SANJANA SHETTY
5-E

A* ALGORITHM - MANHATTAN DISTANCE

```
#MANHATTAN DISTANCE

import heapq

# Define the goal state for the 8 puzzle
GOAL_STATE = [1, 2, 3, 4, 5, 6, 7, 8, 0]  # 0 represents the blank space

# Find the index of the blank space (0)
def find_blank(state):
    return state.index(0)

# Define valid moves based on the blank's position
def possible_moves(blank_index):
    moves = []
    # Move blank up (swap with element above)
    if blank_index > 2:
        moves.append(blank_index - 3)
    # Move blank down (swap with element below)
    if blank_index < 6:
        moves.append(blank_index + 3)
    # Move blank left (swap with element to the left)
    if blank_index % 3 > 0:
        moves.append(blank_index - 1)
    # Move blank right (swap with element to the right)
    if blank_index % 3 < 2:
        moves.append(blank_index + 1)
    return moves

# Swap blank space (0) with the target position
def swap(state, blank_index, target_index):
    new_state = state[:]
    new_state[blank_index], new_state[target_index] = 
new_state[target_index], new_state[blank_index]
    return new_state
```

```python
# Manhattan distance heuristic function
def manhattan_distance(state):
    distance = 0
    for i in range(9):
        if state[i] != 0:  # Ignore the blank space (0)
            # Find the target position in the goal state
            target_index = GOAL_STATE.index(state[i])
            # Calculate Manhattan distance
            current_row, current_col = divmod(i, 3)
            target_row, target_col = divmod(target_index, 3)
            distance += abs(current_row - target_row) + abs(current_col -
target_col)
    return distance

# A* Search Algorithm
def a_star(start_state):
    open_list = []  # priority queue to maintain nodes to be explored
    closed_list = set()  # to store already explored nodes

    # Initial state setup
    g = 0  # Cost to reach the current node (depth/level)
    h = manhattan_distance(start_state)  # Manhattan distance heuristic
    f = g + h  # Total cost (f = g + h)
 heapq.heappush(open_list, (f, g, h, start_state, []))  # Add initial node
to open list

    while open_list:
        # Get the node with the lowest f value
        f, g, h, current_state, path = heapq.heappop(open_list)

        # If current state is the goal, return the solution path
        if current_state == GOAL_STATE:
            return path + [(current_state, g, h)]

        # Add current state to the closed list
        closed_list.add(tuple(current_state))

        # Find blank's position and generate all possible moves
        blank_index = find_blank(current_state)
```

```python
        for move in possible_moves(blank_index):
            new_state = swap(current_state, blank_index, move)

            # If the new state is already explored, skip it
            if tuple(new_state) in closed_list:
                continue

            # Calculate g, h, and f for the new state
            new_g = g + 1
            new_h = manhattan_distance(new_state)
            new_f = new_g + new_h

            # Add the new state to the open list
            heapq.heappush(open_list, (new_f, new_g, new_h, new_state,
path + [(current_state, g, h)]))

    return None  # If no solution is found

# Example usage
start_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Start state of the puzzle
solution = a_star(start_state)

if solution:
    print("Solution found:")
    for state_info in solution:
        state, g, h = state_info
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print(f"Level (g): {g}, Heuristic (h): {h}, Total Cost (f = g +
h): {g + h}\n")
else:
    print("No solution exists.")
```

OUTPUT:

```
Solution found:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]
Level (g): 0, Heuristic (h): 6, Total Cost (f = g + h): 6

[1, 2, 3]
[4, 5, 0]
[6, 7, 8]
Level (g): 1, Heuristic (h): 5, Total Cost (f = g + h): 6

[1, 2, 3]
[4, 5, 8]
[6, 7, 0]
Level (g): 2, Heuristic (h): 6, Total Cost (f = g + h): 8

[1, 2, 3]
[4, 5, 8]
[6, 0, 7]
Level (g): 3, Heuristic (h): 7, Total Cost (f = g + h): 10

[1, 2, 3]
[4, 5, 8]
[0, 6, 7]
Level (g): 4, Heuristic (h): 6, Total Cost (f = g + h): 10

[1, 2, 3]
[0, 5, 8]
[4, 6, 7]
Level (g): 5, Heuristic (h): 7, Total Cost (f = g + h): 12

[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
Level (g): 6, Heuristic (h): 8, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 8]
[4, 0, 7]
Level (g): 7, Heuristic (h): 7, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
Level (g): 8, Heuristic (h): 6, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
Level (g): 9, Heuristic (h): 5, Total Cost (f = g + h): 14

[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Level (g): 10, Heuristic (h): 4, Total Cost (f = g + h): 14

[1, 2, 3]
[0, 5, 6]
```