```
from queue import PriorityQueue

# Goal configuration of the 8-puzzle
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

# Function to calculate the heuristic (Manhattan distance)
def calculate_heuristic(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                row = (state[i][j] - 1) // 3
                col = (state[i][j] - 1) % 3
                distance += abs(i - row) + abs(j - col)
    return distance

# Function to get the possible moves for a given state
def get_possible_moves(state):
    moves = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                # Check possible moves: up, down, left, right
                if i > 0:
                    moves.append((i - 1, j))
                if i < 2:
                    moves.append((i + 1, j))
                if j > 0:
                    moves.append((i, j - 1))
                if j < 2:
                    moves.append((i, j + 1))
                return moves
# Function to perform the A* algorithm
def solve_puzzle(initial_state):
    visited = set()
    priority_queue = PriorityQueue()
    priority_queue.put((0, initial_state, 0))  # (priority, state, cost)

    while not priority_queue.empty():
        _, current_state, cost = priority_queue.get()
        visited.add(tuple(map(tuple, current_state)))  # Convert state to tuple for hashing

        if current_state == goal_state:
            return current_state, cost
```

```
        moves = get_possible_moves(current_state)

        for move in moves:
            new_state = [row[:] for row in current_state]  # Create a copy of the current state
            new_i, new_j = move
            zero_i, zero_j = find_zero_position(new_state)

            new_state[zero_i][zero_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[zero_i][zero_j]

            if tuple(map(tuple, new_state)) not in visited:
                priority = calculate_heuristic(new_state) + cost + 1
                priority_queue.put((priority, new_state, cost + 1))

    return None


# Helper function to find the position of zero (empty tile)
def find_zero_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Example usage
initial_state = [[1, 0, 3],
                 [4, 2, 5],
                 [7, 8, 6]]

result = solve_puzzle(initial_state)

if result:
    solution_state, cost = result
    print("Solution found!")
    print("Cost:", cost)
    print("Solution state:")
    for row in solution_state:
        print(row)
else:
    print("No solution found.")
```