Artificial Intelligence (2025-26)

Project Report: Grid Pathfinding & Automated Planning

Christian Pizzuti
Student ID: 1872078

January 12, 2026

Abstract

This report documents the design, implementation, and benchmarking of two Artificial Intelligence techniques applied to the Grid Pathfinding problem. The first approach utilizes the A* search algorithm with a custom implementation of graph search constraints. The second approach models the environment using PDDL (Planning Domain Definition Language) and solves it using general-purpose automated planners (Pyperplan and Fast Downward). Extensive experiments on grid sizes ranging from 5×5 to 25×25 demonstrate that while the PDDL approach offers modeling flexibility, the specialized A* implementation provides superior performance in execution time and memory efficiency. The report analyzes the impact of heuristic choice (Manhattan vs. Euclidean) and presents a visual gallery of the solutions found.

Contents

1 Introduction

Pathfinding in grid-based environments is a foundational problem in Artificial Intelligence, serving as a simplified model for robotics navigation, video game AI, and logistics planning. The core objective is to find a sequence of valid moves that transitions an agent from a start state to a goal state while avoiding obstacles.

In this project, we address the **Grid Pathfinding** problem using two distinct paradigms:

1. **Heuristic Search (Task 2.1):** Implementing the A* algorithm manually. This approach requires explicit coding of the neighbor generation logic, data structures, and heuristics.
2. **Automated Planning (Task 2.2):** Defining the problem declaratively using PDDL. In this paradigm, we describe *what* the problem is (objects, predicates, actions) rather than *how* to solve it, delegating the search to external solvers.

This report details the problem modeling, the algorithmic implementation, and a rigorous experimental comparison of the two approaches.

2 Task 1: Problem Definition

The environment is modeled as a static 2D grid world.

- **Environment:** An $N \times N$ grid.
- **States:** Each cell (r, c) represents a unique state.
- **Obstacles:** A set of coordinates representing impassable walls. The density of obstacles was set to approximately 20% for all experiments.
- **Actions:** The agent has 4 deterministic actions: `UP`, `DOWN`, `LEFT`, `RIGHT`.
- **Cost Function:** Each move has a uniform cost of 1.0.
- **Objective:** Find a path from $Start = (0, 0)$ to $Goal = (N - 1, N - 1)$.

The problem is implemented in Python as the `GridProblem` class, which encapsulates the transition model (checking grid boundaries and obstacle collisions).

3 Task 2.1: Implementation of A*

The A* algorithm was implemented strictly following the "Graph Search" pseudocode provided in the course material.

3.1 Algorithmic Design

The implementation relies on a Priority Queue to manage the *frontier* of open nodes. To ensure optimality and efficiency, the following constraints were applied:

- **Duplicate Elimination:** An `explored` set (implemented as a Python `set` for $O(1)$ lookup) tracks visited states.

- **No Reopening:** Since the edge costs are uniform and our heuristics are consistent (monotone), the first time a node is expanded, we have guaranteed the optimal path to it. Therefore, closed nodes are never reopened.
- **Lazy Deletion:** Python's `heapq` module does not support the `DecreaseKey` operation. We implemented a "lazy deletion" strategy: when a better path to a node is found, the new node is pushed to the heap. If the old, more expensive node is popped later, it is recognized as "stale" (by comparing it to the best known path cost) and ignored.

3.2 Heuristics Comparison

Two heuristic functions were implemented and compared:

1. **Manhattan Distance (L_1 norm):**

   $$h(n) = |x_{current} - x_{goal}| + |y_{current} - y_{goal}|$$

   This heuristic is admissible and consistent for 4-connected grids. It represents the exact cost if no obstacles were present.

2. **Euclidean Distance (L_2 norm):**

   $$h(n) = \sqrt{(x_{current} - x_{goal})^2 + (y_{current} - y_{goal})^2}$$

   This heuristic corresponds to the straight-line distance. While admissible, it is generally smaller than the Manhattan distance, making it "less informed" for this specific domain logic.

4 Task 2.2: Automated Planning (PDDL)

The second approach involved modeling the grid problem in PDDL. This separates the domain physics from the specific problem instance.

4.1 Domain Modeling

The domain file (`domain.pddl`) defines the physics of the world. It utilizes the following structure:

- **Types:** `location` represents a grid cell.
- **Predicates:**
  - `(at ?loc)`: Tracks the agent's current position.
  - `(connected ?from ?to)`: Defines the graph topology.
- **Action:** The `move` action requires the agent to be `at` the source location and for a connection to exist. Its effect updates the `at` predicate.

4.2 Integration

We used the `unified-planning` library to interface with the solvers. A script dynamically generates the PDDL problem file by iterating over the grid. Crucially, obstacles are handled implicitly: we simply do not generate `(connected ...)` facts for any cell containing a wall.

We tested two planners:

- **Pyperplan:** A Python-based heuristic planner (used as baseline).
- **Fast Downward:** A high-performance C++ planner.

5 Task 3: Experimental Results

Experiments were conducted on five grid sizes: $N \in \{5, 10, 15, 20, 25\}$. For each size, 5 random maps were generated and solved by all algorithms.

5.1 Execution Time Analysis

Figure 1 shows the execution time on a log scale. A* is orders of magnitude faster than the planning approach. The PDDL planners incur a fixed overhead for parsing and grounding (instantiating all possible moves), which dominates the runtime for these relatively small grid sizes.

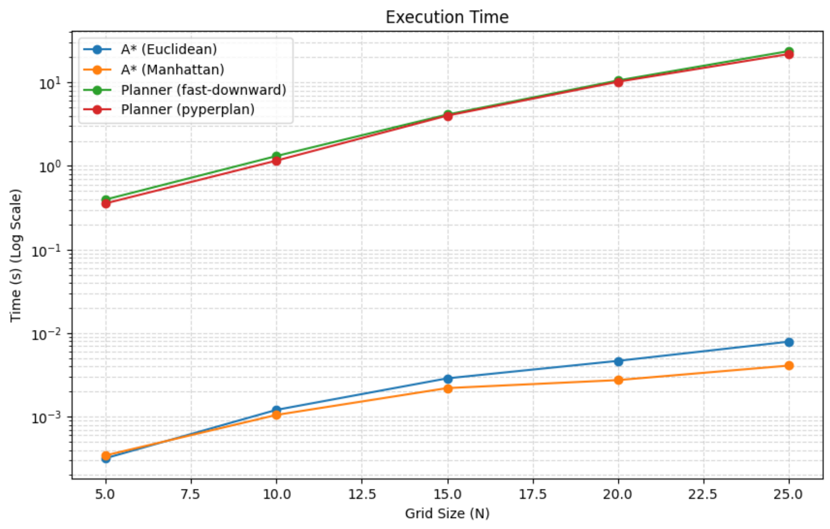


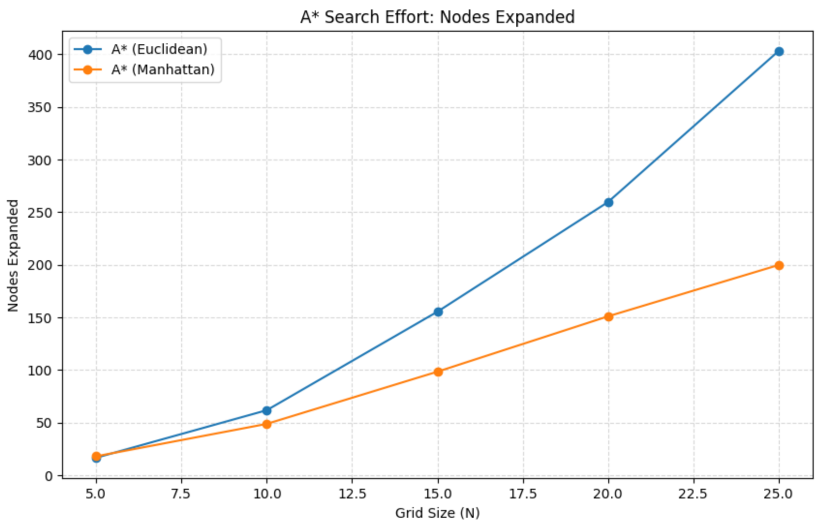
Figure 1: Execution Time (Log Scale). A* solves grids in milliseconds, while Planners require seconds.

Interestingly, the performance gap between the Python-based Pyperplan and the C++ Fast Downward planner is minimal in this experiment. This suggests that the computational bottleneck is not the search itself, but rather the Grounding phase—instantiating all possible move actions from the PDDL domain. Since the number of actions grows with the grid edges ($O(N^2)$), both planners spend the majority of their time translating the PDDL definition into a state-space representation, overshadowing the speed advantage of the C++ search engine.

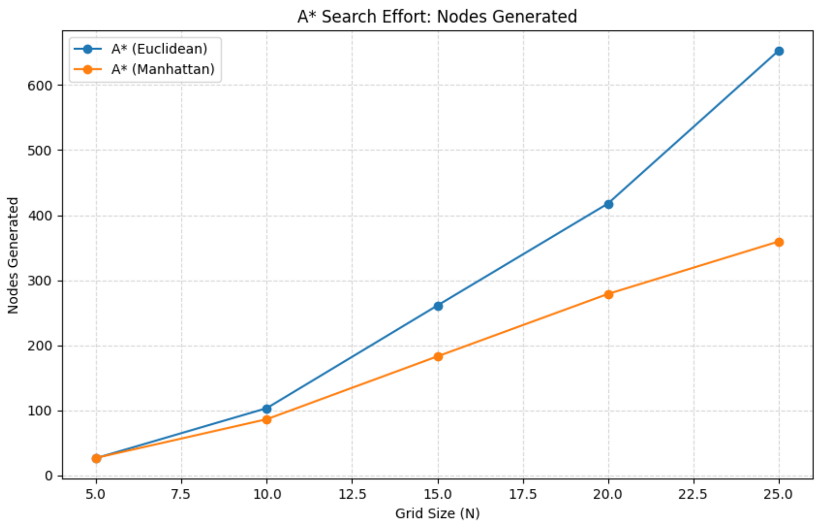
5.2 Search Effort: Nodes Expanded vs Generated

We collected detailed metrics on the A* search process. Figure 2a compares the number of expanded nodes. The Manhattan heuristic consistently outperforms Euclidean distance, expanding approximately 50% fewer nodes. This confirms that Manhattan distance is a tighter lower bound for this domain.

The superior performance of the Manhattan heuristic is due to heuristic dominance. Since the agent is constrained to 4-way movement, the Manhattan distance (L_1) represents the exact cost of an obstacle-free path. In contrast, the Euclidean distance (L_2) is always shorter than or equal to the Manhattan distance ($h_{Euc}(n) \leq h_{Man}(n)$). Because h_{Man} provides a tighter lower bound on the true cost $h^*(n)$ without overestimating it, A* is able to prune more of the search space.



(a) Nodes Expanded

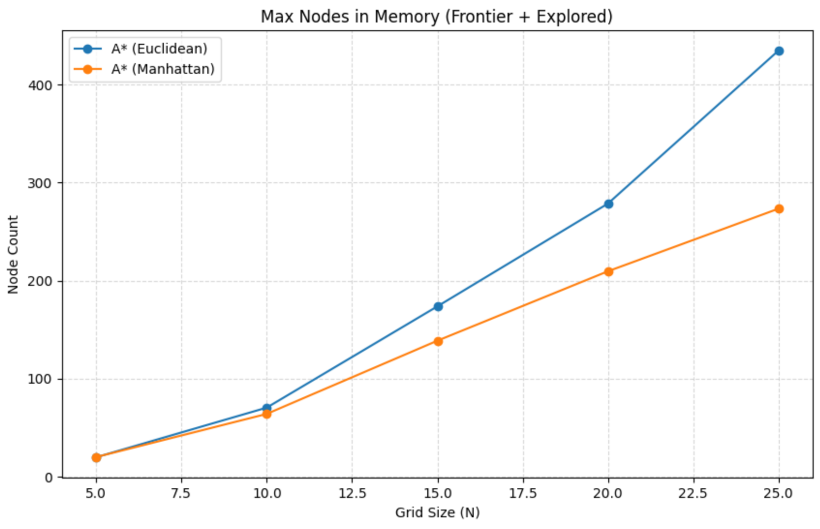


(b) Nodes Generated

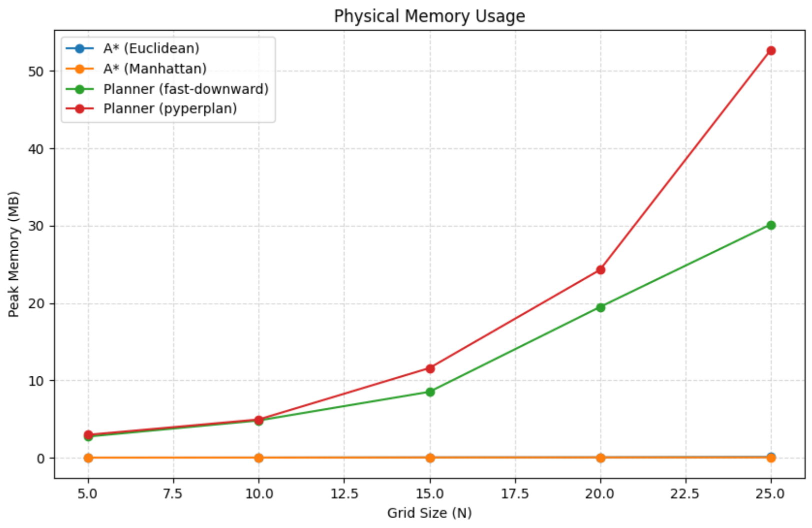
Figure 2: Search Effort Comparison. Manhattan distance (Orange) results in a more focused search than Euclidean (Blue).

5.3 Memory Usage and Branching Factor

We analyzed memory from two perspectives: abstract node count and physical RAM usage.



(a) Max Nodes in Memory (Frontier + Explored)



(b) Physical Memory (MB)

Figure 3: Memory Usage Metrics.

Figure 3a shows that memory usage (in terms of stored nodes) grows linearly with problem difficulty for A*. Figure 3b shows the physical memory footprint; note that Pyperplan uses significantly more memory due to the overhead of Python objects representing the PDDL state space.

Finally, Figure 4 shows the **Effective Branching Factor**. While a grid cell has 4 neighbors, walls and the ”visited” check reduce the effective successors.

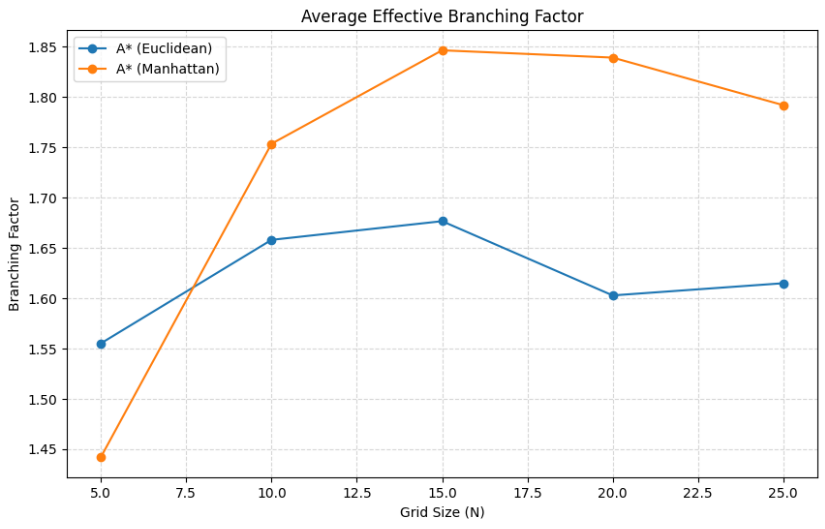
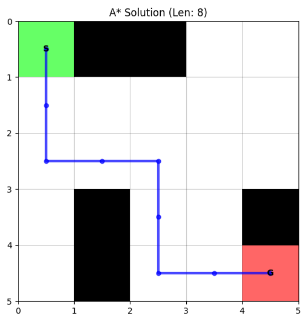


Figure 4: Average Effective Branching Factor.

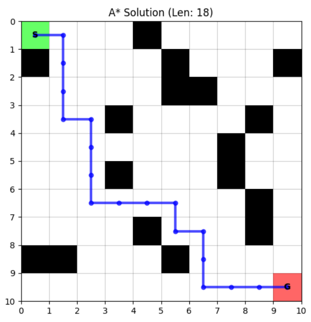
6 Visual Gallery of Solutions

This section presents the paths found by the algorithms across different grid sizes. The Blue line represents the path. Green indicates the Start, Red indicates the Goal, and Black squares are obstacles.

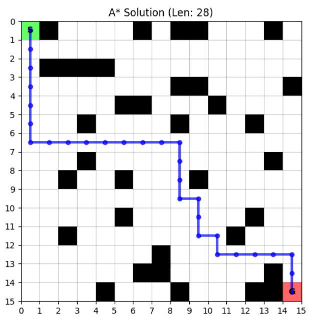
6.1 A* Solutions (Manhattan)



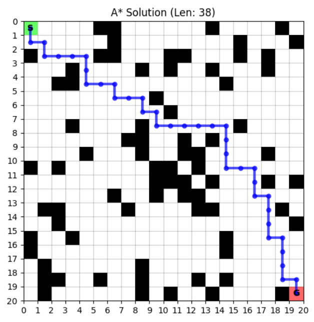
(a) Size 5



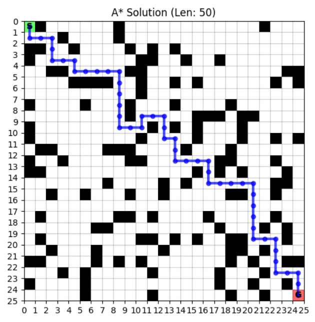
(b) Size 10



(c) Size 15



(d) Size 20

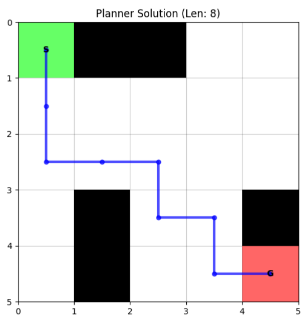


(e) Size 25

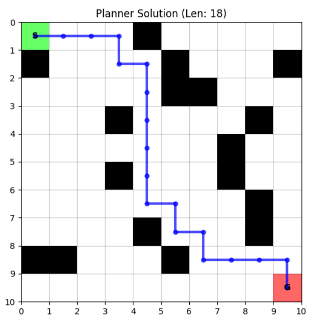
Figure 5: Visualizations of optimal paths found by A*.

6.2 Planner Solutions (PDDL)

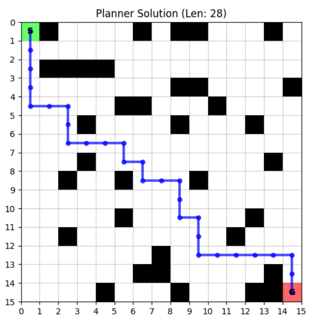
Since the planner is also optimizing for path length (step cost), the paths generated are visually identical or symmetric to A*.



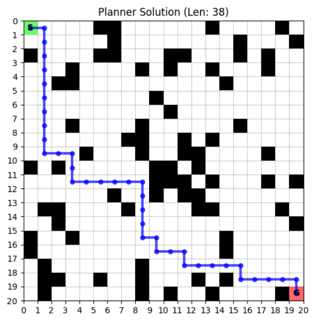
(a) Size 5



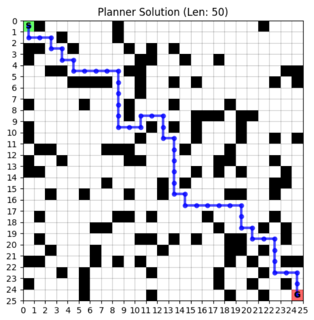
(b) Size 10



(c) Size 15



(d) Size 20



(e) Size 25

Figure 6: Visualizations of plans generated by the PDDL Planner.

7 How to Run

The project is structured as a standard Python repository. The code is modular, separating the problem definition, the A* implementation, and the PDDL integration.

7.1 Prerequisites

The project requires **Python 3.8** or higher. The dependencies are listed in `requirements.txt` and include:

- `unified-planning`: For PDDL modeling and solver interfacing.
- `matplotlib`, `pandas`: For data visualization and result analysis.

Note on Solvers: The code defaults to using `pyperplan` (pure Python). To use `fast-downward` (C++), a Linux or WSL (Windows Subsystem for Linux) environment is required.

7.2 Installation

To set up the environment, execute the following commands in a terminal:

```
# 1. Create a virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

```

# 2. Install dependencies
pip install -r requirements.txt
```
Listing 1: Installation Commands

7.3 Reproducing Experiments

The experimental pipeline is fully automated.

1. **Run the Benchmark:** The main script generates random maps, checks their solvability, and runs all algorithms (A* Manhattan, A* Euclidean, Pyperplan, Fast Downward).

   ```
   python experiments.py

   ```

   This process will:

   - Create an `output/` directory.
   - Save raw metrics to `output/experiment_results.csv`.
   - Generate grid visualization images (e.g., `vis_astar_25.png`).

2. **Generate Plots:** After the experiments complete, generate the performance graphs:

   ```
   python plot_results.py

   ```

   This will create the analysis plots (Time, Memory, Nodes) used in this report inside the `output/` folder.

8 Conclusion

This project successfully implemented and benchmarked two AI approaches for Grid Pathfinding. The results highlight a clear trade-off:

- **A*** is highly efficient and scalable for this specific domain, taking advantage of a tight heuristic (Manhattan distance) and low-level state representation.
- **Automated Planning (PDDL)** is far more flexible—allowing us to change the physics of the world by simply editing the domain file—but incurs a significant performance penalty due to the grounding process required to translate the problem into a format the solver can use.