Artificial Intelligence Homework

AY 2025–26

Thomas Sharp
Student ID: 1986413
Email: sharp.1986413@studenti.uniroma1.it

January 13, 2026

Abstract

This report presents the implementation and analysis of two different Artificial Intelligence techniques applied to the **Water Sort Puzzle**. The problem was first modeled and solved using a custom implementation of the **A* algorithm** in Python. Subsequently, the problem was modeled using the **PDDL** (Planning Domain Definition Language) standard and solved using the generic planner **Fast Downward**. The report discusses the modeling choices, particularly the trade-offs between "atomic" and "chunk" actions in PDDL, and compares the performance of the two approaches in terms of execution time and search space exploration.

Contents

1 Introduction

In this homework, I addressed the *Water Sort Puzzle*, a popular logic game where the objective is to sort colored liquids into beakers until all beakers are either empty or contain only one color.

I approached the problem from two perspectives:

1. **Search-based approach (Task 2.1):** Implementing the A* algorithm from scratch in Python, designing a specific state representation and an admissible heuristic.
2. **Planning-based approach (Task 2.2):** Modeling the domain and problem in PDDL and utilizing an off-the-shelf planner (Fast Downward).

The comparison highlights interesting trade-offs. While the Python A* implementation offers fine-grained control over the heuristic, the PDDL approach leverages highly optimized C++ solvers. A significant challenge in the PDDL modeling phase involved finding the right level of abstraction to avoid the "grounding explosion" problem.

2 Task 1: The Problem

The **Water Sort Puzzle** consists of N beakers, each with a fixed capacity C (usually 4 units). The state is defined by the configuration of colored liquid units within these beakers.

2.1 Rules and Modeling

A valid move consists of pouring liquid from a source beaker (S) to a destination beaker (D) under the following conditions:

- S is not empty.
- D is not full.
- The top color of S matches the top color of D, or D is empty.

The pouring action, from beaker to beaker, is dependant from the capacity of the beakers and is, usually, constrained on the amount of liquid to pour, so that if two adjacent units of liquid are of the same colour both have to be poured — this will be talked again regarding the level of abstraction.
The goal is reached when every beaker is either completely empty or full of a single color (monochromatic).

In my Python implementation, the state is modeled as a list of `Beaker` objects, where each object maintains a stack of colors.

3 Task 2.1: Implementation of A*

For the mandatory task, I implemented the A* algorithm as described in the course slides (Chapter 4), ensuring duplicate elimination and no reopening of closed nodes.

3.1 Data Structures

The state is encapsulated in a `Node` class. To manage the *open list* (frontier), I used Python's `heapq` module, which provides an efficient priority queue. The *closed list* (explored set) is a set of hashed state tuples to allow O(1) lookups for duplicate detection.

3.2 Heuristic Function

The heuristic function implemented for the A* algorithm is designed to be **admissible** and derives from a relaxation of the game's constraints. Specifically, it ignores the physical restrictions of the stack (i.e., only the top liquid can be moved) and the capacity limits of the beakers. It assumes an ideal scenario where any unit of liquid can be moved freely to merge with others of the same color.

The logic proceeds as follows: for every distinct color c present in the puzzle, the algorithm calculates the *fragmentation* of that color across the beakers. Let B_c be the set of beakers that contain at least one unit of color c. To unite all units of color c into a single destination beaker, it is necessary to pour liquid out of at least $|B_c| - 1$ source beakers.

Therefore, the heuristic value $h(n)$ is calculated as the sum of these minimum necessary moves for all colors:

$$h(n) = \sum_{c \in Colors} (|B_c| - 1) \tag{1}$$

This value represents the absolute minimum number of actions required to sort the liquids if the only goal was to group colors together without obstruction constraints. Since the actual game imposes strict rules on movement (LIFO behavior and color matching), the real cost to solve the puzzle is guaranteed to be equal to or greater than $h(n)$, satisfying the admissibility property required for the optimality of A*.
Other heuristics have been considered but not all were admissible, so later one has been chosen.

4 Task 2.2: Implementation of Planning (PDDL)

For the second AI technique, I chose **Automated Planning**. The problem instance is dynamically translated into PDDL files by a Python script (`problem_generator.py`) and then solved by **Fast Downward**.

4.1 Domain Modeling Choices

Modeling the Water Sort Puzzle in PDDL required a geometric abstraction. I defined the types `beaker`, `color`, and `level`. The capacity is enforced by the static predicate `(succ ?l1 ?l2)`, creating a chain of valid slots from bottom (l_0) to top (l_4).

4.1.1 The "Chunking" vs. "Atomic" Dilemma

During the modeling phase, I considered two distinct approaches for defining the `pour` action:

- **Atomic Model:** An action moves exactly one unit of liquid at a time. This would simplify the domain definition and reduce the grounding overhead for the planner by limiting the number of parameters per action.
- **Chunking Model:** An action moves all contiguous units of the same color (a "chunk") in a single step, faithfully replicating the standard rules of the Water Sort Puzzle.

Although the Atomic model would have been computationally lighter for the planner (reducing the number of instantiated actions), I opted for the **Chunking Model**.

4.1.2 Design Rationale and Limitations

The decision to implement the Chunking model was driven by the necessity of **metric consistency**. Since my Python implementation of A* (Task 2.1) treats the movement of a color block as a single step with a cost of 1, using an Atomic model in PDDL would have produced plans

with significantly higher costs (e.g., moving a block of 3 units would cost 3 in PDDL vs. 1 in Python), making a direct comparison of solution quality impossible.

Consequently, I defined specific actions for each possible block size: `pour-chunk-1`, `pour-chunk-2`, `pour-chunk-3`, and `pour-chunk-4`.

It is important to note that this design choice imposes a **hard constraint on the beaker capacity**. Since the actions are explicitly defined for specific amounts of units, the domain is not generic for any capacity C, but is strictly limited to the maximum chunk action defined. Therefore, all experiments in this report are conducted with a maximum beaker capacity of 4 units.

4.2 The Solver

For the resolution process, I utilized the **Fast Downward** planner. Specifically, I selected the `astar(blind())` configuration.

4.2.1 Configuration Choice: Optimality vs. Speed

Initially, I considered using satisficing configurations such as `lama-first` (based on Greedy Best-First Search and the FF heuristic). While these approaches are typically faster, they do not guarantee solution optimality. Since a primary goal of this experiment was to compare the results with my custom Python A* implementation—which is guaranteed to find the optimal path—using an optimal planner was mandatory to ensure a fair and meaningful comparison of plan costs.

4.2.2 The Issue with Heuristic Planners (LM-Cut)

I attempted to employ the state-of-the-art optimal configuration `seq-opt-lmcut`. This configuration utilizes A* search guided by the *Landmark-Cut* admissible heuristic, which typically outperforms blind search by effectively pruning the search space based on relaxation logic.

However, this configuration proved incompatible with the "Chunking" domain model. As detailed in Section 4.1.2, the `pour-chunk` actions rely heavily on ADL features, specifically disjunctions in preconditions (e.g., a move is valid if the level below is the bottom *OR* contains a different color). The Fast Downward translator compiles these complex logical constructs into *axioms* (derived predicates).

Since the current implementation of the LM-Cut heuristic does not support domains containing axioms (resulting in an "unsupported feature" error), I was forced to revert to `astar(blind())`. This configuration performs a uniform-cost search; while it lacks heuristic guidance, it fully supports the expressive PDDL features used in my model and guarantees optimality.

5 Task 3: Experimental Results

This section presents the results obtained by running both algorithms on a set of benchmark instances of increasing difficulty. The metrics considered are execution time, search space exploration (expanded nodes), and solution quality (cost).

5.1 Execution Time Analysis

Figure 1 compares the execution times on a logarithmic scale.

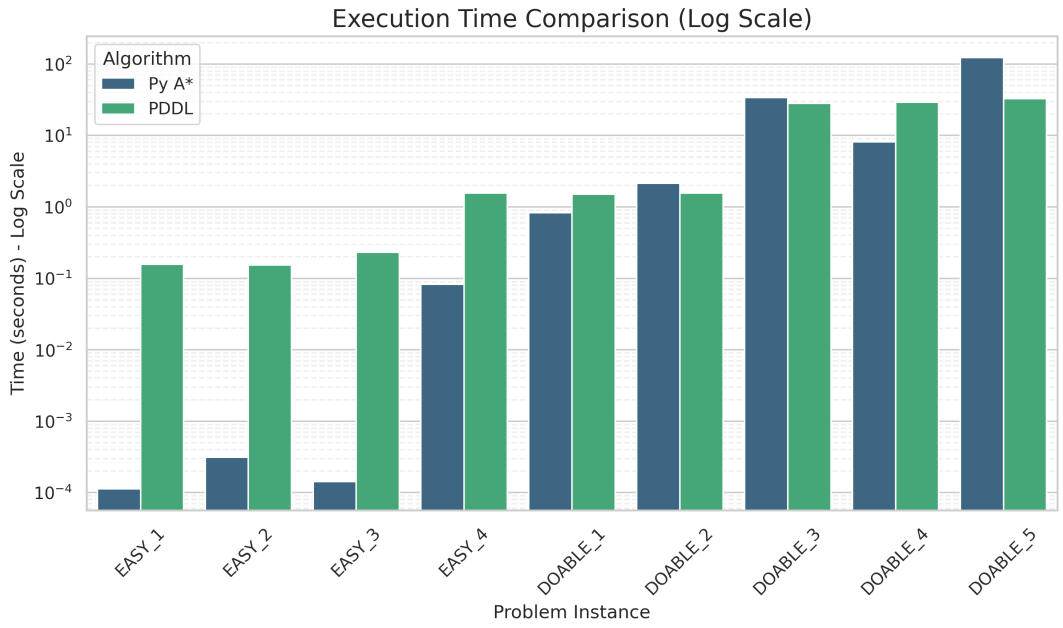

Figure 1: Execution time comparison (Log Scale). Note the overhead of PDDL on small instances versus its raw speed on harder ones.

The results highlight a distinct crossover point. On trivial instances (e.g., *EASY*), the Python A* implementation is instantaneous, whereas the PDDL approach incurs a fixed overhead due to the grounding process (generating `problem.pddl`, translating to SAS+, and initializing the planner).

However, as complexity increases (e.g., *DOABLE_3, 4, 5*), the highly optimized C++ engine of Fast Downward outperforms the Python implementation, despite using a blind search strategy. Python's interpreted nature becomes a bottleneck when managing large priority queues and object allocations, while Fast Downward's raw speed compensates for the lack of heuristic guidance.

5.2 Search Space and Heuristic Power

Figure 2 illustrates the number of nodes expanded by the two algorithms. This comparison offers a clear insight into the "intelligence" of the search strategies.

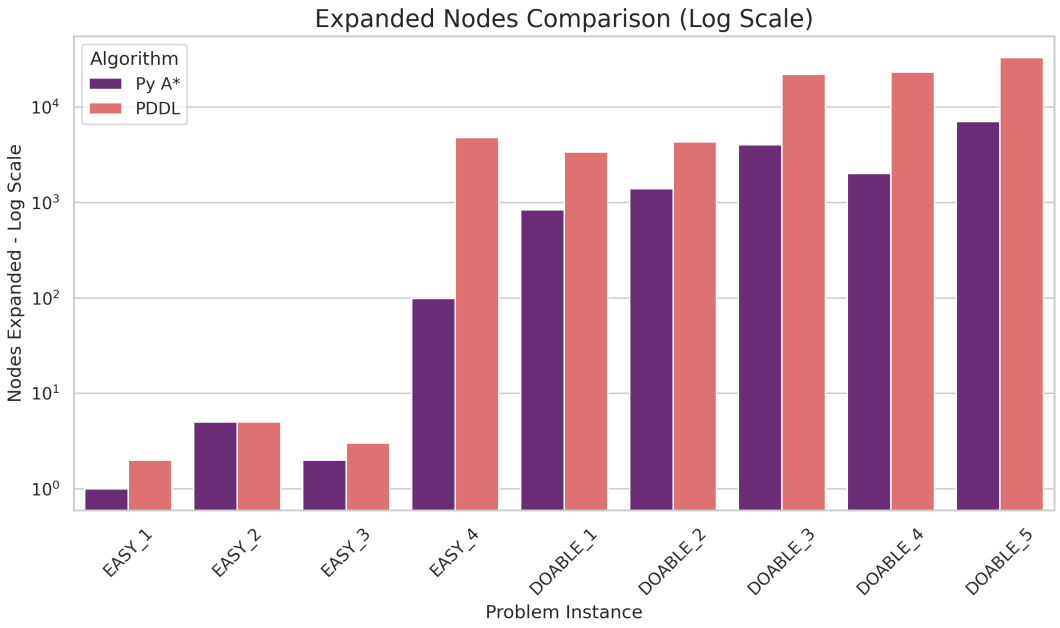

Figure 2: Number of Expanded Nodes (Log Scale). The domain-specific heuristic significantly prunes the search space.

Here, the Python A* implementation demonstrates the power of the domain-specific heuristic defined in Task 2.1. The heuristic effectively guides the search, keeping the number of expanded nodes relatively low even for harder problems. Conversely, the PDDL solver, forced to use `astar(blind())` due to the ADL constraints discussed in Section 4.2.2, performs a Uniform Cost Search. It explores the state space in "concentric circles" without direction, resulting in an exponential growth of expanded nodes.

This confirms that while Fast Downward is faster in terms of nodes-per-second (raw speed), the Python A* is "smarter" (expands fewer nodes).

5.3 Optimality Validation

Finally, Figure 3 validates the correctness of both implementations.

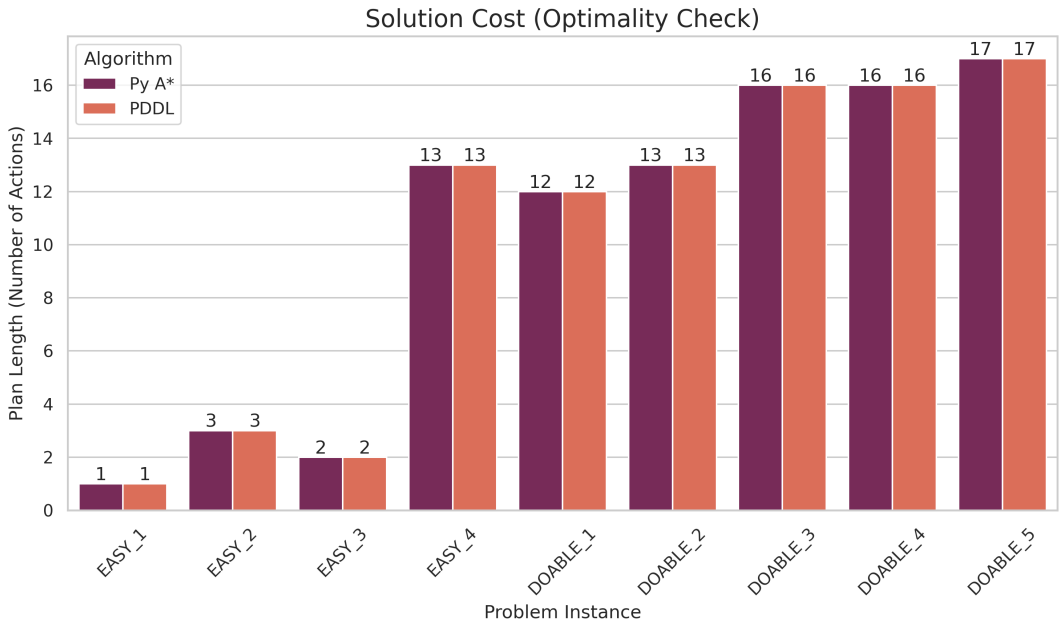

Figure 3: Solution Cost Comparison. Identical heights confirm both algorithms find the optimal solution.

As expected, both algorithms return plans of identical length for every instance. This confirms that:

1. The custom heuristic used in Python is indeed admissible (it never overestimated the cost).
2. The modeling of the PDDL domain faithfully represents the game rules, preserving the optimal path structure despite the chunking abstraction.

6 How to Run

The code provided in the attached GitHub repository allows for the reproduction of all experiments and results presented in this report.

6.1 System Requirements

- **OS:** Linux or Windows (via WSL2) is recommended due to Fast Downward dependencies.
- **Python:** Version 3.8 or higher.
- **Fast Downward:** The planner must be installed and compiled locally.

6.2 Configuration

Before executing the scripts, it is necessary to configure the path to the Fast Downward executable.

1. Open the file `src/task2_2.py`.
2. Locate the variable `PLANNER_EXE` at the beginning of the file.
3. Update the path to match the local installation of the planner (e.g., `/home/user/fast_downward/fast-downward.py`).

6.3 Execution

The project is structured to allow individual execution of tasks or a complete benchmark run.

6.3.1 Reproducing Experiments

To run the full suite of experiments described in Section 5 (Task 3), execute the following command from the root directory:

```
python src/experiments.py
```

This script performs the following operations:

- Runs the Python A* implementation on all test instances.
- Generates the PDDL domain and problem files dynamically for each instance.
- Invokes Fast Downward to solve the PDDL instances.
- Saves the metrics (time, nodes, cost) to `result/experiment_results.csv`.

6.3.2 Generating Plots

To generate the charts used in this report:

```
python src/plot_results.py
```

The images will be saved in the `result/plots/` directory.