Method 2.Manhatan Distance

class Node:

def \_\_init\_\_(self, state, parent=None, move=None, cost=0):

self.state = state # The current state of the puzzle

self.parent = parent # The parent node

self.move = move # The move taken to reach this state

self.cost = cost # The cost to reach this node

def heuristic(self):

"""Calculate the Manhattan distance for the current state."""

goal\_positions = {

1: (0, 0), 2: (0, 1), 3: (0, 2),

4: (1, 0), 5: (1, 1), 6: (1, 2),

7: (2, 0), 8: (2, 1), 0: (2, 2)

}

distance = 0

for i in range(len(self.state)):

for j in range(len(self.state[i])):

tile = self.state[i][j]

if tile != 0:

goal\_x, goal\_y = goal\_positions[tile]

distance += abs(goal\_x - i) + abs(goal\_y - j)

return distance

def get\_blank\_position(state):

"""Find the position of the blank (0) in the state."""

for i in range(len(state)):

for j in range(len(state[i])):

if state[i][j] == 0:

return i, j

def get\_possible\_moves(position):

"""Get possible moves from the blank position."""

x, y = position

moves = []

if x > 0: moves.append((x - 1, y, 'Down')) # Up

if x < 2: moves.append((x + 1, y, 'Up')) # Down

if y > 0: moves.append((x, y - 1, 'Right')) # Left

if y < 2: moves.append((x, y + 1, 'Left')) # Right

return moves

def generate\_new\_state(state, blank\_pos, new\_blank\_pos):

"""Generate a new state by moving the blank tile."""

new\_state = [row[:] for row in state] # Deep copy

new\_state[blank\_pos[0]][blank\_pos[1]], new\_state[new\_blank\_pos[0]][new\_blank\_pos[1]]

= \

new\_state[new\_blank\_pos[0]][new\_blank\_pos[1]],

new\_state[blank\_pos[0]][blank\_pos[1]]

return new\_state

def a\_star\_search(initial\_state):

"""Perform A\* search."""

open\_list = []

closed\_list = set()

initial\_node = Node(state=initial\_state, cost=0)

open\_list.append(initial\_node)

while open\_list:

# Sort the open list by total estimated cost (cost + heuristic)

open\_list.sort(key=lambda node: node.cost + node.heuristic())

current\_node = open\_list.pop(0)

# Print the current state, move, and heuristic value

move\_description = current\_node.move if current\_node.move else "Start"

print("Current state:")

for row in current\_node.state:

print(row)

print(f"Move: {move\_description}")

print(f"Heuristic value (Manhattan distance): {current\_node.heuristic()}")

print(f"Cost to reach this node: {current\_node.cost}\n")

if current\_node.heuristic() == 0: # Goal state reached

# Construct the path

path = []

while current\_node:

path.append(current\_node)

current\_node = current\_node.parent

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

closed\_list.add(tuple(map(tuple, current\_node.state)))

blank\_pos = get\_blank\_position(current\_node.state)

for new\_blank\_pos in get\_possible\_moves(blank\_pos):

new\_state = generate\_new\_state(current\_node.state, blank\_pos, (new\_blank\_pos[0],

new\_blank\_pos[1]))

if tuple(map(tuple, new\_state)) in closed\_list:

continue

cost = current\_node.cost + 1

move\_direction = new\_blank\_pos[2] # Get the direction of the move

new\_node = Node(state=new\_state, parent=current\_node, move=move\_direction,

cost=cost)

if new\_node not in open\_list: # Avoid duplicates in the open list

open\_list.append(new\_node)

return None # No solution found

# Example usage:

initial\_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]] # An example initial state

solution\_path = a\_star\_search(initial\_state)

if solution\_path:

print("Solution path:")

for step in solution\_path:

for row in step.state:

print(row)

print()

else:

print("No solution found.")

Output:

Current state:

[1, 2, 3]

[4, 0, 5]

[7, 8, 6]

Move: Start

Heuristic value (Manhattan distance): 2

Cost to reach this node: 0

Current state:

[1, 2, 3]

[4, 5, 0]

[7, 8, 6]

Move: Left

Heuristic value (Manhattan distance): 1

Cost to reach this node: 1

Current state:

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

Move: Up

Heuristic value (Manhattan distance): 0

Cost to reach this node: 2

Solution path:

[1, 2, 3]

[4, 0, 5]

[7, 8, 6]

[1, 2, 3]

[4, 5, 0]

[7, 8, 6]

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]