```python
import heapq
import copy
# ==========================================
# 1. 定义问题的初始状态和目标状态
# ==========================================
# 0 代表空格
INITIAL_STATE = [
    [5,  1,  2,  4],
    [9,  6,  3,  8],
    [13, 15, 10, 11],
    [14, 0,  7,  12]
]
GOAL_STATE = [
    [1,  2,  3,  4],
    [5,  6,  7,  8],
    [9,  10, 11, 12],
    [13, 14, 15, 0]
]
# ==========================================
# 2. 核心辅助类与函数
# ==========================================
class Node:
    def __init__(self, state, parent=None, move=None, g=0, h=0):
        self.state = state     # 当前棋盘状态
        self.parent = parent   # 父节点 (为了回溯路径)
        self.move = move       # 也就是怎么走到这一步的 (比如 "Up")
        self.g = g             # g(n): 已经走的步数 (代价)
        self.h = h             # h(n): 预计还要走的步数 (启发值)
        self.f = g + h         # f(n) = g(n) + h(n): 总评分
    # 定义比较规则，为了让优先队列知道谁的 f 更小
    def __lt__(self, other):
        return self.f < other.f
def get_pos(state, value):
    """找到某个数字(value)在棋盘中的坐标 (行, 列)"""
    for r in range(4):
        for c in range(4):
            if state[r][c] == value:
                return r, c
    return None
def manhattan_distance(state, goal):
    """
    [关键步骤] 启发函数 h(n)：计算曼哈顿距离
    计算所有数字当前位置与目标位置的横向+纵向距离之和
    """
```

```
    distance = 0
    for r in range(4):
        for c in range(4):
            value = state[r][c]
            if value != 0:  # 空格不计算距离
                target_r, target_c = get_pos(goal, value)
                # abs 是绝对值，计算横向和纵向差距
                distance += abs(r - target_r) + abs(c - target_c)
    return distance
def get_neighbors(node):
    """生成当前状态的所有可能的下一步状态"""
    neighbors = []
    state = node.state
    # 找到 0(空格)的位置
    row, col = get_pos(state, 0)

    # 定义四个移动方向：上，下，左，右 (行偏移，列偏移，动作名)
    moves = [(-1, 0, "Up"), (1, 0, "Down"), (0, -1, "Left"), (0, 1, "Right")]
    for dr, dc, move_name in moves:
        new_row, new_col = row + dr, col + dc

        # 检查是否越界
        if 0 <= new_row < 4 and 0 <= new_col < 4:
            # 复制当前状态用于修改
            new_state = [list(row) for row in state] # 深拷贝
            # 交换空格和相邻数字的位置
            new_state[row][col], new_state[new_row][new_col] = \
                new_state[new_row][new_col], new_state[row][col]

            # 将列表转回元组以便存入集合去重 (列表不可哈希)
            neighbors.append((tuple(tuple(row) for row in new_state), move_name))

    return neighbors
# ========================================
# 3. A* 算法主逻辑
# ========================================
def solve_puzzle(start, goal):
    # 将输入转换为元组格式
    start_tuple = tuple(tuple(row) for row in start)
    goal_tuple = tuple(tuple(row) for row in goal)
    # open_list: 优先队列，自动把 f 值最小的排在前面
    open_list = []

    # 初始节点的 h 值
```

```python
    h_start = manhattan_distance(start_tuple, goal_tuple)
    start_node = Node(start_tuple, None, None, 0, h_start)

    heapq.heappush(open_list, start_node)

    # closed_set: 记录走过的状态，避免死循环和重复计算
    closed_set = set()
    print("正在思考中，请稍候...")
    while open_list:
        # 1. 取出 f 值最小的节点
        current_node = heapq.heappop(open_list)

        # 2. 检查是否到达目标
        if current_node.state == goal_tuple:
            print("找到解了！")
            return current_node
        # 3. 加入已访问集合
        closed_set.add(current_node.state)
        # 4. 扩展邻居节点
        for state_data, move_name in get_neighbors(current_node):
            if state_data in closed_set:
                continue # 如果走过这个状态，就跳过
            # g(n): 新的一步，代价+1
            new_g = current_node.g + 1
            # h(n): 计算新的曼哈顿距离
            new_h = manhattan_distance(state_data, goal_tuple)

            new_node = Node(state_data, current_node, move_name, new_g, new_h)

            # 加入优先队列
            heapq.heappush(open_list, new_node)
    return None # 无解
# ========================================
# 4. 打印结果路径
# ========================================
def print_solution(node):
    path = []
    while node.parent:
        path.append(node)
        node = node.parent
    path.reverse() # 因为是从终点回溯的，所以要反转
    print(f"\n一共需要 {len(path)} 步:\n")

    # 打印初始状态
```

```
    print("Start:")
    for row in INITIAL_STATE:
        print(row)
    print("-" * 20)
    # 打印每一步
    for i, step in enumerate(path):
        print(f"Step {i+1}: Move {step.move}")
        for row in step.state:
            print(list(row))
        print(f"   (g={step.g}, h={step.h}, f={step.f})")
        print("-" * 20)
if __name__ == "__main__":
    result_node = solve_puzzle(INITIAL_STATE, GOAL_STATE)
    if result_node:
        print_solution(result_node)
    else:
        print("未找到解。")
```