```
import heapq
import copy

# Define Puzzle State class
class PuzzleState:

    def __init__(self, board, parent=None, move="Initial"):
        self.board = board
        self.parent = parent
        self.move = move
        self.g = 0
        self.h = 0

    def __lt__(self, other):
        return (self.g + self.h) < (other.g + other.h)

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

    def __hash__(self):
        return hash(tuple(map(tuple, self.board)))


# Goal state
goal_state = [[1,2,3],
              [4,5,6],
              [7,8,0]]


# Moves
moves = [(0,1), (1,0), (0,-1), (-1,0)]
move_names = ["Right", "Down", "Left", "Up"]


# Manhattan Distance
def calculate_manhattan_distance(state):

    distance = 0

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

            value = state.board[i][j]

            if value != 0:

                x, y = divmod(value-1, 3)

                distance += abs(x-i) + abs(y-j)

    return distance


# Print Board
def print_board(board):

    for row in board:
```

```
        print(" ".join(str(x) for x in row))

    print()


# A* Algorithm
def solve_8_puzzle(initial_state):

    open_list = []
    closed_set = set()

    initial_state.h = calculate_manhattan_distance(initial_state)

    heapq.heappush(open_list, initial_state)


    while open_list:

        current_state = heapq.heappop(open_list)


        if current_state.board == goal_state:

            path = []

            temp = current_state

            while temp.parent is not None:

                path.append((temp.move, temp.board))

                temp = temp.parent

            path.reverse()

            return path


        if current_state in closed_set:
            continue


        closed_set.add(current_state)


        # Find blank
        for i in range(3):
            for j in range(3):

                if current_state.board[i][j] == 0:
                    curr_i, curr_j = i, j


        # Generate new states
        for move, name in zip(moves, move_names):

            new_i = curr_i + move[0]
```

```
            new_j = curr_j + move[1]


            if 0 <= new_i < 3 and 0 <= new_j < 3:

                new_board = copy.deepcopy(current_state.board)

                new_board[curr_i][curr_j], new_board[new_i][new_j] = new_board[new_i][new_j],
new_board[curr_i][curr_j]

                new_state = PuzzleState(new_board, current_state, name)

                new_state.g = current_state.g + 1

                new_state.h = calculate_manhattan_distance(new_state)

                if new_state not in closed_set:

                    heapq.heappush(open_list, new_state)

    return None


# Main Function
def main():

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

    initial_state = PuzzleState(initial_board)

    solution = solve_8_puzzle(initial_state)


    if solution:

        print("Initial State:")

        print_board(initial_board)

        print("Solution found in", len(solution), "moves:\n")


        for step, (action, board) in enumerate(solution, 1):

            print("Step", step, ": Move Blank", action)

            print_board(board)

    else:

        print("No Solution Found")


if __name__ == "__main__":
    main()
```