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 √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 √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 motion planning models the environment as a discrete occupancy grid, where each free cell represents a graph vertex and edges connect neighboring cells under a defined motion model (4- or 8-connected). Classical search algorithms traverse this graph to construct feasible paths between specified start and goal cells. Their behavior differs significantly in terms of optimality, runtime, memory usage, and path quality, making them well-suited for systematic comparison in a benchmarking framework.

A. Breadth-First Search (BFS)

BFS explores all states in increasing order of depth, guaranteeing the shortest path **in number of steps** when all edge costs are uniform. Its primary limitation is memory demand: BFS must maintain large frontier layers, especially in open environments, leading to substantial OPEN and CLOSED list sizes. Despite its inefficiency, BFS serves as an important **baseline for completeness and optimality** under uniform-cost assumptions.

B. Dijkstra's Algorithm

Dijkstra generalizes BFS to **weighted** graphs by selecting expansions based on cumulative cost $g(n)$. It guarantees optimality for positive edge weights but expands many nodes because it lacks heuristic guidance. In grid-based domains with unit and diagonal costs, Dijkstra provides a reference point for evaluating how heuristics improve efficiency.

C. A* Algorithm

A* augments Dijkstra's cost model with a heuristic term $h(n)$, enabling goal-directed exploration. When the heuristic is **admissible and consistent**, A* preserves optimality while dramatically reducing node expansions.

Two standard heuristics are used depending on grid connectivity:

- **Manhattan distance** for 4-connected grids (unit step costs)
- **Euclidean distance** for 8-connected grids (diagonal cost $\sqrt{2}$
-)

These choices maintain admissibility and consistency, ensuring that performance differences arise from algorithmic behavior rather than heuristic mismatch. As reflected in the benchmark, A* typically achieves a balance between **speed** and **path quality**.

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. Weighted A*

Weighted A* modifies the A* evaluation function to
$f(n) = g(n) + w \cdot h(n)$ where $w > 1$.
This "heuristic inflation" biases the search toward states closer to the goal, reducing runtime and memory footprint at the expense of optimality. Weighted A* is often used when rapid computation is prioritized over strict optimality.

G. Broader Algorithmic Context

While the present work focuses on classical full-search planners, robotics applications often require adaptability to dynamic or partially known environments. Incremental and anytime planners—including **Lifelong Planning A*** (LPA*), **D* Lite**, and **Field D***—update solutions efficiently as the map changes. At a higher abstraction level, **reinforcement learning** methods learn policies rather than explicit paths. Although these approaches remain outside the scope of this implementation, the benchmarking framework was designed to support them through its unified planner interface and standardized metrics reporting.

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

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

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.

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

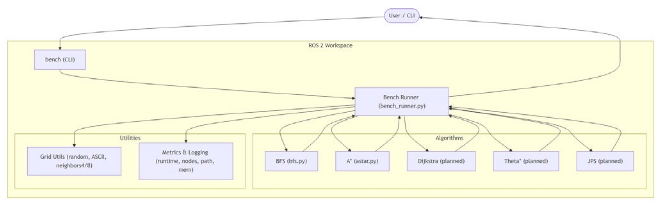

Figure II. High-level system architecture of the benchmarking framework, illustrating interactions between CLI, runner, planners, and metrics components.

for both Manhattan distance (appropriate for 4-connected grids with unit costs) and Euclidean distance (appropriate for 8-connected grids with diagonal moves costing √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.

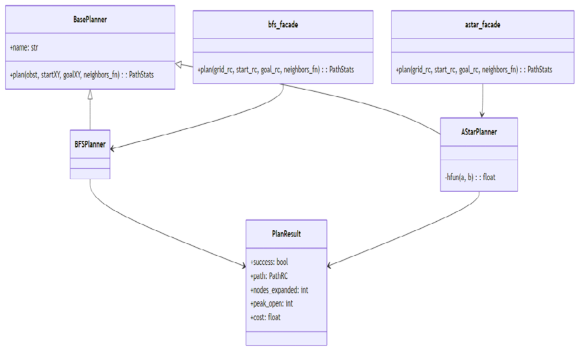
BasePlanner
+name: str
+plan(obst, startXY, goalXY, neighbors_fn) : : PathStats
bfs_facade
+plan(grid_rc, start_rc, goal_rc, neighbors_fn) : : PathStats
astar_facade
+plan(grid_rc, start_rc, goal_rc, neighbors_fn) : : PathStats
BFSPlanner
AStarPlanner
-hfun(a, b) : : float
PlanResult
+success: bool
+path: PathRC
+nodes_expanded: int
+peak_open: int
+cost: float

*Figure III. Class diagram of algorithm modules, showing BFS and A** planners inheriting from a common base planner

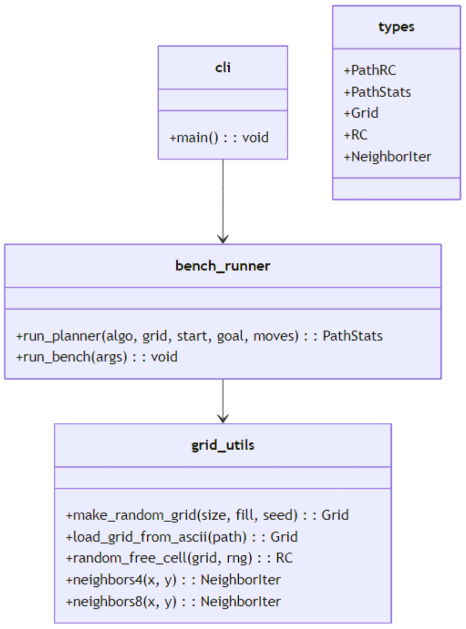
types
+PathRC
+PathStats
+Grid
+RC
+NeighborIter
cli
+main() : : void
bench_runner
+run_planner(algo, grid, start, goal, moves) : : PathStats
+run_bench(args) : : void
grid_utils
+make_random_grid(size, fill, seed) : : Grid
+load_grid_from_ascii(path) : : Grid
+random_free_cell(grid, rng) : : RC
+neighbors4(x, y) : : NeighborIter
+neighbors8(x, y) : : NeighborIter

Figure IV. Class diagram of core data structures used in the framework, including planners, plan results, and grid utilities.

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 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

IV. Results And Analysis

A comparative benchmark was conducted on a uniform 64×64 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 (64×64, 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 measurements show a clear distinction between uninformed and heuristic search. BFS and Dijkstra produced the highest runtimes (approximately 8.9 ms and 12.3 ms), reflecting their exhaustive exploration. In contrast, Greedy Best-First Search, Weighted A*, A*, and JPS completed within 0.4–1.1 ms. Theta* required slightly more time (≈3.9 ms), primarily due to line-of-sight checks. Overall, uristic-driven planners achieve substantially lower runtimes.

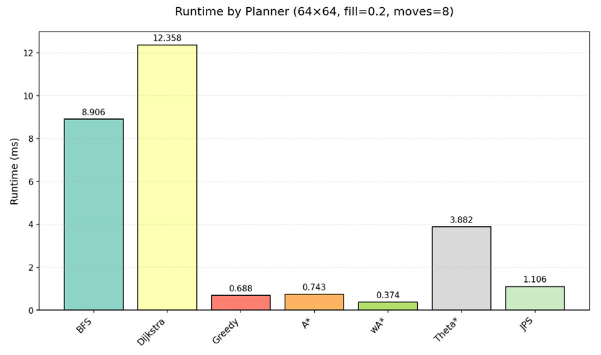

Figure V. Runtime comparison for all planners on a 64×64 grid (20% obstacles, 8-connected).

B. Nodes Expanded

The number of expanded nodes follows the classical expectations of search efficiency. BFS and Dijkstra each expanded more than 2,000 nodes, while Greedy, A*, Weighted A*, and JPS expanded between 45 and 97 nodes. Theta* expanded more than the other heuristics (≈192) because visibility checking may cause additional neighbor evaluations. These results illustrate the benefit of heuristics in pruning unpromising regions of the search space.

C. Memory Footprint

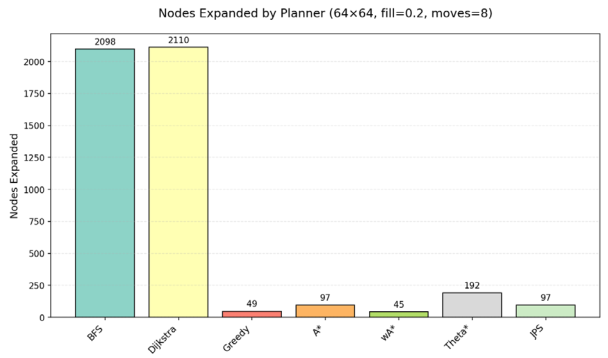

Figure VI. *Number of nodes expanded by each planner.*

Memory footprint correlates strongly with the maximum size of the OPEN and CLOSED lists. BFS and Dijkstra exhibit the highest memory usage (over 2,200 nodes), consistent with their large frontier sizes. Greedy, A*, Weighted A*, and JPS require substantially less memory (≈150–220 nodes). Theta* uses more memory than standard A* because of geometric visibility operations but remains far below the uninformed methods.

D. Path Length

Path length varies significantly across planners.

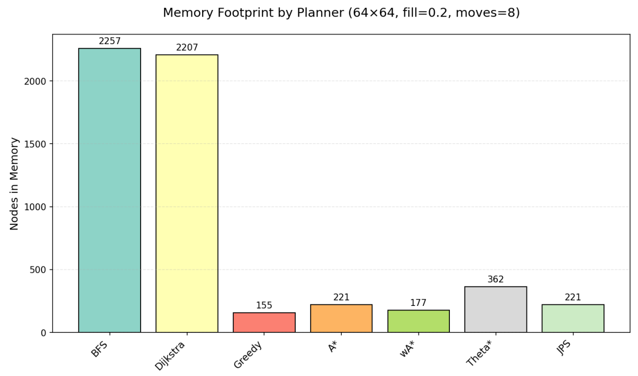

Figure VII. Memory footprint of each planner (peak OPEN + CLOSED list size).

Theta* produced the shortest path (7 cells), consistent with its any-angle formulation that reduces Manhattan-style detours. BFS, Dijkstra, and Greedy generated paths of approximately 45–49 cells; A* and JPS produced slightly shorter paths around 47 cells. Weighted A* produced a path similar in length to BFS and Dijkstra due to its heuristic inflation favoring speed over optimality.

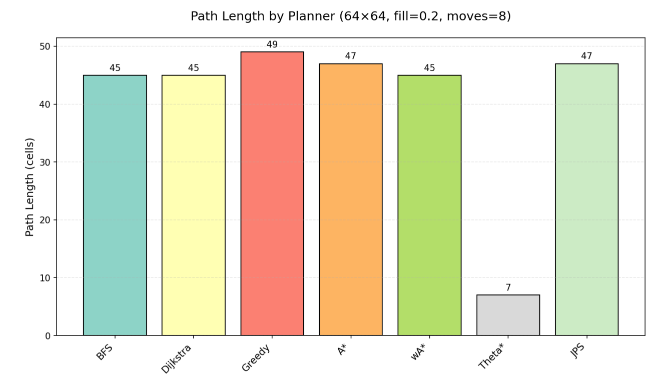

Figure VIII. Resulting path length produced by each planner.

E. Direction Changes

Direction changes serve as a proxy for path smoothness. Weighted A* achieved the smoothest path with only one direction change. Theta* also demonstrated low curvature (5 changes), reflecting its ability to generate continuous, low-turn trajectories. BFS, Dijkstra, Greedy, A*, and JPS exhibited 13–14 changes, which is typical for grid-based discrete planners constrained to 8-connected motion.

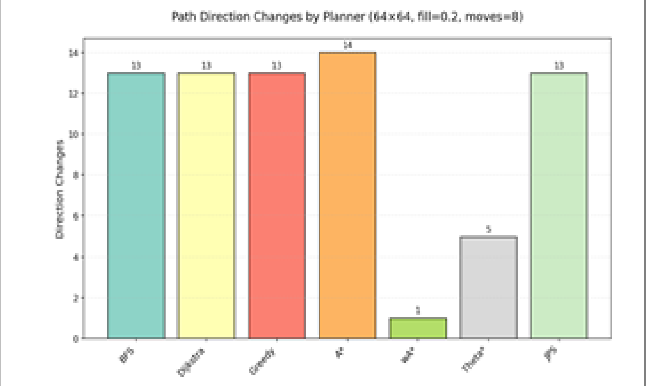

Figure IX. Path smoothness measured by discrete direction changes.

F. Combined Performance Profile

The efficiency scatter plot highlights the trade-offs among algorithms:

- **Theta*** lies on the optimal-quality frontier, providing the shortest and smoothest path at moderate computational cost.
- **Weighted A*** lies on the optimal-speed frontier, achieving the fastest runtime and lowest memory usage while producing reasonable path quality.
- **Greedy and JPS** deliver fast execution but produce longer, more jagged paths.
- **BFS and Dijkstra** serve as completeness baselines but scale poorly in both time and memory.

G. Combined Performance Profile

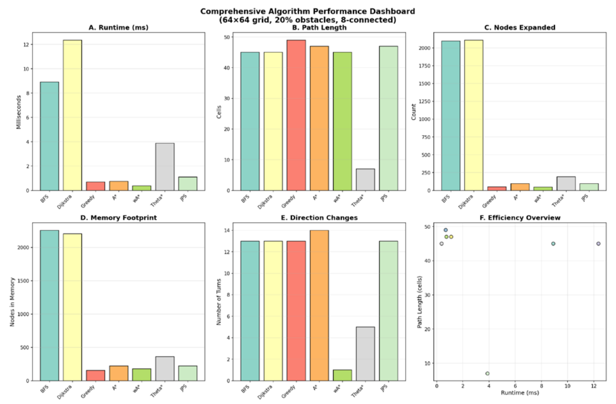

Figure X. Comprehensive six-panel dashboard summarizing runtime, path length, nodes expanded, memory footprint, direction changes, and combined efficiency overview. This visualization consolidates all major performance dimensions to provide a single comparative view of planner behavior.

H. Summary

Across all metrics, Theta* provides the highest path quality, while Weighted A* and JPS offer the best computational efficiency. Traditional uninformed search methods remain significantly more expensive and are unsuitable for real-time navigation in larger grids.

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. Vizualization 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 developed a ROS 2–based benchmarking framework for evaluating grid-based search algorithms under a unified and reproducible experimental pipeline. The system provides standardized planner interfaces, deterministic world generation, and automated reporting of runtime, memory consumption, and path-quality metrics. These capabilities enable transparent comparison across classical algorithms.

The experimental results demonstrate that different planners occupy distinct performance regimes when their behavior is viewed jointly through the **Efficiency Scatter Diagram**, which plots runtime, path length, and nodes expanded in a single visualization (Figure X, page 6 of the report
). This representation highlights interactions that single-metric bar charts cannot capture and provides a more complete understanding of search behavior:

- **Greedy Best-First, A*, Weighted A*, and JPS** cluster in the *fast–long* region: they achieve very low runtimes and small search effort, but their paths are typically longer or contain more discrete directional changes.
- **BFS and Dijkstra** appear in the *slow–long* region, reflecting the exhaustive nature of uninformed uniform-cost exploration. Their runtime, memory footprint, and node expansions are an order of magnitude larger than the heuristic methods.
- **Theta*** uniquely occupies the *fast–short* region, producing both the shortest and smoothest trajectory while maintaining moderate runtime. Its any-angle relaxation allows it to avoid grid-constrained detours that all other planners must follow.
- As expected, no planner falls into a *slow–short* quadrant, since algorithms that limit search or apply strong heuristics tend to sacrifice optimality rather than improve it.

These outcomes confirm that the framework successfully distinguishes planners along multiple performance dimensions and reveals the structural trade-offs inherent in discrete search: uninformed completeness versus heuristic efficiency, grid-constrained optimality versus geometric smoothness, and algorithmic speed versus path quality.

Although the evaluation centered on a set of classical grid-based planners, the architecture is intentionally extensible. The same interface could incorporate incremental algorithms (e.g., LPA*, D*, D*-Lite), any-angle extensions (e.g., Field D*), hierarchical planners, or anytime variants. Additional metrics—such as curvature, replanning latency, and dynamic-obstacle response—can also be introduced with minimal modification.

Overall, the project establishes a robust foundation for systematic empirical study of motion-planning algorithms. With further algorithm integration and broader experimental design, the framework can support deeper insights into search efficiency, path optimality, and memory performance in autonomous navigation systems.

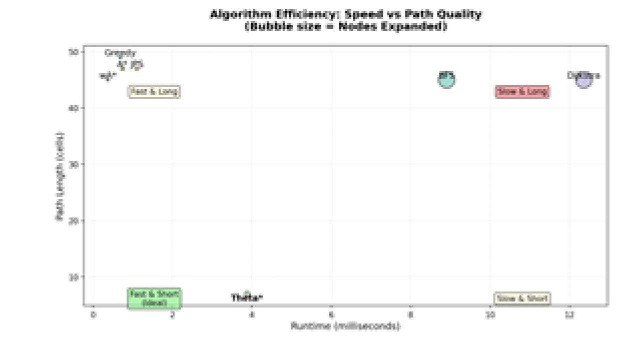

Figure XI. Speed–quality efficiency plot (bubble size ∝ nodes expanded)

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