Africa, 2021.

[9] Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern; July 1968; 4(2): 100-107.

[10] Duchoň F, Babinec A, Kajan M, Beňo P, Florek M, Fico T, Jurišica L. Path planning with modified a star algorithm for a mobile robot. Procedia Eng 2014. 96: 59-69.

[11] Ju C, Luo Q, Yan X. Path planning using an improved a-star algorithm. In: IEEE 11th Int Conf Progn Syst Health Manage; 23 October 2020. pp. 23-26.

[12] Moore EF. The shortest path through a maze. In; Proc Int Symp Theory Switch. Harvard University Press. 1959.

[13] Lee CY. An algorithm for path connections and its applications. IRE Trans Electron Comput 1961; (3): 346-365.

[14] Permana SH, Bintoro KY, Arifitama B, Syahputra A. Comparative analysis of pathfinding algorithms a*, dijkstra, and bfs on maze runner game. Int J Inf Syst Technol 2018; 1(2): 1.

[15] Hopcroft J, Tarjan R. Algorithm 447: efficient algorithms for graph manipulation. Commun ACM 1973; 16(6): 372-378.

[16] Awerbuch B. A new distributed depth-first-search algorithm. Inf Process Lett 1985; 20(3): 147-150.

[17] Dijkstra EW. A note on two problems in connexion with graphs. Edsger Wybe Dijkstra: his life, work, and legacy. 2022; pp. 287-290.

[18] Fredman ML, Tarjan RE. Fibonacci heaps and their uses in improved network optimization algorithms. J ACM 1987; 34(3): 596-615.
[19] Johnson DB. Efficient algorithms for shortest paths in sparse networks. J ACM 1977; 24(1): 1-13.
[20] Burtsev S, Kuzmin YP. An efficient flood-filling algorithm. Comput Graph 1993; 17(5): 549-561.
[21] Kumar B, Tiwari UK, Kumar S, Tomer V, Kalra J. Comparison and performance evaluation of boundary fill and flood fill algorithm. Int J Innov Technol Explor Eng 2020; 8(12): 9-13.
[22] Abu-Khzam FN, Langston MA, Mouawad AE, Nolan CP. A hybrid graph representation for recursive backtracking algorithms. In: Int Workshop Front Algorithmics. Springer 2010.
[23] Lim TH, Ng PL. Evaluating recursive backtracking depth-first search algorithm in unknown search space for self-learning path finding robot. In: Springer Artif Intell Commun Netw: Second EAI International Conference; 19-20 December 2020. pp. 531-543.
[24] Cherroun L, Boumehraz M. Path following behavior for an autonomous mobile robot using neuro-fuzzy controller. Int J Syst Assur Eng Manage 2014; (5): 352-360.
[25] Yadav S, Verma KK, Mahanta S. The Maze problem solved by Micro mouse. I Int J Eng Adv Technol 2012; 2249: 8958.
[26] Dehghani M, Hubálovský Š, Trojovský P. Cat and mouse based optimizer: A new nature-inspired optimization algorithm. Sensors 2021; 21(15): 5214.
[27] Wang H, Lou S, Jing J, Wang Y, Liu W, Liu T. The EBS-A* algorithm: An improved A* algorithm for path planning. PloS ONE 2022; 17(2): e0263841.
[28] Buluç, A, Madduri K. Parallel breadth-first search on distributed memory systems. In: Proc Int Conf High Perform Comput Netw Storage Anal 2011.
[29] Sangamesvarappa V. Parallelizing Depth-First Search for Pathway Finding: A Comprehensive Investigation. Rev Intell Artif 2023; 37(4).
[30] Aviram N, Shavitt Y. Optimizing Dijkstra for real-world performance. arXiv preprint arXiv:1505.05033, 2015.
[31] Abu-Khzam FN, Daudjee K, Mouawad AE, Nishimura N. An easy-to-use scalable framework for parallel recursive backtracking. arXiv preprint arXiv:1312.7626, 2013.
[32] Xie S, Wu P, Liu H, Yan P, Li X, Luo J, Li Q. A novel method of unmanned surface vehicle autonomous cruise. Ind Robot Int J 2016; 43(1): 121-130.
[33] Mahmud MS, Sarker U, Islam MM, Sarwar H. A greedy approach in path selection for DFS based maze-map discovery algorithm for an autonomous robot. In: IEEE 15th Int Conf Comput Inf Technol 2012.
[34] Foltin M. Automated maze generation and human interaction. Brno: Masaryk Univ Fac Inform 2011.
[35] Pame YG, Kottawar VG, Mahajan YV. A Novel Approach to Maze Solving Algorithm. IEEE Int Conf Emerg Smart Comput Inform 2023.
[36] Iloh PC. A Comprehensive and comparative study of DFS, BFS, and A* search algorithms in a solving the maze transversal problem. Int J Soc Sci Sci Stud 2022; 2(2): 482-490.
[37] Sturtevant NR. Benchmarks for grid-based pathfinding. IEEE Trans Comput Intell AI Games 2012; 4(2): 144-148.
[38] Li K. Maze solving robot based on graph algorithm. in AIP Conf Proc. AIP Publishing 2024.
[39] Wang H, Yu Y, Yuan Q. Application of Dijkstra algorithm in robot path-planning. IEEE Second Int Conf Mechan Autom Control Eng 2011.
[40] Kalisiak M, van de Panne M. RRT-blossom: RRT with a local flood-fill behavior. In: Proc IEEE Int Conf Robot Autom 2006.
[41] Kondrak G, Van Beek P. A theoretical evaluation of selected backtracking algorithms. Artif Intell 1997; 89(1-2): 365-387.
[42] Niemczyk, R, Zawiślak S. Review of maze solving algorithms for 2d maze and their visualisation. In: Springer Int Conf Students PhD Young Sci Eng XXI Century 2018.