HILL CLIMBING FOR 4 QUEENS PROBLEM

```
import random

def initial_state(n):
    """Generates a random initial state (a permutation of columns)."""
    state = list(range(n))
    random.shuffle(state)
    return state

def conflicts(state):
    """Calculates the number of conflicts (attacking queens) in a
state."""
    n = len(state)
    conflict_count = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check for conflicts in the same row (not possible with
this state representation)
            # Check for conflicts in diagonals
            if abs(state[i] - state[j]) == abs(i - j):
                conflict_count += 1
    return conflict_count

def get_neighbors(state):
    """Generates all possible neighboring states by swapping two
queens."""
    n = len(state)
    neighbors = []
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(initial_state_list, max_iterations=1000):
    """
    Implements the Hill Climbing algorithm to solve the N-Queens
problem,
    starting from a given initial state and printing states and costs.
    Continues for max_iterations even after finding a solution.

    Args:
        initial_state_list: A list representing the initial state
(column positions).
        max_iterations: The maximum number of iterations to perform.
```

```
    Returns:
        The final state reached after the iterations.
    """
    current_state = list(initial_state_list)
    current_conflicts = conflicts(current_state)

    print(f"Initial State: {current_state}, Conflicts:
{current_conflicts}")

    for i in range(max_iterations):
        if current_conflicts == 0:
            print(f"Solution found at iteration {i}: {current_state},
Conflicts: {current_conflicts}")
            # Small perturbation to encourage exploring other states if
a solution is found
            if i < max_iterations - 1:
                n = len(current_state)
                if n > 1:
                    idx1, idx2 = random.sample(range(n), 2)
                    current_state[idx1], current_state[idx2] =
current_state[idx2], current_state[idx1]
                    current_conflicts = conflicts(current_state)
                    print(f"Perturbing state after solution at
iteration {i}: {current_state}, Conflicts: {current_conflicts}")
            continue # Continue to the next iteration

        neighbors = get_neighbors(current_state)
        if not neighbors:
            print(f"No neighbors to explore at iteration {i}:
{current_state}, Conflicts: {current_conflicts}")
            # Continue printing for the remaining iterations as
requested if no neighbors
            for j in range(i + 1, max_iterations):
                print(f"Iteration {j}: {current_state}, Conflicts:
{current_conflicts}")
            return current_state


        best_neighbor = current_state
        best_conflicts = current_conflicts

        for neighbor in neighbors:
            neighbor_conflicts = conflicts(neighbor)
            if neighbor_conflicts < best_conflicts:
                best_conflicts = neighbor_conflicts
                best_neighbor = neighbor
```

```
        print(f"Iteration {i+1}: {best_neighbor}, Conflicts:
{best_conflicts}")

        if best_conflicts >= current_conflicts:
            # Local optimum reached or no better neighbor
            # Continue printing for the remaining iterations as
requested
            for j in range(i + 1, max_iterations):
                print(f"Iteration {j+1}: {current_state}, Conflicts:
{current_conflicts}")
            return current_state

        current_state = best_neighbor
        current_conflicts = best_conflicts

    print(f"Max iterations reached: {current_state}, Conflicts:
{current_conflicts}")
    return current_state

def print_board(state):
    """Prints the N-Queens board."""
    if state is None:
        print("No solution found.")
        return

    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[row] == col:
                line += " Q "
            else:
                line += " . "
        print(line)

# Get input from the user
initial_state_str = input("Enter the initial state as a comma-separated
list of column positions (e.g., 1,3,0,2 for 4 queens): ")
max_iterations_str = input("Enter the number of iterations to run: ")

try:
    initial_state_list = [int(x.strip()) for x in
initial_state_str.split(',')]
    max_iterations = int(max_iterations_str)
    n = len(initial_state_list)

    print(f"\nSolving {n}-Queens Problem with Hill Climbing from
initial state {initial_state_list} for {max_iterations} iterations:")
```

```
    final_state = hill_climbing(initial_state_list, max_iterations)

    print("\nFinal Board State:")
    print_board(final_state)

except ValueError:
    print("Invalid input. Please enter the initial state as a comma-
separated list of integers and the number of iterations as an
integer.")
except Exception as e:
    print(f"An error occurred: {e}")
```

OUTPUT:

```
Enter the initial state as a comma-separated list of column positions (e.g., 1,3,0,2 for 4 queens): 3,1,2,0
Enter the number of iterations to run: 6

Solving 4-Queens Problem with Hill Climbing from initial state [3, 1, 2, 0] for 6 iterations:
Initial State: [3, 1, 2, 0], Conflicts: 2
Iteration 1: [1, 3, 2, 0], Conflicts: 1
Iteration 2: [1, 3, 0, 2], Conflicts: 0
Solution found at iteration 2: [1, 3, 0, 2], Conflicts: 0
Perturbing state after solution at iteration 2: [3, 1, 0, 2], Conflicts: 1
Iteration 4: [1, 3, 0, 2], Conflicts: 0
Solution found at iteration 4: [1, 3, 0, 2], Conflicts: 0
Perturbing state after solution at iteration 4: [2, 3, 0, 1], Conflicts: 4
Iteration 6: [1, 3, 0, 2], Conflicts: 0
Max iterations reached: [1, 3, 0, 2], Conflicts: 0

Final Board State:
 . Q . .
 . . . Q
 Q . . .
 . . Q .
```