Artificial Intelligence Homework

Boyuan Zhao, 2239514

1. Introduction

In this project, I study the classical 15-puzzle problem and solve it using two different Artificial Intelligence approaches:

(a) a search-based algorithm, namely A*, implemented from scratch;

(b) a planning-based approach, based on PDDL modeling and an external planner.

The goal of the project is not only to correctly implement both approaches, but also to systematically compare their performance under increasing problem difficulty. In particular, I focus on execution time, scalability, and search effort when the scramble depth of the puzzle increases.

By running both solvers on the same problem instances and measuring several performance metrics, this project highlights the different behaviors of search-based and planning-based methods.

2. Task 1: Problem Definition and Modeling

2.1 Problem definition

The classical 15-puzzle consists of a 4×4 grid containing fifteen numbered tiles and one empty cell. A legal move slides one tile that is adjacent to the empty cell into the empty position. The goal is to reach a fixed target configuration in which the tiles are ordered correctly.

Each configuration of the grid corresponds to a state in the problem. From any state, a set of successor states can be generated by applying all legal moves of the empty cell. All moves have the same cost one.

2.2 State representation

I represent a state as a one-dimensional structure of length sixteen, using row-major order. Each entry corresponds to a position in the grid, and the value zero is used to represent the empty cell. This representation is compact, easy to compare, and efficient to store in hash-based data structures.

The position of the empty cell can be easily derived from this representation, which simplifies the generation of successor states. Moreover, this representation allows constant-time indexing and supports fast equality checks, which is important for both the A* algorithm and the planning-based approach.

The same state representation is used consistently in both solvers, ensuring a fair comparison.

2.3 Instance generation and difficulty scaling

To control the difficulty of the problem instances, I generate initial states starting from the goal configuration and applying a sequence of random legal moves. The length of this sequence is called the scramble depth, denoted by k.

Larger values of k correspond to states that are farther from the goal and therefore harder to solve. This parameter is used as the scaling parameter in the experimental evaluation to study how the two approaches scale as problem difficulty increases.

Since the instances are generated by random walks starting from the goal, all generated states are guaranteed to be solvable.

3. Task 2.1 – A* Search Implementation

3.1 Algorithm overview

In this task, I implement the A* search algorithm to solve the 15-puzzle. A* explores the state space by selecting, at each step, the state with the lowest estimated total cost. The evaluation function is defined as :

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

where

g(n) is the cost from the initial state to the current state

h(n) is a heuristic estimate of the remaining cost to reach the goal

3.2 Heuristic function

The heuristic used in the implementation is the Manhattan distance. For each tile, I compute the distance between its current position and its goal position in the grid, and then sum these distances over all tiles, ignoring the empty cell.

This heuristic is admissible and consistent for the 15-puzzle and provides effective guidance for the search while guaranteeing optimality of the solution.

3.3 Implementation details and outputs

The algorithm maintains a priority queue for the frontier and a visited set to eliminate duplicate states. States are not reopened once they have been expanded. When the goal state is reached, the solution path is reconstructed by following parent pointers.

In addition to the solution itself, the implementation records several performance metrics, including execution time, number of expanded nodes, number of generated nodes, branching factor statistics, and maximum frontier size. These values are later used in the experimental evaluation.

4. Task 2.2 – Planning with PDDL

4.1 PDDL modeling

In the planning-based approach, the 15-puzzle is described using the Planning Domain Definition Language (PDDL). The domain defines tiles, grid positions, and actions that move a tile into the empty cell. Preconditions ensure that a move is legal, and effects update the positions of the tile and the empty cell accordingly.

The problem file specifies the initial configuration generated for a given scramble depth and the fixed goal configuration. This modeling ensures that every plan produced by the planner corresponds to a valid solution of the puzzle.

4.2 Planner execution and integration

After generating the domain and problem files, I invoke an external planner directly from my code. The planner computes a sequence of actions that leads from the initial state to the goal. The planner runtime is measured automatically, and the output plan is parsed and converted into the same action representation used by the A* solver.

Since the planner is treated as a black-box solver, internal information such as the number of expanded nodes is not available.

5. Task 3 – Experimental Results

5.1 Experimental setup

The experimental evaluation compares the A* algorithm and the PDDL-

based planner on puzzle instances with increasing scramble depth. For each value of k, several random initial configurations are generated using different random seeds in order to reduce the influence of randomness.

Each configuration is solved independently using both approaches. For A*, both runtime and search-related metrics are recorded, while for the planner only runtime and solution length are available. All reported values are obtained by averaging the results over runs corresponding to the same scramble depth.

5.2 Runtime comparison

The runtime results show different behaviors for the two approaches. For small values of k, the PDDL planner requires more time due to its fixed planning overhead, while A* solves the instances very quickly. As the scramble depth increases, the runtime of A* grows steadily, reflecting the increasing size of the explored search space. In contrast, the runtime of the planner remains relatively stable across different values of k.

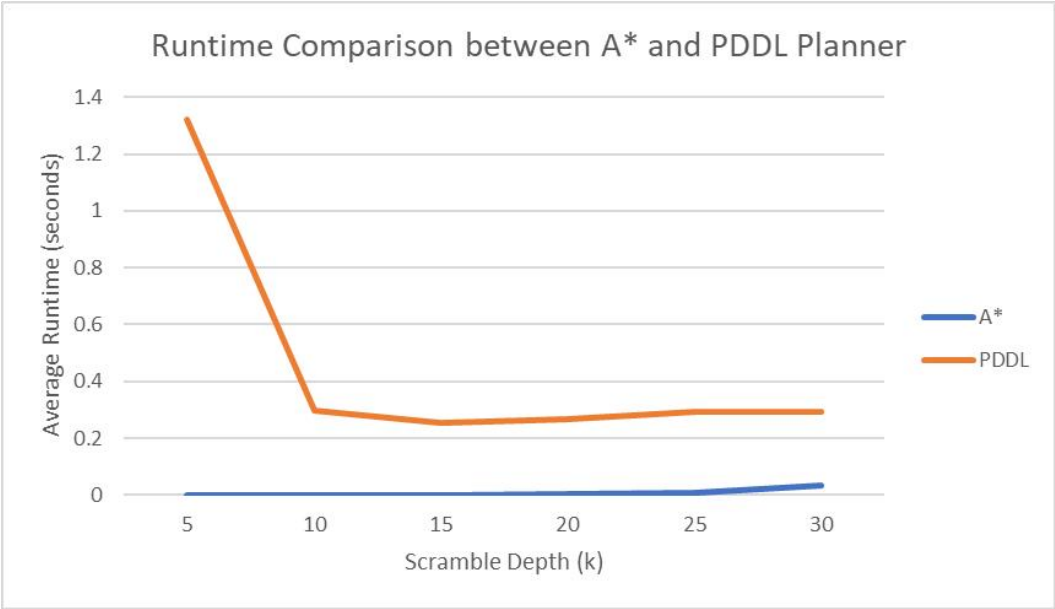

Figure 1: Runtime Comparison (A* vs PDDL)

5.3 Search effort of A*

The number of expanded nodes in the A* algorithm increases rapidly as the scramble depth becomes larger. This clearly shows that, although the heuristic is admissible and effective, the search effort grows significantly for more difficult instances. This behavior explains the observed increase in runtime for A* at higher scramble depths.

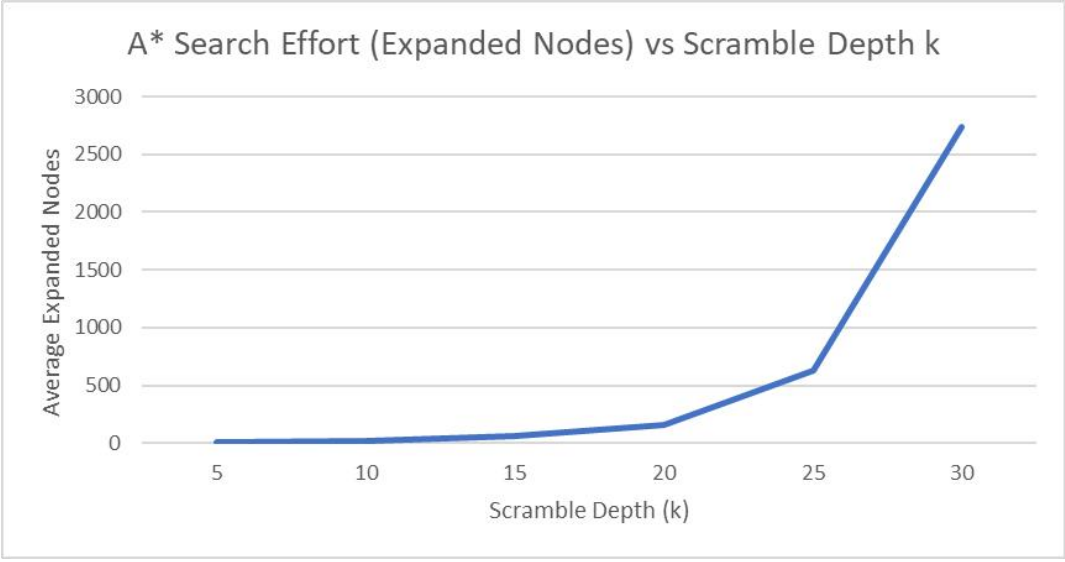

Figure 2: A* Search Effort vs Scramble Depth k

5.4 Discussion

The results highlight the contrasting behaviors of the two approaches. A* provides detailed insight into the search process but suffers from poor scalability as problem difficulty increases. In contrast, the PDDL planner, treated as a black-box solver, offers more stable performance but less transparency into the internal search process.

6. How to Run

This section describes how to set up the execution environment, run the different algorithms, select problem instances, and reproduce the experimental results.

6.1 Execution Environment and Dependencies

The following components are required:

Python 3.8.8

Windows Subsystem for Linux (WSL) with Ubuntu 22.04

Fast Downward planner (used for the PDDL)

All Python scripts rely only on the standard Python library.

Detailed installation instructions for WSL and Fast Downward are provided in the README file.

6.2 Running the Algorithms

A* Search (Task 2.1)

The A* solver can be executed with:

python task1.py

This command generates a solvable 15-puzzle instance and solves it using A* with the Manhattan distance heuristic. The solution and search statistics are printed to the console.

PDDL Planning (Task 2.2)

The PDDL-based approach can be executed with:

python planner_run.py

This script automatically generates the PDDL problem file from the puzzle instance, calls the Fast Downward planner, parses the planner output, and reconstructs the solution.

6.3 Choosing the Problem Instance and Algorithm

The difficulty of the 15-puzzle instances is controlled by the scramble depth k, which specifies the number of random moves applied to the goal state. Larger values of k correspond to more difficult problem instances.

The algorithm is selected by running either the A* script or the PDDL planner script.

The problem instance is generated automatically inside the scripts and can be modified by changing the scramble depth and random seed parameters.

6.4 Reproducing the Experimental Results (Task 3)

To reproduce the experimental evaluation, execute:

python task3_experiment.py

This script runs both A* and the PDDL-based planner on multiple puzzle instances with increasing scramble depth and different random seeds.

All measured metrics are saved to the file:

task3_results.csv

The plots and results presented in the report are obtained by averaging the values in this file over different random seeds.

Github: https://github.com/Bobozbyyy/AIhomework-15-puzzle

Acknowledgement

Generative AI tools were used to assist with code refinement and debugging, as well as with improving the clarity and wording of certain sections of the report