Path Planning Analysis

Ashwin Disa
Robotics Engineering
Worcester Polytechnic Institute
Email: amdisa@wpi.edu

I. INTRODUCTION

Path planning is a fundamental challenge in robotics and autonomous systems, where an agent must determine a feasible route from a given start position to a designated goal while avoiding obstacles and satisfying constraints such as collision avoidance. In practice, path planning algorithms must operate efficiently over discrete or continuous representations of the environment, handle varying levels of complexity in obstacle distribution, and guarantee a path if one exists (or report failure otherwise).

This report focuses on the development and comparative analysis of four distinct path planning algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, and Rapidly-Exploring Random Trees (RRT). BFS and DFS are classical graph-based algorithms that systematically explore nodes (or grid cells) to find paths. Their simplicity and completeness make them a natural starting point for understanding how systematic search can uncover a valid route in a discrete occupancy grid. A* builds upon these ideas by leveraging heuristics to more efficiently guide the search toward the goal, often resulting in shorter paths and reduced exploration time compared to uninformed methods. Finally, RRT is a sampling-based technique commonly used for motion planning in continuous spaces. It incrementally builds a tree of random samples toward the goal region, making it particularly useful for higher-dimensional or more complex environments.

In this assignment, a grid-based map serves as the environment representation, where each cell is either free or occupied. The robot, modeled as a one-meter-diameter circle, must navigate safely through free cells without intersecting any obstacles. By adopting a systematic approach to collision detection and path feasibility, we ensure that the paths each algorithm generates are valid within the map's constraints.

Beyond implementing these algorithms, we also measure and compare their performance across various metrics, including computational time, path length, number of nodes (or states) explored, and overall success rate. This comparison provides valuable insights into how algorithmic design choices—such as breadth-first versus heuristic-driven search or systematic expansion versus random sampling—affect path quality and computational cost. The findings from this study inform decisions about when to select one algorithm over another, balancing trade-offs between efficiency, reliability, and ease of implementation.

II. METHODOLOGY

This section describes the setup procedure and path planning using four algorithms: Breadth-First Search, Depth-First Search, A* Search, and Rapidly-Exploring Random Trees. Each algorithm operates on a two-dimensional occupancy grid represented as a matrix of zeros (free space) and ones (obstacle), with movement permitted in the four directions.

The environment is modeled as a two-dimensional occupancy grid in which each cell holds either a 0 or a 1. A value of 0 indicates free space and a value of 1 indicates an obstacle. Coordinates are expressed in the form (x, y), where x denotes the horizontal coordinate (column index) and y denotes the vertical coordinate (row index). The origin $(0, 0)$ is located at the top-left corner of the grid, and all valid coordinates satisfy the bounds $0 \leq x < \text{cols}$ and $0 \leq y < \text{rows}$.

The input file consists of two main parts: a list of start and goal coordinate pairs, followed by the grid itself represented as lines of characters. In the first part, each line specifies a start position and a goal position in plain text—for example, "Start: (2,5) Goal: (7,3)". A parsing function reads these lines, identifies the start and goal entries, converts each coordinate into a tuple of integers, and appends the tuple pair to a list. In the second part, each subsequent line describes one row of the map using a string of "." characters for free cells and "X" characters for occupied cells. These strings are converted into rows of a two-dimensional list by mapping "." to 0 and "X" to 1. Upon completion of parsing, the variable start_goal_pairs contains the list of all start–goal tuples, and the variable grid contains the corresponding occupancy grid as a matrix of zeros and ones.

The function plot_grid(grid, path, start, goal) generates a grayscale representation of the occupancy grid using Matplotlib. Occupied cells (value 1) appear dark, while free cells (value 0) appear light. The computed path is drawn as blue arrows connecting consecutive waypoints. The start cell is highlighted in green and the goal cell in red. Additionally, the sequence of path coordinates is printed to the console for diagnostic purposes.

Execution begins by selecting the map file (for example, map2.txt) and invoking the parsing routine to load the grid and extract start and goal coordinates. Each start–goal pair is then validated to ensure both cells lie within the grid boundaries and represent free space. The user is prompted to choose one of the four planning algorithms—Breadth-First Search, A*, Rapidly-Exploring Random Trees, or Depth-First Search. After selection, the chosen algorithm runs on the occupancy grid; if a valid path is found, it is plotted, its coordinates are printed, and performance metrics such as computation time, memory usage, and path length are recorded and reported.

A. *Breadth-First Search (BFS)*

Breadth-First Search is implemented with a queue to explore cells in order of increasing distance from the start. A two-dimensional array visited tracks which cells have been examined, and a dictionary parent records each cell's predecessor for later path reconstruction. The start cell is enqueued and marked as visited. During each iteration, the front cell is dequeued and compared to the goal; if it is not the goal, all valid neighbors that have not yet been visited are enqueued and marked as visited. This process repeats until either the goal is reached or the queue becomes empty. The resulting path minimizes the number of grid steps. The results are shown in Figure. 1.

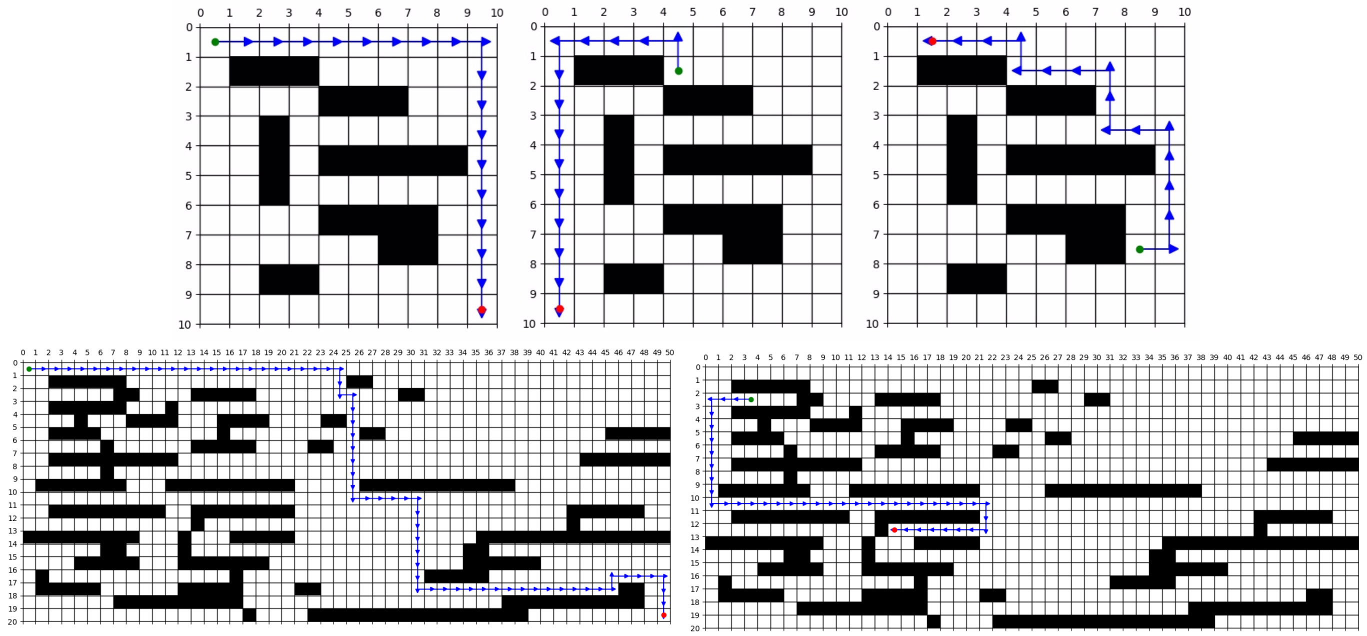

Fig. 1: Breadth-First Search

B. *Depth-First Search (DFS)*

Depth-First Search uses a stack to explore as far along each branch as possible before backtracking. A set visited prevents revisiting cells, and a dictionary parent supports path reconstruction. The start cell is pushed onto the stack. In each iteration, the top cell is popped and checked against the goal; if it is not the goal, all valid, unvisited neighbors are pushed onto the stack and marked as visited. This continues until the goal is found or the stack is emptied. Although this method does not guarantee the shortest path, it will find a valid path if one exists. The results are shown in Figure. 2.

C. *A* Search*

A* Search combines the actual cost from the start with a heuristic estimate of the remaining distance to the goal. The Manhattan distance,

$$h(x, y) = |x - x_{\text{goal}}| + |y - y_{\text{goal}}|,$$

serves as an admissible heuristic for four-connected movement. Two dictionaries, g_score and f_score, maintain the cost from the start to each cell and the estimated total cost, respectively. A priority queue orders cells by their f_score. Initialization sets g_score(start) to zero and f_score(start) to the heuristic value. At each step, the cell with the lowest f_score is removed from the queue and expanded; each neighbor's tentative cost is computed, and if it improves on the current g_score, both g_score and f_score are updated and the neighbor is added to the queue. The algorithm terminates upon reaching the goal, and the path is reconstructed via the parent dictionary. This approach typically finds the optimal path more efficiently than uninformed searches on large grids. The results are shown in Figure. 3.

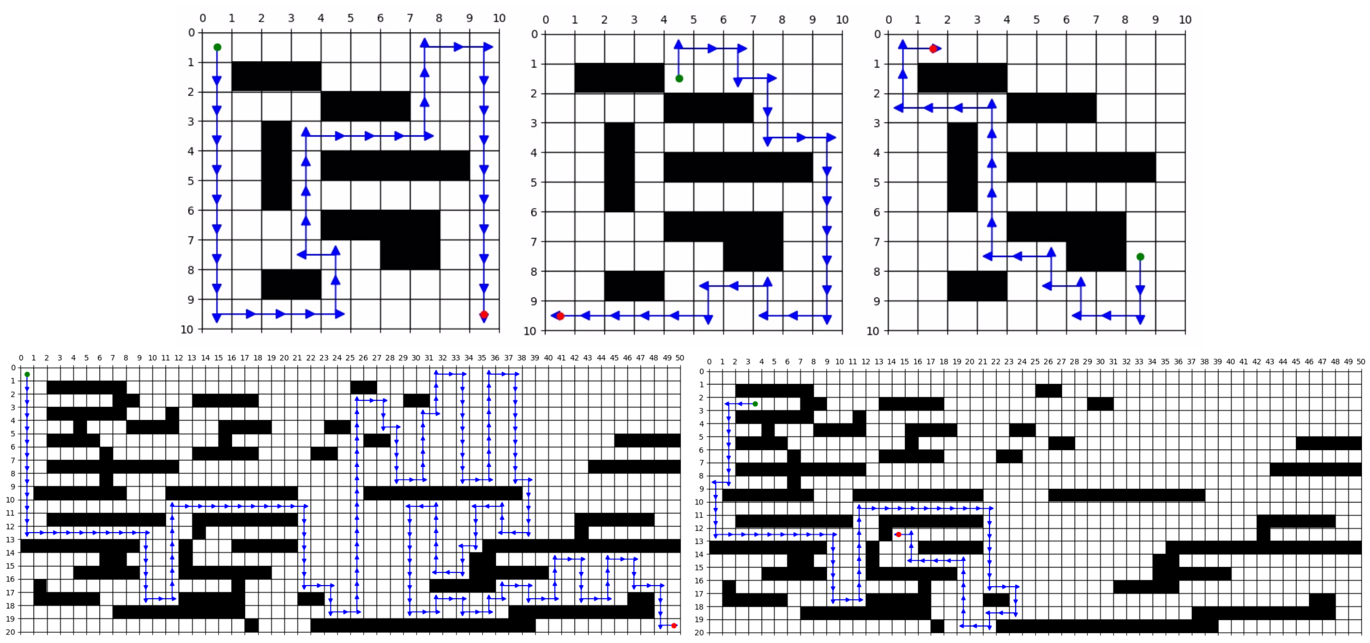

Fig. 2: Depth-First Search

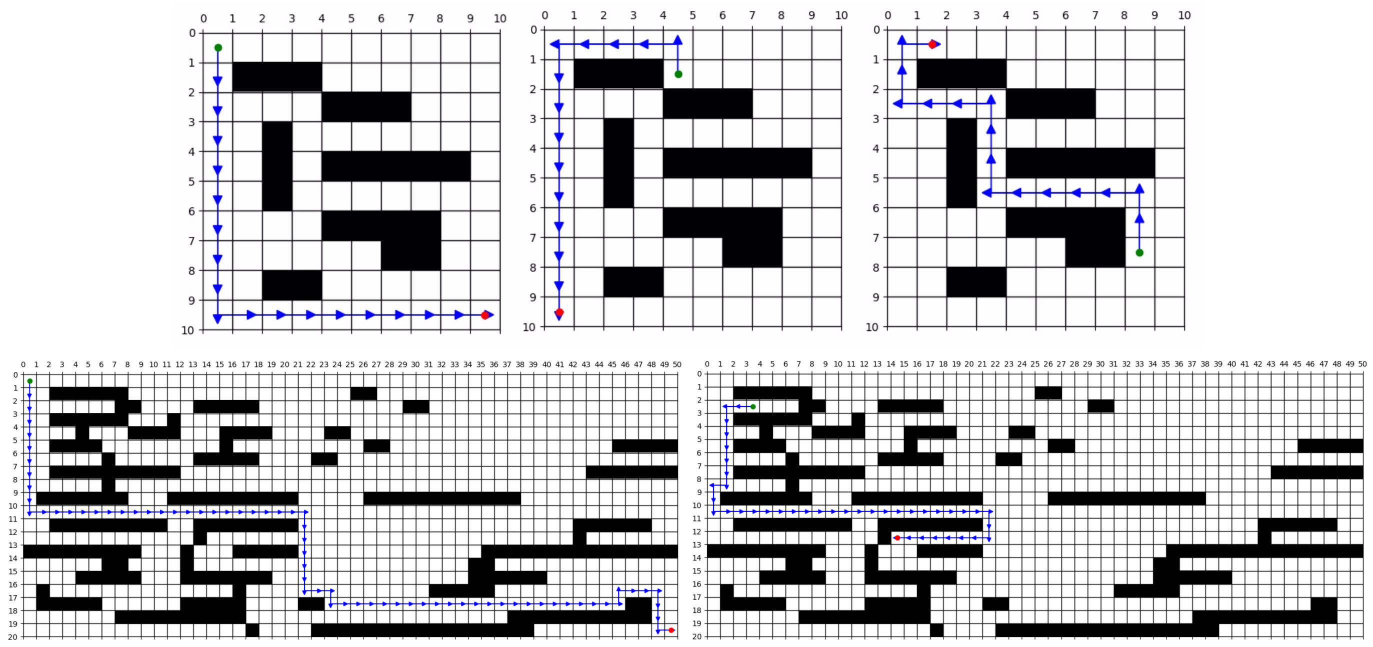

Fig. 3: A* Search

D. Rapidly-Exploring Random Trees (RRT)

Rapidly-Exploring Random Trees constructs a search tree by sampling random free cells, occasionally biasing toward the goal. For each sample, the nearest node in the existing tree is identified using Euclidean distance, and a single grid step is taken from that node toward the sample. If the new cell lies in free space, it is added to the tree with a link to its parent. Sampling continues until the goal is added to the tree, at which point backtracking reconstructs the path from the start to the goal. Although RRT can be less efficient than A* on a discrete grid, it exemplifies how sampling-based methods can scale to more complex or higher-dimensional planning problems. The results are shown in Figure. 4.

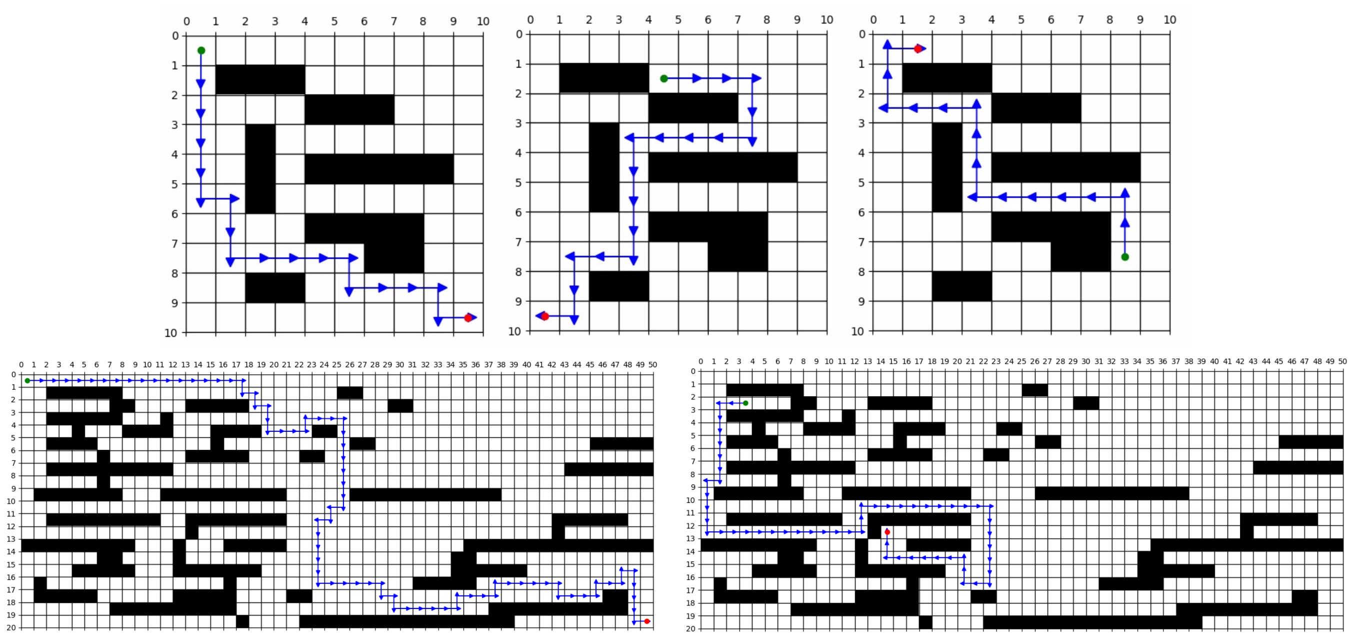

Fig. 4: Rapidly-Exploring Random Trees

E. Results and Analysis

TABLE I: Comparison of BFS, DFS, A*, and RRT on Three Smaller Maps (map1.txt)

Algorithm	Map	Time (s)	Current Mem (MB)	Peak Mem (MB)	Path Length
BFS	Map 1	0.00056	0.012448	0.014216	18.0
	Map 2	0.00056	0.00012	0.004944	14.0
	Map 3	0.00061	0.000136	0.004944	16.0
DFS	Map 1	0.00037	0.006096	0.008904	38.0
	Map 2	0.00042	0.000216	0.005792	26.0
	Map 3	0.00031	0.000168	0.005856	20.0
A*	Map 1	0.00104	0.007224	0.011896	18.0
	Map 2	0.00039	0.00012	0.00412	14.0
	Map 3	0.00059	0.000136	0.004056	16.0
RRT	Map 1	0.03016	0.013853	0.014621	18.0
	Map 2	0.02503	0.00012	0.002032	14.0
	Map 3	0.05628	0.001409	0.004369	20.0

TABLE II: Comparison of BFS, DFS, A*, and RRT on Two Larger Maps (map3.txt)

Algorithm	Map	Time (s)	Current Mem (MB)	Peak Mem (MB)	Path Length
BFS	Map 1	0.00334	0.089248	0.14316	70.0
	Map 2	0.00189	0.013888	0.04144	41.0
DFS	Map 1	0.00049	0.03116	0.08340	196.0
	Map 2	0.00241	0.000592	0.08880	73.0
A*	Map 1	0.00736	0.041576	0.095696	70.0
	Map 2	0.00276	0.000336	0.032584	41.0
RRT	Map 1	10.35449	0.052086	0.103402	0
	Map 2	3.58233	0.007680	0.027780	49.0

Table I presents the execution time, memory usage, and path length for Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, and Rapidly-Exploring Random Trees (RRT). BFS and A* consistently produced the minimal-step solutions, with BFS often exhibiting the lowest execution times on small grids and A* performing equally well when its heuristic effectively guides the search. DFS identified valid but significantly longer paths—for example, 38 steps versus 18 steps for BFS on Map 1—while RRT required more time due to the overhead of random sampling.

Results for the larger maps in Table II reveal more pronounced differences. With two different start and goals, BFS and A* yielded identical optimal paths (70 steps on Map 1 and 41 steps on Map 2). However, BFS's memory usage grew substantially—peaking at 0.14316 MB on Map 1 compared to 0.014216 MB in the smaller cases—while A* leveraged its heuristic to maintain a lower memory footprint (0.095696 MB on Map 1). DFS again returned suboptimal paths (196 steps on Map 1) but occasionally completed in very little time (0.00049 s), demonstrating a trade-off between speed and path quality. RRT struggled to find a valid path within the iteration limit on Map 1 and exhibited long runtimes (over 10 s); on Map 2 it produced a 49-step path in approximately 3.58 s, highlighting the performance penalties of sampling-based methods on large, discrete grids. All results are plotted in Figure. 5.

These findings underscore several key observations. First, BFS and A* remain the strongest candidates for 2D grid-based path planning, consistently delivering optimal paths with manageable runtime and memory demands. Second, DFS can discover paths rapidly in small environments but suffers from significantly increased path lengths as the grid grows. Third, A*'s heuristic guidance reduces the search frontier compared to BFS, especially on larger maps. Finally, RRT's pronounced performance penalty on discrete grids reaffirms its primary advantage in continuous or high-dimensional planning scenarios rather than standard 2D occupancy grids.

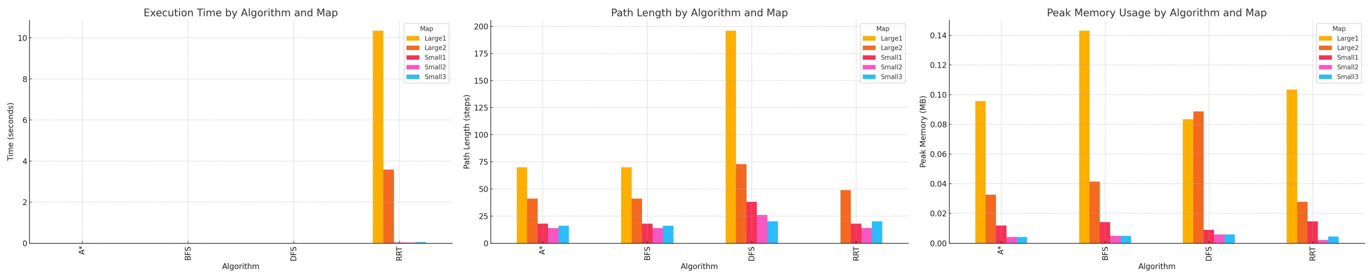

Fig. 5: Results

III. Conclusion

This study has compared the performance of four path planning algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, and Rapidly-Exploring Random Trees (RRT)—on a variety of two-dimensional grid maps differing in size and obstacle density. Across all test cases, BFS and A* consistently produced the shortest paths, confirming their reliability for structured, discrete grid environments. Although BFS demonstrated low execution times on smaller maps, its unguided search led to higher memory consumption as map size increased. A* leveraged a Manhattan-distance heuristic to reduce unnecessary node expansions, achieving both optimality and improved efficiency on larger grids. DFS was able to locate a feasible route quickly but suffered from severely suboptimal path lengths due to its depth-first exploration strategy. RRT, while effective in continuous or high-dimensional spaces, incurred high computational overhead and showed inconsistency in finding valid solutions in discrete grid settings, particularly as map complexity grew. Together, these results highlight A* as offering the best trade-off between path optimality, runtime performance, and memory usage in deterministic grid-based planning.

IV. Future Work

Further research could extend this evaluation to dynamic environments in which obstacles move or appear unpredictably, requiring replanning or incremental search methods. Incorporating variations of A* such as Weighted A* or incremental algorithms like D* Lite may yield additional performance gains in time-critical applications. In addition, integrating sampling-based planners with heuristic guidance—for example, RRT* or informed RRT—could bridge the gap between discrete and continuous planning domains.

Finally, benchmarking these algorithms on robotic platforms operating in real-world settings would validate their practical utility and reveal implementation challenges related to sensor noise, actuator uncertainty, and real-time constraints. Potential applications of these grid-based planning techniques include route optimization for warehouse automation, navigation strategies for planetary exploration rovers traversing unknown terrain, flight path generation for unmanned aerial vehicles in search-and-rescue missions, dynamic path adjustment in autonomous vehicles, and guidance for minimally invasive surgical tools in medical robotics. Evaluating these algorithms in such diverse domains would demonstrate their versatility and inform domain-specific enhancements.