8-puzzle solver:

Name-Radhika

Class-AI(c)

Rollno.202401100300190

INTRODUCTION:

The 8-puzzle is a classic problem in artificial intelligence and search algorithms. It consists of a 3×3 grid containing eight numbered tiles and one empty space. The goal is to rearrange the tiles from a given initial state to a predefined goal state by sliding tiles into the empty space.

This problem is widely used to study state-space search techniques and evaluate the efficiency of various algorithms. It has practical applications in robot motion planning, artificial intelligence, and heuristic search strategies.

METHODOLOGY:

The 8-puzzle problem is represented as a 3×3 matrix, where tiles can move up, down, left, or right into an empty space. The goal is to rearrange tiles from an initial state to a predefined goal state.

We implemented the following search algorithms:

BFS (Breadth-First Search): Explores all possible moves level by level; guarantees the shortest path but uses high memory.

DFS (Depth-First Search): Explores deep paths first; uses less memory but may not find the shortest path.

A Search Algorithm: * Uses heuristics like Manhattan Distance and Misplaced Tiles to find an optimal solution efficiently.

```
import heapq  # Importing heapq for priority queue (min-heap) operations
import numpy as np  # Importing numpy for efficient matrix operations


class Puzzle:
    def __init__(self, board, goal):
        """
        Initializes the puzzle with the given board and goal state.

        Args:
        - board: 2D list representing the current state of the puzzle
        - goal: 2D list representing the goal state of the puzzle
        """
        self.board = np.array(board)  # Convert board to a numpy array
        self.goal = np.array(goal)  # Convert goal state to a numpy array
        self.size = len(board)  # Size of the puzzle (e.g., 3 for a 3x3 puzzle)
        self.blank_pos = tuple(np.argwhere(self.board == 0)[0])  # Find the position of the blank tile (0)

    def get_possible_moves(self):
        """
        Returns a list of possible moves by swapping the blank space with adjacent tiles.

        Returns:
        - List of tuples where each tuple contains (move direction, new puzzle state)
        """
          x, y = self.blank_pos  # Get the blank tile position
        moves = []  # List to store possible moves
        directions = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}  # Possible move directions

        for move, (dx, dy) in directions.items():
            new_x, new_y = x + dx, y + dy  # Calculate new position of the blank tile
```

```
        if 0 <= new_x < self.size and 0 <= new_y < self.size:  # Check if the move is within bounds
            new_board = self.board.copy()  # Create a copy of the board
            # Swap the blank tile with the adjacent tile
            new_board[x, y], new_board[new_x, new_y] = new_board[new_x, new_y], new_board[x, y]
            new_puzzle = Puzzle(new_board, self.goal)  # Create a new puzzle state
            moves.append((move, new_puzzle))  # Store the move and new state

    return moves

def is_goal(self):
    """
    Checks if the current puzzle state matches the goal state.

    Returns:
    - True if the board matches the goal, otherwise False
    """
    return np.array_equal(self.board, self.goal)

def manhattan_distance(self):
    """
    Computes the Manhattan distance heuristic.

    Returns:
    - Total Manhattan distance (sum of tile distances from their goal positions)
      """
    distance = 0
    for num in range(1, self.size ** 2):  # Iterate over all tiles (excluding blank)
        x, y = np.where(self.board == num)  # Get current position of tile
        goal_x, goal_y = np.where(self.goal == num)  # Get goal position of tile
        # Compute Manhattan distance
          distance += abs(x.item() - goal_x.item()) + abs(y.item() - goal_y.item())
```

```
        return distance

    def __eq__(self, other):
        """
        Checks if two puzzle states are equal.
        """
        return np.array_equal(self.board, other.board)

    def __hash__(self):
        """
        Generates a unique hash value for a puzzle state to store it in sets.
        """
        return hash(self.board.tobytes())

    def __lt__(self, other):
        """
        Comparison method for priority queue (min-heap), based on Manhattan distance.
        """
        return self.manhattan_distance() < other.manhattan_distance()

    def print_board(self):
        """
        Prints the current board state in a readable format.
        """
        print("\n".join(" ".join(map(str, row)) for row in self.board))
        print("\n" + "-" * 10)

def a_star_solver(start_board, goal_board):
    """
    Solves the 8-puzzle problem using the A* search algorithm.
```

```
    Args:
    - start_board: Initial board state as a 2D list
    - goal_board: Goal board state as a 2D list

    Returns:
    - List of moves to reach the goal state or None if unsolvable
    """
    start_puzzle = Puzzle(start_board, goal_board)  # Create the initial puzzle state
    pq = []  # Priority queue (min-heap)
    heapq.heappush(pq, (0, 0, start_puzzle, []))  # Push initial state with priority 0
    visited = set()  # Set to track visited states

    while pq:
        _, cost, current_puzzle, path = heapq.heappop(pq)  # Get the puzzle with the lowest cost

        current_puzzle.print_board()  # Print current board state

        if current_puzzle.is_goal():  # Check if the goal state is reached
            return path  # Return the path of moves

        visited.add(current_puzzle)  # Mark the current state as visited

        for move, next_puzzle in current_puzzle.get_possible_moves():  # Explore possible moves
            if next_puzzle not in visited:
                new_cost = cost + 1  # Increment cost
                priority = new_cost + next_puzzle.manhattan_distance()  # Calculate A* priority (f = g + h)
                heapq.heappush(pq, (priority, new_cost, next_puzzle, path + [move]))  # Add to priority
queue

    return None  # Return None if no solution is found
```

```
# Example Usage
start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]  # Initial board state
goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]  # Goal board state
solution = a_star_solver(start, goal)  # Solve the puzzle
print("Solution Path:", solution)  # Print the solution path
```

```
1 2 3
4 7 6
```

Run cell (Ctrl+Enter)
cell might have changed since last executed

executed by RADHIKA CSEAI
10:31AM (39 minutes ago)
executed in 0.086s

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

----------
1 2 3
4 5 6
7 0 8

----------
1 2 3
4 5 6
7 8 0

----------
Solution Path: ['R', 'D', 'L', 'L', 'U', 'R', 'D', 'R', 'U', 'L', 'L', 'D', 'R', 'R']
```