A Review of Various Maze Solving Algorithms

Kanishka Mavi
Department of Computer Science
Indraprastha College For Women, University of Delhi
New Delhi, India

Dr. Manju Bala
Department of Computer Science
Indraprastha College For Women, University of Delhi
New Delhi, India

Abstract **- Maze solving algorithms are a foundational component of Robotics, Navigation, Healthcare services, Artificial Intelligence and Machine Learning, extending beyond simply finding a path in a maze. This paper provides a comparative review of classical, heuristic-driven, sampling-based, and hybrid approaches used in maze-solving. Unlike prior researches that primarily explored various algorithms and lists their features, this review combines empirical findings across multiple studies in a systematic way and analyzes how algorithms are performing in mazes of varied sizes and types. This paper aims to present a systematic review of various Informed and Uninformed search algorithms, Grid-based algorithms, Sampling-based approaches and Heuristic-driven approaches. The review highlights various metrics such as optimality, computational complexity, memory usage, execution time and adaptability to static, dynamic, or continuous environments. Results indicate that no single algorithm performs optimally across all maze types; instead, choice of algorithm is highly context-dependent. The paper concludes with identified research gaps and proposes hybrid algorithm design for future advancements in maze pathfinding.**

Keywords— Pathfinding, Maze Solving, Robot Navigation, AI, Machine Learning, Hybrid algorithms, Pathfinding Algorithms, A*-RRT*, Genetic Algorithm.

INTRODUCTION

Solving mazes is widely enjoyed by players. The goal is to navigate through a maze to reach an exit while avoiding walls and finding the most efficient path within the grid. Though the concept may seem simple, the puzzles can become increasingly complex, requiring strategic planning to complete each maze. A maze can be designed in such a way that it will either have one path, multiple paths or no path. In case of no path existing, the algorithms will terminate after exploring the whole maze, taking a lot of time if the size of maze is large.

With advancing technology, maze solving algorithms are widely used in areas such as: Planning optimal routes for self-driving cars, delivery drones and other autonomous vehicles, guiding robots through complex environments such as warehouses, factories and even in search-and-rescue operations to reach a specific target while avoiding obstacles, determining movement of characters or objects in video games, training and testing AI agents in navigation, decision-making and problem-solving, optimizing traffic flow, network routing and exploring potential molecular pathways and interactions in the search of a new drug and GPS-based navigation.

Several algorithms have been devised to find the shortest path from source to destination by trying to avoid all the walls on the way spanning from uninformed search methods, heuristic-driven approaches, sampling-based techniques and evolutionary models. Each algorithm offers distinct strengths and limitations: uninformed methods such as Breadth-First Search and Depth-First Search provide strong theoretical guarantees but suffer scalability issues; informed strategies like A* and Jump Point Search uses heuristics to improve efficiency; hierarchical and visibility-based approaches optimize navigation in large static spaces; and sampling-based or hybrid algorithms, including RRT* and A*-RRT*, excel in continuous maze environment. Despite this diversity, existing studies often evaluate algorithms under limited conditions, making it difficult to generalize their performance across varied maze sizes and types.

The objective of this paper is to review classical and modern maze solving algorithms by synthesizing cross-study evidence used in maze-solving and to determine the conditions in which they are more optimal. Each of these algorithms has its own advantages and disadvantages, and the results of this research are expected to offer understanding on how different algorithms interact with specific environments .

LITERATURE REVIEW

This section gives consolidated and analytical overview of several maze solving algorithms implemented by various researchers between 2015-2025. All referenced figures (Fig. 1 and 2) and tables (Table I–V) are based on data reported in the reviewed literature and created for illustrative comparison.

A. Uninformed Search Algorithms

This section includes algorithms that systematically explores the mazes without using heuristic function.

Vijaya, Baghel and Mishra (2024) generated mazes of various sizes (ranging from 10×10 to 50×50 grids) using recursive division and highlighted Dijkstra algorithm being a reliable choice when heuristic information is unavailable. Elkari, Ourabah, Sekkat, Hsaine and Essaiouad (2024) used 7 mazes of varied complexity generated with Python library, Pmaze. They depicted that when start and end goal were far or near in the maze, BFS (Breadth First Search) revealed more optimal path cost than A* and DFS (Depth First Search), and DFS showed faster execution time due to its depth first nature.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

Erbil, Özkahraman and Bayrakçı (2025) utilized a 51×51 maze generated using a simple, efficient, and easy to implement DFS algorithm. The maze was represented as a binary matrix where each cell is either a path (0) or a wall (1), starting point of the maze is set at (0,0) and the ending point at (50,50). Algorithms like A*, BFS, DFS, Dijkstra, Flood Fill, Random Mouse, and Recursive Backtracker were evaluated on key performance metrics such as solution speed, memory usage, and CPU consumption. For accurate measurements, the average values of multiple runs were calculated for the algorithms using the tracemalloc, time. perf_counter, cProfile, and pstats libraries. Their results are summarized in Table 1.

[A] TABLE I. COMPARISON OF EFFICIENT SEARCH ALGOROTHMS BASED ON FINDINGS BY ERBIL, ÖZKAHRAMAN AND BAYRAKÇI (2025).

Metric	Fastest/Lowest	Slowest/Highest	Relative Performance
Solution Time (s)	DFS (0.004622)	A* (0.007539)	DFS was the fastest, with Recursive Backtracker (0.005499 s) and BFS (0.0054 s) following closely.
Memory Usage (KB)	DFS (61.28)	A* (164.63)	DFS was the most memory-efficient, followed by Recursive Backtracker (75.59 KB). A* used significantly more memory than the others.
CPU Usage (%)	Flood Fill (33.42)	Dijkstra's (44.06)	Flood Fill demonstrated the lowest CPU overhead, while Dijkstra's and DFS (43.30%) were the most CPU-intensive among this group.

[A]NOTE: The Random Mouse algorithm was excluded from this table and subsequent comparative analysis due to its status as an outlier, taking 23063.39 KB of memory, 89.90% CPU usage and a solution time of 3.292647 seconds, making it hundreds of times less efficient than the other algorithms.

Putra et al. (2025) represented campus environment as a maze-shaped graph and analysed the shortest path by comparing BFS and DFS algorithms. Five tests were conducted and, execution speed, memory usage, and processor efficiency were reviewed. The results highlighted that DFS is more recommended in cases that require time and memory efficiency as it only stores active nodes on the exploration path. Contrary to BFS, which was more consistent in finding optimal solutions and effective in ensuring the shortest path in an environment with a complex graph structure. As BFS stores all nodes at the same level before proceeding to the next level so it took quite a large memory space and moderate processing rate. Since, neither of the two algorithms showed superiority in every parameter so they concluded that optimal results can only be obtained if an appropriate algorithm is chosen based on the specific needs and characteristics of the problem at hand. Table 2 shows the result of their research.

TABLE II. COMPARISON OF BFS AND DFS SEARCH ALGORITHMS BASED ON FINDINGS BY PUTRA ET AL. (2025).

Performance Metric	DFS (Depth-First Search)	BFS (Breadth-First Search)
Execution Time	10 ms to 12 ms (Faster)	14 ms to 17 ms (Slower)
Memory Usage	50 units to 55 units (Lower)	78 to 85 units (Larger)
Processor Efficiency	0.0833 to 0.1 (High)	0.0588 to 0.0714 (Moderate)

Bartaula used Pygame and performed testing on 40×40 grids which were generated under 1 second for all algorithms, took memory below 50MB and showed consistent 60 FPS visualisation. They highlighted that BFS guaranteed shortest path but explored more nodes than A*, and DFS, while memory efficient does not guaranteed optimal solutions.

Narendran, Hothri, Saga and Sahay (2024) examined the efficiency of various maze solving algorithms, including BFS and DFS. Mazes of three different sizes, 50×50, 75×75 and 100×100 were randomly generated using pyamaze library and average of 10 iterations (i.e. average of 10 different results for each maze size) was considered. Performance metrics such as total steps taken, time elapsed, and memory usage were observed. They inferred that DFS took comparatively longer time and, unlike BFS, might end up exploring deeper paths before backtracking and might not find the shortest path overall. DFS required significantly more steps, indicating its inefficiency in finding the shortest path.

B. Informed Search Algorithms (Heuristic-Based Search)

This section highlights A* algorithm's performance in different maze types and sizes. A* algorithm uses a heuristic function to guide the search, significantly improving time efficiency over uninformed methods.

Kumar and Kaur (2019) concluded that in an 8×8 maze, Informed search algorithm, A* performed better than uninformed search algorithms like BFS, DFS and IDDFS (Iterative deepening depth-first search). Narendran, Hothri, Saga and Sahay (2024) also inferred that A* when provided with a good heuristic, outperformed BFS and DFS. Vijaya, Baghel and Mishra (2024) also summarized A*'s superior efficiency due to heuristics. They underlined the fact that the choice of algorithm depends on the availability of heuristic data and specific requirements of computational speed versus exhaustive exploration. These results are consistent with Bartaula who also deduced that A* consistently outperformed DFS and BFS in terms of nodes explored, by taking only 190 steps compared to 450 and 280 steps taken by DFS and BFS respectively. Elkari, Özkahraman and Bayrakçı (2024) also showed that A* showed faster execution time due to its requirement of heuristics and, compared to BFS and DFS, A* was the best algorithm for planning the maze path as it always chose the optimal paths considering the path cost, resource availability and time.

1) Heuristic Comparison: Chan, Mustapha and Lee (2024) developed a program base of A* algorithm to find the shortest path of a rectangular-random mazes of 10×10, 20×20, 30×30, 40×40 and 50×50. These mazes were of several types, such as wider path maze, maze without starting and ending point, blocked ending point and multiple path maze. By taking average of evaluation over 100 runs and using two heuristic approaches, Manhattan and Euclidean distance, they concluded both heuristics spent almost same time but Manhattan distance was better due to its lower operation count which results in less time consumption, contrary to Euclidean distance which takes more time despite its ability to give minimum distance between two points. Their findings are shown in fig.1 and fig.2.

Fig.1. Euclidean Distance vs Manhattan Distance based on findings by Chan, Mustapha and Lee (2024)

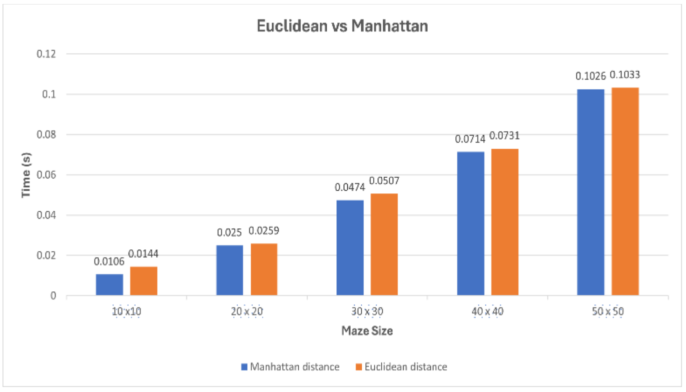
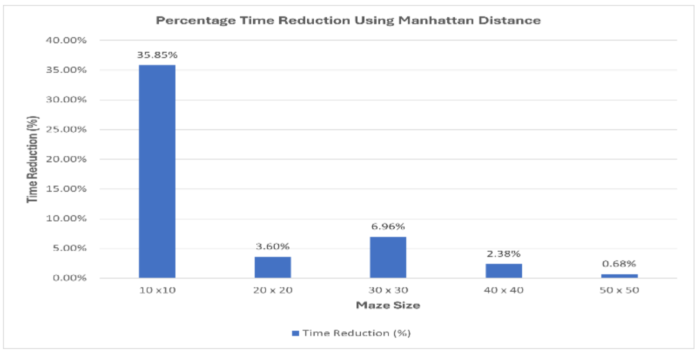


Fig.2. Percentage Time Reduction Using Manhattan Distance based on findings by Chan, Mustapha and Lee (2024).



2) Weighted/Unweighted Scenario: Biju, Gayathri, Jayapandian, Chris and Thaleeparambil (2025) examined A*, Dijkstra and BFS algorithm. They created 4 datasets: Dataset 1 (small unweighted), Dataset 2 (small weighted), Dataset 3 (large unweighted) and Dataset 4 (large weighted), using a queue to store ongoing paths and a visited set to prevent repeated exploration of cells, which stops the queue from growing with duplicate path entries and limits memory demand, resulting in improved efficiency and reducing unnecessary computations. Table 3 summarizes the results obtained by Biju et al. (2025).

TABLE III. COMPARISON OF A*, BFS AND DIJKSTRA SEARCH ALGORITHMS BASED ON FINDINGS BY BIJU, GAYATHRI, JAYAPANDIAN, CHRIS AND THALEEPARAMBIL (2025).

Scenario/Dataset	**A* Algorithm**	**Dijkstra Algorithm**	**BFS Algorithm**
Small Unweighted (Dataset 1)	Processing time was slower than BFS.	Processing time was slower than BFS.	Performed the Fastest.
Small Weighted (Dataset 2)	Demonstrated equivalent processing time to Dijkstra's, but was slightly slower due to heuristic computation.	Demonstrated equivalent processing time to A*.	Slower compared to A*/Dijkstra's (not designed for weighted graphs).
Large Unweighted (Dataset 3)	Processing time was slower than BFS.	Processing time was slower than BFS.	Performed the Fastest. Exhibited quicker execution time due to its linear behaviour relative to graph size.
Large Weighted (Dataset 4)	Demonstrated equivalent processing time to Dijkstra's, but was slightly slower due to heuristic computation.	Demonstrated equivalent processing time to A*.	Slower compared to A*/Dijkstra's.
Favourable Scenario	Excels in weighted scenarios (with heuristics).	Excels in weighted scenarios (without heuristics).	Optimal for unweighted problems.

C. Local Heuristic and Topological Search

The enhanced wall-follower algorithm in this section relies on local rules (e.g. adjacent walls) and topological properties (e.g., simply-connected maze with no loops), rather than systematic exploration of the entire search space.

Narendran et al. (2024) also found out that an enhanced wall-follower algorithm took the least memory among all the algorithms, and scored good time efficiency along with A*. However, graph search algorithms showed better result than the wall follower approach in an 8×8 maze.

D. Optimized, Hybrid and Stochastic Methods

This section covers advanced techniques for highly specific or complex environments, such as polygonal mapping or stochastic decision processes.

1) *Optimized Grid Search Algorithms (JPS, HPA*):* Vinther and Afshani (2015) created ten mazes with a corridor width of one pixel, of sizes 20, 40, 60, 80,…, 180, 200 pixels using an online maze generator and implemented JPS (Jump Point Search) for pathfinding and HPA* (Hierarchical Pathfinding), for preprocessing as well as pathfinding to find path from one corner of the maze to the opposite corner in these mazes. JPS and HPA* (Hierarchical Pathfinding) are pathfinding algorithms which work on 8-connected grid maps. The findings of their research showed that HPA* was one of the fastest pathfinding algorithms as it took the least amount of time for preprocessing. With increasing map size, HPA* performed the best in preprocessing as well as pathfinding. They also highlighted that for a dynamic map in which preprocessing is not allowed, choosing grid algorithm like JPS is a better option.

2) *Polygonal Environment Algorithms:* Vinther and Afshani (2015) created polygonal counterparts for the ten mazes they used and also implemented three pathfinding algorithms for polygonal environments: VG (Visibility Graph) and their own contribution to the field, VGO (Visibility Graph Optimized) and BSP* (Binary Space Partitioning in combination with A*). The findings showed that VG, and VGO were also the fastest pathfinding algorithms as they take the least amount of time for preprocessing. With increasing map size, BSP* was the slowest among all.

3) *Stochastic Algorithms (MDP):* Narendran, Hothri, Saga and Sahay (2024) also examined Markov decision processes with policy iteration (MDP PI) and value iteration (MDP VI). MDP PI, although with the longest time and large space utilization, was found to be the algorithm requiring the least steps. MDP PI and MDP VI showed a solid decision-making capacity which makes them better in action planning in the longer run.

4) *Hybrid Sampling-Based Algorithms (A*-RRT*):* Mahmood, Munir and Iqbal (2025) developed a hybrid of A* and RRT* (Rapidly-exploring Random Tree Star), and to evaluate its performance, conducted a comparative experiment with A* and RRT* on four factors: time taken, number of iterations, explored nodes, and path distance. They used mazes with various complexity levels as indicated by cell numbers (smaller cell number indicates a complex maze). The maze is modelled on a grid where each cell acts like a pixel, representing either free space or an obstacle. A smaller cell size (e.g., 10 px) implies a higher resolution (smaller and more cells overall) and thus a more complex maze with narrower passages and more potential nodes to explore. The results obtained by Mahmood, Munir and Iqbal (2025) at cell size 100 are depicted in Table 4.

TABLE IV. COMPARISON OF A*, RRT* AND HYBRID A*-RRT* SEARCH ALGORITHMS BASED ON FINDINGS BY MAHMOOD, MUNIR AND IQBAL (2025).

Algorithm	**Execution Time (s)**	**Path Distance (units)**	**Nodes Explored and Iteration Count**	**Rationale**
A*	124.64	1,721.20	Highest	Systematic Grid-Based Search leads to exhaustive exploration and highest computational cost, but gives ideal path lengths.
RRT*	97.58	2,239.70	Lower	Sampling-Based Method avoids exhaustive search, lowering computational load, but its sampling bias results in longer (suboptimal) routes.
Hybrid A*-RRT*	69.74	2,100.70	Lower	More robust and versatile as it balances A*'s path optimality and RRT*'s computational efficiency and generates paths nearly as optimal as A*.

E. Metaheuristic Algorithm (Evolutionary)

This section includes genetic algorithm that uses concept of natural selection to find optimized solutions.

Sagming, Heymann and Hurwitz (2019) implemented artificial intelligence (AI) technique known as Genetic Algorithm (GA) for solving randomly generated mazes of sizes 5×5, 7×7, 9×9, 11×11 and 13×13. To illustrate the effectiveness of GA, Sagming et al. implemented several non-AI techniques such as Depth First Search (DFS), Breadth-First Search (BFS), A-Star

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

Algorithm (A*), Dijkstra Algorithm (DA), and Greedy Best-First Search (GBFS). For maze generation and randomization, Prim's algorithm was used. The strongest results for the number of steps and the time taken to solve the maze were recorded over 3 tests, for each algorithm. It was found out that the GA always found the shortest path but became slower than the non-AI algorithms after 10×10 maze. Their results are depicted in Table 5.

TABLE V. COMPARISON OF GA AND NON-AI SEARCH ALGOROTHMS BASED ON FINDINGS BY SAGMIMG, HEYMANN AND HURWITZ (2019).

Maze Size	Time Take by Genetic Algorithm (GA) (ms)	Best Non-AI Algorithm	Non-AI Time (ms)	Relative Performance
5x5	13	DFS	532	GA is 41x faster.
7x7	12	GBFS	556	GA is 46x faster.
9x9	61	GBFS	565	GA is 9x faster.
11x11	1132	GBFS	599	Non-AI is 2x faster.
13x13	184 (Best Test)	GBFS	599	GA is faster in the best test, but inconsistent.

GRAPH ALGORITHMS

1) *A* Algorithm:* A* algorithm is a type of informed search algorithm as it uses a heuristic function to find the shortest path. The node (n) for which the following evaluation function (f(n)) gives the minimum value is explored:

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

Here, g(n) estimates the cost from start to the current node and h(n) is the heuristic function which estimates the cost from current node n to the goal node. Common heuristic function used with A* are Manhattan distance, Euclidean distance and Octile distance.

2) *Breadth First Search (BFS) Algorithm:* An uninformed search algorithm which explores all the nodes at the current level in a search space before moving to the next level.

3) *Depth First Search (DFS) Algorithm:* An uninformed search algorithm which explores as far as possible along a node in a search space.

4) *Dijkstra Algorithm:* In a search space, this uninformed algorithm is capable of finding the shortest path from a single source node to all other nodes with non-negative edge weights. It always chooses the unvisited node with the smallest current distance and updates the distance of the node if,

$$d(u) + w(u,v) < d(v)$$

then,

$$d(v) = d(u) + w(u,v)$$

Here, d(u) is the distance of node u, w(u,v) is weight of the edge between u and v, and d(v) is the currently known distance of node v.

5) *Flood Fill Algorithm:* It works by assigning a colour/value to each node in the search space, starting with the destination and incrementing outward to connected and unblocked cell, and going until the starting node has a value. Then the path is reconstructed by moving from the start node to the neighbouring nodes with smaller values until the destination is reached. The path discovered is the shortest path.

6) *Wall Follower Approach (Left/Right Hand Rule):* This approach does not require to store the previous nodes traversed in the memory as the traversal happens by simply following the left or right wall till the destination node is reached.

7) *Genetic Algorithm (GA):* GA was invented by John Holland in the 1960s and GAs are search heuristics that mimics the process of natural selection [13]. It involves:
   a) *Creation* of population of possible solutions. Each solution is a chromosome representing a path through the maze, often as a sequence of moves.
   b) *Fitness Function:* Each path is then evaluated using a fitness function which typically measures how close the path gets to the exit, how short a path is, etc.
   c) *Selection:* The fittest paths are selected to be parents for the next generation and less fit paths are eliminated.
   d) *Crossover:* Based on the fitness function, a combination of two parent paths is created to form a new 'offspring' path with better fitness.
   e) *Mutation:* Random changes are introduced in the offspring paths to maintain the genetic diversity and prevent the algorithm from getting stuck in local optimum.

Steps b to e are repeated for a number of generations or iterations to improve the average fitness function of the population and path gets closer to reaching the destination node and terminates upon reaching the destination node.

8) *Rapidly-exploring Random Tree Star (RRT*):* RRT* algorithm extends the basic RRT by adding optimization steps. It begins by incrementally growing a tree from the start node toward a randomly sampled point in the search space. In each step, the algorithm steers from the nearest existing tree node toward a new random sample, ensuring the path segment is collision-free. The key difference from standard RRT is the search for an optimal parent: before adding the new node, RRT* checks existing nodes within a certain radius to find the parent that provides the

lowest total cost path from the start. Finally, the algorithm performs rewiring, checking if the new node can act as a better (lower-cost) parent for its neighbours, thus improving the overall structure of the tree. This continuous optimization allows RRT* to converge on the shortest possible path to the goal over time, even if the first path found wasn't optimal.

9) *Hybrid A*-RRT* Algorithm:* Proposed by Mahmood, Munir and Iqbal (2025), the Hybrid A*-RRT* algorithm leverages the strengths of both pathfinding methods by biasing the RRT*'s random sampling toward the optimal A* path. It first computes a fast, deterministic A* path from start to goal. This discrete path is then converted into a continuous guide using an interpolation function. The RRT* process uses a tunable parameter, λ (where $0 \leq \lambda \leq 1$) , to balance its natural probabilistic exploration with this A* guided sampling, aiming to find a path that is both near-optimal and found quickly.

10) *Random Mouse Algorithm:* A non-systematic algorithm as the 'mouse' moves in a straight line until it hits a wall, and then randomly chooses a new direction. It is guaranteed to find the exit eventually but is highly inefficient.

11) *Markov decision processes with policy iteration (MDP PI):* This algorithm aims to the find the optimal policy which in context of maze refers to- the best action to take in every state (maze cell)- to maximize expected rewards (e.g., reaching the end quickly) for an MDP by following two steps:
    a) *Policy Evaluation:* Value of current policy is evaluated. This step takes the current policy π and determines how good it is by calculating the value function V[s] (the expected cumulative reward) for every state 's', assuming the agent follows π. This loop repeats until the value function converges.
    b) *Policy Improvement:* Creating a new policy that is greedy with respect to the value function V calculated in the previous step by selecting the action 'a' in each state 's' that maximizes the expected next reward plus the discounted value of the next state. If the policy changes, the process repeats; otherwise, the optimal policy has been found.

12) *Markov decision processes with value iteration (MDP VI):* This algorithm finds the optimal value function for an MDP directly by combining the evaluation and improvement steps into a single update.
    a) *Iterative Update*: For every state 's', the value V[s] is updated using the Bellman Optimality Equation. This update involves calculating the expected return for all possible actions 'a' and then taking the maximum of those returns. This single step simultaneously improves the value function and implicitly determines a better policy. The process repeats until the change Δ in the value function falls below a small threshold θ.
    b) *Policy Extraction:* Once the optimal value function V* is found, the optimal policy is extracted in a final pass by simply choosing the action 'a' in each state 's' that is greedy with respect to V*.

13) *Greedy Best First Search (GBFS):* It is also an informed search algorithm proposed by a computer scientist Judea Pearl. GBFS is described as a search algorithm that always expands nodes with the smallest heuristic value h(n) [22]. h(n) is the heuristic function which estimates the cost from current node n to the goal node. Euclidean or Manhattan distance are used as heuristic.

$$f(n) = h(n)$$

14) *Jump Point Search (JPS):* It is an optimization of A* algorithm specifically for grid-based maps. It prunes the search space by identifying "jump points"— nodes that are guaranteed to be on an optimal path— and skips over large portions of empty space. During the search, JPS dynamically determines these jump points using strict pruning rules based on forced neighbours. A forced neighbour is a nearby cell that the shortest path must pass through because a wall or obstacle nearby forces the path to turn at that point. The algorithm continuously "jumps" along straight lines, checking the current cell at each step. When the algorithm finds a current cell that, due to its adjacency to an obstacle, creates a situation where a specific neighbour can only be reached by turning through this current cell, it marks the current cell as a jump point and stops the jump.

15) *BSP* (Binary Space Partitioning in combination with A*):* The BSP* algorithm, proposed by Vinther and Afshani (2015), is designed for optimal pathfinding in dynamic polygonal worlds where obstacles can move.

a) BSP* combines A* search with Binary Space Partitioning (BSP) by constructing a new BSP tree before each pathfinding query to model the current environment, instead of using a fixed visibility graph, to achieve its dynamic capability.
b) *Visibility Calculation:* This tree is then used to dynamically calculate visibility between corners by traversing obstacles from closest to farthest.
c) *Angular Intervals:* A key optimization is early termination which relies on 'angular intervals.' These are the portions of a 360° range around a given corner that are blocked by surrounding obstacles, and are represented as an array or list of angle ranges.
d) *Early Termination:* As BSP* processes the obstacles from the nearest to the farthest, it incrementally calculates the accumulated blocked view. The early termination occurs when the combined angular intervals of closer obstacles add up to a full circle (360 degrees), proving that no distant obstacles can possibly be visible to the current corner, thus efficiently pruning the search.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

The algorithm further optimizes the search by restricting the area to be checked based on the agent's previous position. While its expected running time of O(N^2) is generally slower than algorithms relying on static pre-processing, BSP* remains the necessary and primary choice for finding optimal paths in environments that change during runtime.

16) *HPA* (Hierarchical Pathfinding):* The HPA* (Hierarchical Pathfinding A*) algorithm, introduced by Adi Botea, Martin Müller, and Jonathan Schaeffer in 2004, is a technique to speed up A* search on grid maps by employing a hierarchy of abstract maps. The acceleration is achieved in a preprocessing step where the grid is partitioned into equally sized clusters. An abstract graph is then constructed, using "entrances" between clusters as nodes, connected by "inter-edges" (paths between clusters) and "intra-edges" (internal paths within a cluster). During runtime, A* is first executed on this high-level abstract graph to find an approximate path, which then serves as a series of intermediate goals to guide the final A* search on the original detailed grid, thus helping the algorithm avoid large dead ends. The primary trade-off for this significant speed improvement is that the resulting path is no longer guaranteed to be perfectly optimal, though it is typically no longer than 1% of the optimal grid path length [23].

17) *VG (Visibility Graph):* This algorithm works by constructing a visibility graph that describes which corners can be seen from each other. This graph is formed by connecting every corner with every other corner that is in line of sight. For the purpose of pathfinding, the algorithm only includes protruding corners, those that do not point inward (like the corner of a table) as nodes, because the depressed corners, those that point inward (like the inner corner of a U-shaped wall) are excluded as they are not part of an optimal path. The shortest path is then found by adding the start and goal positions to the graph and executing the A* search algorithm.

18) *VGO (Visibility Graph Optimized):* The VGO (Visibility Graph with Optimizations) algorithm, proposed by Vinther and Afshani (2015), is an improvement over the VG algorithm that reduces the required storage and increases pathfinding speed in polygonal worlds. It achieves this through an optimization applied during the preprocessing step to reduce the number of edges, leaving only a minimal visibility graph where every remaining edge is necessary for at least one optimal path . At runtime, VGO further optimizes the A* search by using another optimization to limit the number of protruding corners that are considered as successors, based on the previous position. While VGO is a definite improvement over VG due to its reduced visibility graph size and memory requirements, the full positive effect of the extra calculation required by the runtime optimization is yet to be explored.

19) *IDDFS (Iterative deepening depth-first search):* It is an uninformed search algorithm that efficiently combines the space-efficiency of DFS with the completeness of BFS. It works by starting with a depth limit of zero and increasing the limit by one in each iteration until the goal node is found. Since, each iteration is a depth-first search, IDDFS only needs to store the current path on the stack, while avoiding the massive memory requirements of BFS. Although it repeatedly explores nodes in the upper levels of the search tree, this overhead is surprisingly small, making its time complexity asymptotically similar to BFS and often making it the preferred algorithm when the depth of the solution is unknown.

20) *Recursive Backtracker:* It begins at a starting cell, which it marks as visited, and maintains a stack to track its current path. From the current cell, it randomly selects an unvisited neighbouring cell and moves into it, adding the new cell to the stack. This process of exploring deeper into the graph continues until the algorithm reaches a dead-end, meaning the current cell has no unvisited neighbours. At this point, the algorithm backtracks by removing the cell from the stack and returning to the previous cell to try any remaining unvisited branches. This cycle of random exploration and backtracking repeats until every cell in the entire space has been visited. The algorithm is favoured for its simplicity, quick results in maze tasks, and its space efficiency, because it only needs to store the cells on its current path in the stack.

BTABLE VI. TABLE OF COMPARISON

S.No.	Algorithm	Optimality	Completeness	Search Type	Map Type/ Environment	Preprocessing
1	**A***	Yes (if heuristic is admissible)	Yes	Informed	Grid Maze	No
2	**BFS**	Yes (for unweighted)	Yes	Uninformed	Grid Maze	No
3	**DFS**	No	Yes (for finite graphs)	Uninformed	Grid Maze	No
4	**Dijkstra**	Yes	Yes	Uninformed	Grid Maze	No

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

5	**Flood-Fill**	No	Yes	Uninformed	Grid Maze	No
6	**Wall Follower**	No	Yes (in simple mazes)	Heuristic (Local)	Maze (Boundary-Based)	No
7	**Genetic**	No	No	Meta-Heuristic	Grid Maze	No
8	**RRT***	Yes (Asymptotically)	Yes (Probabilistic)	Informed (Sampling-Based)	Continuous Maze	No
9	**Random Mouse**	No	No	Uninformed (Random)	Grid Maze	No
10	**MDP PI**	Yes	Yes	Dynamic Programming (Model-Based)	Stochastic Grid Maze	Yes
11	**MDP VI**	Yes	Yes	Dynamic Programming (Model-Based)	Stochastic Grid Maze	Yes
12	**GBFS**	No	No (can get stuck)	Informed	Grid Maze	No
13	**JPS**	Yes	Yes	Informed	Grid Maze	Yes
14	**BSP***	Yes	Yes	Informed	Dynamic Polygonal Maze	Yes
15	**HPA***	No (Near-Optimal)	Yes	Informed	Hierarchical Grid Maze	Yes
16	**VG**	Yes	Yes	Informed	Static Polygonal Maze	Yes
17	**VGO**	Yes	Yes	Informed	Static Polygonal Maze	Yes
18	**IDDFS**	Yes (for unweighted graphs)	Yes	Uninformed	Grid Maze	No
19	**Recursive Backtracker**	No	Yes	Uninformed	Grid Maze	No
20	**Hybrid A* - RRT***	No (Near-Optimal)	Yes (Probabilistic)	Informed (Sampling- Based)	Continuous Maze	No

[B]Note: These values in the tables are strictly context-dependent on the papers reviewed in the literature survey.

ACKNOWLEDGMENT

The author sincerely acknowledges Amit Pal for giving his valuable time for reviewing the paper and providing substantive constructive feedback. His suggestions significantly improved the quality of the review paper.

CONCLUSION

This review analyzed a wide range of solving algorithms from informed search algorithms, uninformed search algorithms, metaheuristic algorithms, local algorithms, optimized grid search, to polygonal environment algorithms, stochastic algorithms and hybrid sampling-based algorithms, and evaluated their strengths, limitations across different maze structures.

The findings underscore that for standard grid-based mazes, A* stands out as the most optimal, if the heuristic chosen is admissible. JPS (Jump Point Search) outperforms A* in uniform-cost grids by reducing the effective search space, while HPA*, though not optimal, is tailored for very large maps, providing significant speed gains through its hierarchical preprocessing. For static polygonal environments, the Visibility Graph (VG) and its optimized variant, VGO, algorithms are optimal but require costly initial preprocessing. In a dynamic maze environment, BSP* is the sole optimal solution, although its expected $O(N^2)$ running time is its major weakness, making it generally slower than static methods. Basic algorithms like Recursive Backtracker and DFS offers time and memory efficiency, and are useful for maze generation, but they do not guarantee optimal path length. Random Mouse strategy is the simplest but is unreliable due to its potentially infinite execution time. In complex settings, hybrid algorithms like A*-RRT* combines efficiency with near-optimal paths, making them an ideal choice for diverse challenges in robotics, navigation, and AI.

The comparative study shows that no single algorithm is universally superior; rather, the most effective choice is highly dependent on maze type, heuristic availability, and computational capabilities.

Future work could provide a large-scale empirical data to strongly validate the performance of polygonal maps and unstructured grids. Additionally, hybrids of basic, well-understood classical algorithms could be developed to explore their potential in pathfinding in mazes and integration of heuristic and sampling-based methods could be done. Reinforcement learning could be used to learn optimal heuristics or prune the search space, potentially outperforming manually designed heuristics. Pathfinding algorithms for dynamic obstacles remains underexplored.

REFERENCES

[1] Vinther, A. S. H., & Afshani, P. (2015). Pathfinding in two dimensional worlds. *no. June*.

[2] Kaur, N. K. S. (2019). A review of various maze solving algorithms based on graph theory. IJSRD, 6(12), 431-434.

[3] Vijaya, J., Baghel, A. K., & Mishra, A. S. (2024, March). Dynamic Maze Solver: Integrating Advanced Algorithms for Optimal Path-finding in Varied Environments. In 2024 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI) (Vol. 2, pp. 1-6). IEEE. doi:10.1109/IATMSI60426.2024.10503462

[4] ELKARI, B., OURABAH, L., SEKKAT, H., HSAINE, A., & ESSAIOUAD, C. (2024). Exploring Maze Navigation: A Comparative Study of DFS, BFS, and A. doi: https://doi.org/10.19139/soic-2310-5070-1939

[5] Chan, K. H., Mustapha, A., & Lee, S. C. (2024). Shortest Pathfinding in a Standard Rectangular Maze using A* Search Algorithm. International Journal of Integrated Engineering, 16(3), 244-256. doi: https://doi.org/10.35234/fumbd.1518386

[6] Putra, A. A. D., Maulana, A. N., Febrianti, S. B., Camelia, N., Hutagalung, S. K., & Khudori, A. N. (2025). Comparison of Breadth-First Search (BFS) and Depth-First Search (DFS) Algorithms for Shortest Search in Campus Labyrinth. Journal of Enhanced Studies in Informatics and Computer Applications, 2(2), 43-51. doi: https://doi.org/10.47794/jesica.v2i2.14

[7] Erbil, M. E., Özkahraman, M., & Bayrakçı, H. C. (2025). Comprehensive Performance Analysis and Evaluation of Various Maze Solving Algorithms for Optimized Autonomous Navigation and Pathfinding. *Fırat Üniversitesi Mühendislik Bilimleri Dergisi*, *37*(1), 151-166.

[8] Bartaula, B. Interactive Maze Generation and Pathfinding Simulation: A Comprehensive Educational Platform for Algorithm Visualization.

[9] Narendran, A., Hothri, A., Saga, H., & Sahay, A. (2024, June). MazeSolver: Exploring Algorithmic Solutions for Maze Navigation. In 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT) (pp. 1-8). IEEE. doi: 10.1109/ICCCNT61001.2024.10725996

[10] Biju, A., Gayathri, M. P., Jayapandian, N., Chris, A., & Thaleeparambil, N. R. (2025, March). Efficient Pathfinding in a Maze to overcome Challenges in Robotics and AI Using Breadth-First Search. In 2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT) (pp. 727-732). IEEE. doi: 10.1109/CSNT64827.2025.10968383

[11] Sagming, M. N., Heymann, R., & Hurwitz, E. (2019, September). Visualising and solving a maze using an artificial intelligence technique. In 2019 IEEE AFRICON (pp. 1-7). IEEE. doi: 10.1109/AFRICON46755.2019.9134044

[12] Mahmood, A., Munir, M., & Iqbal, S. (2025, May). Adaptive A* RRT* for Robot Navigation in Maze. In 2025 18th International Conference on Engineering of Modern Electric Systems (EMES) (pp. 1-5). IEEE. doi: 10.1109/EMES65692.2025.11045571

[13] M. Mitchell, An introduction to genetic algorithms, Massachusetts: The MIT Press, 1999.

[14] Suhaib Al-Ansarry and Salah Al-Darraji. Hybrid rrt-a*: An improved path planning method for an autonomous mobile robots. Iraqi Journal for Electrical & Electronic Engineering, 17(1), 2021. doi: https://doi.org/10.37917/ijeee.17.1.13

[15] Sivasankar Ganesan, Balakrishnan Ramalingam, and Rajesh Elara Mohan. A hybrid sampling-based rrt* path planning algorithm for autonomous mobile robot navigation. Expert Systems with Applications, 258:125206, 2024. doi: https://doi.org/10.1016/j.eswa.2024.125206

[16] Iram Noreen, Amna Khan, and Zulfiqar Habib. Optimal path planning using rrt* based approaches: a survey and future directions. International Journal of Advanced Computer Science and Applications, 7(11), 2016.

[17] Colin Brennan, Nicholas Ciordas, Samhita Pokkunuri, Nickolas Regas, Adam Wahid, Will Wands, and Shreya Srikanth. Comparative analysis of a* and rrt* pathfinding algorithms for autonomous drone navigation in different environments. 2024. doi: 10.1109/URTC65039.2024.10937509

[18] Benjamin B Spratling IV and Daniele Mortari. A survey on star identification algorithms. Algorithms, 2(1):93–107, 2009. doi: https://doi.org/10.3390/a2010093

[19] Karthik Karur, Nitin Sharma, Chinmay Dharmatti, and Joshua E Siegel. A survey of path planning algorithms for mobile robots. Vehicles, 3(3):448–468, 2021. . doi: https://doi.org/10.3390/vehicles3030027

[20] Farzad Kiani, Amir Seyyedabbasi, Royal Aliyev, Murat Ugur Gulle, Hasan Basyildiz, and M Ahmed Shah. Adapted-rrt: novel hybrid method to solve three-dimensional path planning problem using sampling and metaheuristic-based algorithms. Neural Computing and Applications, 33(22):15569–15599, 2021. doi: https://doi.org/10.1007/s00521-021-06179-0

[21] Jauwairia Nasir, Fahad Islam, and Yasar Ayaz. Adaptive rapidly exploring-random-tree-star (rrt*)-smart: algorithm characteristics and behavior analysis in complex environments. 2013. doi: https://doi.org/10.17576/apjitm-2013-0202-04

[22] L. Beck. A Case Study on the Search Topology of Greedy Best-First Search.

[23] Berg, Cheong, Kreveld, Overmars; Computational Geometry, Algorithms and Applications; Springer; 3rd edition. . doi: https://doi.org/10.1007/978-3-540-77974-2_4

[24] Botea, Muller, Schaeffer; Near Optimal Hierarchical Path-Finding; Department of Computing Science, University of Alberta; 2004.

[25] T.H. Coremen, C.E. Leiserson, R.L. Rivest and C. Stein, “Introduction to Algorithms”, Third edition, Prentice Hall of India, pp. 587-748, 2009.

[26] Abu-Khzam FN, Langston MA, Mouawad AE, Nolan CP. A hybrid graph representation for recursive backtracking algorithms. In: Int Workshop Front Algorithmics. Springer 2010. doi: https://doi.org/10.1007/978-3-642-14553-7_15

[27] Kondrak G, Van Beek P. A theoretical evaluation of selected backtracking algorithms. Artif Intell 1997; 89(1-2): 365 387. doi: https://doi.org/10.1016/S0004-3702(96)00027-6

[28] Dehghani M, Hubálovský Š, Trojovský P. Cat and mouse based optimizer: A new nature-inspired optimization algorithm. Sensors 2021; 21(15): 5214. doi: https://doi.org/10.3390/s21155214

[29] Yadav S, Verma KK, Mahanta S. The Maze problem solved by Micro mouse. I Int J Eng Adv Technol 2012; 2249: 8958.

[30] 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. doi: https://doi.org/10.1007/978-3-030-13321-4_22

[31] Chen, G., Gaebler, J.D., Peng, M., Sun, C. and Ye, Y., 2021. An Adaptive State Aggregation Algorithm for Markov Decision Processes. arXiv preprint arXiv:2107.11053 unpublished doi: https://doi.org/10.48550/arXiv.2107.11053

[32] Abadi, E. and Brafman, R.I., 2020. Learning and solving regular decision processes. arXiv preprint arXiv:2003.01008 unpublished. doi: https://doi.org/10.48550/arXiv.2003.01008

[33] Sapio, S. S. Bhattacharyya and M. Wolf, ”Efficient Model Solving for Markov Decision Processes,” 2020 IEEE Symposium on Computers and Communications (ISCC), Rennes, France, 2020, pp. 1-5, doi: 10.1109/ISCC50000.2020.9219668.