

# A Benchmarking Framework for Grid-Based Motion Planning: BFS, Dijkstra, A\*, Theta\*, and JPS — Preliminary Evaluation in ROS 2 (Runtime, Path Quality, Memory)

Mohamed Eljahmi  
Robotics Engineering Department  
Worcester Polytechnic Institute  
100 Institute Road  
Worcester, Massachusetts 01609

**Abstract**— This project began with a narrow focus: implementing and benchmarking five classical grid-based search algorithms— Breadth-First Search (BFS), Dijkstra’s algorithm, A\*, Weighted A\*, Theta\*, and Jump Point Search (JPS). These methods remain the immediate scope, providing a well-understood testbed where trade-offs in runtime, path length, and node expansions can be compared. The framework is implemented in ROS 2 with Python, with utilities for random obstacle generation, ASCII map loading, start/goal specification, and reproducible benchmarking. At this stage, all the aforementioned algorithms are integrated, with benchmarks contrasting uniform-cost exploration against heuristic-guided search under both Manhattan and Euclidean distance heuristics. These comparisons highlight how cost models and heuristic selection affect efficiency and path quality. While the foundation is grid-based, the architecture is deliberately extensible: its unified planner interface and benchmarking harness could support future extensions to incremental graph searches for dynamic maps, and even policy-based algorithms from reinforcement learning. Manhattan is used for 4-connected grids with unit costs and Euclidean for 8-connected grids with  $\sqrt{2}$  diagonal costs to maintain heuristic admissibility and consistency.

**Keywords**— Grid-based motion planning, Breadth-First Search (BFS), Dijkstra’s algorithm, A\* algorithm, Weighted A\*, Manhattan heuristic, Euclidean heuristic, Theta\*, Jump Point Search (JPS), incremental search, policy-based planning, reinforcement learning, benchmarking framework, ROS 2.

## I. INTRODUCTION

Path planning is a fundamental problem in robotics and artificial intelligence. Mobile robots, manipulators, and autonomous agents must be able to navigate from an initial configuration to a goal while avoiding obstacles. Among the many approaches to path planning, grid-based search provides a simple yet powerful model: the workspace is discretized into cells that are either free or occupied, and algorithms explore this graph to connect the start and goal states.

Classical grid-based planners such as Breadth-First Search (BFS), Dijkstra’s algorithm, and A\* are widely studied because they guarantee optimal solutions under appropriate cost models. BFS explores in uniform steps without considering costs, while Dijkstra extends this to weighted graphs by minimizing cumulative cost. A\* further introduces heuristics, and its performance depends strongly on the heuristic function chosen—for example, Manhattan

distance is consistent with 4-connected grids, while Euclidean distance matches 8-connected grids where diagonal moves cost  $\sqrt{2}$ . More advanced methods such as Theta\* and Jump Point Search (JPS) improve efficiency by reducing node expansions or enabling any-angle paths. Each algorithm exhibits trade-offs in runtime, memory usage, and path quality, making systematic evaluation important for both education and practical deployment.

This project began with a deliberately narrow focus: to implement and benchmark these five classical grid-based planners in a consistent setting. Implemented in ROS 2 with Python, the framework provides random obstacle generation, ASCII map loading, start/goal specification, and benchmarking utilities. Algorithms can be executed under identical conditions, and metrics such as path length, number of turns, nodes expanded, runtime, and memory consumption are recorded.

## II. BACKGROUND

Grid-based path planning treats the environment as a discrete occupancy grid where each free cell corresponds to a vertex in a graph and edges connect neighboring cells. Search algorithms then explore this graph to connect a start cell and a goal cell. Several classical algorithms are widely studied:

### A. Breadth-First Search (BFS)

BFS explores the grid in layers outward from the start, guaranteeing the shortest path in terms of number of steps when all moves have equal cost. Its drawback is high memory usage, since all frontier nodes are stored.

### B. Dijkstra’s Algorithm

Dijkstra extends BFS to weighted graphs by maintaining a priority queue over cumulative path cost. It guarantees an optimal path in graphs with positive edge weights but tends to expand many nodes because it does not use heuristics.

### C. A\* Algorithm

A\* improves on Dijkstra by adding a heuristic estimate of cost-to-go, enabling the search to focus on states that appear closer to the goal. Its effectiveness depends on both the cost model and the heuristic function. For 4-connected grids with unit step costs, the Manhattan distance is admissible and consistent. For 8-connected grids, where diagonal moves are allowed with cost  $\sqrt{2}$ , the Euclidean distance provides a more accurate admissible heuristic. These variations highlight that A\* is not a single method but a family of

approaches, and the choice of heuristic directly influences runtime, node expansions, and overall efficiency while still preserving optimality.

#### D. Theta\* Algorithm

Theta\* is a variation of A\* that allows any-angle shortcuts rather than restricting paths to grid edges. This reduces unnecessary turns and often produces shorter, more natural-looking paths, while maintaining efficiency.

#### E. Jump Point Search (JPS)

JPS is an optimization for uniform-cost grids that skips over symmetric intermediate states. By “jumping” directly to critical nodes, JPS dramatically reduces the number of expansions while preserving optimality.

#### F. Beyond Classical Grid Search

Although the immediate scope of this project is restricted to the five grid-based planners above, robotics often requires planning in environments that are partially known or dynamic. Incremental algorithms such as D\* and D\* Lite update solutions efficiently when obstacles are discovered, while reinforcement learning methods such as Dyna-Q maintain policy-based decision structures through interaction with the environment. These approaches are not implemented here, but the framework is designed to be flexible and could support them in future work. Evaluation of all these algorithms typically considers metrics such as:

- Path length (or cost)
- Number of turns
- Runtime performance
- Nodes expanded
- Memory consumption of open/closed lists

### III. METHODOLOGY

The goal of this project is not only to implement individual planning algorithms, but to develop a consistent and extensible benchmarking framework. This section describes the design choices, software environment, and evaluation setup.

#### A. Software Environment

The framework is implemented in **ROS 2 Humble** using **Python 3.10**. A custom ROS 2 package, *rbe550\_grid\_bench*, provides the benchmarking utilities and planner implementations. The build system is based on **colcon**, and helper scripts (*build.sh* and *run.sh*) simplify compilation, environment setup, and experiment execution.

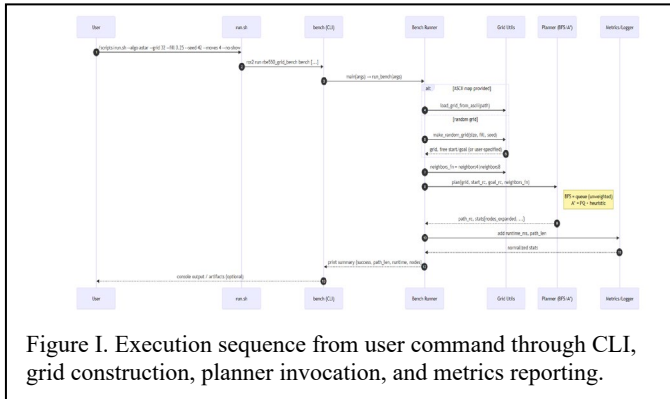


Figure I. Execution sequence from user command through CLI, grid construction, planner invocation, and metrics reporting.

#### B. Grid Representation

The environment is modeled as a 2D occupancy grid. Each cell is marked as either free or occupied, and planners operate on the induced grid graph. The framework supports:

- **Random grids**, generated with a target fill percentage of obstacles. A random seed is specified for reproducibility.
- **ASCII map files**, where obstacles and free spaces are explicitly encoded.

Start and goal cells are either randomly selected from free cells or specified by the user. To ensure validity, both start and goal are forced to free space.

#### C. Planner Interface

To maintain consistency, each algorithm is wrapped in a common Python interface:

```
def plan(grid, start, goal, neighbors_fn) -> (path, stats)
where:
```

- *grid* is a binary occupancy matrix.
- *start/goal* are (row, col) coordinates
- *neighbors\_fn* selects 4- or 8-connected motion.

Each planner returns a path (list of grid cells) and a stats dictionary (runtime, nodes expanded, path length in steps; memory/turns where available).

#### D. Implemented Algorithms

At this stage, all algorithms are now integrated. BFS is implemented with a queue-based frontier and uniform step costs. A\* uses a priority queue guided by admissible heuristics, with support

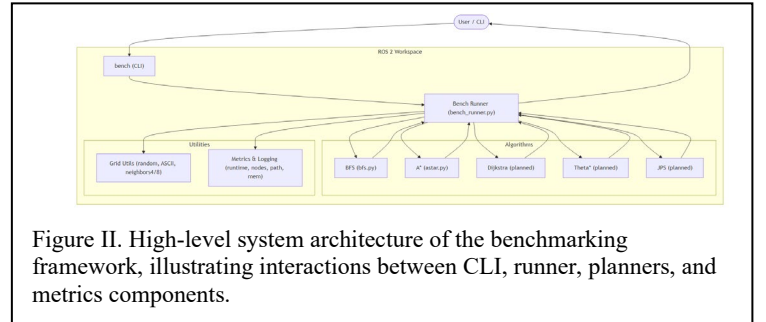
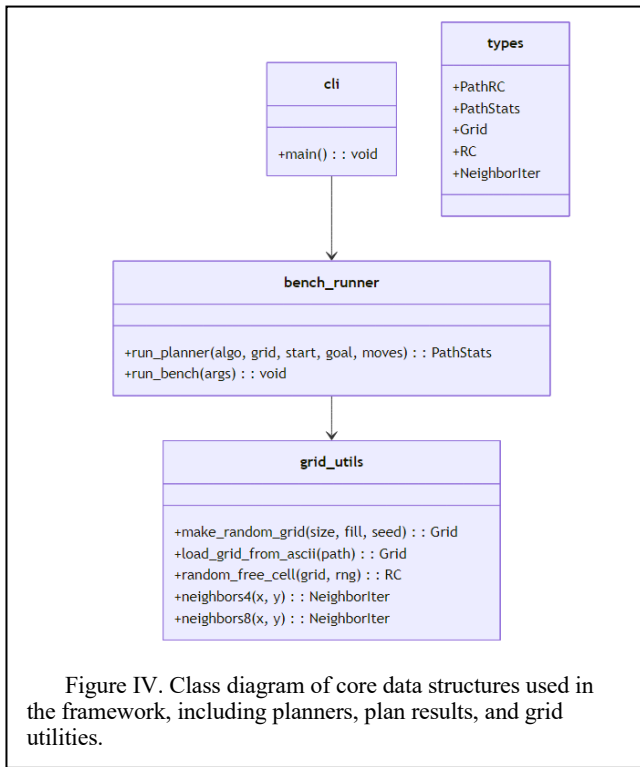
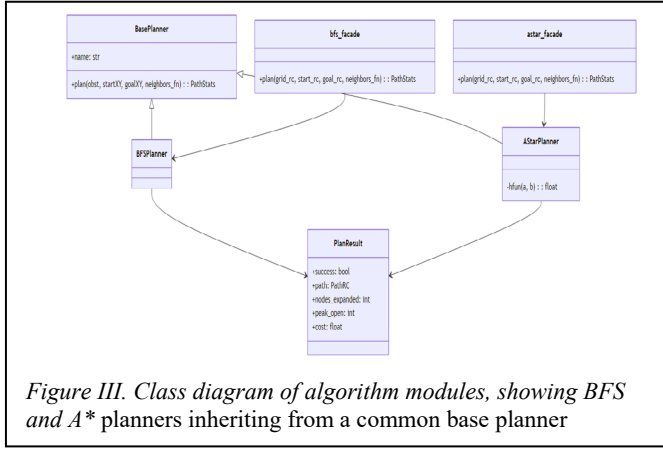


Figure II. High-level system architecture of the benchmarking framework, illustrating interactions between CLI, runner, planners, and metrics components.

for both Manhattan distance (appropriate for 4-connected grids with unit costs) and Euclidean distance (appropriate for 8-connected grids with diagonal moves costing  $\sqrt{2}$ ). This allows benchmarking of multiple A\* variants to study how heuristic selection influences efficiency and path quality. Additional algorithms—Dijkstra, Theta\*, and Jump Point Search (JPS)—will be incorporated using the same interface to enable fair comparisons.



### E. Benchmarking Harness

The benchmarking harness executes planners on specified grids and records standardized metrics:

- Runtime measured in milliseconds
- Nodes expanded during search
- Path length (number of steps)
- Number of turns (optional, for later integration)
- Memory usage of open and closed lists

Each run reports results in a structured format, printed to the console and optionally saved for later analysis. The harness ensures that all planners are evaluated under identical conditions. In particular, it supports varying the cost model (4-connected vs. 8-connected grids) and heuristic functions (Manhattan vs. Euclidean) so that differences

between algorithmic approaches can be measured systematically. The harness systematically varies both the cost model (4 vs 8 connectivity) and the heuristic (Manhattan vs Euclidean) to measure their impact on efficiency and path quality.

### F. Reproducibility and Scripts

Experiments are launched via `run.sh`, which sources the ROS 2 environment, builds the workspace, and invokes `ros2 run rbe550_grid_bench` with user-specified parameters. Command-line options allow configuration of grid size, fill percentage, seed, start/goal, algorithm selection, and connectivity. This design enables reproducible benchmarking across random and structured maps.

### G. Packaging and submission

The complete ROS 2 workspace RBE550-Workspace is hosted on GitHub and packaged for fully reproducible runs through Docker. The repository includes:

- `src/rbe550_grid_bench/` — core ROS 2 package (BFS and A\* implementations).
- `maps/` — ASCII map files (e.g., `maze_32.txt`).
- `scripts/` — helper scripts for build and execution, both local and Docker.
- `Dockerfile` and `.dockerignore` — define a self-contained, reproducible build.
- `README.md` — detailed usage instructions.

1. Reproducibility via Docker (Recommended): Docker ensures identical behavior across platforms without requiring a local ROS 2 installation. From any Linux or Windows system with Docker installed:
 

```
git clone git@github.com:meljahmi-personal/RBE550-Workspace.git
cd RBE550-Workspace
./scripts/run_docker.sh --steps 10 --render-every 2 --no-show
```

The last command builds the image, copies both `src/` and `maps/` into `/ws/`, compiles the workspace with `colcon`, and runs the benchmark. Outputs (logs, images, and GIFs) are saved to `./outputs/` on the host.

2. Local Build (Optional, ROS 2 Humble required) Users with ROS 2 installed can alternatively build and run locally:

```
git clone git@github.com:meljahmi-personal/RBE550-Workspace.git
cd RBE550-Workspace
./scripts/build.sh
./scripts/run.sh --grid 64 --fill 0.20 --seed 42 --algo astar --moves 8 --no-show
```

3. Submission and Exclusions:

The submission excludes generated artifacts to keep the repository lightweight:

```
build/
install/
log/
outputs/
```

Only source code, maps, scripts, and configuration files are included.

These components are sufficient for anyone with Docker and SSH access to reproduce the reported results exactly.

4. Project Documentation Repository:

In addition to the code workspace, all written project documents — including the proposal, status reports, and presentation materials — are hosted in a separate GitHub repository for transparency and version control: [git@github.com:meljahmi-personal/RBE550-project-proposal.git](https://github.com/meljahmi-personal/RBE550-project-proposal.git)

#### IV. RESULTS AND ANALYSIS

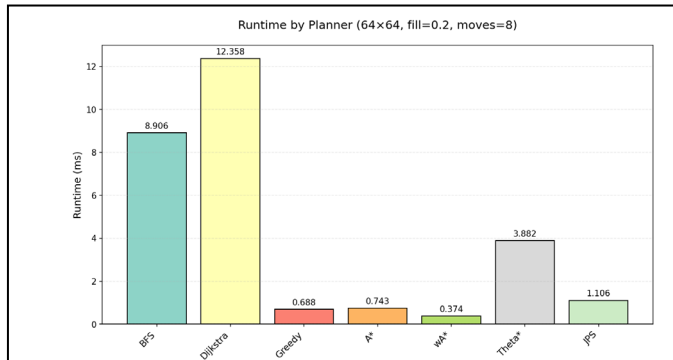
A comparative benchmark was conducted on a uniform  $64 \times 64$  grid with an obstacle density of 20% (fill=0.2) and 8-connected movement. All planners were tested on an identical problem instance (seed=42) with start (5,53) and goal (49,34) coordinates. The results, summarized in Table I and visualized in Figures. V–X, evaluate performance across runtime, path length, path smoothness, and search efficiency.

**Table I.** Benchmark results summary ( $64 \times 64$ , Obstacle Fraction=0.2061)

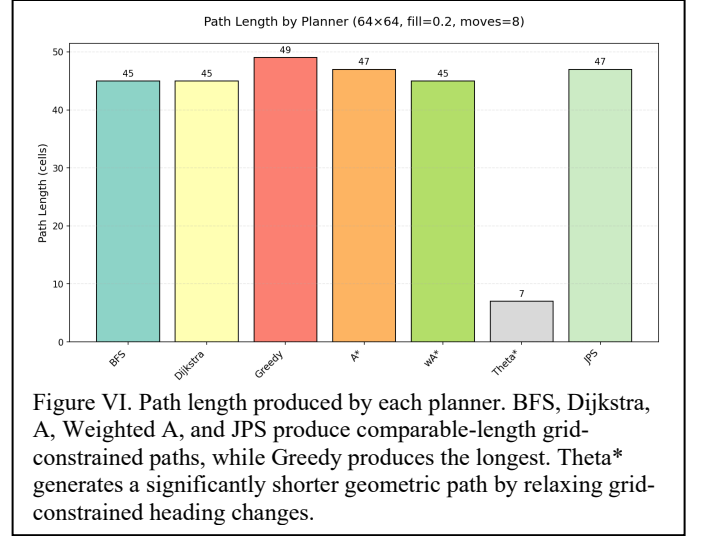
Algorithm	Path Length	Runtime (ms)	Nodes Expanded	Turns
BFS	45	8.906	2098	13
Dijkstra	45	12.358	2110	13
Greedy	49	0.688	49	13
A*	47	0.743	97	14
Weighted A*	45	0.374	45	1
Theta*	7	3.882	192	5
JPS	47	1.106	97	13

##### A. Runtime Performance

Runtime performance varied significantly across planners, as depicted in Figures V. and VI. Dijkstra's algorithm was the slowest (12.36 ms), followed closely by BFS (8.91 ms), due to their exhaustive, uninformed search strategies. In contrast, heuristic-based planners demonstrated markedly superior speed. Greedy search was the fastest (0.688 ms), followed by Weighted A\* (0.374 ms) and A\* (0.743 ms). Jump Point Search (JPS) achieved a runtime of 1.106 ms, improving upon A\* by skipping redundant nodes. Theta\* incurred the highest cost among informed planners (3.882 ms), a trade-off for its any-angle path capability.



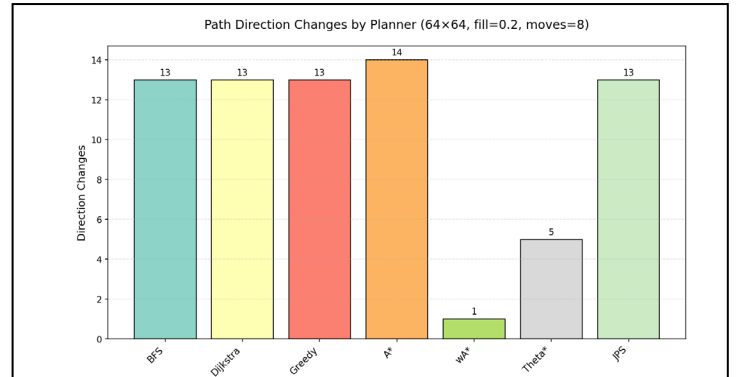
**Figure V.** Runtime performance of all planners on a  $64 \times 64$  grid with 20% obstacle density and 8-connected movement. Weighted A, A, Greedy, and JPS achieve sub-millisecond execution, while BFS and Dijkstra exhibit the highest latency due to exhaustive exploration.



**Figure VI.** Path length produced by each planner. BFS, Dijkstra, A, Weighted A, and JPS produce comparable-length grid-constrained paths, while Greedy produces the longest. Theta\* generates a significantly shorter geometric path by relaxing grid-constrained heading changes.

##### B. Path Length and Smoothness

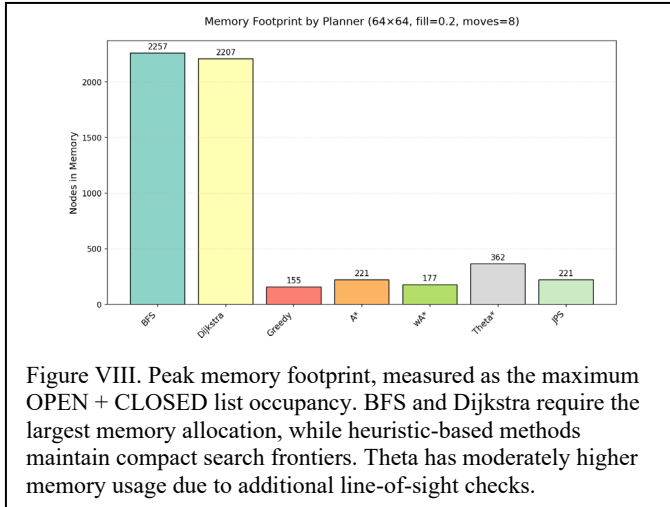
Path optimality, measured in cell count, and smoothness, measured by the number of direction changes (turns), are critical for robotic traversal. As shown in Table I and Figure VII, Theta\* produced the shortest path by a substantial margin (length=7), leveraging line-of-sight smoothing to cut across grid cells. It also generated a smoother path with only 5 turns. BFS, Dijkstra, and Weighted A\* all found an optimal grid-aligned path of length 45, though Weighted A\* achieved this with a perfectly straight path (1 turn), demonstrating effective heuristic weighting. A\* and JPS found a near-optimal path of length 47. Greedy search produced the longest path (49) among the informed searchers, a consequence of its heuristic myopia.



**Figure VII.** Number of direction changes (turns) along the produced paths. A, BFS, Dijkstra, and JPS produce highly discretized paths with many turns, whereas Theta produces smoother paths with far fewer heading changes. Weighted A\* yields the fewest turns (1), producing a highly directed trajectory.

##### C. Search efficiency and memory footprint

Search efficiency, measured by nodes expanded, and memory footprint, measured by peak nodes stored, are directly correlated and shown in **Figure. VIII & IX.** Uninformed algorithms (BFS, Dijkstra) expanded and stored over 2,000 nodes, reflecting exhaustive exploration. In stark contrast, heuristic-based methods drastically reduced this burden. Greedy and Weighted A\* were the



most efficient, expanding only 49 and 45 nodes respectively. A\* and JPS expanded 97 nodes, demonstrating the balanced pruning of a consistent heuristic. Theta\* expanded 192 nodes due to the overhead of line-of-sight checks. This visualizes the critical role of the heuristic: it compresses the search space, reducing both computational effort and memory consumption by orders of magnitude.

#### D. Path analysis

*F. Search Efficiency*

Node expansion directly correlates with memory usage and computational effort. Greedy and Weighted A\* were the most efficient, expanding only 49 and 45 nodes, respectively. A\* and JPS expanded 97 nodes, while Theta\* expanded 192. The uninformed algorithms (BFS, Dijkstra) expanded over 2000 nodes, underscoring the critical role of the heuristic in pruning the search space. The results confirm that a well-tuned heuristic, as in Weighted A\*, can yield optimal paths with search efficiency rivaling that of greedy approaches.

The analysis demonstrates a clear multi-objective trade-off:

- **For minimum path length and smoothness**, Theta\* is superior, despite its higher runtime.
- **For optimal grid-aligned paths with minimal runtime and memory**, Weighted A\* is the most efficient, achieving the same path length as BFS/Dijkstra orders of magnitude faster.
- **For fastest feasible solution**, Greedy search is adequate, though path quality suffers.
- **For classic completeness with no heuristic**, Dijkstra and BFS guarantee optimality at a severe computational cost. JPS provides a practical speed-up over A\* in uniform grids, while A\* itself remains a robust, balanced baseline.

## V. CHALLENGES

### A. Scope and time constraints

The project expanded from basic planner implementation to include benchmarking, visualization, and multi-interface execution within a short development window. Prioritizing features through a structured task hierarchy ensured that core planning and RViz visualization were completed first, enabling timely delivery of all required components.

### B. System integration issues

- **GPU and OpenGL compatibility:**  
RViz initially failed to launch due to mismatched NVIDIA driver and NVML versions, which prevented OpenGL context creation. A complete driver removal followed by a clean installation resolved the incompatibility and restored stable rendering.
- **ROS2 Package Configuration:**  
Launch execution errors occurred when executables (e.g., `bench`, `planner_node`) were not properly registered in `setup.py`. Correcting the console script definitions and rebuilding the workspace ensured that ROS 2 recognized all installed nodes.

### C. Visualization Pipeline:

- **Occupancy Grid Alignment:**  
Despite correct data publication, RViz displayed an empty grid due to non-centered origin coordinates and invalid timestamps in the message header. Adjusting the origin and publishing valid ROS timestamps resolved the visualization discrepancy.
- **Planner Interface Consistency:**  
Several planners reported errors arising from mismatched neighbor-function signatures. A unified neighbor adapter was introduced to enforce consistent output formatting, enabling all algorithms to operate correctly under both 4- and 8-connected motion models.

### D. Development Interruption

A brief illness caused loss of development time during system integration. Modular code organization and detailed version control practices enabled efficient resumption of work without significant schedule impact.

### E. QoS Configuration Challenges

Intermittent “No map received” warnings were traced to inappropriate QoS durability settings for visualization topics. Refining the QoS profile to use reliable delivery ensured consistent RViz updates.

## VI. CONCLUSION

This project presents an extensible benchmarking framework for grid-based motion planning, developed in ROS 2 with Python. The framework provides consistent interfaces, reproducible experiment setup, and metrics reporting across multiple planners. Early experiments have confirmed the functionality of the system and

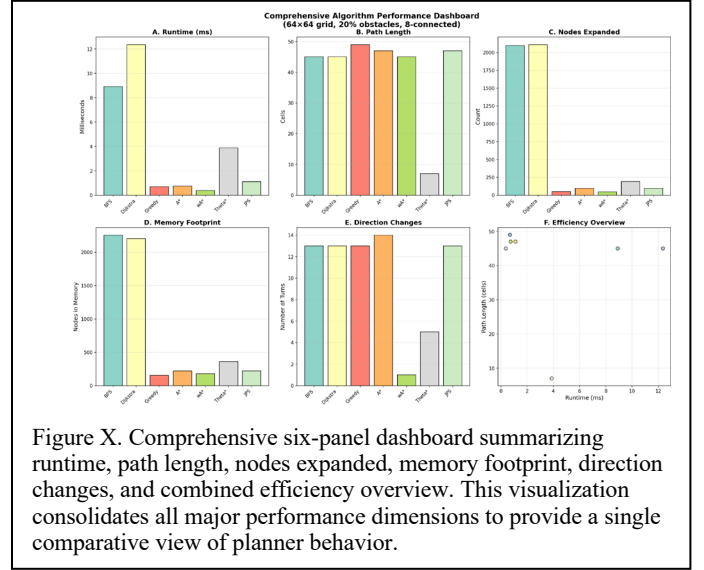


Figure X. Comprehensive six-panel dashboard summarizing runtime, path length, nodes expanded, memory footprint, direction changes, and combined efficiency overview. This visualization consolidates all major performance dimensions to provide a single comparative view of planner behavior.

demonstrated how cost models and heuristic choices influence search efficiency and path quality.

Although only a subset of the planned algorithms has been integrated so far, the foundation is in place for systematic evaluation of Breadth-First Search, Dijkstra’s algorithm, A\*, Theta\*, and Jump Point Search. The framework is deliberately designed to support additional planners, richer metrics, and automated analysis. Future iterations will expand both the algorithm set and the benchmarking scope, ultimately enabling a comprehensive study of classical search methods and their trade-offs in runtime, path quality, and memory usage.

## REFERENCES

- [1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.
- [2] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [3] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. Switching Theory*, 1959, pp. 285–292.
- [4] D. Harabor and A. Grastien, “Online graph pruning for pathfinding on grid maps,” in *Proc. AAAI*, 2011.
- [5] R. A. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of A\*,” *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985.
- [6] S. Koenig and M. Likhachev, “D\* Lite,” in *Proc. AAAI*, 2002.
- [7] S. M. LaValle, *Planning Algorithms*. Cambridge Univ. Press, 2006.
- [8] B. P. Gerkey, R. T. Vaughan, and A. Howard, “The Player/Stage Project: Tools for multi-robot and distributed sensor systems,” in *Proc. Int. Conf. Advanced Robotics (ICAR)*, 2003, pp. 317–323.
- [9] M. Quigley et al., “ROS: an open-source Robot Operating System.” In *Proc. ICRA Workshop on Open Source Software*, 2009.