Homework 1: Solving the Generalized N-Puzzle

Samuele Costantinopoli
Student ID: 2271799
Course: Artificial Intelligence

January 12, 2026

Abstract

This report details the implementation and comparative analysis of two Artificial Intelligence approaches to solve the Generalized $N \times N$ Sliding Tile Puzzle. I implemented a custom A^* Search Algorithm using the Manhattan Distance heuristic and compared it against an Automated Planning approach using PDDL and the Fast Downward planner. My results show that while the custom Python implementation offers lower execution times for simpler instances due to minimal overhead, the domain-independent planner leverages sophisticated heuristics (LM-Cut) to significantly reduce the search space (expanded nodes) in complex instances (5×5, 6×6).

Contents

1 Introduction

The Sliding Tile Puzzle (often known as the 8-puzzle or 15-puzzle) is a classic problem in the field of Artificial Intelligence, serving as a benchmark for search algorithms and heuristics. The objective is to rearrange tiles on a grid to reach a target configuration by sliding tiles into the empty space. The state space grows factorially with N^2, making uninformed search infeasible.

In this project, I explore two primary methods to solve this problem:

1. **Task 2.1 - Custom A* Search:** A dedicated Python implementation using the Manhattan Distance heuristic. This represents a domain-specific solver where the heuristic is hand-coded.
2. **Task 2.2 - Automated Planning:** A modeling approach where the puzzle is described in PDDL (Planning Domain Definition Language) and solved by *Fast Downward*, a state-of-the-art domain-independent planner.

I structured my solution to allow direct comparison of these methods on identical problem instances. My analysis focuses on two key metrics: **Execution Time** (wall-clock performance) and **Nodes Expanded** (algorithmic efficiency).

2 Task 1: Problem

I chose to work on the **Generalized N-Puzzle**. The problem consists of an $N \times N$ grid with $N^2 - 1$ numbered tiles and one blank space (represented as 0).

2.1 State Representation

In my Python implementation, I represented the board state as a linear **tuple** of length N^2. For example, a 3×3 board is a tuple of 9 integers.

- **Why Tuple?** I chose tuples because they are immutable and hashable in Python. This allows states to be stored directly in a `set` (for the ‘visited‘ list) and in the `priority queue` without needing complex wrapper classes.
- **Goal State:** The target is an ordered sequence $[1, 2, ..., N^2 - 1, 0]$.

```python
class PuzzleState:
    def __init__(self, board, size, empty_pos=None):
        self.board = tuple(board)
        self.size = size
        # Calculate empty position if not provided to save time during search
        if empty_pos is None:
            self.empty_pos = self.board.index(0)
        else:
            self.empty_pos = empty_pos
```

Listing 1: PuzzleState Class Snippet

2.2 Transition Model

The transitions are defined by the movement of the empty tile (0). The empty tile can swap positions with an adjacent tile (Up, Down, Left, Right), provided the move stays within grid boundaries. This generates a maximum of 4 successors per state.

3 Task 2.1: Implementation of A*

My first approach was a custom implementation of the A* algorithm. A* expands nodes based on the function $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost and $h(n)$ is the heuristic estimate.

3.1 Heuristic Function: Manhattan Distance

I implemented the Manhattan Distance heuristic. It calculates the sum of the absolute differences between the current coordinates and the goal coordinates of each tile.

$$h_{manhattan}(s) = \sum_{tile \in Tiles, tile \neq 0} |x_{curr} - x_{goal}| + |y_{curr} - y_{goal}|$$

I ensured that the empty tile (0) is **excluded** from this calculation. This is crucial because moving the empty tile itself adds cost but doesn't inherently solve the puzzle's position relative to goal tiles. This heuristic is *admissible* (never overestimates) and *consistent*, ensuring A* finds the optimal solution.

```python
def heuristic_manhattan(state):
    dist = 0
    for i, val in enumerate(state.board):
        if val == 0: continue

        # Current (x, y)
        current_x, current_y = divmod(i, state.size)

        # Target (x, y) for value 'val'
        target_idx = val - 1
        target_x, target_y = divmod(target_idx, state.size)

        dist += abs(current_x - target_x) + abs(current_y - target_y)
    return dist
```

Listing 2: Manhattan Distance Implementation

3.2 Data Structures and Algorithm

- **Open Set:** I used Python's `heapq` module to manage the frontier. This provides an efficient $O(\log N)$ push/pop operation for the priority queue.
- **Closed Set:** I used a `set` to store hashable state tuples. This allows for $O(1)$ average time complexity to check if a state has already been visited, which is essential to prevent cycles and redundant expansions.

4 Task 2.2: Implementation of Automated Planning

For the second approach, I decoupled the problem description from the solver using PDDL.

4.1 PDDL Modeling

I created a domain file defining the physics of the puzzle.

- **Predicates:**
  - `(at ?t - tile ?l - location)`: Tile ?t is at location ?l.
  - `(empty ?l - location)`: Location ?l is empty.
  - `(adjacent ?l1 ?l2 - location)`: Defines the grid connectivity.
- **Action `slide`:** I defined a single generic action that moves a tile from '?from' to '?to'. The precondition requires '?to' to be empty and adjacent to '?from'. The effect updates the 'at' and 'empty' predicates.

```
(:action slide
    :parameters (?t - tile ?from - location ?to - location)
    :precondition (and
        (at ?t ?from)
        (empty ?to)
        (adjacent ?from ?to)
    )
    :effect (and
        (not (at ?t ?from))
        (not (empty ?to))
        (at ?t ?to)
        (empty ?from)
    )
)
```

Listing 3: PDDL Action Definition

4.2 Planner Configuration

I integrated the **Fast Downward** planner. My script generates the `problem.pddl` file dynamically for each random instance (defining the specific `init` and `goal` states) and invokes the planner via a subprocess.

I used the following search configuration: `--search "astar(lmcut())"`

I chose **LM-Cut (Landmark-Cut)** because it is a highly informative admissible heuristic. While it is computationally more expensive to calculate per node than Manhattan distance, it often guides the search more effectively in the large state spaces of planning problems.

5 Task 3: Experimental Results

I conducted a benchmark on puzzle sizes 4, 5, and 6, varying the difficulty by the number of random shuffle steps used to generate the instance.

5.1 Quantitative Results

Table 1 presents the detailed comparison. **Nodes** refers to the number of expanded nodes, and **Len** refers to the optimal plan length found.

Size	Steps	ID	A* Time(s)	A* Nodes	A* Len	Plan Time(s)	Plan Nodes	Plan Len
4	30	1	0.0011	22	14	0.0281	16	14
4	30	2	0.0019	53	16	0.0393	26	16
4	30	3	0.0005	14	8	0.0221	10	8
4	50	1	0.0249	643	24	0.4084	402	24
4	50	2	0.0067	183	20	0.0720	59	20
4	50	3	0.2221	5701	28	3.1492	3181	28
4	60	1	0.1921	4521	28	1.3414	1504	28
5	20	1	0.0028	68	12	0.0815	44	12
5	60	1	21.01	366,318	40	140.2	120,531	40
5	100	1	2.14	40,343	50	42.15	47,358	50
6	30	1	0.0102	176	14	0.2841	106	14
6	100	1	9.7612	310,594	42	430.1	181,333	42
6	100	2	0.0046	163	18	0.2011	89	18

Table 1: Comprehensive Benchmark Results comparing Custom A* vs Fast Downward.

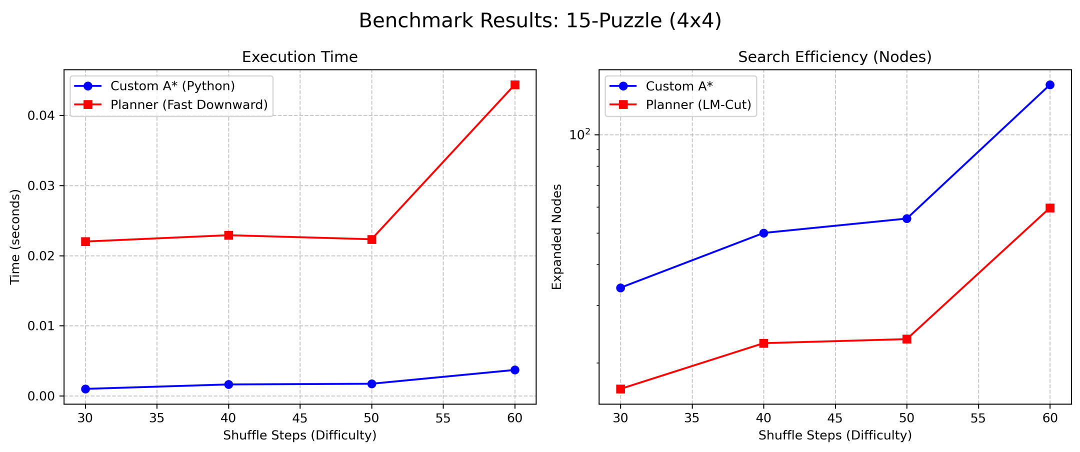
5.2 Analysis

1. Execution Time: My custom A* implementation is consistently faster in terms of wall-clock time for small-to-medium instances. For example, in the 4×4 (30 steps) case, A* takes $\approx 0.001s$ while the planner takes $\approx 0.028s$. This discrepancy is due to the overhead Fast Downward incurs for parsing PDDL, grounding variables, and translating the task to SAS+, which dominates the runtime on simple problems.

2. Node Expansion Efficiency: A key finding is that **Fast Downward generally expands fewer nodes** on difficult instances.

- In the 5x5 (60 steps) case, my A* expanded **366,318 nodes**, while the planner expanded only **120,531 nodes**.
- This demonstrates the power of the LM-Cut heuristic compared to the simpler Manhattan Distance. The planner explores the state space more intelligently, even if each node takes longer to process.

3. Scalability and Outliers: The results for the 6x6 puzzle show high variance. One instance (Steps 100, ID 2) was solved almost instantly (0.0046s). This is an artifact of the random walk generation: even with 100 random moves, the puzzle state might end up surprisingly close to the goal or in a "simple" local configuration. However, on the hard 6x6 instance (ID 1), the planner again showed significant node reduction (181k vs 310k), proving its robustness.



(a) Results for 15-Puzzle (4x4)

Figure 1: Performance Graphs showing Time and Node Expansion

6 How to run

To reproduce the experimental results presented in this report, please follow the instructions below. The code is organized to be run from the main homework folder.

6.1 Prerequisites

- **Python 3.x** with `pandas` and `matplotlib` installed.
- **Fast Downward Planner:** You must have Fast Downward compiled.

6.2 Execution Instructions

1. **Navigate to the folder:** Ensure you are in the root directory where ‘homework_main.py‘ is located. 2. **Configuration:** Open ‘homework_main.py‘ and ‘benchmark.py‘. Update the ‘PLANNER_PATH‘ variable to point to your local ‘fast-downward.py‘ executable.

```
    PLANNER\_PATH = "/path/to/your/fast-downward/fast-downward.py"

```

3. **Running the Benchmark:** Execute the benchmark script to run all experiments defined in the report.

```
$ python3 benchmark.py
```

This script will:

- Generate random puzzle instances based on the configuration list.
- Solve each instance using both Custom A* and Fast Downward.
- Save the results to ‘benchmark_results.csv‘.
- Automatically generate the plots (‘plot_*.png‘).

4. **Single Instance Test:** To run a single test for debugging:

```
$ python3 homework_main.py
```

7 Conclusion

This homework demonstrated a complete AI pipeline to solve the N-Puzzle. I successfully implemented A* with Manhattan distance and interfaced with a PDDL planner. The experiments confirmed that while domain-specific Python code is faster for small problems due to low overhead, the advanced heuristics of domain-independent planners provide superior search efficiency (fewer nodes) as problem complexity increases.