A* Algorithm for Solving the 3×3 Sliding Puzzle

Dahmani Abdelmadjid

1 Introduction

The A* algorithm is a widely used pathfinding and graph traversal algorithm. It is particularly effective for solving puzzles like the 3×3 sliding puzzle (also known as the 8-puzzle or taquin). A* combines the benefits of uniform-cost search (Dijkstra's algorithm) and greedy best-first search by using a heuristic to estimate the cost to reach the goal.

1.1 Key Features of A*

- **Optimality:** A* is guaranteed to find the shortest path if the heuristic is admissible (never overestimates the cost).
- **Efficiency:** A* explores fewer states compared to brute-force methods like DFS or BFS by prioritizing states with lower estimated total cost.
- **Heuristic Function:** A* uses a heuristic function (e.g., Manhattan distance) to estimate the cost from the current state to the goal state.

2 A* Algorithm Implementation

The A* algorithm is implemented as follows:

- A priority queue (heap) is used to manage states based on their total cost $f = g + h$, where:
  - g is the cost from the initial state to the current state.
  - h is the heuristic estimate of the cost from the current state to the goal state.
- The algorithm generates valid neighboring states by sliding tiles into the empty space ('0').
- It avoids revisiting states by keeping track of visited states using a set.
- The Manhattan distance is used as the heuristic function.

3 Python Code for A* Algorithm

Below is the Python implementation of the A* algorithm for solving the 3×3 sliding puzzle. The code is explained in detail with comments.

```python
import heapq

# Define the initial and goal states
initial_state = [2, 8, 3, 1, 6, 4, 7, 0, 5]
goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

# Define possible moves (up, down, left, right)
moves = [(0, -1, 'Left'), (0, 1, 'Right'), (-1, 0, 'Up'), (1, 0, 'Down')]

# Function to find the index of the empty tile (0)
def find_empty_tile(state):
    return state.index(0)

# Function to generate valid neighboring states
def generate_neighbors(state):
    neighbors = []
    empty_index = find_empty_tile(state)
    row, col = empty_index // 3, empty_index % 3  # Convert index to (row, col)

    for dr, dc, move_name in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:  # Check if the move is within bounds
            new_state = state.copy()
            # Swap the empty tile with the neighboring tile
            new_index = new_row * 3 + new_col
            new_state[empty_index], new_state[new_index] = new_state[new_index], new_state[empty_index]
            neighbors.append((new_state, move_name))
    return neighbors

# Heuristic function: Manhattan distance
def manhattan_distance(state):
    distance = 0
    for i in range(9):
        if state[i] != 0:  # Ignore the empty tile
            goal_index = goal_state.index(state[i])
            current_row, current_col = i // 3, i % 3
            goal_row, goal_col = goal_index // 3, goal_index % 3
            distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

# A* Algorithm
def a_star(initial_state, goal_state):
    priority_queue = []  # Priority queue for A*: (f, g, state, path)
    heapq.heappush(priority_queue, (0 + manhattan_distance(initial_state), 0, initial_state, []))
    visited = set()  # Set to keep track of visited states
```

```python
    visited.add(tuple(initial_state))

    while priority_queue:
        f, g, current_state, path = heapq.heappop(priority_queue)

        # Check if the current state is the goal state
        if current_state == goal_state:
            return path  # Return the path to the goal state

        # Generate and explore neighbors
        for neighbor, move_name in generate_neighbors(current_state):
            if tuple(neighbor) not in visited:
                visited.add(tuple(neighbor))
                # Calculate f = g + h (heuristic)
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(priority_queue, (new_f, new_g, neighbor, path + [move_name]))

    return None  # If no solution is found

# Function to print the puzzle state
def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Solve the puzzle
solution_path = a_star(initial_state, goal_state)

# Output the solution
if solution_path:
    print("Solution Found!")
    print(f"Number of Moves: {len(solution_path)}")
    print("Path to Solution:", solution_path)
else:
    print("No solution found.")
```

Listing 1: A* Algorithm for 3×3 Sliding Puzzle

4 Explanation of the Python Code

The Python code implements the A* algorithm to solve the 3×3 sliding puzzle. Below is a detailed explanation of each part of the code:

4.1 Initial and Goal States

- The puzzle states are represented as lists of length 9, where ‘0‘ represents the empty tile.
- Example:
  - `initial_state = [2, 8, 3, 1, 6, 4, 7, 0, 5]`

– `goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]`

4.2 Generating Neighboring States

- The function `generate_neighbors` generates valid neighboring states by sliding tiles into the empty space.
- It calculates the row and column of the empty tile and checks if moves (up, down, left, right) are within bounds.
- For each valid move, it creates a new state by swapping the empty tile with the neighboring tile.

4.3 Heuristic Function

- The heuristic function `manhattan_distance` calculates the Manhattan distance for each tile from its current position to its goal position.
- The Manhattan distance is the sum of the horizontal and vertical distances.
- This heuristic is admissible, meaning it never overestimates the cost to reach the goal.

4.4 A* Algorithm

- The A* algorithm uses a priority queue (heap) to explore states with the lowest total cost $f = g + h$.
- It starts with the initial state and explores its neighbors, adding them to the priority queue if they haven't been visited.
- The algorithm terminates when the goal state is found or the priority queue is empty.

4.5 Output

- The solution path is a list of moves (e.g., '['Up', 'Left', 'Down', ...]').
- The number of moves and the sequence of moves are printed.

5 Example Output

For the given initial and goal states, the output might look like this:

```
Solution Found!
Number of Moves: 5
Path to Solution: ['Up', 'Left', 'Down', 'Right', 'Down']
```

6 Conclusion

The A* algorithm is an efficient and optimal method for solving the 3×3 sliding puzzle. By using the Manhattan distance as a heuristic, A* explores fewer states compared to brute-force methods while guaranteeing the shortest solution path. This implementation demonstrates the power of A* in solving combinatorial puzzles.