WEEK-04

Implement **Iterative Deepening Search Algorithm** using **8 puzzle problem**

import copy

class Node:

    def \_\_init\_\_(self, state, parent=None, action=None, depth=0):

        self.state = state

        self.parent = parent

        self.action = action

        self.depth = depth

    def \_\_lt\_\_(self, other):

        return self.depth < other.depth

    def expand(self):

        children = []

        row, col = self.find\_blank()

        possible\_actions = []

        if row > 0:  # Can move the blank tile up

            possible\_actions.append('Up')

        if row < 2:  # Can move the blank tile down

            possible\_actions.append('Down')

        if col > 0:  # Can move the blank tile left

            possible\_actions.append('Left')

        if col < 2:  # Can move the blank tile right

            possible\_actions.append('Right')

        for action in possible\_actions:

            new\_state = copy.deepcopy(self.state)

            if action == 'Up':

                new\_state[row][col], new\_state[row - 1][col] = new\_state[row - 1][col], new\_state[row][col]

            elif action == 'Down':

                new\_state[row][col], new\_state[row + 1][col] = new\_state[row + 1][col], new\_state[row][col]

            elif action == 'Left':

                new\_state[row][col], new\_state[row][col - 1] = new\_state[row][col - 1], new\_state[row][col]

            elif action == 'Right':

                new\_state[row][col], new\_state[row][col + 1] = new\_state[row][col + 1], new\_state[row][col]

            children.append(Node(new\_state, self, action, self.depth + 1))

        return children

    def find\_blank(self):

        for row in range(3):

            for col in range(3):

                if self.state[row][col] == 0:

                    return row, col

        raise ValueError("No blank tile found")

def depth\_limited\_search(node, goal\_state, limit):

    if node.state == goal\_state:

        return node

    if node.depth >= limit:

        return None

    for child in node.expand():

        result = depth\_limited\_search(child, goal\_state, limit)

        if result is not None:

            return result

    return None

def iterative\_deepening\_search(initial\_state, goal\_state, max\_depth):

    for depth in range(max\_depth):

        result = depth\_limited\_search(Node(initial\_state), goal\_state, depth)

        if result is not None:

            return result

    return None

def print\_solution(node):

    path = []

    while node is not None:

        path.append((node.action, node.state))

        node = node.parent

    path.reverse()

    for action, state in path:

        if action:

            print(f"Action: {action}")

        for row in state:

            print(row)

        print()

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

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

max\_depth = 5

solution = iterative\_deepening\_search(initial\_state, goal\_state, max\_depth)

if solution:

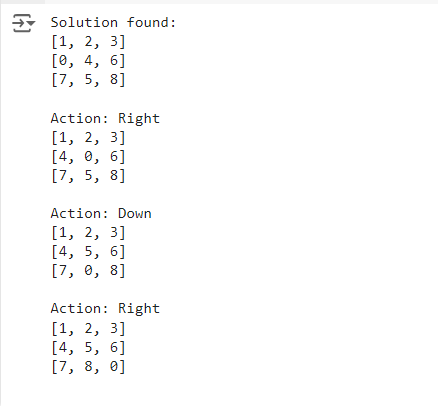
    print("Solution found:")

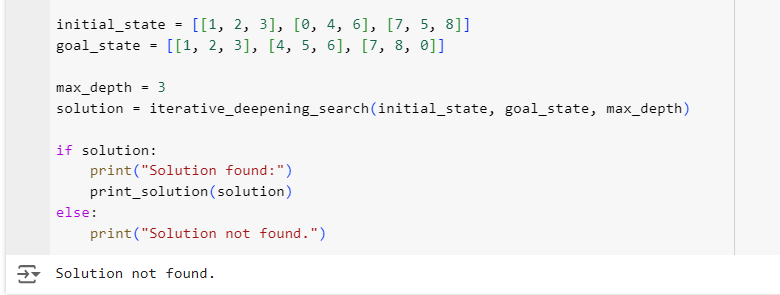
    print\_solution(solution)

else:

    print("Solution not found.")

**OUTPUT:**





Implement **Hill Climbing Search** algorithm to **solve N-queens problem**

import random

def calculate\_cost(state):

    """Calculate the number of conflicts in the current state."""

    cost = 0

    n = len(state)

    for i in range(n):

        for j in range(i + 1, n):

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

                cost += 1

    return cost

def get\_neighbors(state):

    """Generate all possible neighbors by moving each queen in its column."""

    neighbors = []

    n = len(state)

    for col in range(n):

        for row in range(n):

            if state[col] != row:  # Move the queen in column `col` to a different row

                new\_state = list(state)

                new\_state[col] = row

                neighbors.append(new\_state)

    return neighbors

def hill\_climbing(n, max\_iterations=1000):

    """Perform hill climbing search to solve the N-Queens problem."""

    current\_state = [random.randint(0, n - 1) for \_ in range(n)]

    current\_cost = calculate\_cost(current\_state)

    for iteration in range(max\_iterations):

        if current\_cost == 0:  # Found a solution

            return current\_state

        neighbors = get\_neighbors(current\_state)

        neighbor\_costs = [(neighbor, calculate\_cost(neighbor)) for neighbor in neighbors]

        next\_state, next\_cost = min(neighbor\_costs, key=lambda x: x[1])

        if next\_cost >= current\_cost:  # No improvement found

            print(f"Local maximum reached at iteration {iteration}. Restarting...")

            return None  # Restart with a new random state

        current\_state, current\_cost = next\_state, next\_cost

        print(f"Iteration {iteration}: Current state: {current\_state}, Cost: {current\_cost}")

    print(f"Max iterations reached without finding a solution.")

    return None

# Get user-defined input for the number of queens

try:

    n = int(input("Enter the number of queens (N): "))

    if n <= 0:

        raise ValueError("N must be a positive integer.")

except ValueError as e:

    print(e)

    n = 4  # Default to 4 if input is invalid

solution = None

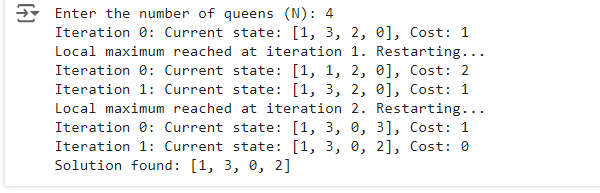
# Keep trying until a solution is found

while solution is None:

    solution = hill\_climbing(n)

print(f"Solution found: {solution}")

**OUTPUT:**

****