```
import random
def random_initial_state(size):
    """Generates a random initial state for the given size, placing one queen in each column."""
    return list(range(size)) # Start with one queen in each column (0 to size-1)
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(row1, col1, row2, col2):
    """Checks if two queens threaten each other."""
    return (row1 == row2) or (abs(col1 - col2) == abs(row1 - row2))
def generate_neighbors(state):
    """Generates all possible neighbor states by moving queens to different rows in the same column."""
    neighbors = []
    for col in range(len(state)):
        for row in range(len(state)):
            if row != state[col]: # Don't move to the same row
                new_state = state.copy()
                new_state[col] = row # Move queen to new row
                neighbors.append(new_state)
    return neighbors
def hill climbing(size):
    """Main function to solve the n-Queens problem using hill climbing."""
    current_state = [3, 1, 2, 0]
    current_h = calculate_heuristic(current_state)
    print(current_h)
    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)
            print(neighbor)
            print(neighbor h)
            # Check if this neighbor is better
            if neighbor_h < next_h:</pre>
                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)
```

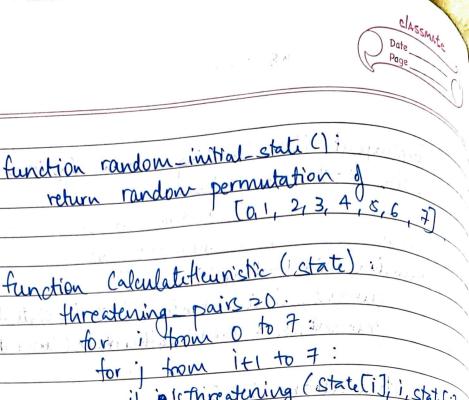
```
2
[0, 1, 2, 0]
4
[1, 1, 2, 0]
2
[2, 1, 2, 0]
3
[3, 0, 2, 0]
2
[3, 2, 2, 0]
4
[3, 3, 2, 0]
3
[3, 1, 0, 0]
3
[3, 1, 1, 0]
4
[3, 1, 3, 0]
2
[3, 1, 2, 1]
3
[3, 1, 2, 2]
2
[3, 1, 2, 3]
4
Solution for 4-Queens: No solution found.
```

Start coding or generate with AI.

Implement till Climbing search algorithm to solve N-Queene problem. function hill Climbing & Queens ()

current_state = random_initial_state() current_h = calculate Heuristic (current_state). while current - h > 0: neighbour - state = generate Neighbour sleument dats next-state = null. next_h = ament-h for each neighbour in neighbour-starte neighbour - h = calculate Heunistic (neighbour) if neighbor-h < next-h: next-state = neighbor. next-haneighbor-h if next-state is nell: current state = next state. current h = next-h if current_h = = 0. return current_state.

return "No solution found within given iterations:



function Calculateteuristic (state) threatening pairs 20.

return threatening pairs.

tunction is Threatening (col 1, 00w1, col 2, mon2): return (col 1 = 2 col 2) or (abs (no) - now 2) == abs (cd1 - co(2))

function generate Neighbors (state) neighbour 3 CJ for now from 0 to 7: tor col from 0 to 7: if col! = state (row) new_state = state copy () new_state trows wo neighbors append (new state) return neighbour.



State space tree:	
	-
Column index representation, index represents	
column and value represents on	
Waster Was	
Q	
-0 [3120]	
Q - h=2.	
a (no. g queen pair	
attre	k]
(i). [0,1,2,0] h24	
(ū) [1,1,2,0] h=2	
(ii) [21, 210] h23	
(iv) [3,0,20] and = 2.	
(V) [3.2, 2,0] & h 24	
(vi) (3,20) + h 2 3 mm and do not	
(vii) (3,1,0,0) z h z 3.	
The state of the s	
(viii) (3,1,3,0) = h = 2.	
(3,1,2,1) = h23	
(A) [311,22] 2 h22	
(21) -[31,23] h = 4.	
The state of the s	
Solution for 4 gauns: No solution found.	
Source to grand to the	
10/19	