***Puzzle - Solver***

import heapq

import copy

import time

class Puzzle:

    def \_\_init\_\_(self, state, parent=None, move=None, depth=0, cost=0):

        self.state = state

        self.parent = parent

        self.move = move

        self.depth = depth

        self.cost = cost

        self.empty\_pos = self.find\_empty()

    def find\_empty(self):

        for i in range(4):

            for j in range(4):

                if self.state[i][j] == 0:

                    return (i, j)

    def get\_neighbors(self):

        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

        neighbors = []

        x, y = self.empty\_pos

        for dx, dy in moves:

            nx, ny = x + dx, y + dy

            if 0 <= nx < 4 and 0 <= ny < 4:

                new\_state = copy.deepcopy(self.state)

                new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

                neighbors.append(Puzzle(new\_state, self, (dx, dy), self.depth + 1))

        return neighbors

    def \_\_lt\_\_(self, other):

        return self.cost < other.cost  # Needed for heapq priority queue

def heuristic\_manhattan(state, goal):

    """Optimized Manhattan Distance heuristic."""

    dist = 0

    misplaced = 0

    positions = {goal[i][j]: (i, j) for i in range(4) for j in range(4)}  # Precompute goal positions

    for i in range(4):

        for j in range(4):

            tile = state[i][j]

            if tile != 0:

                goal\_x, goal\_y = positions[tile]

                dist += abs(i - goal\_x) + abs(j - goal\_y)

                if (i, j) != (goal\_x, goal\_y):  # Count misplaced tiles

                    misplaced += 1

    return dist, misplaced

def astar(start, goal, heuristic):

    start\_time = time.time()  # Track execution time

    open\_set = []

    closed\_set = set()

    start\_node = Puzzle(start)

    goal\_node = Puzzle(goal)

    start\_node.cost, start\_misplaced = heuristic(start, goal)

    heapq.heappush(open\_set, (start\_node.cost, start\_node))

    nodes\_explored = 0

    while open\_set:

        \_, current = heapq.heappop(open\_set)

        nodes\_explored += 1

        # Logging to check progress

        current\_heuristic, current\_misplaced = heuristic(current.state, goal)

        if nodes\_explored % 1000 == 0:

            elapsed\_time = time.time() - start\_time

            print(f"Nodes Explored: {nodes\_explored}, Depth: {current.depth}, Misplaced: {current\_misplaced}, Time: {elapsed\_time:.2f} sec")

        if current.state == goal\_node.state:

            path = []

            while current:

                path.append(current)

                current = current.parent

            total\_time = time.time() - start\_time

            print(f"\nSolution Found! Moves: {len(path) - 1}, Nodes Explored: {nodes\_explored}, Time: {total\_time:.2f} sec\n")

            return path[::-1], nodes\_explored  # Reverse to get correct order

        # Use frozenset to store closed states efficiently

        state\_tuple = frozenset(tuple(map(tuple, current.state)))

        closed\_set.add(state\_tuple)

        for neighbor in current.get\_neighbors():

            neighbor\_tuple = frozenset(tuple(map(tuple, neighbor.state)))

            if neighbor\_tuple in closed\_set:

                continue

            neighbor.cost, neighbor\_misplaced = heuristic(neighbor.state, goal)

            heapq.heappush(open\_set, (neighbor.cost, neighbor))

    print("No solution found.")

    return None, nodes\_explored

def print\_puzzle(state):

    """Prints the puzzle grid in a readable format."""

    for row in state:

        print(" ".join(f"{num:2}" if num != 0 else " \_" for num in row))

    print("-" \* 20)

# Test Case

initial\_state = [[11, 5, 13, 4],

                 [8, 6, 10, 2],

                 [12, 7, 3, 1],

                 [9, 14, 0, 15]]

goal\_state = [[1, 2, 3, 4],

              [5, 6, 7, 8],

              [9, 10, 11, 12],

              [13, 14, 15, 0]]

# Solve with Manhattan Distance heuristic

path\_manhattan, nodes\_manhattan = astar(initial\_state, goal\_state, heuristic\_manhattan)

# Print Solution Steps

if path\_manhattan:

    print("\nSolution Path:")

    for i, step in enumerate(path\_manhattan):

        print(f"Move {i} (Misplaced Tiles: {heuristic\_manhattan(step.state, goal\_state)[1]}):")

        print\_puzzle(step.state)

# Compare Performance

print("\nPerformance Summary:")

print(f"Manhattan Distance -> Moves: {len(path\_manhattan) - 1 if path\_manhattan else 'N/A'}, Nodes Explored: {nodes\_manhattan}")

