Solving the 8-Puzzle Problem with A*

An Implementation in Python

Yule Liu

HKUST(GZ) - UFUG 1601

July 8, 2025

What is the 8-Puzzle Problem?

The 8-puzzle is a classic sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space.

Objective: Rearrange the tiles from a given initial configuration to a target goal configuration by sliding tiles into the empty space.

Initial State

1	2	3
4		5
7	8	6

Goal State

1	2	3
4	5	6
7	8	

Introducing the A* Search Algorithm

A* (pronounced "A-star") is a powerful and widely used pathfinding and graph traversal algorithm.

Key Features:

- **Informed Search:** It uses a heuristic to guide its search, making it much more efficient than uninformed methods like Breadth-First Search (BFS).
- **Optimality:** If the heuristic is "admissible" (never overestimates the cost), A* is guaranteed to find the shortest path.
- **Completeness:** It will always find a solution if one exists.

A* is a perfect fit for the 8-puzzle because it intelligently explores the most promising moves first, avoiding a brute-force search of all possible states.

The Core of A*: The Evaluation Function

A* decides which path to explore next based on the following evaluation function for a node (or state) n:

Evaluation Function

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

Where:

- $g(n)$: The actual cost of the path from the start node to node n.
  - For the 8-puzzle, this is simply the number of moves made so far.
- $h(n)$: The **heuristic** estimated cost from node n to the goal.
  - This is the "intelligent" part of A*. It's an educated guess.

The algorithm always expands the node with the **lowest** $f(n)$ value.

Heuristics for the 8-Puzzle

A good heuristic is crucial for A*'s performance. Two common admissible heuristics for the 8-puzzle are:

1. **Number of Misplaced Tiles**
   - Simple: Count the number of tiles that are not in their goal position.
   - Fast to compute, but less accurate.

2. **Manhattan Distance (Most Common)**
   - For each tile, sum the number of horizontal and vertical moves required to get it to its goal position.
   - More accurate than misplaced tiles, leading to a more efficient search.

Example: Manhattan Distance

For tile '6' in the initial state, its goal is one step left and one step up. Its Manhattan distance is $1 + 1 = 2$.

A* Algorithm Step-by-Step

Data Structures Needed:

- **Open List (Priority Queue):** Stores nodes that have been generated but not yet visited. Nodes are ordered by their $f(n)$ value.
- **Closed List (Set or Hash Table):** Stores nodes that have already been visited to avoid cycles and redundant work.

The Process:

1. Create the start node and add it to the Open List.
2. While the Open List is not empty:
    1. Pop the node with the smallest $f(n)$ value from the Open List. Let's call it *current*.
    2. If *current* is the goal state, we're done! Reconstruct the path.
    3. Add *current* to the Closed List.
    4. Generate all valid successor nodes of *current*.
    5. For each successor:
        - If it's already in the Closed List, ignore it.
        - Calculate its $g(n)$ and $h(n)$ values.
        - If it's not in the Open List, add it.
3. If the Open List becomes empty, no solution exists.

Your Goal

Your objective is to complete a Python script that finds the shortest solution for the 8-puzzle problem using the A* search algorithm.

You have been provided with a script containing:

- A ‘Node‘ class to represent puzzle states.
- Several helper functions.
- A complete testing framework.

Your Task

You must implement the logic for three key functions that are currently empty or incomplete.

Task 1: Implement the Heuristic

Function to Complete: `calculate_manhattan_distance`

What it does

This function is the heuristic ($h(n)$) for our A* algorithm. It must calculate the total Manhattan distance for the given puzzle 'state'.

Input:

- `state`: A 3x3 list of lists (the current board).
- `goal_positions`: A dictionary mapping each tile to its goal '(row, col)'.

Output:

- An `integer` sum of the Manhattan distances for all tiles.

Hint:

- Loop through each cell of the 'state'.
- For each tile, find its distance to its goal position: `abs(row - goal_row) + abs(col - goal_col)`.
- Sum these distances.

Task 2: Generate Successor States

Function to Complete: `generate_successors`

What it does

This function finds all valid next states by swapping the blank tile (‘0‘) with an adjacent tile.

Input:

- `state`: A 3x3 list of lists (the current board).

Output:

- A `list` of new 3x3 states, one for each valid move.

Hint:

- Find the location of the blank tile (‘0‘).
- For each potential move (up, down, left, right), check if it’s within the grid.
- If a move is valid, create a **deep copy** of the state, perform the swap, and add the new state to your list.

Task 3: Implement the A* Solver

Function to Complete: `solve`

What it does

This is the main A* algorithm logic. It uses a priority queue (‘open_list‘) and a visited set (‘closed_set‘) to find the optimal path.

Process to Implement:

1. Start a ‘while‘ loop that runs as long as the ‘open_list‘ is not empty.
2. Pop the node with the lowest f-score from the ‘open_list‘.
3. **Goal Check:** If it’s the goal state, reconstruct and return the path.
4. Add the current node to the ‘closed_set‘.
5. Generate all successors for the current node.
6. For each successor:
    - If it’s in the ‘closed_set‘, ignore it.
    - Otherwise, create a new ‘child_node‘ and add it to the ‘open_list‘.

How to Test Your Code

Once you have completed the functions, run the script from your terminal:

```
python your_script_name.py
```

The testing framework will automatically evaluate your solution against several test cases.

You will see:

- **PASSED:** Correct!
- **FAILED:** Incorrect.

A passing grade requires:

- Finding the optimal (shortest) path.
- Correctly identifying unsolvable puzzles.

Good Luck!