**A Comparative Study of Various Pathfinding Algorithms**

**Pranav Balaji, Rishi Dinesh, Sruthi Srinivasan**

**Abstract**

Pathfinding algorithms find their application in several domains such as google maps, satellite navigation systems and outing packets over the internet. Pathfinding algorithms require a search technique to get the fastest route, efficient, and shortest time. This paper presents a comparative analysis of some of the most popular path finding algorithms such as DFS, BFS, A\* and Dijkstra. These algorithms were implemented and visualized in python to find the shortest path between the starting and ending point of a maze consisting of obstacles. Two maze generation algorithms- Recursive division and DFS maze were used to generate the maze and these two algorithms have also been highlighted in this paper. This paper describes the working of the path finding algorithms and subsequently attempts to compare the efficiency of the four algorithms based on metrics such as time taken, number of cells considered, cost of path and reachability. The comparative analysis showed that the A\* algorithm was the most reliable of all four algorithms.

**Introduction**

Pathfinding algorithms are usually an attempt to solve the shortest path problem in graph theory. They try to find the best path given a starting point and ending point based on some predefined criteria. Path finding algorithms are important because they are used in applications like google maps, satellite navigation systems, routing packets over the internet. The usage of pathfinding algorithms isn’t just limited to navigation systems. The overarching idea can be applied to other applications as well.

Path finding algorithms are extensively used in gaming applications especially the A\* algorithm. Games like role-playing games and real-time strategy games often have characters sent on missions from their current location to a predetermined or player determined destination. The most common issue of pathfinding in a video game is how to avoid obstacles cleverly and seek out the most efficient path over different terrain. Dijkstra’s algorithm finds its application in Link State Routing of packets over a network. Path finding algorithms are also used in transit planning, telephone traffic routing, maze navigation and robot path planning. This paper draws a comparison between four path finding algorithms in maze navigation.

The rest of the paper is structured as follows: The next section is literature review where several papers related to this domain are discussed and summarized. This is followed by the methodology section which elucidates the existing path finding algorithms along with an explanation for each existing algorithm. Next in the sequence is Existing Algorithms which talks about the implementation and visualization of the path finding algorithms in maze navigation. All the path finding algorithms are discussed in detail. Additionally, significant weightage has also been given to the two maze generation algorithms in this section. Following this are the results and associated discussion. This is where a detailed comparison of the algorithms has been presented and explained. The paper is finally wrapped with a conclusion section which provides a concise summary of the experimental observations. The last leg of the paper is the references section which contains all the necessary citations.

**Literature Review**

Path-finding is an important problem for many applications, including network traffic, robot planning, military simulations, and computer games. A literature survey was conducted in the various domains where pathfinding algorithms were applied. It was found that these algorithms were widely implemented in games, from simple Android-based applications to games developed with Augmented Reality. This was done to understand which pathfinding algorithm worked best in what kind of scenario.

[1] Conducted a comparative analysis of A\* and the basic Theta\* algorithm in Android-based pathfinding games. The goal of this paper was to compare the performance of these search techniques using metrics like completeness, time complexity, optimality etc. when run on square grid maps on Android-based pathfinding games. Various tools such as Android SDK, Eclipse IDE and IBM Worklight with HTML5 and JavaScript were used.

The authors used the simulation method to compare the two algorithms using Euclidean distance as the heuristic function. They applied the two algorithms to a simulation of a pathfinding game named “Finding the Bones Benchmark”. The input parameter was a square grid map (which was represented as a two-dimensional array) with a start point, target point and obstacles. The presence/absence of the obstacles was represented using 1’s and 0’s respectively. The output was a table with the values of the following metrics: completeness (if the pathfinding algorithm found a solution or not), running time (time taken to find the shortest path), path length (distance from start point to target point) and the number of nodes searched. The simulation was performed several times with different scenarios (different input parameters) and the results generated were verified and validated.

The performance comparison showed that both A\* and Basic Theta\* had relatively the same values for completeness and time complexity but A\* had the advantage of optimality in terms of fewer number of nodes searched whereas Basic Theta\* had the advantage of optimality in terms of shortest path generated.

[2] Conducted a comparative analysis of various pathfinding algorithms in Augmented Reality using Unity3D. The Intel RealSense Camera was used to construct an environment with virtual objects. These virtual objects were Non-Player Characters that used A\*, A\* smooth and Navigation Mesh algorithms as navigation systems.

The Navigation Mesh is a data structure that consists of a convex polygon covering an empty space. It describes the walkable surfaces of the gaming world, making it possible to find paths from one walkable location to another in the gaming world. This mesh is a pathfinding algorithm that is integrated with unity 3D and can be used in AR based environments. The A\* smooth algorithm on the other hand is built by modifying the A\* algorithm using the ray cast function found in Unity 3D [3].

The data used was the position of the coordinates of virtual objects consisting of x and z coordinates on the simulation of Unity 3D and x, y, and z coordinates in the real world. The Pathfinding simulation in Unity 3D had three scenarios: two scenarios with different, fixed start and end points and one scenario with the same moving start point and end point. The experiment was conducted five times to test the consistency of the path through which the object passes. The results showed that A\* smooth algorithm was superior compared to the A\* and Navigation Mesh algorithms in terms of runtime.

[4] Focused on the A\* algorithm for determination of shortest path in the domain of game development along with a comparative study of A\*, Dijkstra and Greedy Breadth First Search algorithms. The primary discussion of this research paper was the A\* algorithm which is the most popular algorithm for pathfinding in game AI. The paper also gives a brief on Dijkstra’s algorithm and Greedy Best-First-Search algorithm and compares their efficiency with the A\* algorithm. This paper also reviews a number of potential optimizations and future research areas for the A\* algorithm.

The comparative studies showed that the A\* algorithm was the most efficient one as it was found to have the good qualities of both the Dijkstra’s and Best-First-Search algorithms but it had the drawbacks of none. It was as fast as Best-FirstSearch algorithm and found a path as good as Dijkstra’s algorithm.

[5] Explored the application of pathfinding algorithms in a racing game. In a race car game, the NPC (Non-Playable Character) needs pathfinding to be able to walk on the track and avoid obstacles to reach the finish line. The Pathfinding method used by NPC in this game was Al Dynamic Pathfinding Algorithm to avoid the static and dynamic obstacles in track. The experimental results showed that NPCs using combined Dynamic Pathfinding Algorithm and Algorithm A \* get the results from NPCs that use only DPA Algorithm A \* while the obstacle position and trajectory shape have a big effect on DPA.

The Combined Algorithm A\* and Dynamic Pathfinding Algorithm were based on the advantages of each algorithm. The ability of A\* Algorithm in finding the shortest route and the ability of Dynamic Pathfinding Algorithm in avoiding dynamic obstacles were combined and executed together on an NPC in Car Racing Game. Before the race car game began, A\* Algorithm found the shortest route on the track from the start line to the finish line without being affected by the obstacles on the track. When the game began NPCl followed the shortest route of results from Algorithm A \* and run Dynamic Pathfinding Algorithm to detect obstacles on the path. When the NPC detected any obstacles, the NPC followed the Dynamic Pathfinding Algorithm to avoid such obstacles. When the obstacle was not detected then the NPC followed the route from Algorithm A \* again. The process was restarted until the NPC reached the finish line.

The conclusions of this research were:

1. Combined Dynamic Pathfinding Algorithm and A \* Algorithm can be implemented on Car Racing Game.

2. NPC using the combined Dynamic Pathfinding Algorithm and Algorithm A \* get better results from NPCs that only use the DPA Algorithm on empty track paths and race tracks that have obstacles.

3. The grid representation method used has an effect on the route result obtained by A\* Algorithm.

4. The obstacle position and trajectory shape have a big effect on DPA.

Apart from the game development domain, it was found that these algorithms were also used in some interesting fields such as in UAV path planning, word recognition, scale-free networks etc. In addition to this, it was also observed that various domains used grid-based path finding to find the shortest path.

[6] Focused on the application of pathfinding algorithms in multi UAVs path planning. The various algorithms that were studied included Dijkstra, Bellman Ford, Floyd-Warshall and A\*. Their performances were compared to pick the best algorithm that could help establish a path for communication between the UAVs and the human supervisors.

Given a connected graph with a source and destination node, the goal was to find the shortest path possible. The links between the nodes were weighted: they were given a value based on the physical distance between the nodes, the time for communication between the two nodes, the sensor capability of a node, the actual type of node, the availability of the nodes, the reliability of the node, the bandwidth constraints between the nodes, or a combination of some or all the above factors.

The four algorithms were studied using the MATLAB environment and their results were compared. It was observed that although BFS could find paths faster than Dijkstra’s, it was greedy, which meant that the paths produced weren’t always optimal. A\* was developed to combine the virtues of heuristic approaches like BFS and formal approaches like Dijkstra’s algorithm. It was concluded that the best-established algorithm for the general searching of optimal paths was A\*.

[7] Conducted a comparative study of two search strategies namely Dynamic Programming and Heuristic Search in word recognition. A successful approach to recognizing continuous speech is to model the recognition problem as one of finding an optimal path through a finite state network and this was done using two strategies- Dynamic Programming and Heuristic search. This comparison was based on theoretical considerations and experimental tests on a digit string task.

The recognition experiments showed that in a digit-string recognition task, dynamic programming beam search outperformed heuristic search with respect to both the computation time and memory requirements.

[8] proposed a two-stage solution algorithm to solve the most reliable path problem. In the first stage, the upper and lower bounds of on-time arrival probability were estimated. Dominance conditions and the monotonic property of the most reliable path problem were then established. In the second stage, the multi-criteria label-setting approach was utilized to efficiently determine the most reliable path.

To illustrate the applicability of the proposed MRP-TS algorithm, a comprehensive case study was carried out using the Wuhan network with real traffic data. Several typical distributions, including normal, lognormal and gamma, were tested. It is observed in the testing network that no single type of distribution can be the best-fit distribution for all link travel times. The lognormal, gamma and normal distributions, respectively, account for 50.29%, 30.25% and 19.46% of the best-fit results.

The Monte Carlo simulation technique was adopted to validate the assumption of path travel times following the normal distributions. Simulation results showed that the normal distribution could approximate the path travel time distributions by achieving 98.3% and 94.9% of approximate accuracy at 10th and 90th percentiles. Computational performance of the proposed MRP-TS algorithm was examined using five networks with different sizes. The results of computational experiments showed that the proposed MRP-TS algorithm could efficiently determine the most reliable path for all testing networks, and had potential applications in developing online route guidance systems. The proposed MRP-TS algorithm had a remarkable computational advantage over the existing multi-criteria label-correcting approach built on the FSD rule (Nie and Wu 2009).

[9] Conducted a survey on various grid-based path finding algorithms. Typically, a grid is superimposed over a region, and a graph search is used to find the optimal (minimal cost) path. The most common scenario is to use a grid of tiles and to search using A\*. This paper discussed the trade-offs for different grid representations and grid search algorithms. Grid representations discussed were 4-way tiles, 8-way tiles, and hexes. This paper introduced texes as an efficient representation of hexes. The search algorithms used are A\* and iterative deepening A\* (IDA\*).

The results of this paper apply not only to computer games, but to any type of pathfinding on a grid. This paper introduced results that increase our understanding of the algorithms and data representations used:

1. Hexagonal grids provide a better topological representation of the underlying problem space. Each hex is equidistant and uniquely shares one side with each adjacent hex.

2. The tex grid retains the advantages of a hexagonal grid but is easier to implement.

3. While the choice of grid does not affect the asymptotic performance of A\*, it does for IDA\*.

4. It is mathematically proven and empirically shown that the hexagonal grid is superior to the conventional tile grid for IDA\* searches. Furthermore, searching on a hex grid instead of a tile or octile grid will result in exponentially faster searches. It can also be proven that a hex grid is optimal in terms of search speed for all regular planar tessellations.

Further research was also done about the various studies that performed a comparative analysis of the various pathfinding and maze-generating algorithms. The goal of this research was to understand the working and the different kinds of advantages and disadvantages of these algorithms, and the various kind of metrics used to evaluate them.

[10] Conducted a comparative analysis of various search algorithms. It focused mainly on uninformed search strategies such as BFS, DFS, UCS and informed search strategies like A\* and Best First Search. It presented a comparison based on complexity, optimality and completeness. It also included the working of the search techniques (open list and closed list), their advantages, disadvantages and applications. A cross comparison of the above algorithms using the previously mentioned metrics clearly showed that heuristic search was more efficient that blind search.

[11] Conducted an examination of representational expression in maze generation algorithms. It provided an overview of three basic two-dimensional maze generation algorithms: Depth-First search, Prim’s and Recursive Definition. It compared the path lengths of each of the algorithms to pick the best base for new maze representations.

A comparison of the three algorithms showed that DFS produced mazes with long and winding passes whereas Prim’s algorithm generated mazes with shorter passes. Mazes generated by recursive division had long straight walls, making it easy to pass through them. A pairwise two-tailed T-test between the means showed that DFS provided the longest path length among the three algorithms, making it an ideal maze generator.

**Methodologies**

Various pathfinding algorithms were encountered during the literature survey. The most commonly recurring ones were the A\* algorithm, Dijkstra algorithm, BFS, DFS and Best First Search algorithms. Apart from this, other algorithms like the Bellman Ford algorithm, Floyd-Warshall algorithm and interesting variations of the A\* algorithm such as the Basic Theta\* algorithm and the IDA\* algorithm were also encountered.

Broadly, these algorithms were also grouped into and explained by various search strategies such as uninformed search, informed search, dynamic programming, heuristic search, etc. One paper [8] even proposed a unique two-stage algorithm called the MRP-TS algorithm.

A\* combines the concept of Dijkstra and Greedy Best First Search Algorithms [1]. It calculates the cost using a heuristic function and then prioritizes the nodes accordingly. A\* finds grid paths (paths constrained to grid edges) quickly, but grid paths were often not the true shortest path. [12] Basic Theta\* is a variant of the A\* algorithm that aims to find the shortest path without constraining itself the grid edges (Any-angle path planning) [1].

[2] Performed a comparative study of A\*, Dijkstra and Greedy Breadth First Search algorithms. Dijkstra’s algorithm unfailingly finds the shortest path from the starting point to the goal. However, when searching for a single target or goal, the use of this algorithm is not recommended because it consumes extra time and resources due to the additional number of nodes this algorithm inspects.

Greedy Best-First-Search runs much faster than Dijkstra’s algorithm because it uses the heuristic function to estimate the distance to the goal which helps it filter its paths to save time and resources. However, this algorithm, unlike Dijkstra’s, does not guarantee a shortest path. A\* is believed to be based on the above-mentioned algorithms because A\* is like Dijkstra’s algorithm in that it finds the shortest path without fail and it is like Greedy Best First Search in that it uses a heuristic function to estimate the distance to the goal.

[6] Studied various algorithms such as Dijkstra, Bellman Ford, Floyd-Warshall and A\* in the application of multi UAVs path planning. The distance vector routing algorithm, also known as the Bellman Ford algorithm [13], is used by routers in various networks to exchange routing information about the current status of the network and how to route packets to their destinations. Routers that use this algorithm have to maintain the lookup tables, which gives the distances and shortest path to sending packets to each node in the network. The information in the lookup table is always updated by exchanging information with the neighbouring nodes.

The Dijkstra’s algorithm in [6] created labels associated with vertices that represent the cost from the source vertex to that particular vertex. Within the graph, there exist two kinds of labels: temporary and permanent. The temporary labels (whose values can vary) are given to vertices that have not yet been reached. Permanent labels are given to vertices that have been reached and their cost to the source vertex is known. For any given vertex, there must be a permanent label or a temporary label, but not both. The algorithm begins at a specific vertex and extends outward within the graph, until all vertices have been reached.

The Floyd-Warshall Algorithm is an application of dynamic Programming. It is a technique that takes advantage of overlapping sub problems, optimal substructure, and trades space for time to improve the runtime complexity of algorithms. The A\* algorithm is like a combination of every other graph-searching algorithm. It’s like Dijkstra’s as it can find the shortest path, it’s like BFS as it can use a heuristic to guide itself, and in the simple case, it is as fast as BFS.

[7] Conducted a comparative study of two search strategies namely Dynamic Programming and Heuristic Search in word recognition.

Utilizing the technique of dynamic programming, a minimum accumulated distance D(i,j,k) is defined along any path to the grid point (i,j,k). This minimum accumulated distance is also referred to as matching score. The dynamic programming recursion was implemented by using only one column from the whole D(i,j,k) matrix, which was propagated along the time axis i. In order to achieve this storage reduction and to put the bookkeeping into evidence, a technique requiring three arrays- Backpointer array B(i,j,k), “from frame” array F(i) and “from reference” array R(i) was used.

Heuristic search can be considered to be a best-first strategy, which does not proceed in a time-synchronous fashion strictly from left to right but extends paths of different lengths. A path hypothesis is a sequence of nodes which are grid points (i,j,k). This search algorithm considered all nodes and the node with the best evaluation function was expanded by generating its successor nodes within the local transition rules in the graph. The evaluation function was computed for these successor nodes and the new nodes were added to the set of path hypotheses. The path hypotheses were then sorted using the scores of the evaluation function. All the operations were repeated until the node with the best evaluation function was the terminal node.

[8] Conducted a comprehensive case study using a real road network with stochastic travel times to illustrate the applicability of their proposed algorithm. This study investigated the problem of finding the most reliable path that maximized travel time reliability in such road networks.

A two-stage solution algorithm (namely, MRP-TS) was proposed to exactly solve the most reliable path problem. In the first stage, the upper and lower bounds of travel time reliability were estimated by calculating the least expected time path. Based on the estimated range of travel time reliability, the effective M-B dominance rule and monotonic property of objective function were established for solving the most reliable path problem. In the second stage, a multi-criteria label-setting approach was utilized to efficiently determine the most reliable path for different risk-taking scenarios. The optimality of the proposed MRP-TS algorithm was rigidly proved.

[9] Conducted a cross-comparative study of the A\* and IDA\* algorithm in various grid-based settings. A\* is the classic artificial intelligence optimization search algorithm. It uses a best-first search strategy, exploring the most likely candidate while eliminating provably inferior solutions. Its effectiveness is based on having a good heuristic estimator, H, on the remaining distance from the current state to a goal state. If the heuristic is admissible (does not overestimate), then an optimal answer is guaranteed. On a grid, A\* was shown to explore a search space that is proportional to D2, where D is the distance to a goal state [14].

Iterative-deepening A\* (IDA\*) is a memory-efficient version of A\*. It eliminates the open and closed lists by trading off space for time. For many applications, space is the limiting factor and thus IDA\* is preferred. However, since IDA\* iterates and repeatedly explores paths, this may result in a horribly inefficient search that is still asymptotically optimal (e.g., DNA sequence alignment). The speed of an IDA\* search depends on the number of nodes it needs to examine. Analysis showed that the size of the nodes to be searched was proportional to O(bD−H ) [15], where b is the average branching factor and H is the effect of the heuristic. Intuitively, this is because IDA\* checks every path of length D, and at each depth, each node branches into b more nodes.

[10] Conducted a comparative analysis of both uninformed search strategies such as BFS, DFS and UCS and informed search strategies like A\* and Best First Search. It also listed out the various applications of each of the algorithms.

Uninformed Search or blind search is a type of brute force technique wherein the search algorithm possesses no information about the number of steps required to reach the final state from the current state. It often does not guarantee the shortest path. Informed search on the other hand is a heuristic search that will estimate (or guess) how close the final state is from the given current state before picking the next node to traverse.

In BFS, nodes are traversed level by level in the search tree using a queue data structure. It is not employed when memory requirements are high. It is however used in social networking websites, GPS Navigation systems, etc. In contrast to this, DFS expansion starts from the source node and goes all the way to the deepest level of the tree. It is a recursive algorithm that uses a stack data structure. It is used to identify loops in a graph, and also identify components that are strongly connected in the graph. Uniform Cost Search (UCS) expands the node with the lowest cost path. It is implemented using a priority queue and is widely used in solving maze problems and path finding.

A\* combines UCS with pure heuristic search to provide an optimal solution [16]. Various heuristics such as Euclidean distance, Manhattan distance, Diagonal distance, etc. are used. It is used in traffic navigation system [17] and in games to find the shortest path. Best First Search, also known as Greedy Search, is simply a combination of BFS and DFS. It is used mainly in web crawlers and in some games.

In addition to pathfinding algorithms, the literature survey also covered a few maze generating algorithms. [11] provided an overview of three basic two-dimensional maze generation algorithms: Depth-First search, Prim’s and Recursive Definition.

Maze generation using DFS is fairly simple [18]. Each cell is initialized with walls on all sides. The DFS algorithm then picks a random starting point and goes from cell to cell by breaking the walls in between until all the cells have been visited.

Prim’s algorithm [19] on the other hand initializes all cells as walls, picks a random cell, detects its neighbors and adds one of them to the maze. This process is repeated until all the cells are in the maze. The Recursive Definition Algorithm [20] starts with an empty grid, randomly divides the grid into two parts using a wall and adds a random pass in the wall.

**Existing Algorithms**

The implementation consists of two major components- the design view and runtime view. The design view is the interface where the user can set the maze. The user can either choose to set their own maze or they can opt for automatic maze generation which can be carried out using two different algorithms- Recursive Division algorithm and Depth First Search maze algorithm. The interface consists of options to set the maze using either of the two automatic maze generation algorithms. This is accompanied by an option to store the generated maze for future use, which can then be accessed using the load button. The load button opens up the most recently saved maze. Subsequently, there is also an option to clear out the currently displayed maze. The user can exit the interface by pressing the exit button. Once the maze has been set, the user has to set a start and end point before clicking on the view algorithms button to navigate to the runtime view.

The runtime view is where the path computation using the four path finding algorithms can be visualized. The interface is split into four quadrants with each quadrant housing one algorithm. The four algorithms run simultaneously, thus enabling the users to have a real time visualization of the path computation and a comparison of all the four algorithms at once. The path traversed by each algorithm will be visibly highlighted. Once all the computations are completed, several comparison metrics for the analysis and comparison of the algorithms are then displayed on the screen. The metrics used for this purpose are- time taken, number of cells considered, cost of path and reachability. Additionally, there is also a description beneath the name of each algorithm indicating if the algorithm guarantees the shortest path or not.

The four algorithms used for the visualization and comparison are- Depth-First Search, Breadth-First Search, Dijkstra’s algorithm and the A\* Search algorithm.

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored. The process is similar to the BFS algorithm. In DFS, the edges that lead to an unvisited node are called discovery edges while the edges that lead to an already visited node are called block edges. The data structure which is being used in DFS is stack**.** The time complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges in the graph. This algorithm does not guarantee the shortest path.

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertex and checks its adjacent vertices again. To implement this algorithm the Queue data structure has been used. All the adjacent vertices are added into the queue, when all adjacent vertices are completed, one item is removed from the queue and we start traversing through that vertex again. The time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. This algorithm guarantees the shortest path.

Dijkstra’s Algorithm is a very widely known path-finding and graph-traversal algorithm, The algorithm has multiple variants including source-to-destination plotting and source-to-nodes plotting (the latter creating a shortest-path tree).It keeps track of visited and unvisited nodes, checks for the next unvisited node with the minimum distance of traversal and progressively builds on that till it reaches the target node (if it exists) or the queue becomes empty (unbounded).Internally, it uses a Minimum-Priority Queue (combined with Heap for optimization) for figuring out the minimum heuristic traversal node. This results in a time complexity of Θ((|V| + |E|)log(|V|)), where |V| is the number of nodes and |E| is the number of edges connecting the nodes. However, this algorithm can also be implemented in Θ(|V|2) using arrays. This algorithm guarantees the shortest path.

A\* is a computer [algorithm](https://brilliant.org/wiki/algorithm/) that is widely used in pathfinding and [graph](https://brilliant.org/wiki/graphs/) traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. It uses the sum of 'cost' (distance from source to the current node, defined by g(x)) as well as 'heuristic' (estimated distance from current node to destination, defined by h(x)). The time complexity of A\* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(bd), where b is the branching factor (the average number of successors per state). The data structure typically used in the A\* search algorithm is a priority queue. This algorithm guarantees the shortest path.

The two automatic maze generation algorithms used in this project are the Recursive Division Algorithm and Depth first Search Maze Algorithm.

The recursive division algorithm is one that was implemented as a wall adder. This algorithm is particularly fascinating because of its fractal nature as theoretically the process can be continued indefinitely at progressively finer and finer levels of detail.

The working is as follows:

Step 1: Begin with an empty field.

Step 2: Bisect the field with a wall, either horizontally or vertically. Add a single passage through the wall.

Step 3: Repeat step 2 with the areas on either side of the wall.

Step 4: Continue, recursively, until the maze reaches the desired resolution.

A valid maze is still obtained at every step. Each repetition of the algorithm simply increases the level of detail.

In the depth first search algorithm, a maze is a series of paths separated by walls, and to simplify the generation one can think of the maze as a 2-dimensional grid. The grid has a width and height, and each x/y position in the grid can be represented as a cell. When considered in this manner, the grid can be considered a graph G, in which each cell is a node connected to each of its four neighbours by a wall (the exception to this rule is for edge and corner cells which have 3 and 2 neighbours, respectively). The algorithm then finds, based upon a random seed, a spanning tree - or tree composed of all vertices but only some of the edges - of this graph G. The algorithm does so as follow:

Step 1: Randomly select a node (or cell) N.

Step 2: Push the node N onto a stack S.

Step 3: Mark the cell N as visited.

Step 4: Randomly select an adjacent cell A of node N that has not been visited.

Step 5: If all the neighbours of N have been visited, continue to pop items off the stack S until a node is encountered with at least one non-visited neighbour - assign this node to N and go to step 4. Otherwise, if no nodes exist, stop.

Step 6: Break the wall between N and A.

Step 7: Assign the value A to N.

Step 8: Go to step 2.

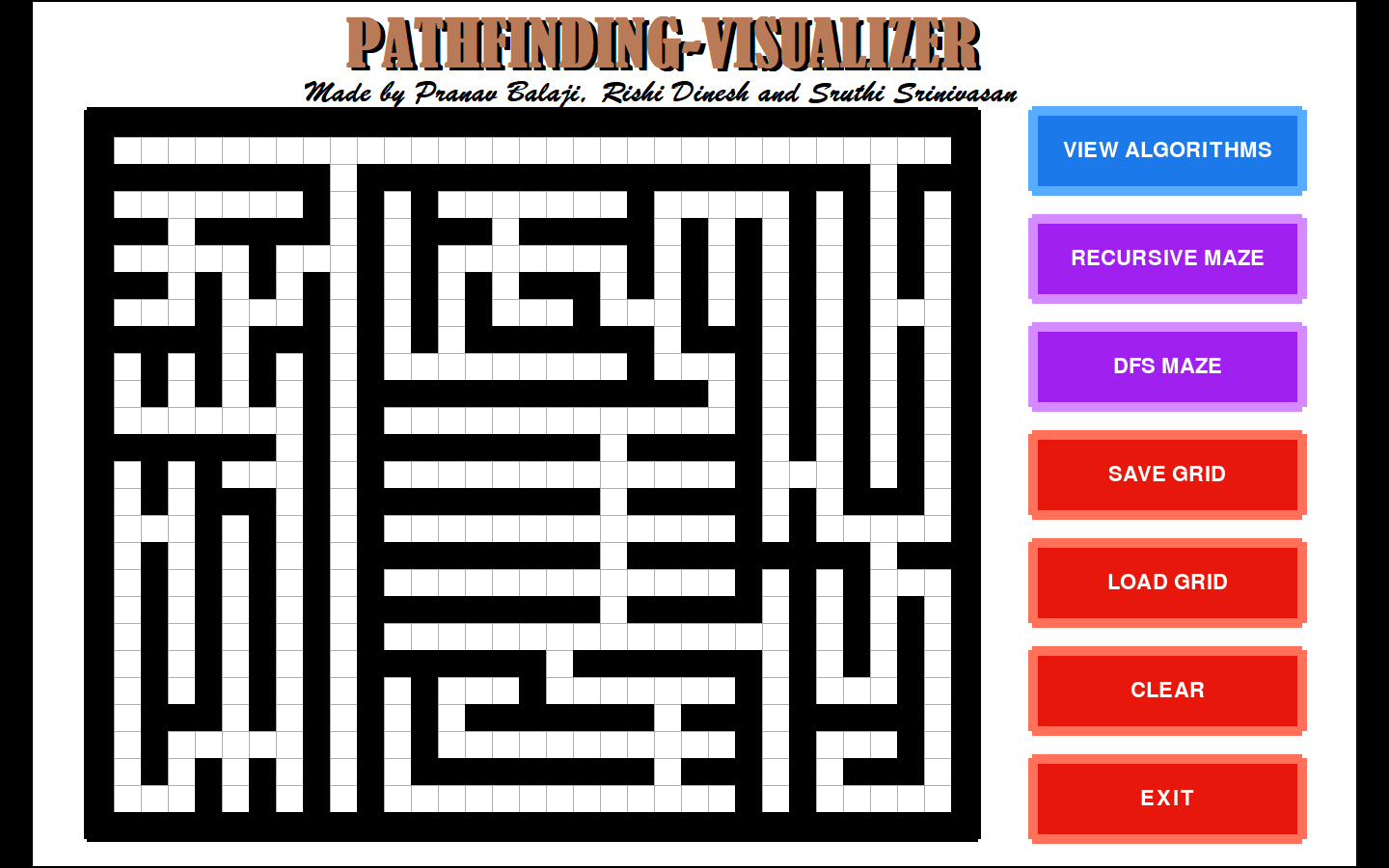
**Comparative Analysis**

An experiment with a series of trials was conducted to analyze the four different pathfinding algorithms. The experiment was designed in the following manner: Four mazes (or scenarios) were generated, two using the recursive division algorithm and two using the DFS algorithm. For each scenario, a series of 10 trials were conducted to collect data and build observation tables.

For every trial under one scenario, the parameters varied to get different results were the starting point and the ending point. As mentioned in the previous subsection, four different metrics were used to analyze these algorithms – time complexity (M1), number of nodes considered (M2), final path length (M3) and reachability (M4). The results of the experiment, i.e. the values of these four metrics for each algorithm under each trial have been tabulated below.

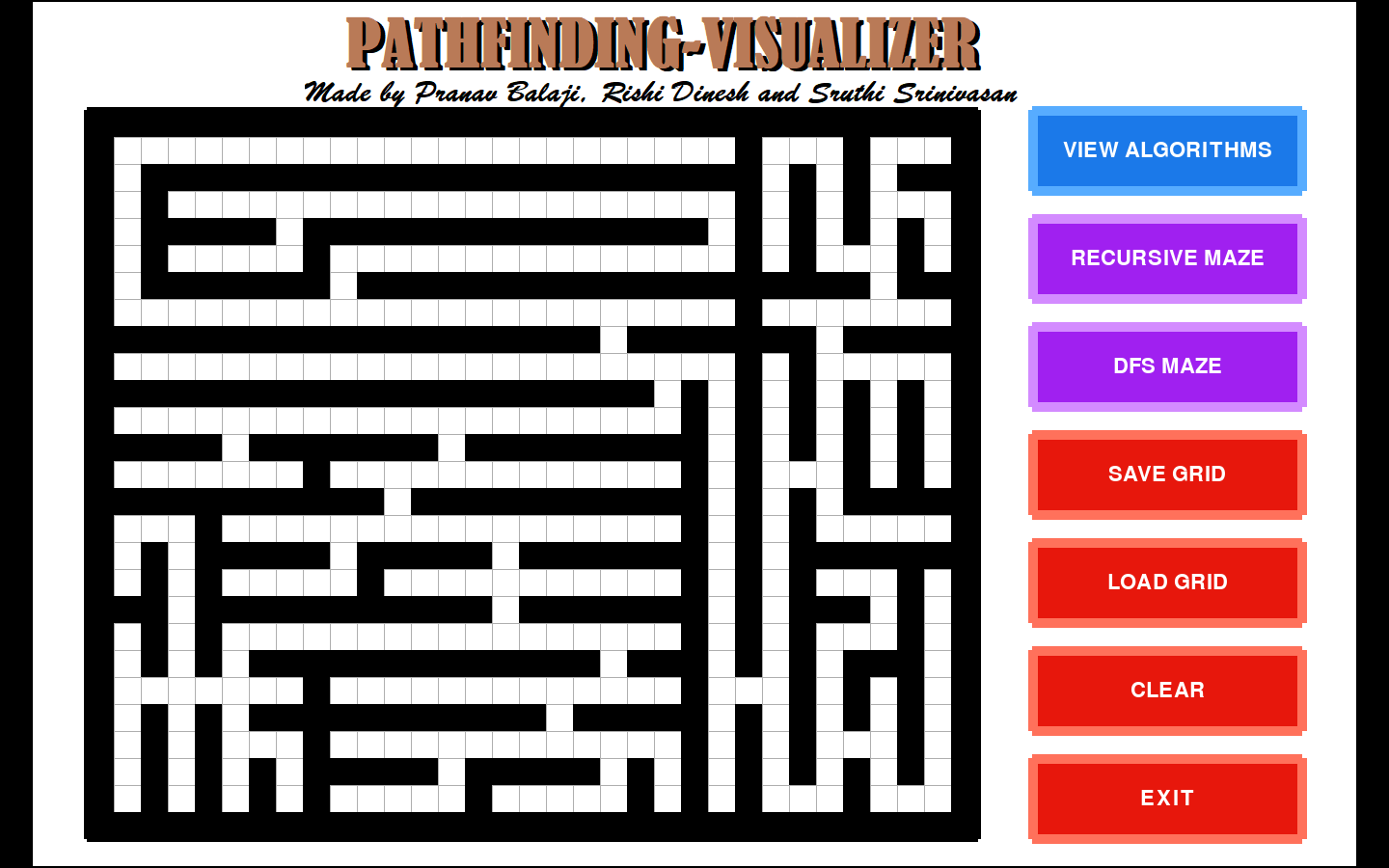
**RECURSIVE DIVISION MAZE**

***Scenario 1***



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | A\* Algorithm | | | | Dijkstra’s algorithm | | | | Breadth first search | | | | Depth first search | | | |
| # | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 |
| 1 | 4.07 | 163 | 57 | Yes | 2.86 | 216 | 57 | Yes | 2.87 | 218 | 57 | Yes | 5.34 | 407 | 57 | Yes |
| 2 | 5.40 | 134 | 83 | Yes | 5.68 | 267 | 83 | Yes | 5.67 | 268 | 83 | Yes | 6.23 | 301 | 83 | Yes |
| 3 | 6.31 | 155 | 85 | Yes | 4.56 | 195 | 85 | Yes | 4.55 | 196 | 85 | Yes | 3.45 | 136 | 85 | Yes |
| 4 | 3.15 | 79 | 55 | Yes | 4.38 | 220 | 55 | Yes | 4.41 | 223 | 55 | Yes | 5.85 | 308 | 55 | Yes |
| 5 | 13.12 | 322 | 125 | Yes | 9.34 | 411 | 125 | Yes | 9.32 | 411 | 125 | Yes | 6.26 | 256 | 125 | Yes |
| 6 | 3.88 | 101 | 65 | Yes | 3.55 | 169 | 65 | Yes | 3.53 | 169 | 65 | Yes | 2.85 | 122 | 65 | Yes |
| 7 | 8.76 | 226 | 89 | Yes | 5.90 | 280 | 89 | Yes | 5.88 | 280 | 89 | Yes | 5.10 | 235 | 89 | Yes |
| 8 | 10.69 | 266 | 113 | Yes | 8.21 | 386 | 113 | Yes | 8.19 | 386 | 113 | Yes | 5.54 | 237 | 113 | Yes |
| 9 | 14.06 | 407 | 0 | No | 6.60 | 407 | 0 | No | 6.58 | 407 | 0 | No | 6.58 | 407 | 0 | No |
| 10 | 2.83 | 95 | 0 | No | 1.49 | 95 | 0 | No | 1.47 | 95 | 0 | No | 1.46 | 95 | 0 | No |

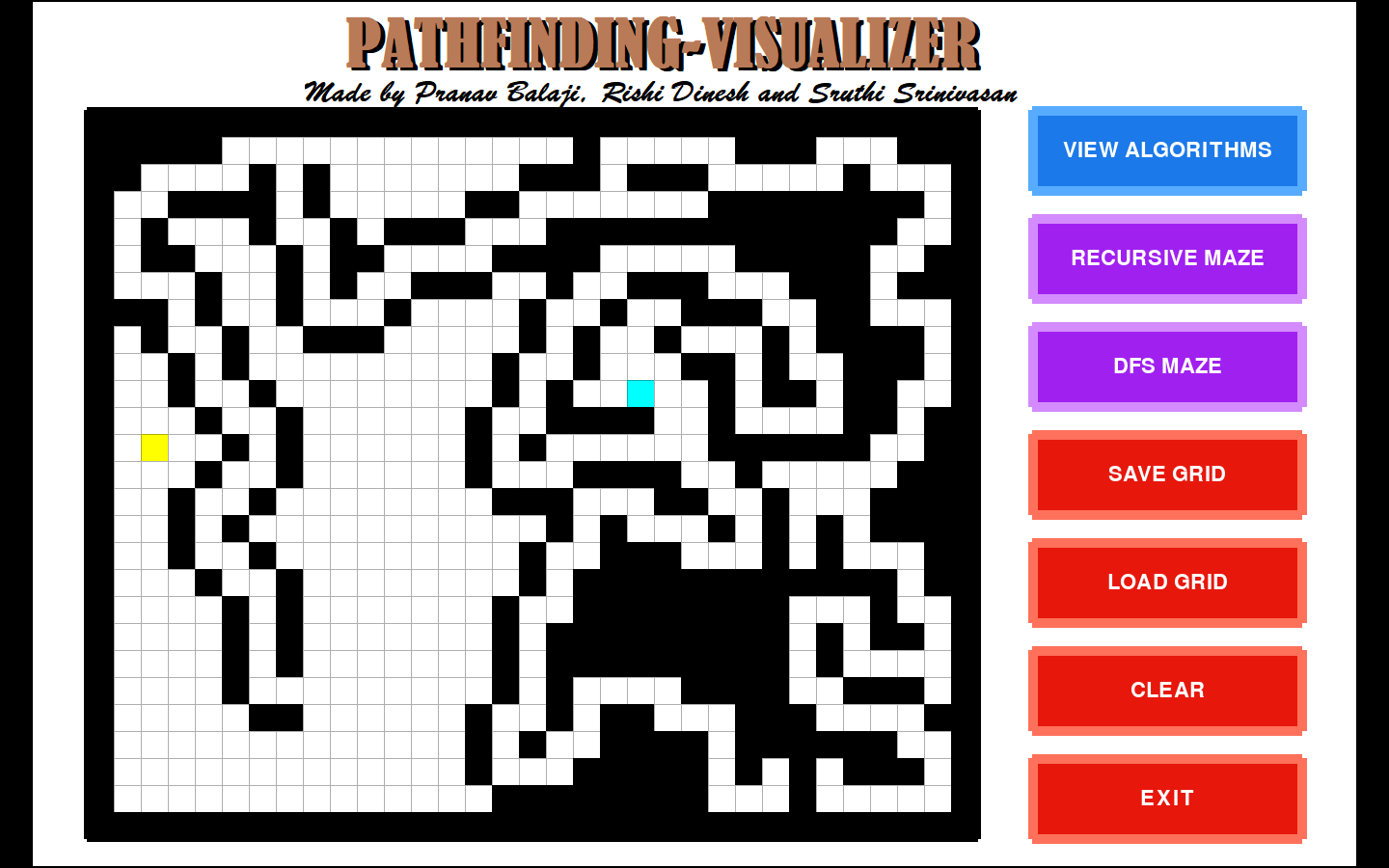
***Scenario 2***



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | A\* Algorithm | | | | Dijkstra’s Algorithm | | | | Breadth First Search | | | | Depth First Search | | | |
| # | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 |
| 1 | 7.91 | 193 | 87 | Yes | 7.99 | 371 | 87 | Yes | 7.90 | 368 | 87 | Yes | 5.53 | 241 | 87 | Yes |
| 2 | 6.18 | 160 | 75 | Yes | 6.81 | 335 | 75 | Yes | 6.82 | 337 | 75 | Yes | 4.36 | 198 | 75 | Yes |
| 3 | 3.75 | 95 | 57 | Yes | 5.33 | 276 | 57 | Yes | 5.32 | 277 | 57 | Yes | 4.91 | 252 | 57 | Yes |
| 4 | 7.75 | 207 | 77 | Yes | 7.72 | 395 | 77 | Yes | 7.73 | 397 | 77 | Yes | 4.60 | 223 | 77 | Yes |
| 5 | 4.07 | 109 | 71 | Yes | 4.95 | 251 | 71 | Yes | 4.89 | 248 | 71 | Yes | 3.36 | 160 | 71 | Yes |
| 6 | 2.70 | 73 | 53 | Yes | 3.92 | 213 | 53 | Yes | 3.88 | 211 | 53 | Yes | 5.42 | 306 | 53 | Yes |
| 7 | 6.72 | 134 | 65 | Yes | 6.75 | 256 | 65 | Yes | 6.80 | 261 | 65 | Yes | 4.30 | 190 | 65 | Yes |
| 8 | 14.30 | 407 | 0 | No | 7.14 | 407 | 0 | No | 7.12 | 407 | 0 | No | 7.12 | 407 | 0 | No |
| 9 | 1.33 | 31 | 26 | Yes | 2.42 | 132 | 26 | Yes | 2.38 | 130 | 26 | Yes | 3.28 | 189 | 26 | Yes |
| 10 | 1.03 | 35 | 0 | No | 0.55 | 35 | 0 | No | 0.53 | 35 | 0 | No | 0.53 | 35 | 0 | No |

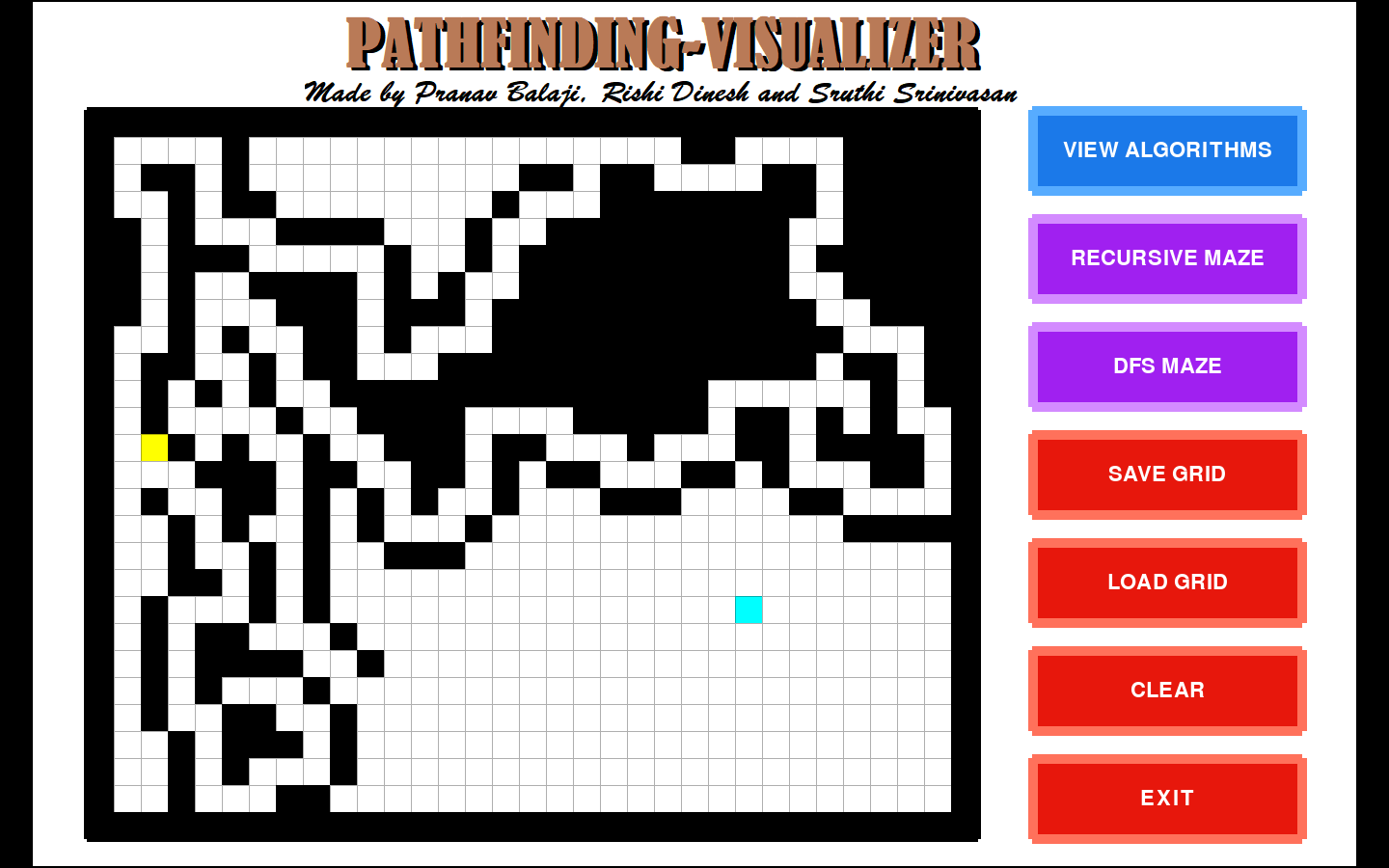
**DEPTH FIRST SEARCH MAZE**

***Scenario 3***



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | A\* Algorithm | | | | Dijkstra’s Algorithm | | | | Breadth First Search | | | | Depth First Search | | | |
| # | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 |
| 1 | 12.01 | 281 | 130 | Yes | 7.59 | 339 | 130 | Yes | 7.61 | 341 | 130 | Yes | 5.28 | 194 | 138 | Yes |
| 2 | 8.74 | 172 | 43 | Yes | 4.70 | 259 | 43 | Yes | 4.69 | 259 | 43 | Yes | 5.02 | 243 | 81 | Yes |
| 3 | 39.29 | 468 | 228 | Yes | 11.89 | 470 | 228 | Yes | 11.86 | 470 | 228 | Yes | 11.53 | 420 | 262 | Yes |
| 4 | 4.48 | 106 | 78 | Yes | 3.12 | 132 | 78 | Yes | 3.11 | 132 | 78 | Yes | 2.43 | 87 | 78 | Yes |
| 5 | 2.72 | 66 | 58 | Yes | 3.11 | 153 | 58 | Yes | 3.09 | 153 | 58 | Yes | 6.42 | 369 | 58 | Yes |
| 6 | 4.28 | 104 | 41 | Yes | 4.22 | 235 | 41 | Yes | 4.21 | 236 | 41 | Yes | 2.91 | 130 | 61 | Yes |
| 7 | 8.16 | 211 | 91 | Yes | 4.90 | 223 | 91 | Yes | 4.88 | 223 | 91 | Yes | 3.06 | 106 | 91 | Yes |
| 8 | 26.02 | 432 | 95 | Yes | 5.52 | 275 | 95 | Yes | 5.50 | 275 | 95 | Yes | 3.11 | 120 | 95 | Yes |
| 9 | 20.06 | 375 | 0 | No | 5.89 | 375 | 0 | No | 5.87 | 375 | 0 | No | 5.86 | 375 | 0 | No |
| 10 | 3.08 | 102 | 0 | No | 1.50 | 102 | 0 | No | 1.48 | 102 | 0 | No | 1.48 | 102 | 0 | No |

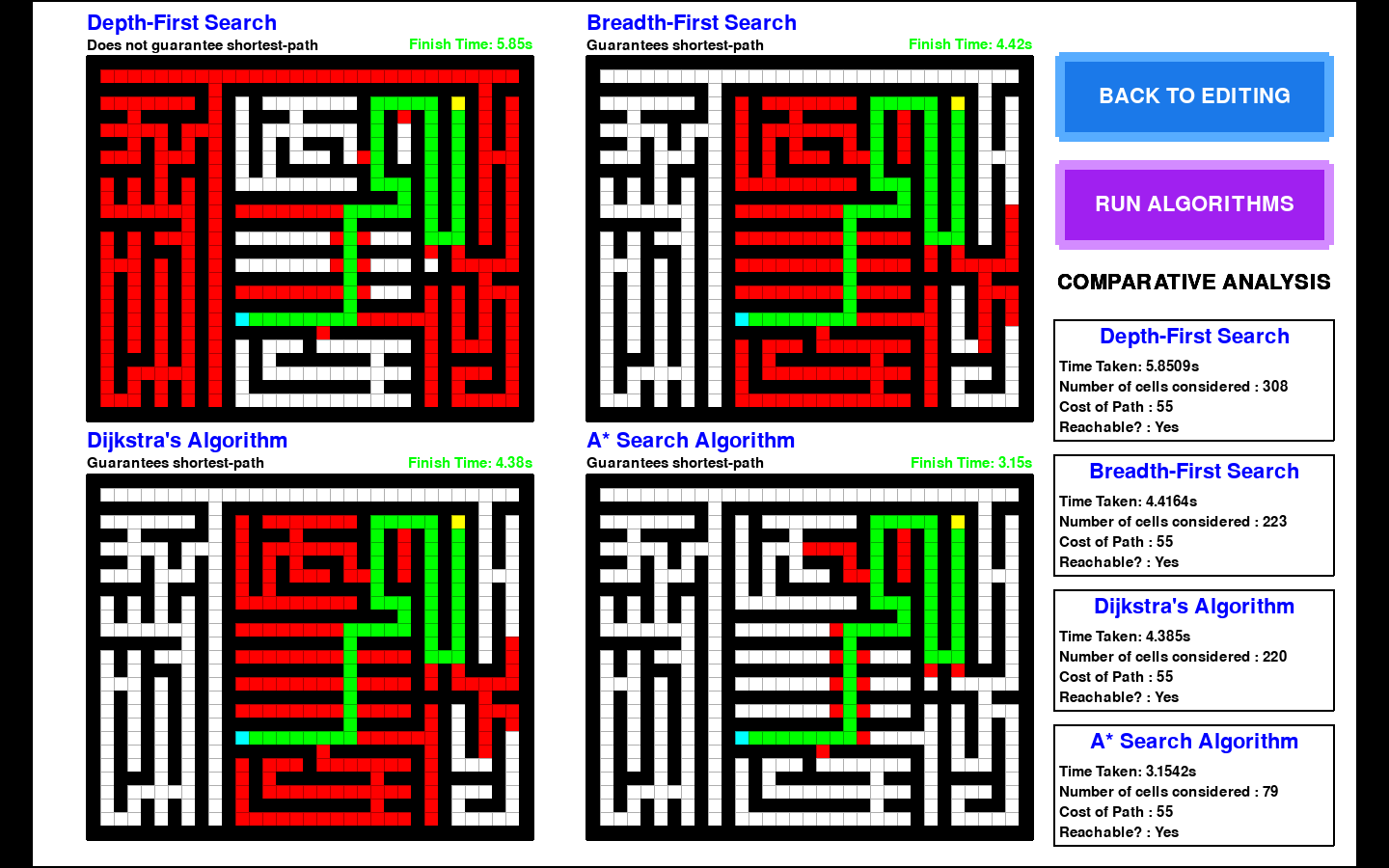
***Scenario 4***

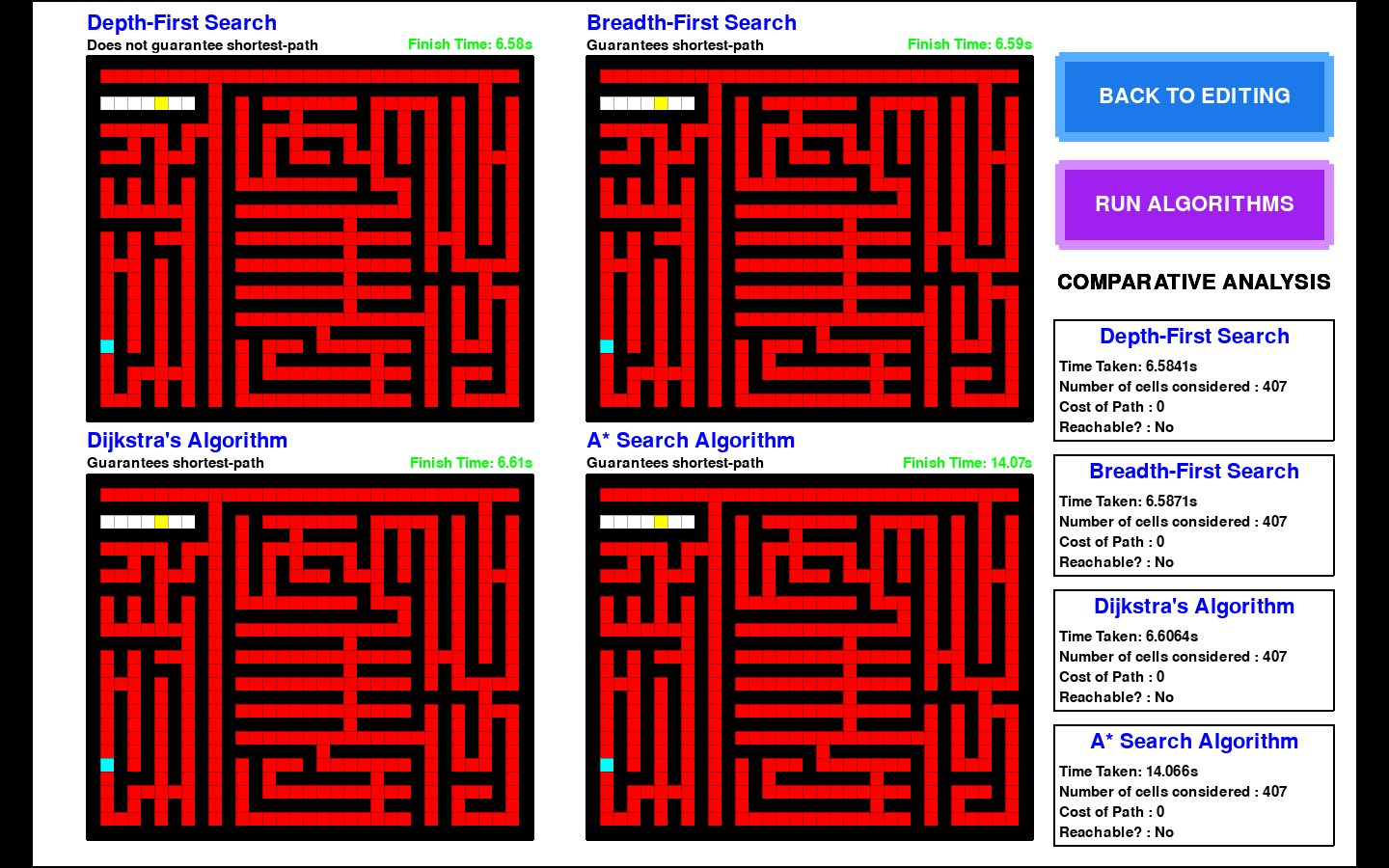


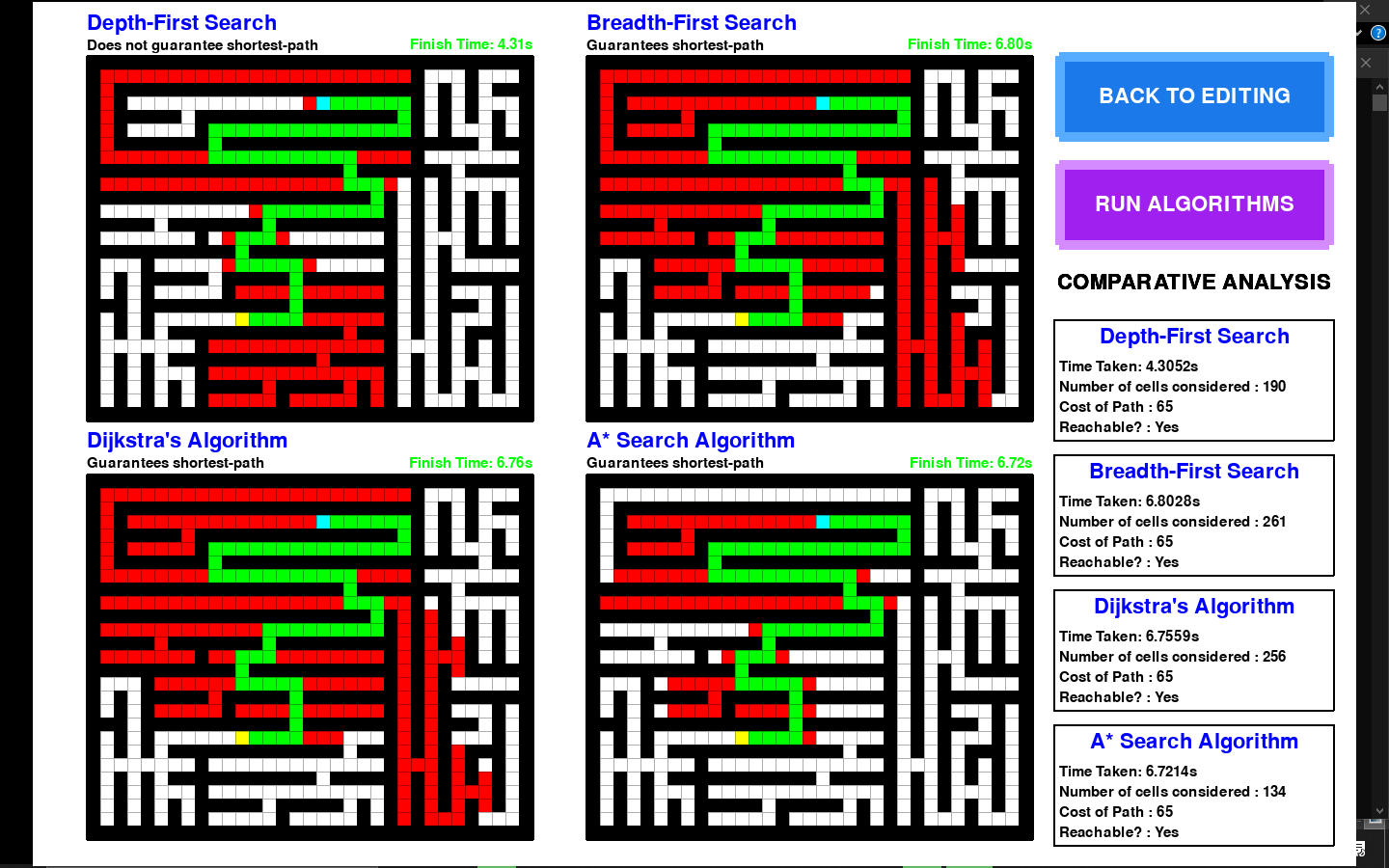
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Trial | A\* Algorithm | | | | Dijkstra’s Algorithm | | | | Breadth First Search | | | | Depth First Search | | | |
| # | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 | M1 | M2 | M3 | M4 |
| 1 | 25.94 | 338 | 119 | Yes | 7.66 | 376 | 119 | Yes | 7.64 | 376 | 119 | Yes | 7.96 | 361 | 151 | Yes |
| 2 | 3.66 | 89 | 56 | Yes | 2.65 | 141 | 56 | Yes | 2.64 | 141 | 56 | Yes | 7.66 | 465 | 62 | Yes |
| 3 | 3.34 | 80 | 69 | Yes | 4.21 | 201 | 69 | Yes | 4.19 | 201 | 69 | Yes | 2.72 | 110 | 71 | Yes |
| 4 | 5.25 | 128 | 77 | Yes | 3.65 | 163 | 77 | Yes | 3.63 | 163 | 77 | Yes | 2.91 | 118 | 77 | Yes |
| 5 | 6.56 | 113 | 66 | Yes | 3.57 | 183 | 66 | Yes | 3.49 | 180 | 66 | Yes | 8.22 | 472 | 66 | Yes |
| 6 | 1.98 | 78 | 30 | Yes | 4.76 | 287 | 30 | Yes | 4.74 | 287 | 30 | Yes | 7.17 | 405 | 60 | Yes |
| 7 | 23.98 | 479 | 0 | No | 7.84 | 479 | 0 | No | 7.80 | 479 | 0 | No | 7.79 | 479 | 0 | No |
| 8 | 7.42 | 225 | 42 | Yes | 5.42 | 301 | 42 | Yes | 5.39 | 301 | 42 | Yes | 4.27 | 204 | 74 | Yes |
| 9 | 4.04 | 82 | 43 | Yes | 2.42 | 129 | 43 | Yes | 2.40 | 129 | 43 | Yes | 6.33 | 393 | 43 | Yes |
| 10 | 1.43 | 32 | 28 | Yes | 2.18 | 113 | 28 | Yes | 2.3 | 123 | 28 | Yes | 0.99 | 32 | 28 | Yes |

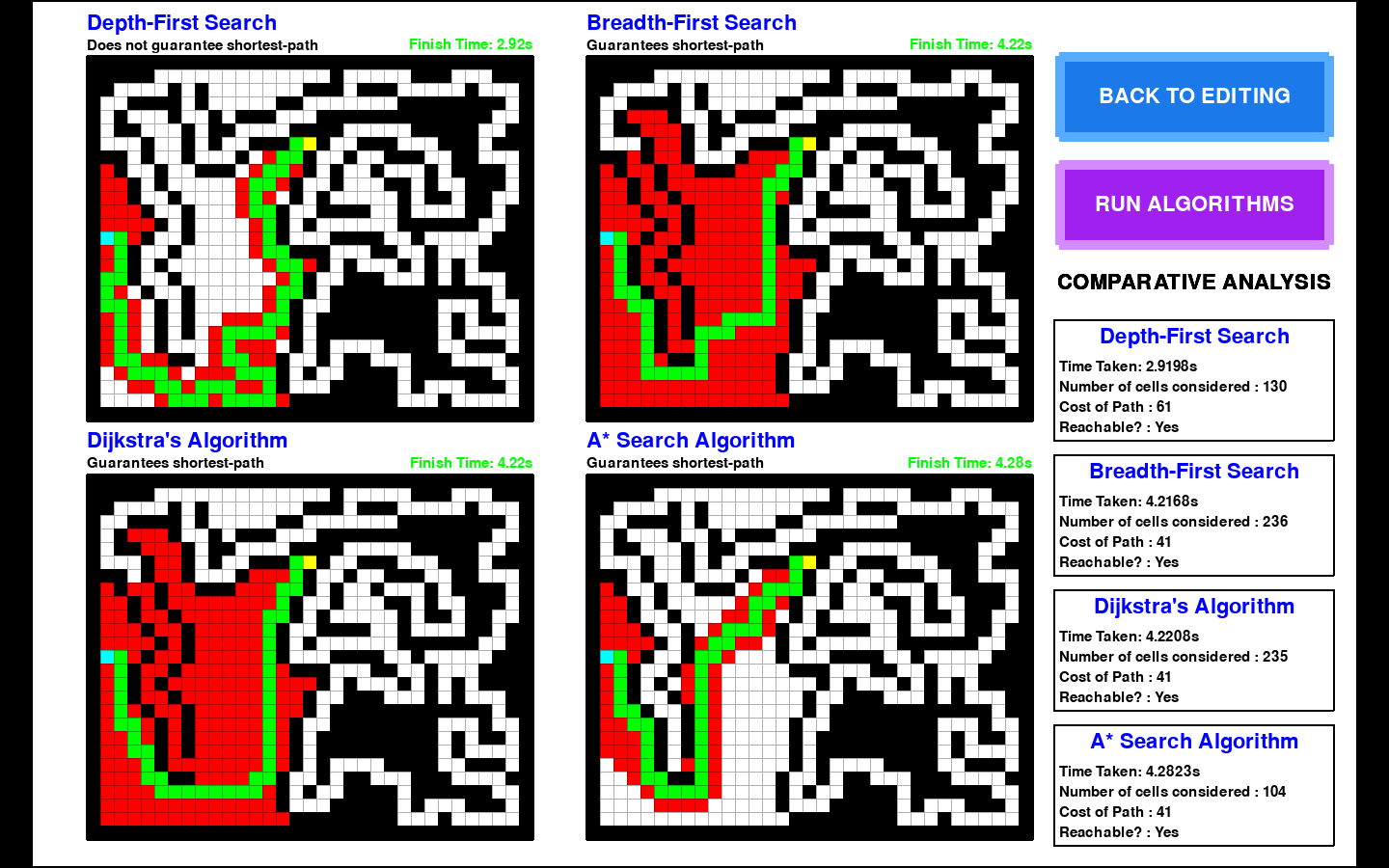
**Results**

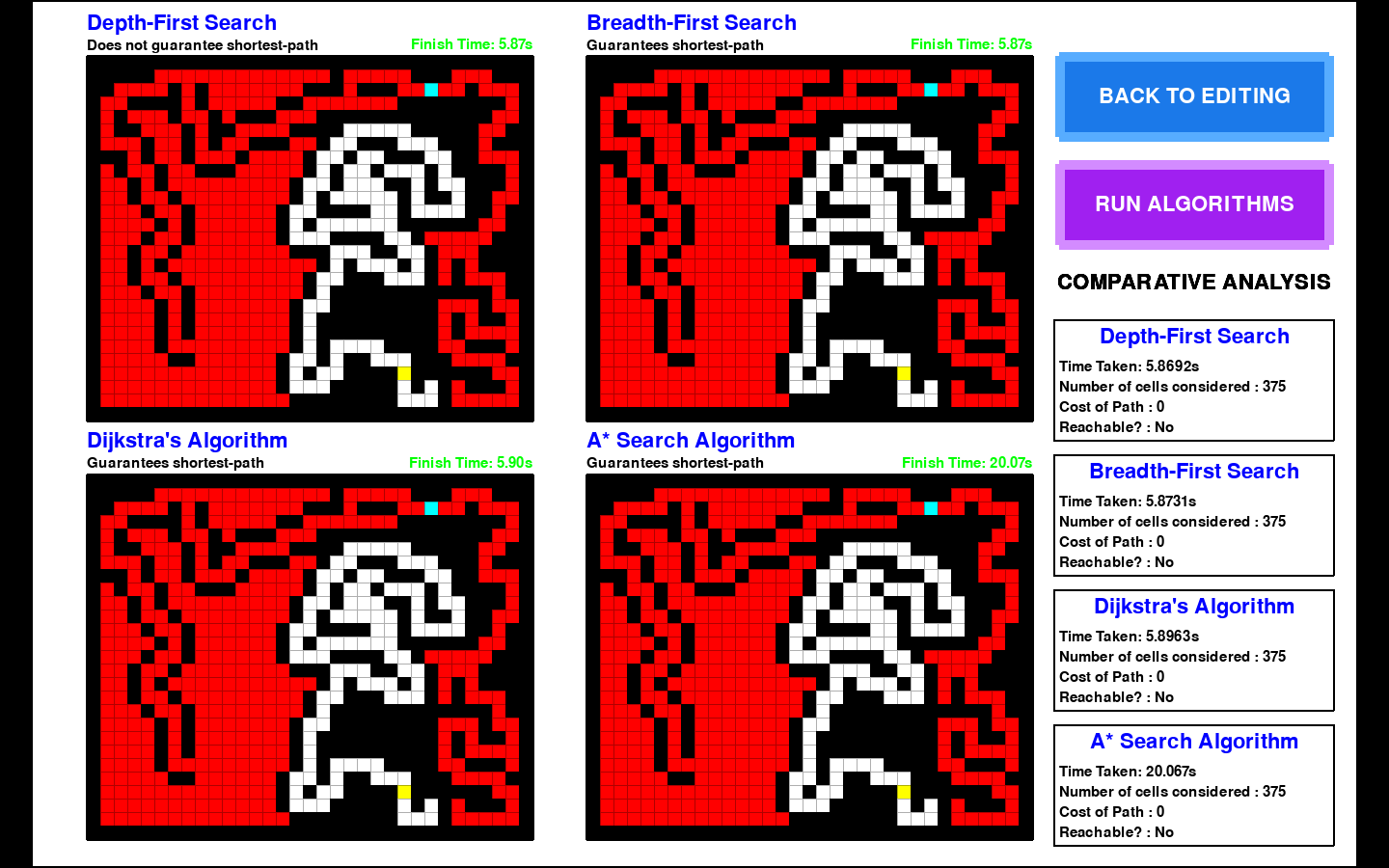
Given below are sample screenshots of the trials conducted as part of the experiment. Each screenshot shows how the four pathfinding algorithms determined the route from the source to the destination. The blue tile indicates the start point, the yellow tile indicates the end point and the red tiles indicate the nodes explored by the algorithm. The path determined is highlighted by the green tiles. For each trial, the values of the metrics for the four algorithms are also displayed. The screenshots also show how the algorithms react in case the destination is unreachable.

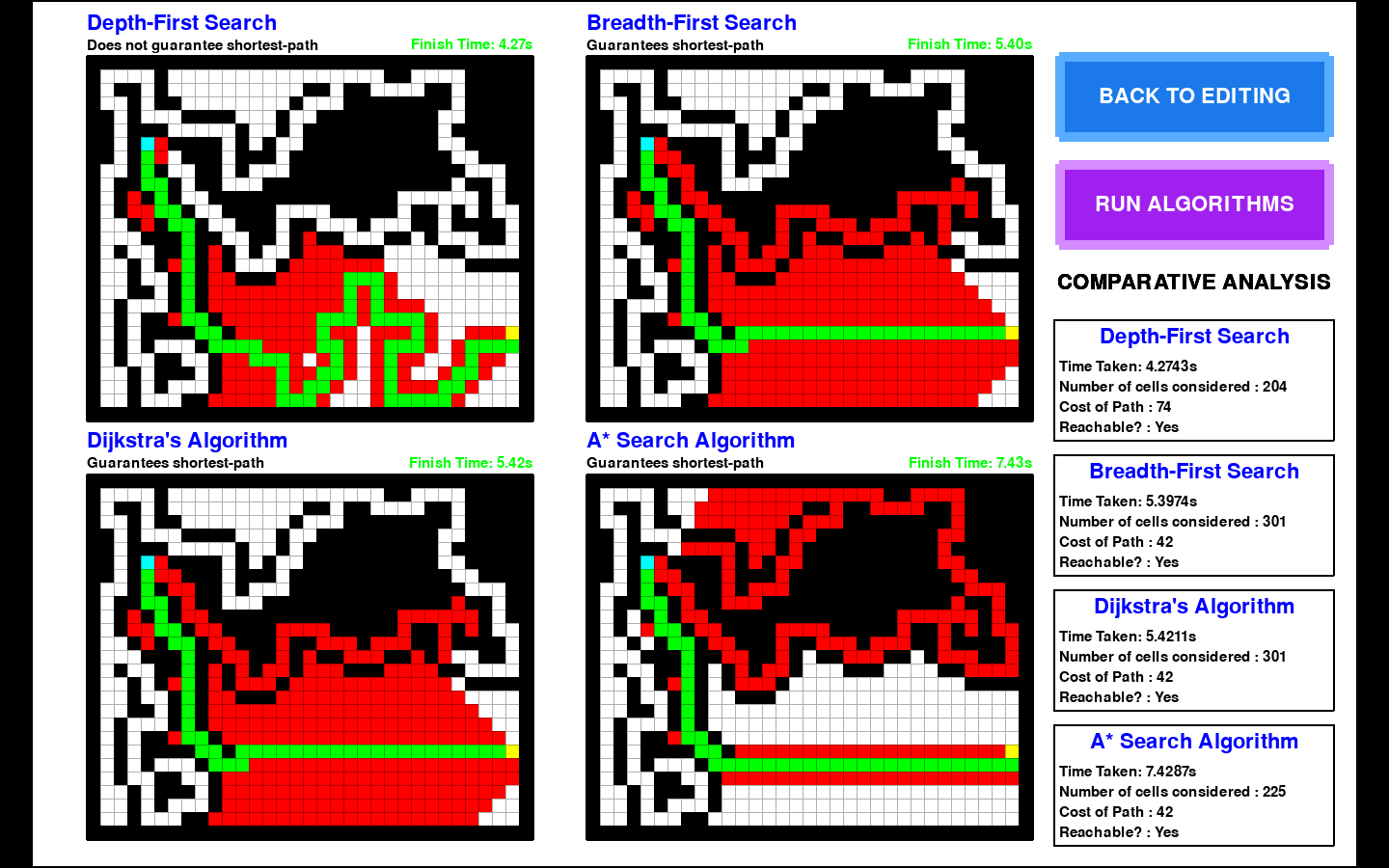












The averages of each of the metrics for each of the algorithms under all four scenarios have been computed and summarized in the table given below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Scenario | A\* Algorithm | | | Dijkstra’s Algorithm | | | Breadth First Search | | | Depth First Search | | |
| # | M1 | M2 | M3 | M1 | M2 | M3 | M1 | M2 | M3 | M1 | M2 | M3 |
| 1 | 7.22 | 195 | 84 | 5.25 | 265 | 84 | 5.24 | 265 | 84 | 4.86 | 250 | 84 |
| 2 | 5.57 | 144 | 64 | 5.35 | 267 | 64 | 5.33 | 267 | 64 | 4.34 | 220 | 64 |
| 3 | 12.88 | 232 | 95 | 5.24 | 256 | 95 | 5.23 | 257 | 95 | 4.71 | 215 | 108 |
| 4 | 8.36 | 164 | 59 | 4.43 | 237 | 59 | 4.42 | 238 | 59 | 5.60 | 304 | 70 |

From the above table, it can be inferred that the A\* algorithm always guarantees the shortest path (as indicated by M3). Another observation that can be made is that A\* explores minimum number of nodes while finding the shortest path (as indicated by M2). However, it can be seen that the time taken to find the shortest path is high compared to the other three algorithms, especially in mazes generated using DFS (as indicated by M1). This can be attributed to the fact that the speed of execution of A\* is heavily dependent on the accuracy and quality of the heuristic function used.

The Dijkstra algorithm, much like the A\* algorithm, guarantees the shortest path in any case (as indicated by M3). However, unlike A\*, it explores a lot of nodes before determining the shortest path (as indicated by M2). It has a fairly good time complexity (as indicated by M1) owing to the fact that it is a greedy algorithm.

It can be observed that the results of the Breadth First Search are very close to that of the Dijkstra algorithm. This is a result of the fact that all the nodes in the maze are unweighted. In an unweighted graph, both Dijkstra and BFS work with more or less the same principle. In fact, Breadth-first search can be viewed as a special-case of Dijkstra's algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue. Dijkstra is more efficient than BFS only in situations with weighted graphs.

The Depth First Search has the least amount of running time out of all the four algorithms. This verifies the fact that DFS is more efficient than BFS (in terms of speed and number of nodes explored). Another reason for its surprisingly low run time is due to the fact that the mazes generated restricted the possible movements to single-tiled paths, thus allowing DFS to traverse the maze quickly without making a lot of deviations. It is also known that DFS works better than BFS when the destination is far away from source (which was the case in many of the conducted trials). However, it does not always guarantee the shortest path as indicated by the table above.

**Conclusion**

The results of the comparative analysis showed how the different pathfinding algorithms worked in different maze settings. It was observed that A\* always produced the shortest path by exploring a minimum number of nodes, making it the most reliable of all four algorithms. It, however, did not have an optimal running time owing to a poor heuristic function. The Dijkstra algorithm, being the abridged version of A\*, displayed a similar performance. Due to the lack of a heuristic function to guide its search, the number of nodes explored were comparatively high. It, however, triumphed where A\* failed by maintaining its running time even in DFS generated maze settings.

The striking similarities between the performance results of the Dijkstra algorithm and the Breadth First Search proved the fact they are indeed based on the same concepts when tested in unweighted maze settings. The Depth First Search, despite using an uninformed search strategy, always managed to find a path between the source and destination in very little time. It, however, explored a lot of nodes to achieve this and did not always guarantee the shortest path.

**References**

[1] E. R. Firmansyah, S. U. Masruroh and F. Fahrianto, "Comparative Analysis of A\* and Basic Theta\* Algorithm in Android-Based Pathfinding Games," 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Jakarta, 2016, pp. 275-280, doi: 10.1109/ICT4M.2016.063.

[2] Nuryono, Aninditya & Ma'arif, Alfian & Iswanto,. (2020). Comparative Analysis of Path-finding Algorithm on Unrestricted Virtual Object Movable for Augmented Reality. International Journal of Scientific & Technology Research. 9. 160-165.

[3] J. Stamford, A. S. Khuman, J. Carter, and S. Ahmadi, “Pathfinding in partially explored games environments: The application of the A\* Algorithm with occupancy grids in Unity3D,” in 2014 14th UK Workshop on Computational Intelligence (UKCI), 2014, pp. 1–6.

[4] Mehta, Parth & Shah, Hetasha & Verma, Saurav. (2015). A Review on Algorithms for Pathfinding in Computer Games.

[5] Y. Sazaki, H. Satria and M. Syahroyni, "Comparison of A∗ and dynamic pathfinding algorithm with dynamic pathfinding algorithm for NPC on car racing game," 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA), Lombok, 2017, pp. 1-6, doi: 10.1109/TSSA.2017.8272918.

[6] Moses, Beulah & Jain, Lakhmi & Finn, R. & Drake, Samuel. (2008). Multiple UAVs path planning algorithms: A comparative study. Fuzzy Optimization and Decision Making. 7. 257-267. 10.1007/s10700-008-9035-0.

[7] H. Ney, "A Comparative Study of Two Search Strategies for Connected Word Recognition: Dynamic Programming and Heuristic Search" in *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 14, no. 05, pp. 586-595, 1992.

[8] Bi Yu Chen, Chaoyang Shi, Junlong Zhang, William H. K. Lam, Qingquan Li & Shujin Xiang (2017) Most reliable path-finding algorithm for maximizing on-time arrival probability, Transportmetrica B: Transport Dynamics, 5:3, 248-264, DOI: [10.1080/21680566.2016.1169953](https://doi.org/10.1080/21680566.2016.1169953)

[9] Yap P. (2002) Grid-Based Path-Finding. In: Cohen R., Spencer B. (eds) Advances in Artificial Intelligence. Canadian AI 2002. Lecture Notes in Computer Science, vol 2338. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-47922-8\_4

[10] Patel, Ronit & Pathak, Maharshi. (2018). Comparative Analysis of Search Algorithms.

[11] A. Kozlova, J. A. Brown and E. Reading, "Examination of representational expression in maze generation algorithms," 2015 IEEE Conference on Computational Intelligence and Games (CIG), Tainan, 2015, pp. 532-533, doi: 10.1109/CIG.2015.7317902.

[12] Daniel, Kenny & Nash, Alex & Koenig, Sven & Felner, Ariel. (2014). Theta\*: Any-Angle Path Planning on Grids. J. Artif. Intell. Res. (JAIR). 39. 10.1613/jair.2994.

[13] Cavendish, D. & Gerla, M.. (2000). Internet QoS Routing using the Bellman-Ford Algorithm. 10.1007/978-0-387-35388-3\_37.

[14] J. Pearl. Heuristics: Intelligent search strategies. In Addison-Wesley, 1984. 46

[15] R. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening A\*. Artificial Intelligence, 129(2):199–218, 2001. 46

[16] Deepika Garg, COMPARATIVE STUDY OF VARIOUS SEARCHING ALGORITHMS, National Conference on Innovative Trends in Computer Science Engineering (ITCSE-2015) held at BRCMCET, Bahal on 4 th April 2015.

[17] LIU Jingang, and LIU Yujun, Application of A\* algorithm in Traffic Navigational System, Information Engineering and Electronic Commerce (IEEC), 2010 2nd International Symposium on. IEEE, 2010.

[18] Ki Sang Lee, “Maze-generating algorithms”, in Physical Mathematics Conference, 2009.

[19] Jamis Buck, “Maze generation: Prim’s algorithm”, January 2011, http://weblog.jamisbuck.org/2011/1/10/maze-generation-prims-algorithm.

[20] Jamis Buck, “Maze generation: Recursive division”, January 2011, http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursivedivision-algorithm.