15-Puzzle Game

Alessandro Pio n.294417

October 29, 2023

1 Introduction

The 15-Puzzle problem is a classic example of an artificial intelligence search problem. The goal is to find a sequence of actions that leads from an initial state, where the tiles are randomly arranged on a 4x4 grid with one empty square, to a final state where the tiles are arranged in order. This report discusses the implementation of algorithms to solve the 15-Puzzle problem using Python.
To solve the problem, a search in the state space is required, starting from the initial state and aiming to reach the final state. Search algorithms like BFS and A* are used to explore this state space.

1.1 Initial State

The problem begins with an initial state, representing the random arrangement of tiles on the grid. This state is represented by a 4x4 matrix where each element represents a numbered tile from 1 to 15, and a 0 to represent the empty square.

1.2 Final State

The final state is fixed and represents the ordered arrangement of tiles on the grid, where tiles are numbered from 1 to 15, and the empty square is in the last position.

2 Construction of the Solution

2.1 State Class

The `State` class is designed to represent a puzzle state. It keeps track of the puzzle's current configuration, its parent state (the state from which it was reached), the action that led to this state, and the cost to reach it. Additionally, it calculates the Manhattan distance and Misplaced tiles as an A* heuristic.

2.2 Game Class

The `Game` class is created to define the rules of the game. It contains methods to check if a state is the final state, get possible actions from a state, and generate the next state based on an action. It also provides two heuristic functions: Manhattan distance and the count of misplaced tiles.

2.3 Agent Class

The `Agent` class is the core of the solving algorithm. It implements three different search strategies:

1. **Breadth-First Search (BFS)**: This algorithm starts from the root (initial state) and expands gradually in breadth, exploring all states at each level before moving on to the next level. The goal is reached when a state matching the final state is found.

2. **A* with Manhattan Distance Heuristic**: This algorithm uses the A* algorithm to search for the solution. The heuristic used is the Manhattan distance, which calculates the sum of horizontal and vertical distances between the tiles in the puzzle and their correct positions in the goal state.

3. **A* with Misplaced Tiles Heuristic**: This algorithm is similar to A* with Manhattan distance but uses the count of misplaced tiles as the heuristic. The heuristic estimates the cost to reach the final state based on the number of tiles out of place compared to the goal state.

2.4 Initialization and Execution

The code firstly defines the ‘goal_state‘, which represents the final state of the 15-Puzzle.
The initial state of the puzzle, ‘initial_state‘, can be randomized using the ‘randomise4x4matrix()‘ function, but in the provided code, it is explicitly set to a predefined configuration for consistency.
The code then attempts to solve the puzzle using three different search strategies: Breadth-First Search (BFS), A* with Manhattan Distance Heuristic, and A* with Misplaced Tiles Heuristic. The results of each solution attempt are stored in ‘solution_bfs‘, ‘solution_a_star_manhattan‘, and ‘solution_a_star_misplaced‘.
If a solution is found for all three search strategies, the code prints information about the initial state and the number of nodes explored by each search strategy.
If no solution is found, the code prints a message indicating that no solution was found because there is no solution for a given configuration.

NB A puzzle is solvent if:

- N is odd and the number of inversions is even on the size of the input;
- N is even and:
  - The blank is in an even position counting from the bottom while the number of reversals is odd
  - Or the blank is in an odd position counting from the bottom while the number of inversions is even;

2.5 Reconstructing the Solution Path

The ‘reconstruct_path()‘ function takes a solution in the form of a list of states and reconstructs the path leading to the solution. It starts from the final state and iteratively finds the parent state by matching it with the parent information stored in the solution data. The reconstructed path is returned as a list.

3 Conclusion

In conclusion, the provided Python code offers implementations of Breadth-First Search (BFS) and two variants of the A* search algorithm to solve the 15-Puzzle problem. These algorithms use heuristics such as Manhattan distance and misplaced tiles to estimate the cost of reaching the goal state. Manhattan distance certainly shows better results than misplaced tiles.
We can notice also that the bfs search finds a solution after a longer time than the two heuristics.