```python
import heapq


class Node:
    def __init__(self, state, parent=None):
        self.state = state  # Current state of the board
        self.parent = parent  # Parent node
        self.g = 0  # Cost from start to this node (depth)
        self.h = 0  # Heuristic cost
        self.f = 0  # Total cost


    def __lt__(self, other):
        return self.f < other.f


def misplaced_tiles(state):
    # Count the number of misplaced tiles
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != i * 3 + j + 1:
                count += 1
    return count


def manhattan_distance(state):
    # Calculate the Manhattan distance
    distance = 0
    goal_position = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                     4: (1, 0), 5: (1, 1), 6: (1, 2),
                     7: (2, 0), 8: (2, 1)}


    for i in range(3):
        for j in range(3):
```

```
            value = state[i][j]
            if value != 0:
                target_x, target_y = goal_position[value]
                distance += abs(target_x - i) + abs(target_y - j)
    return distance

def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    x, y = zero_pos

    # Possible moves (up, down, left, right)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for move in moves:
        new_x, new_y = x + move[0], y + move[1]

        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]  # Deep copy of the current state
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]  #
Swap
            neighbors.append(new_state)

    return neighbors

def astar(start, goal, heuristic):
    open_list = []
    closed_list = set()

    start_node = Node(start)
    goal_node = Node(goal)
```

```
heapq.heappush(open_list, start_node)

while open_list:
    current_node = heapq.heappop(open_list)
    closed_list.add(tuple(map(tuple, current_node.state)))

    # Goal check
    if current_node.state == goal:
        path = []
        while current_node:
            path.append(current_node)
            current_node = current_node.parent
        return path[::-1]  # Return reversed path

    # Generate neighbors
    for neighbor_state in get_neighbors(current_node.state):
        neighbor_tuple = tuple(map(tuple, neighbor_state))

        if neighbor_tuple in closed_list:
            continue

        neighbor_node = Node(neighbor_state, current_node)
        neighbor_node.g = current_node.g + 1

        # Calculate heuristic
        if heuristic == "misplaced":
            neighbor_node.h = misplaced_tiles(neighbor_state)
        elif heuristic == "manhattan":
            neighbor_node.h = manhattan_distance(neighbor_state)
```

```
            neighbor_node.f = neighbor_node.g + neighbor_node.h

            # Check if this neighbor is already in the open list
            if any(neighbor.state == neighbor_node.state and neighbor.f <= neighbor_node.f for neighbor in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)

    return []  # Return empty path if no path found

def get_user_input():
    """Get the initial state from the user."""
    print("Enter the initial state of the board (0 for empty tile):")
    board = []
    for i in range(3):
        while True:
            try:
                row = input(f"Row {i+1} (comma-separated values 0-8): ")
                row_values = list(map(int, row.split(',')))
                if len(row_values) != 3 or any(v < 0 or v > 8 for v in row_values):
                    raise ValueError
                board.append(row_values)
                break
            except ValueError:
                print("Invalid input. Please enter three numbers (0-8) for each row.")
    return board

# Example usage
if __name__ == "__main__":
    start = get_user_input()
```

```
    goal = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    print("Using Misplaced Tiles Heuristic:")
    path_misplaced = astar(start, goal, heuristic="misplaced")
    for step in path_misplaced:
        for row in step.state:
            print(row)
        print(f"g(n): {step.g}, h(n): {step.h}, f(n): {step.f}\n")

    print("Using Manhattan Distance Heuristic:")
    path_manhattan = astar(start, goal, heuristic="manhattan")
    for step in path_manhattan:
        for row in step.state:
            print(row)
print(f"g(n): {step.g}, h(n): {step.h}, f(n): {step.f}\n")
```

```
Enter the initial state of the board (0 for empty tile):
Row 1 (comma-separated values 0-8): 1,2,3
Row 2 (comma-separated values 0-8): 4,5,6
Row 3 (comma-separated values 0-8): 0,7,8
Using Misplaced Tiles Heuristic:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
g(n): 0, h(n): 0, f(n): 0

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
g(n): 1, h(n): 1, f(n): 2

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
g(n): 2, h(n): 0, f(n): 2

Using Manhattan Distance Heuristic:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
g(n): 0, h(n): 0, f(n): 0

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
g(n): 1, h(n): 1, f(n): 2

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
g(n): 2, h(n): 0, f(n): 2
```