Artificial Intelligence – Homework

Solving the *n-Puzzle* with Search and Planning Techniques

Name: Francesco Zezza
Student ID: 2067707
Course: Artificial Intelligence
Academic Year: 2025–2026

1. Introduction

In this project, we address the *n-puzzle* problem, a classical benchmark in Artificial Intelligence commonly used to evaluate search and planning algorithms. The n-puzzle consists of a sliding-tile puzzle in which the objective is to transform an initial configuration into a predefined goal configuration by sliding tiles into an empty position.

The problem is tackled using two different Artificial Intelligence techniques:

1. A custom implementation of the A* search algorithm with an admissible heuristic.
2. A planning-based approach based on PDDL modeling and solved using the Fast Downward planner.

The main objective of the project is to compare these two approaches from both an algorithmic and an experimental perspective. To this end, multiple problem instances of increasing size and difficulty are generated, and several performance metrics are collected, including runtime, number of expanded and generated nodes, and memory usage.

2. Task 1 – Problem Description

The *n-puzzle* is defined on an $n \times n$ grid containing $n^2 - 1$ numbered tiles and one empty cell. A state of the problem is represented by the current arrangement of tiles on the grid. A legal action consists of sliding a tile adjacent to the empty cell into the empty position.

The goal state is defined as the ordered configuration:

$$(1,2,\dots,n^2-1,0)$$

where the value 0 denotes the empty cell.

The state space of the problem grows exponentially with n, making the n-puzzle a suitable benchmark for evaluating the scalability and efficiency of AI techniques. In this project, all problem instances are generated through random walks starting from the goal state, ensuring that all generated instances are solvable.

3. Task 2.1 – Implementation of A*

The A* search algorithm was implemented from scratch in Python. The algorithm maintains a frontier (open list) implemented as a priority queue ordered by the evaluation function:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost from the initial state to node n, and $h(n)$ is the heuristic estimate of the remaining cost to reach the goal.

The heuristic used is the Manhattan distance, which is admissible and consistent for the n-puzzle. Duplicate elimination is enforced by maintaining a closed set of visited states. Node reopening is disabled, in accordance with the assignment requirements.

During execution, the following metrics are collected:

- number of expanded nodes;
- number of generated nodes;
- minimum, maximum, and average branching factor;
- maximum size of the frontier;
- maximum number of stored states in memory.

This implementation allows a detailed analysis of both time and space complexity as the problem size increases.

From an implementation perspective, the A* algorithm is structured around a priority queue that stores search nodes ordered by their $f = g + h$ value.

Each state is represented as an immutable tuple, allowing it to be efficiently stored and queried inside hash-based data structures.

The closed set is implemented using a Python set and is used to enforce duplicate elimination: once a state has been expanded, it is never reconsidered.
This choice simplifies the implementation and ensures compliance with the required A* variant without node reopening.

The Manhattan distance heuristic is implemented in a separate module, making the heuristic computation independent from the search logic.
During the search, additional counters are updated to track expanded and generated nodes, branching factors, and memory usage, which are later used for experimental analysis.

4. Task 2.2 – Implementation of Planning (PDDL + Fast Downward)

As an alternative AI technique, the n-puzzle was modeled as an automated planning problem using the Planning Domain Definition Language (PDDL). A parametric domain was defined to represent tiles, positions, adjacency relations, and slide actions.

For each experiment, a corresponding PDDL problem file is automatically generated from the current puzzle configuration. This is handled by a dedicated Python module that translates the internal state representation into PDDL predicates, ensuring consistency between the search-based and planning-based formulations.

The planner used is Fast Downward, this planner is executed through the Windows Subsystem for Linux using Python's subprocess interface.
After execution, the generated plan is parsed from the sas_plan file and converted into a sequence of actions, allowing the planner's output to be integrated into the experimental framework.

The LM-Cut heuristic combined with an A*-based search strategy is used. Unlike the custom A* implementation, Fast Downward acts as a black-box planner; however, it provides reliable performance and allows solving larger instances without manually managing the search process.

5. Task 3 – Experimental Results

Experiments were conducted by varying both the puzzle size *n* and the length *k* of the random walk used to generate the initial state. For each pair *(n,k)*, multiple runs were executed using different random seeds to ensure statistical robustness.

Observations

For small puzzle sizes (e.g., *n=3*), both A* and the planning-based approach solve all instances quickly. However, A* consistently exhibits lower runtime and fewer expanded nodes due to its problem-specific heuristic and lightweight implementation.

As the difficulty increases (higher values of *k*), A* shows a noticeable increase in the number of expanded and generated nodes, leading to higher memory usage. The planning-based approach, while incurring a higher constant overhead due to the translation and planning phases, remains robust across different instances.

For larger puzzle sizes, A* becomes increasingly constrained by memory and time limits, whereas Fast Downward continues to find solutions for a wider range of instances. This highlights the trade-off between fine-grained control offered by search-based methods and the scalability provided by automated planning systems.

Overall, the results confirm that A* is highly effective for small to medium-sized instances, while planning-based approaches offer better scalability for larger or more complex problems.

6. How to Run

Execution Environment

The project requires the following software:

- Python 3.10 or higher
- Windows Subsystem for Linux (WSL) with Ubuntu
- Fast Downward planner

Fast Downward setup

Fast Downward must be installed inside the WSL environment.
In particular, the planner is expected to be located at:

```
$HOME/tools/downward/fast-downward.py
```

This can be achieved by cloning the official Fast Downward repository inside the WSL home directory.

Running the Experiments

When running the experiments on Windows, the Python module search path must be set as follows:

```
$env:PYTHONPATH="src"
```

Afterwards, the full experimental evaluation can be launched from the project root directory with:

```
python scripts/run_experiments.py
```

The script automatically generates problem instances, runs both A* and the planning-based approach, and stores the collected metrics in CSV format.

Execution Time

Depending on the selected parameters, the complete execution may take a non-negligible amount of time.
On the reference machine used for this project, generating the full set of experimental results required approximately 25/30 minutes.

Output

Experimental results are saved in:

```
results/runs.csv
```

These results can be used to reproduce the experimental analysis presented in this report.

7. Conclusions

In this project, two different AI techniques were applied to the same problem, allowing a direct comparison between heuristic search and automated planning. The experimental results demonstrate that while the A*

implementation offers full control over data structures and metrics collection, the planning-based approach trades transparency for scalability and robustness.

This comparison highlights the importance of selecting appropriate AI techniques depending on the characteristics and constraints of the problem at hand.