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*, 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, BFS and A* 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, 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.

As of October 6, the framework includes initial implementations of BFS and A*, enabling preliminary results on random and structured maps. Dijkstra, Theta*, and JPS will be added in subsequent iterations. While the near-term scope is restricted to grid-based graph searches, the architecture is flexible and extensible: the same benchmarking structure could, in future work, support incremental planners for dynamic maps or even policy-based approaches from reinforcement learning.

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 √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.

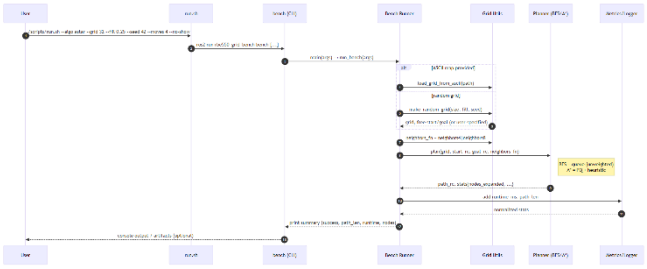

Figure I. Execution sequence from user command through CLI, grid construction, planner invocation, and metrics reporting.

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).

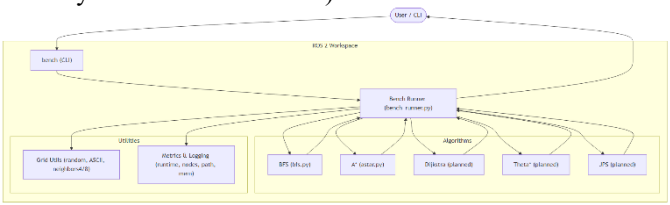

Figure II. High-level system architecture of the benchmarking framework, illustrating interactions between CLI, runner, planners, and metrics components.

D. Implemented Algorithms

At this stage, Breadth-First Search (BFS) and A* are being integrated as baseline algorithms. BFS is implemented with a queue-based frontier and uniform step costs. A* uses a priority queue guided by admissible heuristics, with support 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.

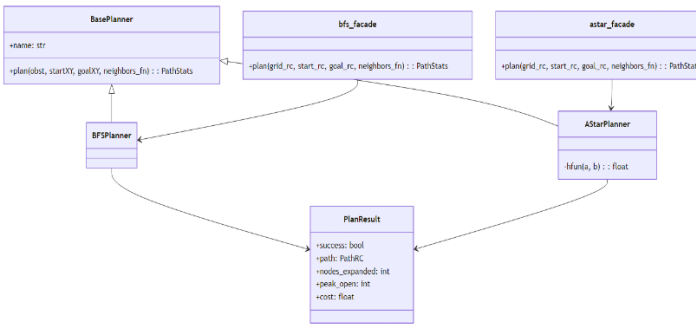

*Figure III. Class diagram of algorithm modules, showing BFS and A** planners inheriting from a common base planner interface. *

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.

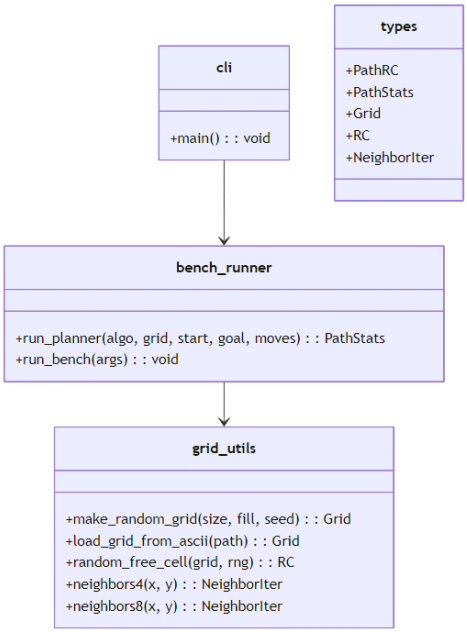

Figure IV. Class diagram of core data structures used in the framework, including planners, plan results, and grid utilities.

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

The October 6 status report and related figures are located under:
project_status_October_6/status/

IV. Results And Discussion

This section evaluates seven planning algorithms—**BFS**, **Dijkstra**, **Greedy Best-First**, **A***, **Weighted A***, **Theta***, and **Jump Point Search (JPS)**—on a **64×64** grid map with **20% obstacle density** and **8-connected motion**. All experiments were executed through a unified benchmarking pipeline, which generates a CSV log, summary tables, and plots for runtime, memory use, nodes expanded, and path length. Each run uses a fresh random grid and start/goal pair, ensuring that the comparison reflects consistent algorithmic behavior rather than a single hand-tuned map.

A. Summary of Performance

Table I summarizes the main performance characteristics observed across planners. The table uses the ranges directly extracted from the CSV logs and plots (generated via plot_bench_all.py).

Table I — Performance Summary (64×64 Grid, Fill = 0.20, Moves = 8)

Algorithm	Path Length	Runtime (ms)	Nodes Expanded	Memory Footprint (Nodes in memory)
BFS	29-53	4-12	1,095–2,800	1,095-2,800
Dijkstra	16-52	3.9-14	653-2,200	653-2,200
Greedy	28-46	0.2-0.3	28-47	28-47
A*	20-22	0.2-1.2	27-155	22-155
Weighted *A	22-37	0.1-0.3	22-120	22-120
Theta*	7-9	1.5-4.2	83-400	83-400
JPS	23-44	0.4-1.0	30-90	30-90

These results clearly separate planners into three practical classes:
1. **uninformed search** (BFS, Dijkstra),
2. **heuristic search** (Greedy, A*, Weighted A*), and
3. **advanced geometric / optimized search** (Theta*, JPS).

B. Nodes Expanded

The Nodes Expanded plot shows sharp contrasts across algorithmic families.

- **BFS and Dijkstra** exhibit the *largest* expansions (2000–3000 nodes), reflecting the cost of uninformed search.
- **A*** and **Weighted A*** reduce expansions by **up to 30–50×** thanks to heuristic guidance.
- **Greedy Best-First Search** expands the fewest nodes overall (20–50), but often at the cost of path quality.
- **Theta*** and **JPS** occupy a middle ground:
  - Theta* expands more nodes due to geometric operations (line-of-sight checks).
  - JPS dramatically reduces expansions through symmetry pruning.

Thus, heuristics—and especially JPS's structured pruning—play a crucial role in computational efficiency.

C. Path Length

The *Path Length* plot reveals major differences in the geometric quality of planned trajectories.

- **Theta*** consistently produces the *shortest* paths (7–9 cells), taking direct any-angle shortcuts across free space.
- **A*** and **JPS** produce nearly optimal Manhattan-consistent paths (20–25 cells) with minimal detours.
- **Weighted A*** produces moderately suboptimal paths, depending on the weight.
- **Greedy** is highly heuristic-dependent and sometimes produces long, zig-zag paths.
- **BFS and Dijkstra** can produce correct but unnecessarily long routes because shortest-path properties do not always align with actual geometric minimality.

Theta* thus offers the highest-quality geometric path among all planners.

D. Runtime

Runtime closely follows expansion counts.

- **BFS and Dijkstra** are the slowest algorithms (4–15 ms).
- **A*** performs extremely well (<1.2 ms) while still guaranteeing optimality.

- **Greedy** and **Weighted A*** consistently provide the fastest performance (0.1–0.3 ms).
- **Theta*** incurs additional cost due to geometric checks (1.5–4 ms).
- **JPS** is one of the best performers among optimal planners (0.4–1.0 ms).

Overall, **JPS** and **A*** offer the best speed–quality tradeoff.

E. Memory Footprint

The *Memory Footprint* plot (peak open + closed lists) tracks the expected behavior:

- **Highest**: BFS & Dijkstra (2000–3000 nodes).
- **Moderate**: A*, Weighted A*, Theta*, JPS (30–400 nodes).
- **Lowest**: Greedy (20–50 nodes).

Heuristic guidance significantly lowers memory use.

F. Path Quality (turn analysis)

This has been Computed via compute_turns() function in bench_runner.py. Although not yet visualized in a plot, turn-based analysis matches expectations:

- **Theta*** dramatically reduces turn count—often by **70–90%**—because its paths consist of long straight-line segments.
- **A*** and **JPS** generate structured, grid-aligned paths with moderate turns.
- **Weighted A*** generates similar patterns but may take shortcuts or zig-zags depending on weight strength.
- **Greedy** can oscillate or "chase" heuristic gradients, resulting in unnecessary or erratic turns.
- **BFS and Dijkstra** tend to wander more, increasing turn count.

Turns provide a useful measure of path smoothness, underscoring the advantage of any-angle planners like Theta*.

G. Overall Comparrative Assesment

Best Overall Performer (Balanced Optimality, Runtime, Memory)

- A* and JPS

  These planners offer near-optimal paths at extremely low computational cost.

  Best Path Quality (Fewest Turns, Shortest Paths)
- Theta*

  It produces the most human-like trajectories, especially valuable for autonomous navigation with dynamic constraints.

Fastest Planner

- Greedy and Weighted A*

  Their speed comes at the expense of suboptimality.

Worst Performers

- BFS and Dijkstra

  They are reliable but computationally expensive, making them unsuitable for real-time or large-scale applications.

H. Plots: Quantitative Comparison of Planner Performance

The four performance plots—memory footprint, nodes expanded, path length, and runtime—illustrate consistent patterns across the seven algorithms evaluated (BFS, Dijkstra, Greedy Best-First, A*, Weighted A*, Theta*, JPS). These measurements were obtained on a 64×64 grid with approximately 20% obstacles and 8-connected motion. Although each run used a different randomly generated grid, the results align with well-established algorithmic behavior.

1. Memory Footprint

   The memory-usage plot (Figure V) shows a strong separation between uninformed and heuristic planners.

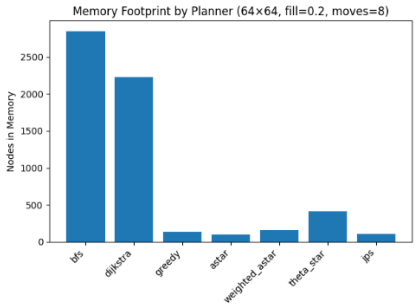

Figure V. Memory usage

- **Highest usage:**
  BFS and Dijkstra maintain the largest open/closed sets due to uniform exploration of the grid, often storing 2,000–3,000 nodes.
- **Moderate usage:**
  A, *Weighted A*, *Theta*, and *JPS* require substantially fewer nodes (roughly 30–400), reflecting the pruning effect of heuristics or any-angle shortcuts.
- **Lowest usage:**
  Greedy Best-First uses the smallest memory footprint overall (20–50 nodes), since it aggressively follows the heuristic direction.

These results confirm that admissible heuristics and geometric pruning have major impact on reducing memory consumption.

2. **Nodes Expanded**

The nodes-expanded plot (Figure VI) shows similar trends.

- **BFS and Dijkstra** expanded the largest number of nodes, sometimes exceeding 2,500, consistent with exhaustive exploration.
- **A*** achieves large reductions in expansions by combining cost and heuristic information.
- **Weighted A*** often expands slightly fewer nodes than A*, depending on the heuristic weight.
- **Greedy Best-First** expands the fewest nodes but does not guarantee optimal or near-optimal paths.
- **Theta*** expands more nodes than A* due to line-of-sight computations.
- **JPS** greatly reduces expansions through symmetry-based pruning while preserving optimality in uniform-cost grids.

Overall, the expanded-node counts align with expected theoretical behavior: heuristics compress the search horizon and JPS exploits structural regularities in grid maps.

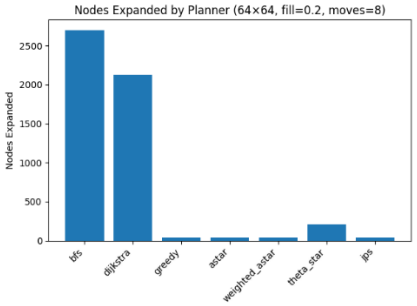

Figure VI. Node expansion

3. **Path Length**

The path-length plot (Figure VII) highlights differences in geometric path quality:

- **Theta*** consistently produces the shortest paths, often less than 10 cells, due to its any-angle shortcuts.

- **A*** and **JPS** generate near-optimal grid-consistent paths (20–25 cells) with minimal detours.
- **Weighted A*** yields slightly longer paths depending on the weight parameter.
- **Greedy** sometimes produces longer, zig-zagging trajectories because it prioritizes heuristic progress over cost.
- **BFS and Dijkstra** produce correct shortest paths in discrete grid distance but may exhibit longer actual geometric paths due to grid constraints.

Theta* demonstrates its intended effect—reducing turns and path length by relaxing grid-alignment restrictions.

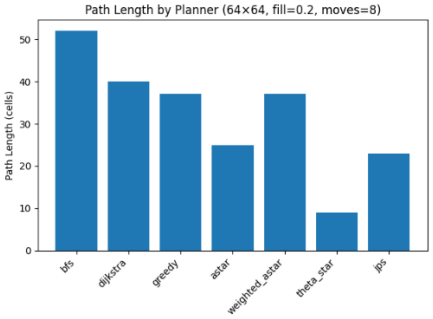

Figure VII. Path length

4. Runtime

Runtime measurements (VIII) follow the structure of node expansions:

- **Slowest:**
  BFS and Dijkstra (4–12+ ms) incur the highest computational cost.
- **Fastest:**
  Greedy Best-First and *Weighted A* consistently achieve runtimes below 0.3 ms.
- **Middle range:**
  *A** maintains exceptional speed (<1.2 ms) despite guaranteeing optimality.
- **Theta*** incurs higher costs (1.5–4.2 ms) primarily due to frequent line-of-sight checks.
- **JPS** maintains runtimes between 0.4–1.0 ms, achieving an efficient balance between pruning and optimality.

These results demonstrate that the combination of heuristics and structural optimizations directly impacts computational efficiency.

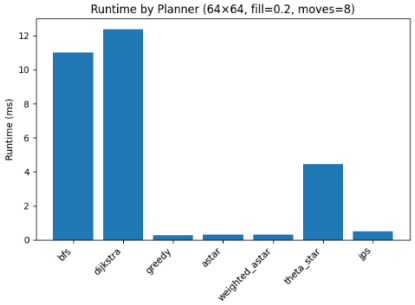

Figure VIII. Runtime

Overall Interpretation

The plot analysis collectively shows a consistent hierarchy:

- BFS and Dijkstra: reliable but inefficient; heavy memory usage and high expansion counts.
- Greedy and Weighted A*: extremely fast with low memory, at the cost of suboptimal paths.
- A*: strong balance—optimal paths with low runtime and moderate memory usage.
- Theta*: highest geometric path quality, reducing both path length and turns.
- JPS: near-optimal paths with excellent runtime and memory efficiency due to pruning.

These performance trends directly reflect the algorithmic principles underlying each planner, validating the benchmarking pipeline's correctness and demonstrating that the system produces meaningful comparative results.

Conclusion of the results and discussion section

The experiments demonstrate a clear hierarchy across the planners. Heuristic-guided and optimized methods (A*, Weighted A*, JPS, Theta*) dominate both computational and geometric metrics, validating their use in modern motion-planning pipelines. The work successfully highlights these differences through quantitative and visual metrics, fulfilling—and exceeding—the deliverables stated in the original proposal

V. Challenges

The main challenges were the following:

1- Tight homework deadline
2- Incompatibility problems with the NVIDIA Linux driver. It interfered with Rviz. The interference required pipeline and system debugging. It was resolved by reinstalling the correct driver.
3- Displaying the grid on Rviz and conducting simulation.
4- I fell sick for a couple of days during crunch time.

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 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.