A* SEARCH USING MANNHATTAN DISTANCE METHOD

```
 import heapq
from itertools import count

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        """
        board: a tuple of length 9 representing the 3x3 board,
0 is blank.
        moves: g-cost (number of moves from start)
        previous: parent PuzzleState for path reconstruction
        """
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3  # 3x3 puzzle

    def __eq__(self, other):
        return isinstance(other, PuzzleState) and self.board
== other.board

    def __hash__(self):
        return hash(self.board)

    def get_neighbors(self):
        """Return list of PuzzleState neighbors (by sliding
blank)."""
        neighbors = []
        zero_index = self.board.index(0)
        x, y = divmod(zero_index, self.size)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.size and 0 <= new_y <
self.size:
                new_zero_index = new_x * self.size + new_y
                new_board = list(self.board)
                new_board[zero_index],
new_board[new_zero_index] = (
                    new_board[new_zero_index],
                    new_board[zero_index],
                )
                neighbors.append(PuzzleState(tuple(new_board),
self.moves + 1, self))
        return neighbors

    def manhattan_distance(self, goal_positions):
        """
        Compute Manhattan distance to goal.
```

```
        goal_positions: dict mapping tile -> index in goal
board
        """
        dist = 0
        for idx, tile in enumerate(self.board):
            if tile == 0:
                continue
            goal_idx = goal_positions[tile]
            x1, y1 = divmod(idx, self.size)
            x2, y2 = divmod(goal_idx, self.size)
            dist += abs(x1 - x2) + abs(y1 - y2)
        return dist

    def __lt__(self, other):
        # Needed for heap operations when f-cost ties occur.
        # We don't compare states by any meaningful metric
here; heap tie-breaker uses counter.
        return False


def is_solvable(board):
    """Check solvability for 8-puzzle using inversion count
(blank ignored)."""
    arr = [x for x in board if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    # For 3x3 puzzle, solvable iff inversions count is even
    return inv % 2 == 0


def a_star(start, goal):
    """
    A* search using Manhattan distance heuristic.
    Returns the path (list of PuzzleState) from start to goal,
or None if not found.
    """
    # Precompute goal positions for fast Manhattan lookup
    goal_positions = {tile: idx for idx, tile in
enumerate(goal.board)}

    open_heap = []
    entry_counter = count()  # tie-breaker for heap

    start_h = start.manhattan_distance(goal_positions)
    heapq.heappush(open_heap, (start_h + start.moves,
next(entry_counter), start))
```

```
    g_score = {start: 0}
    f_score = {start: start_h}

    closed_set = set()

    while open_heap:
        current_f, _, current = heapq.heappop(open_heap)

        if current == goal:
            # reconstruct path
            path = []
            node = current
            while node is not None:
                path.append(node)
                node = node.previous
            path.reverse()
            return path

        closed_set.add(current)

        for neighbor in current.get_neighbors():
            if neighbor in closed_set:
                continue

            tentative_g = g_score[current] + 1

            if neighbor not in g_score or tentative_g <
g_score[neighbor]:
                # Update neighbor
                neighbor.previous = current
                g_score[neighbor] = tentative_g
                h =
neighbor.manhattan_distance(goal_positions)
                f = tentative_g + h
                f_score[neighbor] = f
                heapq.heappush(open_heap, (f,
next(entry_counter), neighbor))

    return None


def print_path(path):
    for state in path:
        for i in range(3):
            print(state.board[i * 3 : (i + 1) * 3])
        print()


def get_input_state(prompt):
    print(prompt)
```

```
    while True:
        try:
            values = list(
                map(int, input("Enter 9 numbers (0 for blank)
separated by spaces: ").strip().split())
            )
            if len(values) != 9 or sorted(values) !=
list(range(9)):
                raise ValueError
            return tuple(values)
        except ValueError:
            print("Invalid input. Please enter numbers 0 to 8
without duplicates.")


if __name__ == "__main__":
    # Read start and goal
    start_board = get_input_state("Enter initial state:")
    goal_board = get_input_state("Enter goal state:")

    if not is_solvable(start_board):
        print("The given initial state is NOT solvable.
Exiting.")
    else:
        start_state = PuzzleState(start_board)
        goal_state = PuzzleState(goal_board)

        solution_path = a_star(start_state, goal_state)

        if solution_path:
            print(f"Solution found in {len(solution_path) - 1}
moves:")
            print_path(solution_path)
        else:
            print("No solution found.")
```

OUTPUT:

```
Enter initial state:
Enter 9 numbers (0 for blank) separated by spaces: 2 8 3 1 6 4 7 0 5
Enter goal state:
Enter 9 numbers (0 for blank) separated by spaces: 1 2 3 8 0 4 7 6 5
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```