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 step-by-step approach used to parse the environment, plan a path, and evaluate performance using four path planning algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), A*, and Rapidly-Exploring Random Trees (RRT). The methodology encompasses the data structure design for the occupancy grid, the parsing of start and goal positions, the algorithmic details of each path planner, and the metrics gathered to assess performance.

A. Environment Representation

- **Occupancy Grid:** The map is represented as a 2D occupancy grid, a matrix of 0s and 1s, where each cell's coordinates are given in (x, y) format:
  - x: horizontal coordinate (column index),
  - y: vertical coordinate (row index),
  - A cell with a value of 0 indicates free space,
  - A cell with a value of 1 indicates an obstacle.
- **Grid Boundaries:**
  - The origin $(0, 0)$ is located at the top-left corner.
  - Valid coordinates satisfy $0 \leq x <$ cols and $0 \leq y <$ rows.

B. Input Parsing

- **File Structure:** The input file contains:
  1) A section with lines beginning with `# Start goal pairs:`, followed by lines that specify start and goal positions in the format `# Start: (x,y)` Goal: (x,y)—.
  2) A section representing each row of the grid, where a row is a string of "." and "X" characters ("." = free, "X" = occupied).
- **Extraction of Start-Goal Pairs:** The function `parse_input_file(file_path)`:
  1) Reads each line and detects the `# Start:` and `Goal:` markers.
  2) Stores the start-goal pairs in a list of tuples $(\text{start}, \text{goal})$.
  3) Each coordinate is parsed as a tuple (x, y) of integers.
- **Conversion of Grid:** Lines corresponding to the map are converted into a 2D Python list `grid`, where "X" maps to 1 and "." maps to 0.

This results in:

- `start_goal_pairs`: A list of valid starting and ending coordinates.
- `grid`: A 2D list of 0s (free) and 1s (occupied).

C. Graph-Based Path Planning Methods

All grid-based methods (BFS, DFS, and A*) use 4-connected movement: left, right, up, and down. This is often referred to as a "4-connected grid."

1) Breadth-First Search (BFS)

- **Data Structures:**
  - A queue (`deque`) holding the cells to explore.
  - A 2D list `visited[y][x]` that marks whether a cell is already processed.
  - A dictionary `parent` mapping each visited cell to its predecessor for path reconstruction.
- **Algorithm:**
  1) Enqueue the start cell (x_s, y_s) and mark it *visited*.
  2) Dequeue the front cell (x_c, y_c).
  3) If (x_c, y_c) is the goal, backtrack with `parent` to form the path.
  4) Otherwise, enqueue its valid, unvisited neighbors and mark them visited.
  5) Repeat until the queue is empty or the goal is found.
- **Result:** BFS returns the *shortest path* in terms of the number of grid steps (if it exists). Otherwise, it returns an empty list.

2) Depth-First Search (DFS)

- **Data Structures:**
  - A stack (implemented via a Python list).
  - A set `visited` to track visited cells.
  - A dictionary `parent` to store each cell's predecessor.
- **Algorithm:**
  1) Push (x_s, y_s) onto the stack.
  2) Pop the top cell (x_c, y_c).
  3) If (x_c, y_c) is the goal, reconstruct the path using `parent`.
  4) Otherwise, push all valid, unvisited neighbors onto the stack, marking them visited.
  5) Continue until the stack is empty or the goal is found.
- **Result:** DFS explores one branch fully before backtracking. It may not yield the shortest path, but will find *a* valid path if one exists.

3) A* Search

- **Heuristic Function:** The Manhattan distance $|x_1 - x_2| + |y_1 - y_2|$ is used as an admissible heuristic for 4-connected movement.
- **Data Structures:**
  - A priority queue (min-heap) for the open set, where each entry is $(f_{\text{score}}, \text{position})$.
  - `g_score[position]` stores the cost from the start to `position`.
  - A dictionary `parent` for path reconstruction.
- **Algorithm:**
  1) Initialize $g_{\text{score}}(\text{start}) = 0$ and $f_{\text{score}}(\text{start}) = \text{heuristic}(\text{start}, \text{goal})$.
  2) Insert $(f_{\text{score}}(\text{start}), \text{start})$ into the priority queue.
  3) Repeatedly pop the cell with the smallest f_{score}.
  4) If the popped cell is the goal, reconstruct and return the path.
  5) Otherwise, for each valid neighbor (x_n, y_n):
     - Let tentative_g $= g_{\text{score}}(\text{current}) + 1$.
     - If tentative_g $< g_{\text{score}}(\text{neighbor})$, update:
       * $g_{\text{score}}(\text{neighbor}) = \text{tentative\_g}$
       * $f_{\text{score}}(\text{neighbor}) = \text{tentative\_g} + \text{heuristic}(\text{neighbor}, \text{goal})$
       * `parent[neighbor] = current`
     - Push the neighbor into the priority queue if it is not already in it or if a better path is found.
- **Result:** A* generally finds the shortest path more efficiently than BFS in larger grids due to its goal-directed heuristic.

D. Sampling-Based Path Planning: Rapidly-Exploring Random Trees (RRT)

- **Tree Representation:** A dictionary `tree = \{node: parent_node\}` tracks how the tree grows from the start node to new nodes.
- **Random Sampling:** Returns a random free cell in the grid; with probability `goal_bias`, it returns the goal itself, directing exploration toward the goal.
- **Nearest Node and Steer:**
  - `nearest_node(tree, random_point)` identifies the node in the tree that is closest in Euclidean distance to `random_point`.
  - `steer(from_node, to_node)` moves one step in a 4-connected manner from `from_node` toward `to_node`.
- **Collision Checking:** Interpolates cells between the two points and verifies that each intermediate cell is free (0).
- **Tree Expansion:**
  1) Sample a `random_point`.
  2) Identify the `nearest` node in the tree.
  3) Generate a `new_point` by steering one step from `nearest` to `random_point`.
  4) If `new_point` is collision-free, add it to `tree` with `tree[new_point] = nearest`.
  5) If `new_point` is the goal, stop and reconstruct the path by backtracking.
- **Result:** RRT may be less efficient than A* in a discrete grid environment, but it demonstrates how sampling-based methods scale to complex or high-dimensional spaces.

E. Performance Tracking

- **Timing:** The runtime of each algorithm is measured using

- **Memory Usage:** Python's `tracemalloc` monitors memory usage. After each algorithm finishes, the current and peak memory consumption are recorded.
- **Path Length:** The function `calculate_path_length(path)` sums the Euclidean distances between consecutive waypoints:

$$\text{length} = \sum_{i=1}^{|\text{path}|-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}.$$

F. Visualization

- `plot_grid(grid, path, start, goal)` generates a grayscale occupancy-grid plot using Matplotlib.
- Cells of value 1 appear dark, while free cells (0) appear light.
- The path is drawn with blue arrows between consecutive waypoints.
- Start and goal cells are marked in green and red, respectively.
- The path coordinates are printed in the console for diagnostics.

G. Execution Flow

1) **Map Selection and Parsing:** The main block selects a file path (e.g., `map2.txt`) and calls `parse_input_file`.
2) **Validation of Start and Goal:** For each start-goal pair, the script checks if the cells are within bounds and free.
3) **Algorithm Selection:** The user is prompted to choose BFS, A*, RRT, or DFS.
4) **Execution and Output:** The chosen algorithm runs. The path, if found, is plotted, printed, and evaluated in terms of time, memory, and path length.

This methodology ensures consistency in environment loading, algorithm execution, and performance measurement. Standardizing the grid representation, parsing strategy, and evaluation metrics simplifies comparing BFS, DFS, A*, and RRT, highlighting each algorithm's particular strengths and limitations for discrete 2D path planning.

III. Performance Comparison and Analysis

Table I summarizes the key performance metrics (execution time, memory usage, and path length) for each of the four algorithms—BFS, DFS, A*, and RRT—across three smaller maps contained in `map1.txt`. Table II then presents results for two larger maps drawn from `map3.txt`, illustrating how these algorithms scale with increased map size.

A. Results on Smaller Maps (`map1.txt`)

As shown in Table I, BFS and A* consistently find an optimal path (i.e., minimal number of steps). BFS often has very low execution times on small grids, though A* can be equally fast or faster if the heuristic effectively guides the search. DFS finds valid paths but they can be substantially longer (e.g., 38 steps vs. 18 steps in Map 1). RRT works but is comparatively slower due to random sampling overhead.

B. Results on Larger Maps (`map3.txt`)

For the two larger maps from `map3.txt` (Table II), the differences between algorithms become more pronounced:

- **BFS vs. A*:** Both again produce an optimal path (70 steps for Map 1, 41 steps for Map 2). BFS uses more memory than on the smaller maps due to expanded frontier nodes, which is evident in the peak memory usage of 0.14316 MB for Map 1, compared to 0.014216 MB in the smaller cases. A* remains efficient in terms of path quality and typically uses less memory than BFS (peak of 0.095696 MB vs. 0.14316 MB for BFS on Map 1).
- **DFS:** The path lengths illustrate the major drawback of a depth-first approach. On Map 1, DFS yields a 196-step path, significantly longer than the 70-step path produced by BFS or A*. Nevertheless, DFS can remain fast in raw execution time on some maps (0.00049 s for Map 1), but the cost is suboptimal path quality.
- **RRT:** The random sampling nature of RRT leads to much higher execution times (over 10 s on Map 1), and in fact, it failed to return a valid path for Map 1 within the allotted iterations (the length reported is 0, indicating no path was found). For Map 2, RRT required around 3.58 s and produced a 49-step path, which is less efficient than the 41-step solution by BFS and A*. This highlights that while RRT is powerful for more complex or higher-dimensional planning, it can be highly inefficient in a larger 2D grid.

C. Discussion of Observations

- **Path Optimality:** BFS and A* consistently yield the same (shortest) solutions in both small and larger 2D grid maps.
- **Execution Time:** DFS can appear faster but returns longer paths. As the grid grows, BFS and A* remain reliable, though BFS's frontier expansion can increase memory usage.
- **Memory Usage:** Across the larger maps, we see BFS holding more nodes in memory at once, compared to its usage on smaller maps. A* leverages its heuristic to guide the search more selectively. DFS tends to keep fewer nodes in memory overall but can backtrack extensively.
- **RRT:** The use of random sampling is noticeably slower for grid-based problems of increasing size, underscoring how RRT is best-suited for more complex continuous or high-dimensional scenarios rather than standard 2D grids.

Overall, for these 2D grid-based maps, A* and BFS remain the strongest candidates for consistently finding an optimal path with manageable runtime and memory. DFS is appealing for rapid path discovery in small grids but suffers from significant path-length inefficiency. RRT demonstrates the most pronounced performance penalty as map size grows and obstacles increase, reaffirming its primary benefits in continuous or high-dimensional spaces rather than large-scale discrete grids.

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

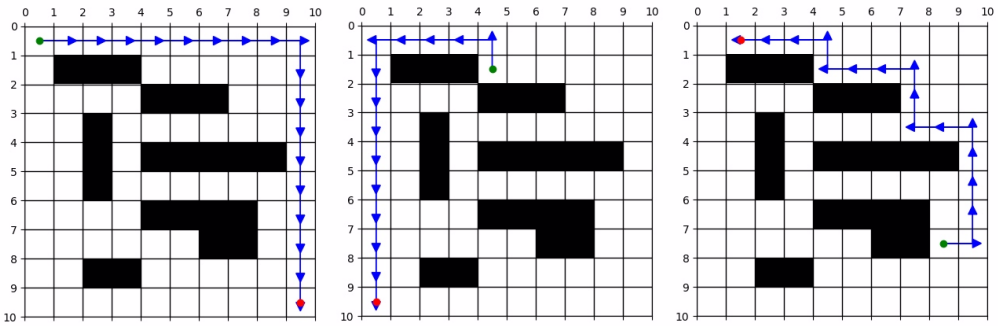

Fig. 1: BFS small map

IV. Conclusion and Future Work

A. Summary of Findings

This study evaluated four path planning algorithms—BFS, DFS, A*, and RRT—across different grid-based maps of varying sizes and complexities. The performance was analyzed based on execution time, memory usage, and path optimality. The key observations from our experiments are:

- **Optimality:** BFS and A* consistently produced the shortest path across all test cases, making them the most reliable algorithms for structured, discrete grid-based path planning.
- **Efficiency:** A* was generally more efficient than BFS in terms of execution time, especially for larger maps, due to its heuristic-guided search reducing unnecessary expansions.
- **DFS Limitations:** While DFS can quickly find a path, it often results in highly suboptimal solutions due to its depth-first nature, leading to significantly longer paths.
- **Scalability Issues in RRT:** RRT demonstrated high computational cost and inconsistency in finding valid paths in grid-based maps. As map size increased, RRT struggled significantly, confirming that it is not well-suited for structured 2D grid environments but remains advantageous for high-dimensional motion planning in continuous spaces.
- **Memory Consumption:** BFS required more memory than DFS due to its expansive search frontier, while A* often had lower memory demands than BFS due to its selective node expansion. RRT's memory footprint was not significantly high, but its execution time made it impractical for discrete 2D planning.

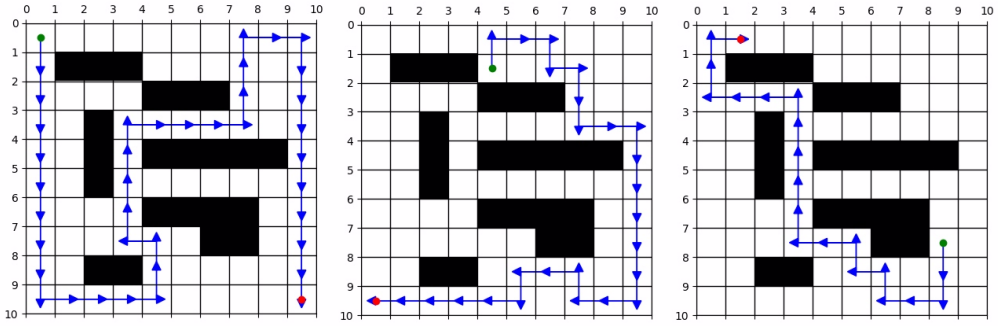

Fig. 2: DFS small map

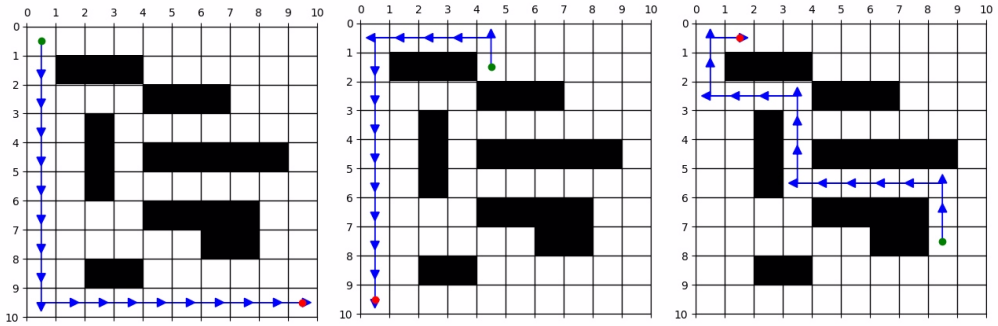

Fig. 3: A* small map

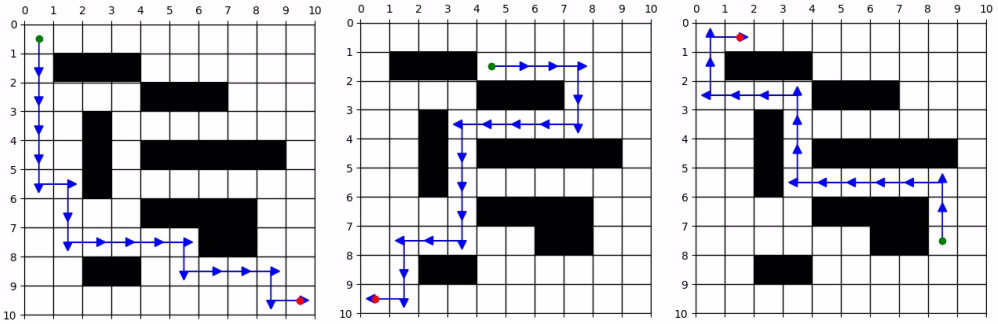

Fig. 4: RRT small map

From these results, BFS and A* emerge as the preferred algorithms for deterministic, grid-based path planning, with A* offering the best trade-off between computational efficiency and optimality.

B. Future Work

While this study provides valuable insights into classical path planning algorithms, several extensions can further improve performance and applicability:

- **Alternative Heuristics for A*:** Exploring more advanced heuristics (e.g., Euclidean distance, octile distance, or adaptive heuristics) can enhance A*'s efficiency, especially in irregular grid maps.
- **Hybrid Approaches:** Combining A* with probabilistic methods like RRT* or RRT-Connect may offer improved performance in larger, more complex environments.
- **Obstacle Inflation and Cost Maps:** Incorporating obstacle inflation (e.g., adding a safety margin around obstacles) and weighted cost maps can provide more realistic navigation behavior.

Overall, this study establishes a strong foundation for understanding the trade-offs between different classical path planning algorithms in discrete grid environments. By integrating more advanced techniques and optimizations, future research can further enhance efficiency and applicability in real-world robotic and autonomous navigation systems.

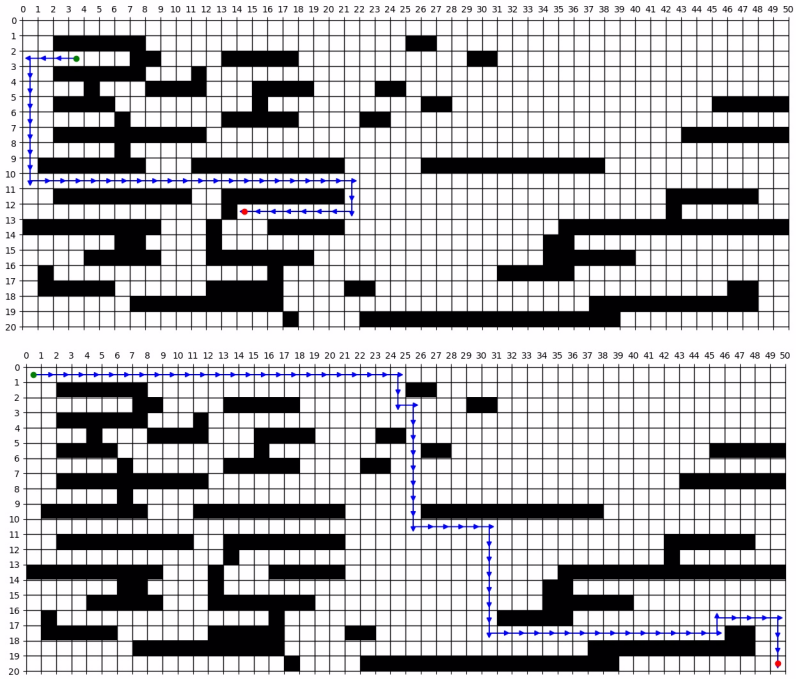

Fig. 5: BFS large map

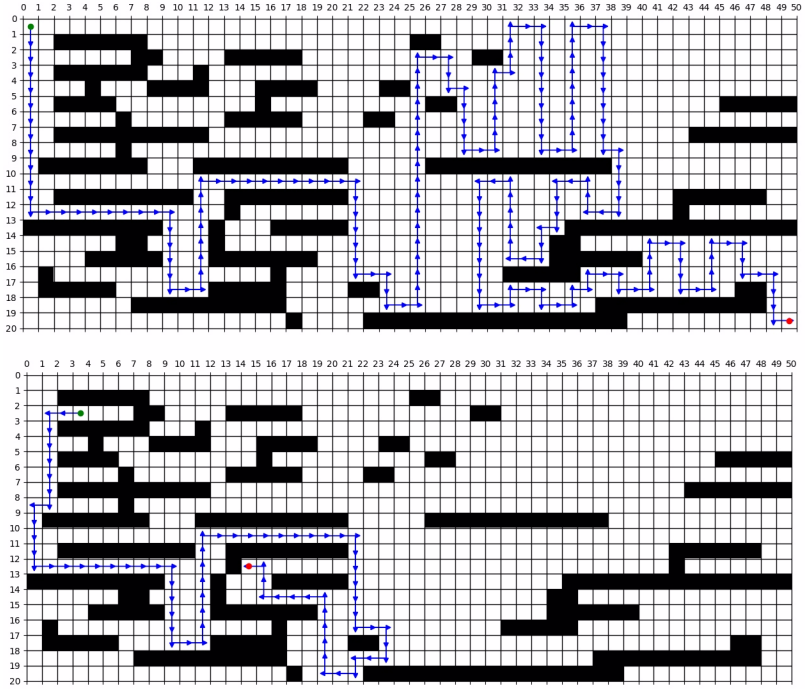

Fig. 6: DFS large map

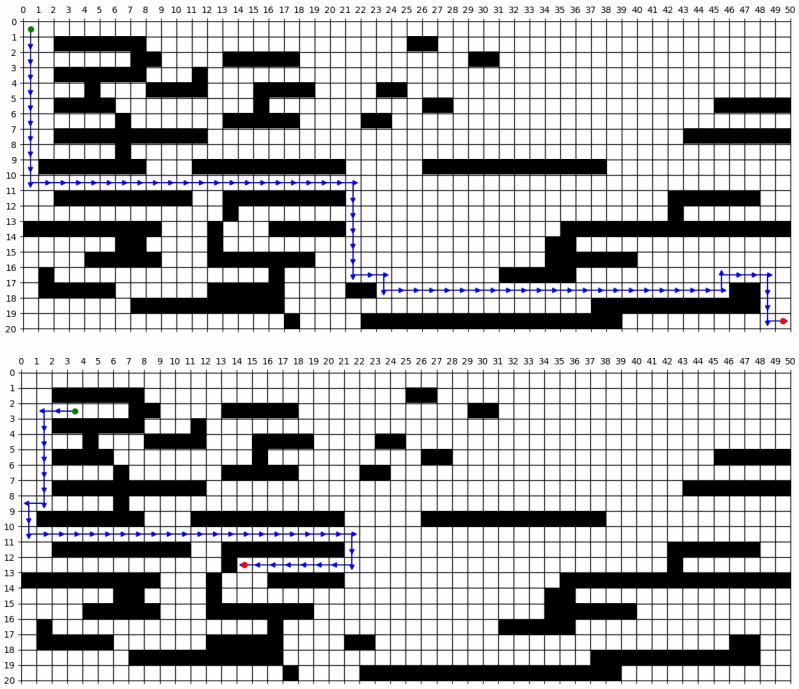

Fig. 7: A* large map

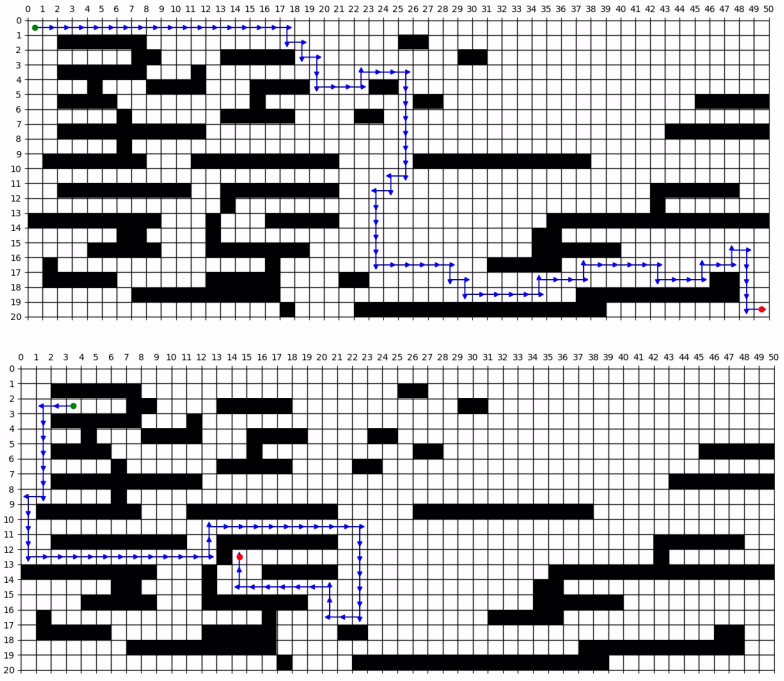

Fig. 8: RRT large map