```
import heapq
import copy

# Define the 8-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)))


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

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


# Manhattan distance heuristic
def calculate_manhattan_distance(state):
distance = 0
for i in range(3):
for j in range(3):
if state.board[i][j] != 0:
x, y = divmod(state.board[i][j] - 1, 3)
distance += abs(x - i) + abs(y - j)
return distance


# Helper function to print board
def print_board(board):
for row in board:
print(" ".join(str(x) for x in row))
print()


# A* search algorithm
def solve_8_puzzle(initial_state):
open_list = []
closed_set = set()

# Set heuristic for initial state
initial_state.h = calculate_manhattan_distance(initial_state)
heapq.heappush(open_list, initial_state)
```

```
while open_list:
current_state = heapq.heappop(open_list)

# Check if goal reached
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 tile (0)
curr_i, curr_j = -1, -1
for i in range(3):
for j in range(3):
if current_state.board[i][j] == 0:
curr_i, curr_j = i, j
break
if curr_i != -1:
break

# Generate new states
for move, move_name in zip(moves, move_names):
new_i, new_j = curr_i + move[0], curr_j + move[1]

if 0 <= new_i < 3 and 0 <= new_j < 3:
new_board = copy.deepcopy(current_state.board)

# Swap blank tile
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, move_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(f"Solution found in {len(solution)} moves:\n")
for step, (action, board) in enumerate(solution, 1):
print(f"Step {step}: Move blank {action}")
print_board(board)
else:
print("No solution found.")


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

OUTPUT:
Initial state:
1 2 4
6 5 0
7 8 3

Solution found in 15 moves:

Step 1: Move blank Down
1 2 4
6 5 3
7 8 0

Step 2: Move blank Left
1 2 4
6 5 3
7 0 8

Step 3: Move blank Up
1 2 4
6 0 3
7 5 8

Step 4: Move blank Left
1 2 4
0 6 3
7 5 8

Step 5: Move blank Up
0 2 4
1 6 3
7 5 8

Step 6: Move blank Right
2 0 4
1 6 3

7 5 8

Step 7: Move blank Right
2 4 0
1 6 3
7 5 8

Step 8: Move blank Down
2 4 3
1 6 0
7 5 8

Step 9: Move blank Left
2 4 3
1 0 6
7 5 8

Step 10: Move blank Up
2 0 3
1 4 6
7 5 8

Step 11: Move blank Left
0 2 3
1 4 6
7 5 8

Step 12: Move blank Down
1 2 3
0 4 6
7 5 8

Step 13: Move blank Right
1 2 3
4 0 6
7 5 8

Step 14: Move blank Down
1 2 3
4 5 6
7 0 8

Step 15: Move blank Right
1 2 3
4 5 6
7 8 0