N-Queens Problem Report

NAME :- PRITHVI NAWADIYA
BRANCH :- CSE-AI (C)
UNIV. ROLL NO. :- 202401100300183
CLASS ROLL NO. :- 36

Introduction

The N-Queens problem is a classic combinatorial problem where the goal is to place N chess queens on an N×N chessboard such that no two queens threaten each other. This means that no two queens share the same row, column, or diagonal. The problem is an example of constraint satisfaction and has various solutions depending on the size of the board.

For example:

- **N = 4:** One valid solution is placing queens at positions (0,1), (1,3), (2,0), and (3,2).
- **N = 8:** This is the classic chess problem, and there are 92 distinct solutions.

This report demonstrates solving the N-Queens problem.

Methodology

Steps Followed:

1. **Initial State:** Start with a random permutation representing queen positions.
2. **Heuristic Function:** Calculate the number of conflicts (pairs of queens attacking each other).
3. **Neighbour Generation:** Move each queen within its row to generate possible neighbours.
4. **Selection:** Choose the neighbour with the least conflicts. If multiple, select randomly among them.
5. **Sideways Move:** Allow a limited number of sideways moves (where conflicts don't reduce) to help escape plateaus.
6. **Termination:** Stop if a solution is found (zero conflicts) or if no better neighbour exists.

Code

```
import numpy as np
import random

# Function to calculate the number of conflicting pairs of queens
def calculate_conflicts(state):
    """Calculate the number of conflicting pairs of queens."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same column or on the same diagonal
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

# Function to generate neighboring states by moving each queen within its row
def get_neighbors(state):
    """Generate neighbors by moving each queen within its row."""
    neighbors = []
    n = len(state)
    for row in range(n):
        for col in range(n):
            if col != state[row]:
```

```
            new_state = list(state)
            new_state[row] = col
            neighbors.append(new_state)
    return neighbors

# Hill Climbing algorithm with sideways moves to solve the N-Queens problem
def hill_climbing(N, max_sideways=100):
    """Hill Climbing algorithm with sideways moves."""
    state = list(np.random.permutation(N))  # Start with a random permutation
    current_conflicts = calculate_conflicts(state)
    sideways_moves = 0

    while True:
        neighbors = get_neighbors(state)
        neighbor_conflicts = [calculate_conflicts(neighbor) for neighbor in
neighbors]
        min_conflict = min(neighbor_conflicts)

        # Find all neighbors with the minimum conflicts
        best_neighbors = [neighbors[i] for i in range(len(neighbors)) if
neighbor_conflicts[i] == min_conflict]
        next_state = random.choice(best_neighbors)  # Choose randomly among
best

        # Check if we found a better state
        if min_conflict < current_conflicts:
            state = next_state
```

```
            current_conflicts = min_conflict
            sideways_moves = 0  # Reset sideways moves
        elif min_conflict == current_conflicts:
            if sideways_moves < max_sideways:
                state = next_state
                sideways_moves += 1
            else:
                break  # Stop if too many sideways moves
        else:
            break  # No better state found

        # If no conflicts, solution is found
        if current_conflicts == 0:
            return state

    return None

# Function to print the chessboard for a given state
def print_board(state):
    """Print the chessboard for a given state."""
    N = len(state)
    board = np.full((N, N), '.', dtype=str)  # Initialize an empty board
    for row in range(N):
        board[row, state[row]] = 'Q'  # Place queens on the board
    for line in board:
        print(" ".join(line))  # Print each row of the board
```

```
# Function to solve the N-Queen problem using Hill Climbing
def solve_n_queen(N):
    """Solve the N-Queen problem using Hill Climbing."""
    solution = hill_climbing(N)
    if solution:
        print(f"\nSolution for N={N}: {solution}")
        print_board(solution)
    else:
        print(f"\nNo solution found for N={N}")

# Test the algorithm for multiple values of N
test_values = [4, 8, 12]

for N in test_values:
    print(f"\n=== Solving for N={N} ===")
    solve_n_queen(N)
```

Screenshots of Output

```
=== Solving for N=4 ===

Solution for N=4: [2, 0, 3, 1]
. . Q .
Q . . .
. . . Q
. Q . .

=== Solving for N=8 ===

Solution for N=8: [3, 0, 4, 7, 5, 2, 6, 1]
. . . Q . . . .
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .

=== Solving for N=12 ===

Solution for N=12: [9, 6, 1, 7, 4, 11, 0, 10, 5, 2, 8, 3]
. . . . . . . . . Q . .
. . . . . . Q . . . . .
. Q . . . . . . . . . .
. . . . . . . Q . . . .
. . . . Q . . . . . . .
. . . . . . . . . . . Q
Q . . . . . . . . . . .
. . . . . . . . . . Q .
. . . . . Q . . . . . .
. . Q . . . . . . . . .
. . . . . . . . Q . . .
. . . Q . . . . . . . .
```