A* SEARCH USING MANNHATTAN DISTANCE METHOD

```
import heapq

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3

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

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

    def get_neighbors(self):
        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 misplaced_tiles(self, goal):
        count = 0
        for i in range(len(self.board)):
            if self.board[i] != 0 and self.board[i] != goal.board[i]:
                count += 1
        return count

    def __lt__(self, other):
        return False


def a_star(start, goal):
```

```
    open_set = []
    heapq.heappush(open_set, (start.misplaced_tiles(goal), start))

    g_score = {start: 0}
    f_score = {start: start.misplaced_tiles(goal)}

    closed_set = set()

    while open_set:
        current_f, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current:
                path.append(current)
                current = current.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]:
                neighbor.previous = current
                g_score[neighbor] = tentative_g
                f = tentative_g + neighbor.misplaced_tiles(goal)
                f_score[neighbor] = f
                heapq.heappush(open_set, (f, 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.")


# Take inputs
start_board = get_input_state("Enter initial state:")
goal_board = get_input_state("Enter goal state:")

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.")

print("1BM23CS333")
```

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)

1BM23CS333
```