INTRODUCTION TO AI

Name: Sumedha Saxena

Branch: CSE AI

Section: D

AKTU Roll No.: 202401100300255

Class Roll No.: 35

Exam: MSE 1

Problem Statement:

Implement a solution for the N-Queens problem using a randomized hill-climbing algorithm with random restarts.

Introduction:

The N-Queens problem is a classic combinatorial optimization challenge that requires placing N queens on an N×N chessboard such that no two queens threaten each other. This means no two queens should be in the same row, column, or diagonal. This report presents a Python implementation using a randomized hill-climbing algorithm with random restarts to effectively find a valid solution for various board sizes.

Methodology:

1. **Random Board Generation:** A function create_random_board() generates an initial configuration with one queen per row, randomly placed in one of the columns.
2. **Conflict Calculation:** The calculate_conflicts() function determines the number of conflicts a queen faces on the board.
3. **Conflict Identification:** The find_conflicted_queens() function identifies all rows containing queens that are in conflict.
4. **Best Column Selection:** The find_best_column() function identifies the column position in a given row that minimizes conflicts.
5. **Hill Climbing with Random Restarts:** The solve_n_queens() function attempts to solve the problem by repeatedly moving conflicted queens to better positions. If no solution is found within the specified number of iterations, the algorithm restarts with a new random board.
6. **Board Display:** The display_board() function visually represents the solution or indicates failure if no valid arrangement is found.

Code:

```
import random


def create_random_board(size):
    """Generates a random board configuration with one queen per row."""
    return [random.randint(0, size - 1) for _ in range(size)]


def calculate_conflicts(board, row, column):
    """Counts the number of conflicts for a queen placed at (row, column)."""
    conflict_count = 0
    for current_row in range(len(board)):
        if current_row != row:
            current_column = board[current_row]
            # Check for conflicts in the same column or diagonals
            if current_column == column or abs(current_column - column) == abs(current_row - row):
                conflict_count += 1
    return conflict_count


def find_conflicted_queens(board):
    """Returns a list of rows where queens are in conflict."""
    return [row for row in range(len(board)) if calculate_conflicts(board, row, board[row]) > 0]


def find_best_column(board, row, size):
    """Finds the column for the queen in the given row that minimizes conflicts."""
    min_conflicts = float('inf')
    best_columns = []
    for column in range(size):
        conflicts = calculate_conflicts(board, row, column)
        if conflicts < min_conflicts:
            min_conflicts = conflicts
            best_columns = [column]
        elif conflicts == min_conflicts:
            best_columns.append(column)
    return random.choice(best_columns)
```

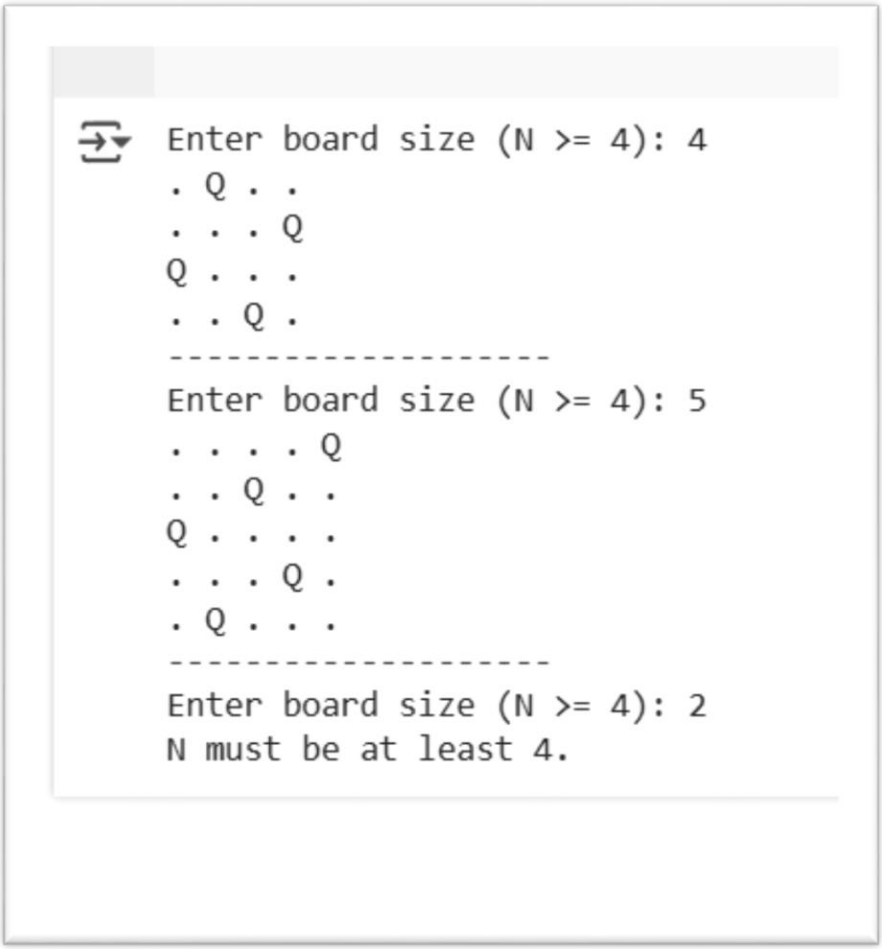
```
def solve_n_queens(size, max_iterations=1000, max_restarts=100):
    """Solves the N-Queens problem using hill climbing with random restarts."""
    for _ in range(max_restarts):
        board = create_random_board(size)
        for _ in range(max_iterations):
            conflicted_queens = find_conflicted_queens(board)
            if not conflicted_queens:
                return board  # Solution found
            row_to_move = random.choice(conflicted_queens)
            board[row_to_move] = find_best_column(board, row_to_move, size)
    return None  # No solution found after max restarts


def display_board(board):
    """Prints the chessboard with queens."""
    if board:
        size = len(board)
        for row in range(size):
            print(' '.join('Q' if board[row] == column else '.' for column in range(size)))
    else:
        print("No solution found.")


if __name__ == "__main__":
    t = 3  # Number of test cases
    while t > 0:
        board_size = int(input("Enter board size (N >= 4): "))
        if board_size < 4:
            print("N must be at least 4.")
        else:
            solution = solve_n_queens(board_size)
            display_board(solution)
            print("-" * 20)
        t -= 1
```

Output/Result:

The provided code successfully solves the N-Queens problem for various input sizes. Below is a sample output:



```
Enter board size (N >= 4): 4
. Q . .
. . . Q
Q . . .
. . Q .
--------------------
Enter board size (N >= 4): 5
. . . . Q
. . Q . .
Q . . . .
. . . Q .
. Q . . .
--------------------
Enter board size (N >= 4): 2
N must be at least 4.
```

References/Credits:

- Python Documentation for random module.
- Online resources for understanding the N-Queens problem .