**Lab 4: Hill Climbing 22/10/2024**

Psedocode

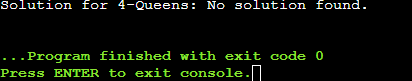
function hillClimbing8Queens():  
    // Step 1: Generate an initial random configuration  
    current\_state = random\_initial\_state() // An array of size 8 with unique column indices  
    current\_h = calculateHeuristic(current\_state)  
  
    while current\_h > 0: // While there are threatening pairs  
        neighbor\_states = generateNeighbors(current\_state)  
        next\_state = null  
        next\_h = current\_h  
  
        // Step 2: Evaluate neighbors  
        for each neighbor in neighbor\_states:  
            neighbor\_h = calculateHeuristic(neighbor)  
  
            // Step 3: Check if this neighbor is better  
            if neighbor\_h < next\_h:  
                next\_state = neighbor  
                next\_h = neighbor\_h  
  
        // Step 4: If no better neighbor is found, exit the loop  
        if next\_state is null:  
            break  
         
        // Step 5: Move to the best neighbor  
        current\_state = next\_state  
        current\_h = next\_h  
  
    if current\_h == 0:  
        return current\_state // Solution found  
    else:  
        return "No solution found within given iterations."  
  
function random\_initial\_state():  
    return random permutation of [0, 1, 2, 3, 4, 5, 6, 7] // Each index represents a row, value represents the column  
  
function calculateHeuristic(state):  
    threatening\_pairs = 0  
    for i from 0 to 7:  
        for j from i + 1 to 7:  
            if isThreatening(state[i], i, state[j], j):  
                threatening\_pairs += 1  
    return threatening\_pairs  
  
function isThreatening(col1, row1, col2, row2):  
    return (col1 == col2) or (abs(row1 - row2) == abs(col1 - col2)) // Same column or diagonal  
  
function generateNeighbors(state):  
    neighbors = []  
    for row from 0 to 7:  
        for col from 0 to 7:  
            if col != state[row]: // Don't move to the same column  
                new\_state = state.copy()  
                new\_state[row] = col // Move queen to new column  
                neighbors.append(new\_state)  
    return neighbors

CODE:

----------

import random  
  
def random\_initial\_state(size):  
    """Generates a random initial state for the given size."""  
    columns = list(range(size))  # Possible column positions  
    random.shuffle(columns)       # Randomly shuffle the columns  
    return columns                 # Each index represents a row  
  
def calculate\_heuristic(state):  
    """Calculates the number of threatening pairs of queens."""  
    threatening\_pairs = 0  
    size = len(state)  
    for i in range(size):  
        for j in range(i + 1, size):  
            if is\_threatening(state[i], i, state[j], j):  
                threatening\_pairs += 1  
    return threatening\_pairs  
  
def is\_threatening(col1, row1, col2, row2):  
    """Checks if two queens threaten each other."""  
    return (col1 == col2) or (abs(row1 - row2) == abs(col1 - col2))  
  
def generate\_neighbors(state):  
    """Generates all possible neighbor states by moving queens in the same row."""  
    neighbors = []  
    for row in range(len(state)):  
        for col in range(len(state)):  
            if col != state[row]:  # Don't move to the same column  
                new\_state = state.copy()  
                new\_state[row] = col  # Move queen to new column  
                neighbors.append(new\_state)  
    return neighbors  
  
def hill\_climbing(size):  
    """Main function to solve the n-Queens problem using hill climbing."""  
    current\_state = random\_initial\_state(size)  
    current\_h = calculate\_heuristic(current\_state)  
  
    while current\_h > 0:  # While there are threatening pairs  
        neighbors = generate\_neighbors(current\_state)  
        next\_state = None  
        next\_h = current\_h  
  
        for neighbor in neighbors:  
            neighbor\_h = calculate\_heuristic(neighbor)  
  
            # Check if this neighbor is better  
            if neighbor\_h < next\_h:  
                next\_state = neighbor  
                next\_h = neighbor\_h  
  
        if next\_state is None:  # No better neighbor found  
            break  
  
        current\_state = next\_state  
        current\_h = next\_h  
  
    if current\_h == 0:  
        return current\_state  # Solution found  
    else:  
        return "No solution found."  
  
# Example usage  
size = 4  
solution = hill\_climbing(size)  
print("Solution for 4-Queens:", solution)

Output



Observation:

