8-puzzle problem Report

Name: Saumya Gupta

Roll no: _02

Branch: CSE(AI)

Project title: 8 puzzle problem

AKTU Roll no: 2024011003002222

Overview:

This code implements a solution to the **8-puzzle problem** using the *A search algorithm**. The goal of the 8-puzzle is to move tiles on a 3x3 grid until they match a goal state. In this case, the goal state is a specific arrangement of numbers (1 through 8) with the empty space represented by 0.

Key Components of the Code:

1. **Puzzle State Representation**:
    a. The puzzle is represented as a 3x3 grid (a tuple of tuples) where each number (1-8) represents a tile, and 0 represents the empty space.
    b. Each `PuzzleState` object represents a specific configuration of the puzzle, along with additional information about its parent state (the previous puzzle configuration), the move that led to this configuration, and the cost to reach this state.
2. **Heuristic (Manhattan Distance)**:
    a. The **Manhattan distance** heuristic is used to estimate the "cost" to reach the goal state from the current state. It calculates the total distance that each tile is from its goal position, where distance is measured as the number of

moves required to align the tile's current position with its goal position.

b. This heuristic helps the A* algorithm prioritize moves that are more likely to lead to the goal quickly.

3. *Algorithm**:
   a. The *A algorithm** is a search algorithm used to find the shortest path to the goal state. It uses a priority queue (implemented via a heap) to explore states in the order of their **f-value**, where:
      i. `f = g + h`
      ii. g is the cost to reach the current state (number of moves made).
      iii. h is the heuristic (estimated cost from the current state to the goal).
   b. The algorithm explores possible states by considering valid moves (up, down, left, right) and generates new states accordingly.
4. **Solution Reconstruction**:
   a. When the algorithm reaches the goal state, it backtracks through the parent states to reconstruct the solution path, which is the sequence of moves taken to solve the puzzle.
   b. The solution is printed in a sequence where each step shows the configuration of the puzzle at that stage.

Steps to Solve the Puzzle:

1. **Initialization**:

a. The puzzle is represented as a 3x3 grid, with the initial state being:

```
Copy
1  2  3
8  0  4
7  6  5
```

The `start_state` is an object of the `PuzzleState` class representing this configuration.

2. *A Search**:
    a. The A* algorithm starts with the initial state and continuously explores the possible moves (up, down, left, right) until it reaches the goal state.
    b. During this process, the algorithm uses the Manhattan distance heuristic to prioritize states that are closer to the goal.
3. **Goal State**:
    a. The goal state is represented as:

```
Copy
1  2  3
8  0  4
7  6  5
```

    b. Once the algorithm finds a state matching the goal, it traces back through the sequence of states leading to this solution and prints the steps.
4. **Solution Output**:

a. The solution is a sequence of states, printed one after the other, showing the puzzle's configuration at each step.

Code Breakdown:

- **Puzzle State Class**:
  - **`__init__`**: Initializes a puzzle state, calculates its heuristic (Manhattan distance), and computes the f-value (f = g + h).
  - **`manhattan_distance`**: Calculates the Manhattan distance for the current puzzle state.
  - **`get_possible_moves`**: Generates all valid next states by moving the empty tile (0) in the allowed directions (up, down, left, right).
  - **`__lt__`**: Allows states to be compared based on their f-values so they can be ordered in a priority queue.
  - **`__repr__`**: Provides a human-readable string representation of the puzzle state.
- *A Search Function**:
  - **`a_star`**: Implements the A* algorithm to find the optimal solution by exploring all valid moves, checking the f-value, and backtracking from the goal state once it is found.
- **Solution Printing**:
  - **`print_solution`**: Prints the sequence of moves leading to the goal state.

Example Output:

If the algorithm successfully finds a solution, it will print something like:

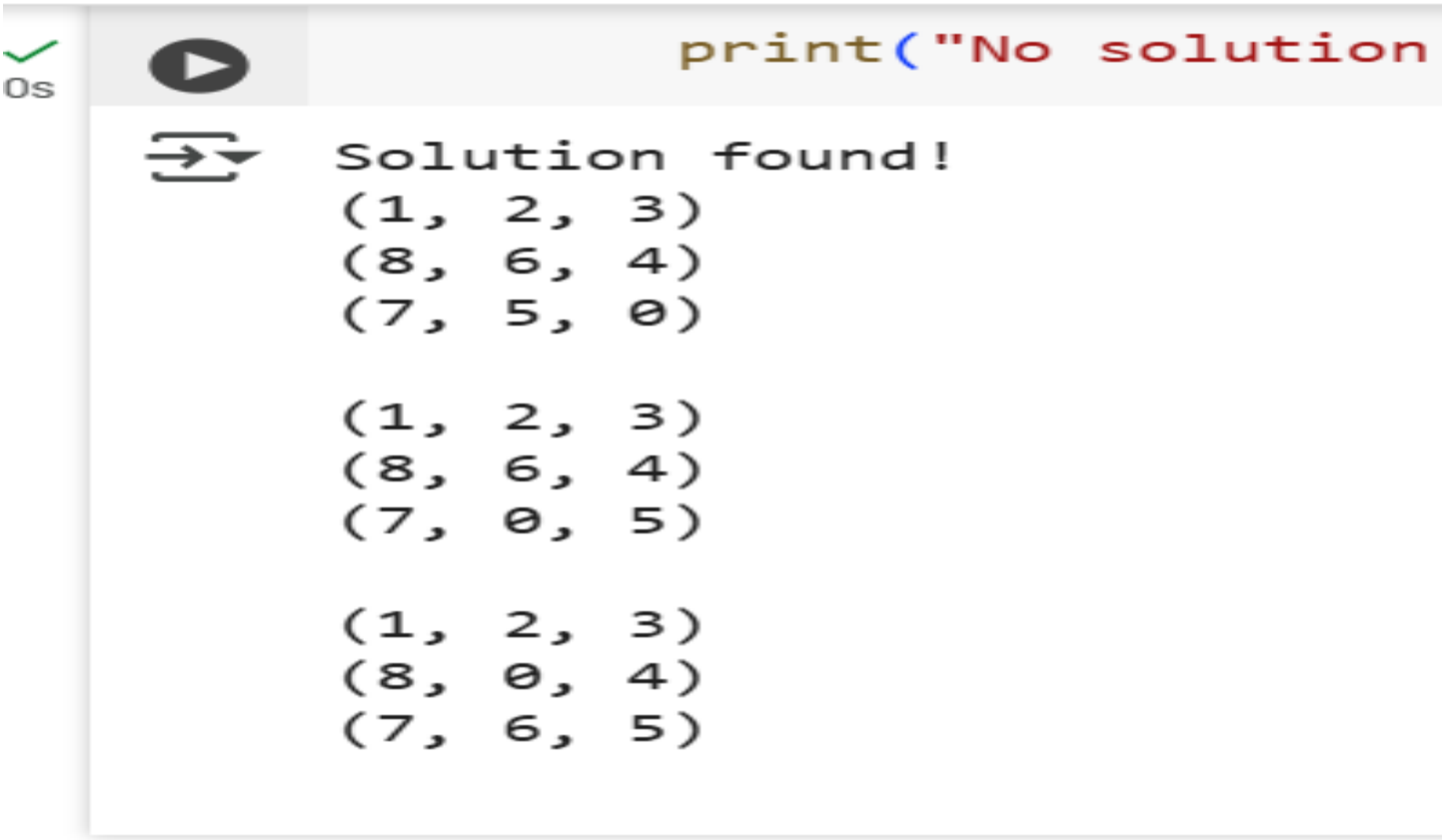
```
csharp
Copy
Solution found!
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[4, 8, 0]
[7, 6, 5]

...
```

Each puzzle configuration is printed step by step, showing the state of the puzzle after each move.



Conclusion:

This program demonstrates how the A* search algorithm can be used to solve the 8-puzzle problem efficiently. By combining the cost of the path (g) and a heuristic estimate of the remaining steps (h), the algorithm intelligently searches for the optimal solution, ensuring minimal moves to solve the puzzle.

```
import heapq
# Goal state
GOAL_STATE = ((1, 2, 3), (8, 0, 4), (7, 6, 5))
```

```
# Directions for movement (up, down, left, right)
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]


class PuzzleState:
    def __init__(self, puzzle, parent=None, move=None, g=0):
        self.puzzle = puzzle  # 2D tuple (state of the puzzle)
        self.parent = parent  # Parent PuzzleState
        self.move = move  # Move that led to this state
        self.g = g  # Cost to reach this state (g-value)
        self.h = self.manhattan_distance()  # Heuristic (h-value)
        self.f = g + self.h  # f = g + h (total cost)

    def manhattan_distance(self):
        """Calculate the Manhattan distance heuristic."""
        distance = 0
        for r in range(3):
            for c in range(3):
                value = self.puzzle[r][c]
                if value == 0:
                    continue
                goal_r, goal_c = divmod(value - 1, 3)
                distance += abs(r - goal_r) + abs(c - goal_c)
        return distance

    def get_possible_moves(self):
        """Generate possible moves (up, down, left, right) from the current
state."""
        empty_r, empty_c = [(r, c) for r in range(3) for c in range(3) if
self.puzzle[r][c] == 0][0]
        for dr, dc in MOVES:
            new_r, new_c = empty_r + dr, empty_c + dc
            if 0 <= new_r < 3 and 0 <= new_c < 3:
                new_puzzle = list(list(row) for row in self.puzzle)  # Create a
mutable copy of the puzzle
                new_puzzle[empty_r][empty_c], new_puzzle[new_r][new_c] =
new_puzzle[new_r][new_c], new_puzzle[empty_r][empty_c]
                yield PuzzleState(tuple(tuple(row) for row in new_puzzle),
parent=self, move=(new_r, new_c), g=self.g + 1)

    def __lt__(self, other):
        """For the priority queue to sort PuzzleState objects by f-value."""
```

```
        return self.f < other.f

    def __repr__(self):
        return f"PuzzleState(puzzle={self.puzzle}, f={self.f}, g={self.g},
h={self.h})"


def a_star(start_state):
    """Solve the 8-puzzle problem using the A* algorithm."""
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, start_state)

    while open_list:
        current_state = heapq.heappop(open_list)

        # If we reach the goal state, reconstruct the solution
        if current_state.puzzle == GOAL_STATE:
            solution = []
            while current_state:
                solution.append(current_state)
                current_state = current_state.parent
            return solution[::-1]  # Return the solution in the correct order

        closed_list.add(current_state.puzzle)

        # Generate possible moves and add them to the open list
        for next_state in current_state.get_possible_moves():
            if next_state.puzzle not in closed_list:
                heapq.heappush(open_list, next_state)

    return None  # No solution found


def print_solution(solution):
    """Print the sequence of moves to reach the goal."""
    for state in solution:
        for row in state.puzzle:
            print(row)
        print()
```

```python
if __name__ == "__main__":
    # New initial puzzle state (this state is different and more scrambled to
show multiple steps)
    start_state = PuzzleState(((1, 2, 3), (8, 6, 4), (7, 5, 0)))

    solution = a_star(start_state)

    if solution:
        print("Solution found!")
        print_solution(solution)
    else:
        print("No solution found.")
```