Code:

import heapq

def solve(src, target):

queue = []

heapq.heappush(queue, (0, src, 0, [])) # (cost, state, depth, path of moves)

visited = {}

visited[tuple(src)] = None # Store the parent of the initial state as None

while len(queue) > 0:

cost, source, depth, moves = heapq.heappop(queue)

print("--------------------------------")

print\_state(source)

print("Cost:", cost)

print("Depth:", depth)

print("Moves:", " ".join(moves))

if source == target:

total\_cost = cost + depth

print("Success with total cost:", total\_cost)

print("Path to target:", reconstruct\_path(visited, source))

return

poss\_moves\_to\_do = possible\_moves(source, visited)

for move, direction in poss\_moves\_to\_do:

move\_tuple = tuple(move)

if move\_tuple not in visited:

move\_cost = calculate\_cost(move, target)

heapq.heappush(queue, (move\_cost, move, depth + 1, moves + [direction]))

visited[move\_tuple] = tuple(source) # Record the parent state

def print\_state(state):

for i in range(9):

if i % 3 == 0:

print("\n")

if state[i] == 0:

print("\_ ", end="")

else:

print(str(state[i]) + " ", end="")

print("\n")

def possible\_moves(state, visited\_states):

b = state.index(0) # Index of empty spot

directions = []

# Add all the possible directions with corresponding moves

if b not in [0, 1, 2]: # Up

new\_state = gen(state, 'u', b)

directions.append((new\_state, 'u'))

if b not in [6, 7, 8]: # Down

new\_state = gen(state, 'd', b)

directions.append((new\_state, 'd'))

if b not in [0, 3, 6]: # Left

new\_state = gen(state, 'l', b)

directions.append((new\_state, 'l'))

if b not in [2, 5, 8]: # Right

new\_state = gen(state, 'r', b)

directions.append((new\_state, 'r'))

# Filter out visited states

return [(move, direction) for move, direction in directions if tuple(move) not in visited\_states]

def gen(state, move, b):

temp = state.copy()

if move == 'd':

temp[b], temp[b + 3] = temp[b + 3], temp[b]

elif move == 'u':

temp[b], temp[b - 3] = temp[b - 3], temp[b]

elif move == 'l':

temp[b], temp[b - 1] = temp[b - 1], temp[b]

elif move == 'r':

temp[b], temp[b + 1] = temp[b + 1], temp[b]

return temp

def calculate\_cost(state, target):

# Count the number of misplaced tiles

cost = sum(1 for i in range(len(state)) if state[i] != target[i] and state[i] != 0)

return cost

def reconstruct\_path(visited, target):

path = []

current = tuple(target)

while current is not None:

path.append(current)

current = visited.get(current) # Use get to avoid KeyError

return path[::-1] # Return reversed path

# Example usage

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

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

solve(src, target)

Output:





