Code:

import heapq

# Function to check if the current state is the goal state

def is\_goal(state, goal\_state):

return state == goal\_state

# Function to get the possible moves from the current state

def get\_neighbors(state):

neighbors = []

index = state.index(0) # Find the blank (0)

row, col = divmod(index, 3)

# Define possible moves (up, down, left, right)

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

for move in moves:

new\_row, new\_col = row + move[0], col + move[1]

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_index = new\_row \* 3 + new\_col

new\_state = list(state)

new\_state[index], new\_state[new\_index] = new\_state[new\_index], new\_state[index]

neighbors.append(tuple(new\_state))

return neighbors

# Heuristic 1: Number of misplaced tiles

def misplaced\_tiles(state, goal):

return sum(1 for i in range(9) if state[i] != 0 and state[i] != goal[i])

# Heuristic 2: Manhattan distance

def manhattan\_distance(state, goal):

distance = 0

for i in range(9):

if state[i] != 0:

current\_row, current\_col = divmod(i, 3)

goal\_row, goal\_col = divmod(goal.index(state[i]), 3)

distance += abs(current\_row - goal\_row) + abs(current\_col - goal\_col)

return distance

# Function to print the state in a 3x3 format

def print\_state(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print() # Blank line for readability

# A\* algorithm with level-wise output

def a\_star\_level\_wise(start, goal, heuristic):

# Priority queue to store the nodes to explore (f(n), g(n), state, path)

priority\_queue = []

heapq.heappush(priority\_queue, (0, 0, start, []))

visited = set()

print("Level-wise output:")

while priority\_queue:

f\_n, g\_n, current\_state, path = heapq.heappop(priority\_queue)

if current\_state in visited:

continue

visited.add(current\_state)

# Print level information

print(f"\nLevel {g\_n} (g(n) = {g\_n}):")

print(f"f(n) = {f\_n}, h(n) = {heuristic(current\_state, goal)}")

print\_state(current\_state)

if is\_goal(current\_state, goal):

return path + [current\_state]

# Gather and print all neighbors for the current level

neighbors = get\_neighbors(current\_state)

for neighbor in neighbors:

if neighbor not in visited:

g\_new = g\_n + 1

h\_new = heuristic(neighbor, goal)

f\_new = g\_new + h\_new

# Print the neighboring state for the next level

print(f" Adjacent Node (g(n) = {g\_new}, h(n) = {h\_new}, f(n) = {f\_new}):")

print\_state(neighbor)

# Push neighbors into the priority queue

heapq.heappush(priority\_queue, (f\_new, g\_new, neighbor, path + [current\_state]))

return None # If no solution found

# Function to take 8-puzzle input from the user

def input\_puzzle(prompt):

print(prompt)

puzzle = []

for i in range(3):

row = input(f"Enter row {i + 1} (3 numbers separated by spaces): ").split()

puzzle.extend([int(x) for x in row])

return tuple(puzzle)

# Main code

start\_state = input\_puzzle("Enter the start state (use 0 for the blank space):")

goal\_state = input\_puzzle("Enter the goal state (use 0 for the blank space):")

# Choose heuristic

print("Select Heuristic:")

print("1. Number of Misplaced Tiles")

print("2. Manhattan Distance")

choice = input("Enter 1 or 2: ")

if choice == '1':

heuristic = misplaced\_tiles

else:

heuristic = manhattan\_distance

print("\nSolving using A\* Search with level-wise output...")

a\_star\_solution = a\_star\_level\_wise(start\_state, goal\_state, heuristic)

if a\_star\_solution:

print("A\* Solution found! Steps:")

for i, step in enumerate(a\_star\_solution):

print(f"Step {i + 1}:")

print\_state(step)

else:

print("No solution found using A\*.")

Algorithm and state space tree:





