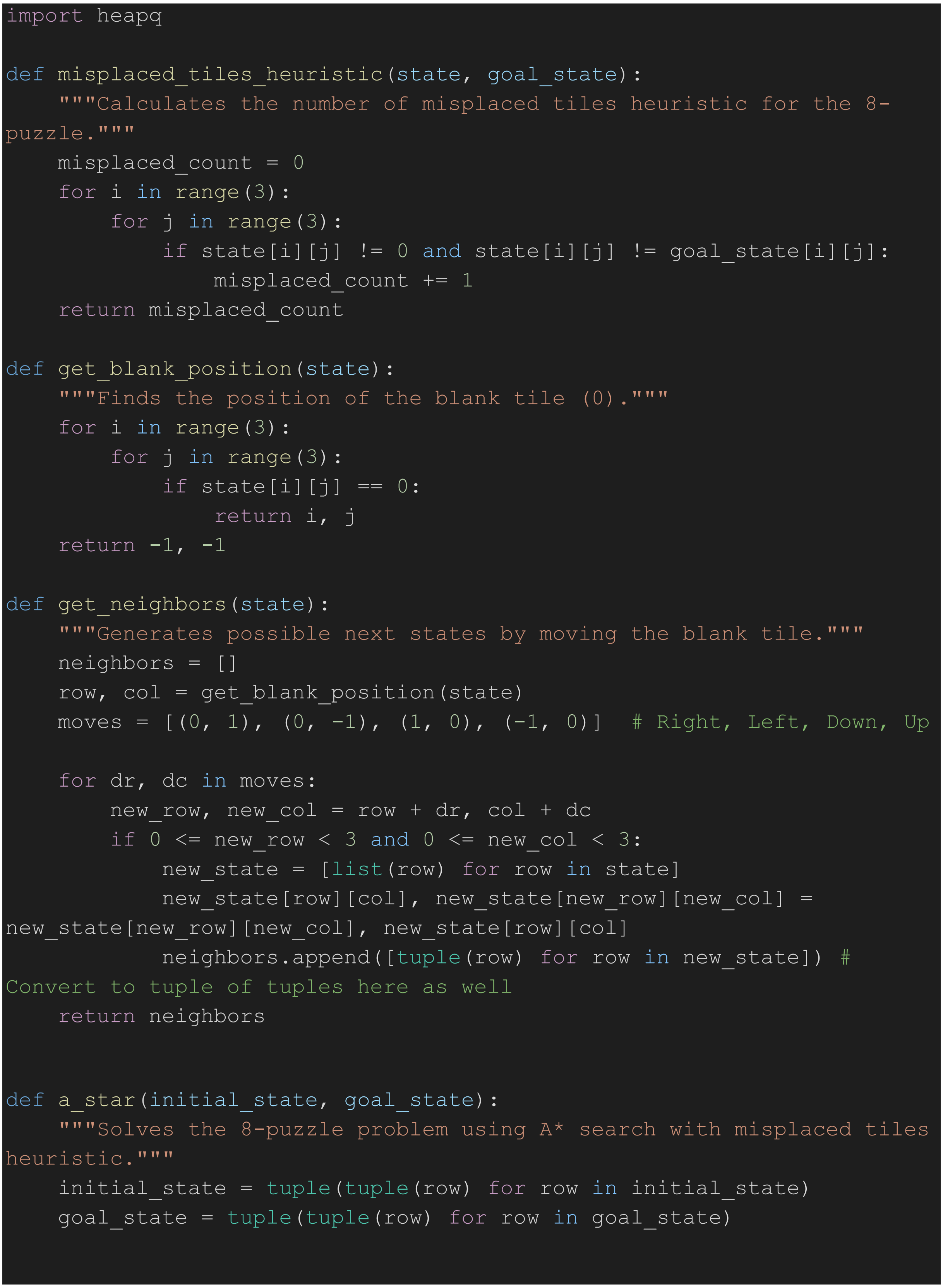
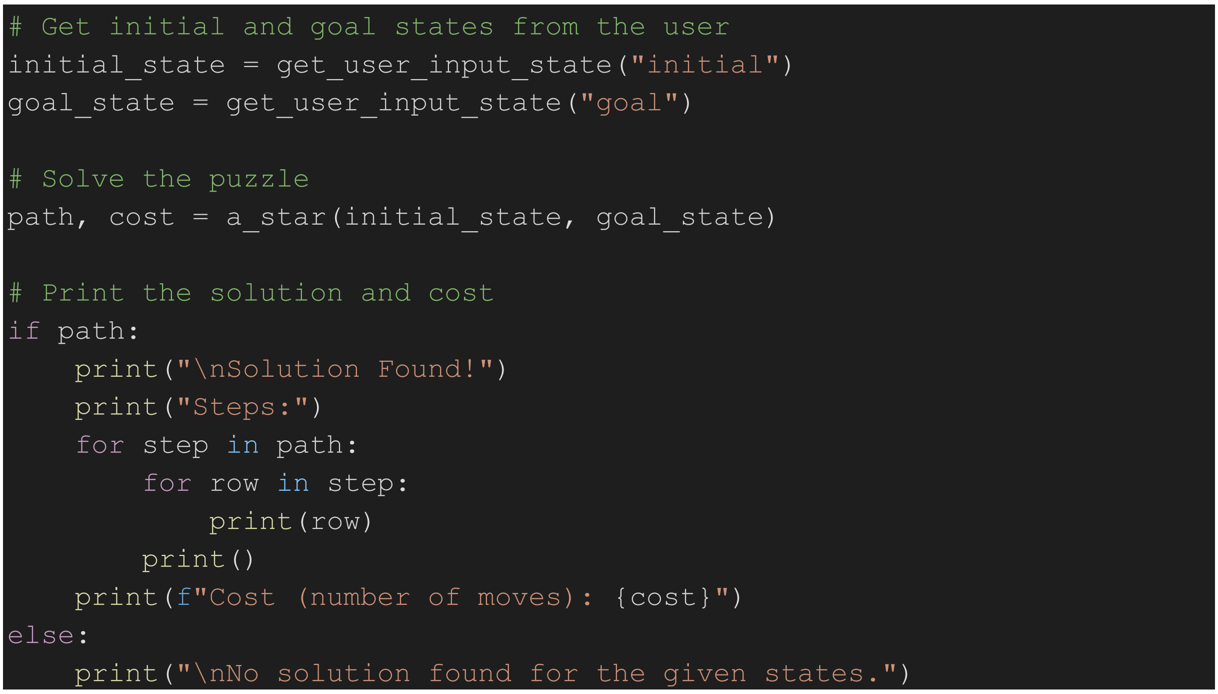
A * SEARCH USING MISPLACED TILES METHOD

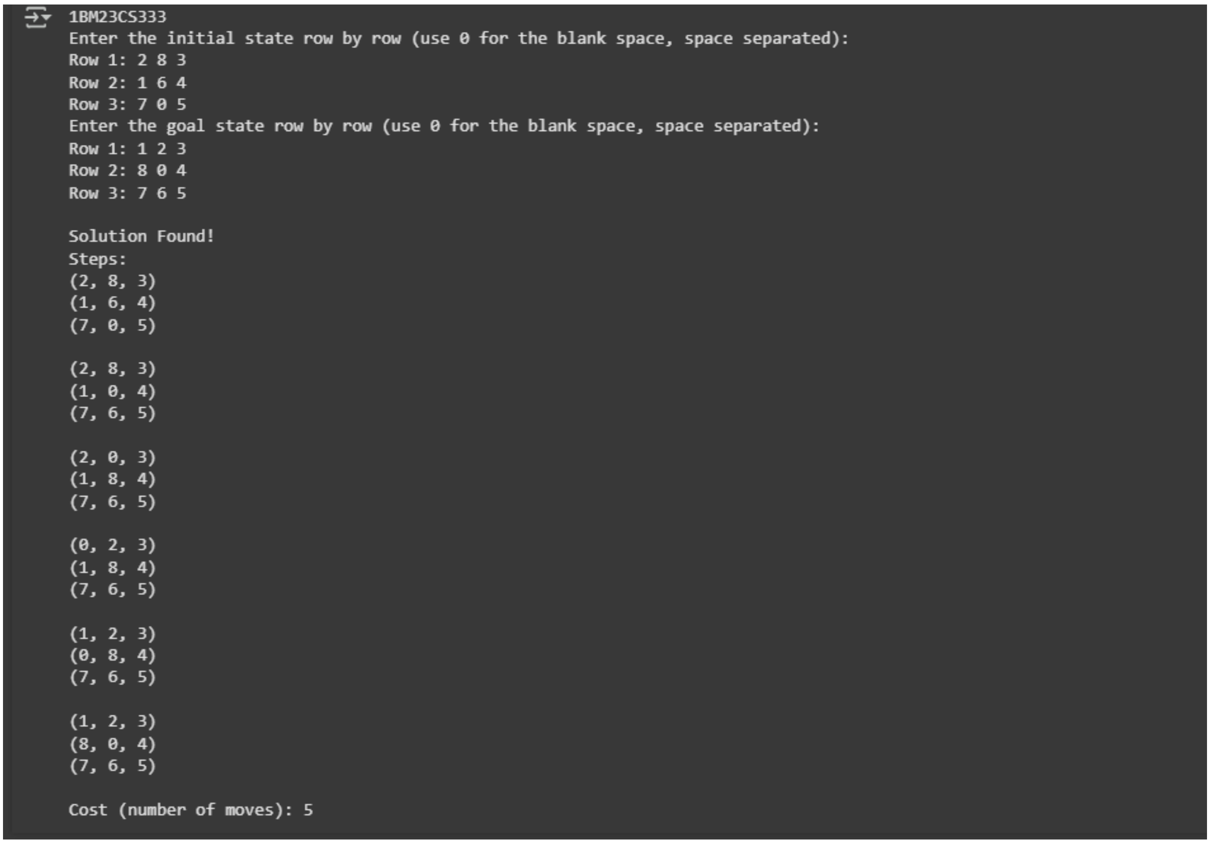


```
import heapq

def misplaced_tiles_heuristic(state, goal_state):
    """Calculates the number of misplaced tiles heuristic for the 8-
puzzle."""
    misplaced_count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced_count += 1
    return misplaced_count

def get_blank_position(state):
    """Finds the position of the blank tile (0)."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return -1, -1

def get_neighbors(state):
    """Generates possible next states by moving the blank tile."""
    neighbors = []
    row, col = get_blank_position(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up

    for dr, dc in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = [list(row) for row in state]
            new_state[row][col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[row][col]
            neighbors.append([tuple(row) for row in new_state]) #
Convert to tuple of tuples here as well
    return neighbors


def a_star(initial_state, goal_state):
    """Solves the 8-puzzle problem using A* search with misplaced tiles
heuristic."""
    initial_state = tuple(tuple(row) for row in initial_state)
    goal_state = tuple(tuple(row) for row in goal_state)
    open_set = [(misplaced_tiles_heuristic(initial_state, goal_state), 0, initial_state, [])]  # (f_cost, g_cost, state, path)
    closed_set = set()

    while open_set:
        f_cost, g_cost, current_state, path = heapq.heappop(open_set)

        if current_state == goal_state:
            return path + [current_state], g_cost  # Return path and cost

        if current_state in closed_set:
            continue

        closed_set.add(current_state)


        for neighbor_state_list in get_neighbors(current_state):
            neighbor_state = tuple(tuple(row) for row in neighbor_state_list) # Convert to tuple of tuples
            if neighbor_state not in closed_set:
                new_g_cost = g_cost + 1
                new_f_cost = new_g_cost + misplaced_tiles_heuristic(neighbor_state, goal_state)
                heapq.heappush(open_set, (new_f_cost, new_g_cost, neighbor_state, path + [current_state]))

    return None, -1  # No solution found

def get_user_input_state(state_name):
    """Gets a 3x3 puzzle state as input from the user."""
    print(f"Enter the {state_name} state row by row (use 0 for the blank space, space separated):")
    state = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").split()))
                if len(row) == 3 and all(0 <= x <= 8 for x in row):
                    state.append(row)
                    break
                else:
                    print("Invalid input. Please enter 3 numbers between 0 and 8.")
            except ValueError:
                print("Invalid input. Please enter numbers separated by spaces.")
    return state
```



```
# Get initial and goal states from the user
initial_state = get_user_input_state("initial")
goal_state = get_user_input_state("goal")

# Solve the puzzle
path, cost = a_star(initial_state, goal_state)

# Print the solution and cost
if path:
    print("\nSolution Found!")
    print("Steps:")
    for step in path:
        for row in step:
            print(row)
        print()
    print(f"Cost (number of moves): {cost}")
else:
    print("\nNo solution found for the given states.")
```

OUTPUT:



```
1BM23CS333
Enter the initial state row by row (use 0 for the blank space, space separated):
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5
Enter the goal state row by row (use 0 for the blank space, space separated):
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5

Solution Found!
Steps:
(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)

Cost (number of moves): 5
```