N-Queens Problem Solver Report

1. Introduction

The N-Queens problem is a classic combinatorial problem where N queens must be placed on an N×N chessboard such that no two queens attack each other. This problem has significant applications in backtracking and constraint satisfaction problems.

2. Problem Statement

Given an integer N, the objective is to place N queens on an N×N chessboard so that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal.

3. Algorithm Explanation

The N-Queens problem is typically solved using a backtracking algorithm. The approach is as follows:

1. Start with an empty board.
2. Place a queen in the first available row.
3. Move to the next row and place another queen in a safe column.
4. If a conflict occurs, backtrack to the previous row and try a different column.
5. Repeat until all queens are placed successfully.
6. If a solution is found, store it; otherwise, try different placements recursively.

4. Implementation Details

The implementation is done using Python and executed in Google Colab. The core logic is built using recursion and backtracking. The solution set is printed in a matrix form, where 'Q' represents a queen and '.' represents an empty space.

Code :-

```
import random  # Importing random module to generate random numbers


# Function to print the chessboard

def print_board(board):

    for row in range(len(board)):

        line = ''

        for col in range(len(board)):

            if board[row] == col:

                line += 'Q '  # Placing a Queen (Q)

            else:
```

```
            line += '. '  # Empty space
        print(line)  # Print each row
    print()

# Function to count conflicts for each queen
def get_conflicts(board):
    conflicts = [0] * len(board)  # Initialize conflict list
    for i in range(len(board)):
        for j in range(len(board)):
            if i != j:
                # Checking if queens attack each other in the same column or diagonals
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    conflicts[i] += 1
    return conflicts  # Return conflict counts

# Hill climbing algorithm to reduce conflicts
def hill_climb(board):
    while True:
        conflicts = get_conflicts(board)  # Get conflict list
        max_conflict = max(conflicts)  # Find max conflicts

        if max_conflict == 0:
            return board  # If no conflicts, return solution

        # Get all indexes with maximum conflict
        max_conflict_indexes = [i for i in range(len(board)) if conflicts[i] == max_conflict]
        queen_to_move = random.choice(max_conflict_indexes)  # Pick a random queen with max
conflict

        best_position = board[queen_to_move]  # Store current position
        min_conflict = conflicts[queen_to_move]  # Store current conflicts
```

```
        for col in range(len(board)):
            if col != board[queen_to_move]:
                temp_board = board[:]  # Create a temporary board
                temp_board[queen_to_move] = col  # Move queen to new column
                temp_conflicts = get_conflicts(temp_board)  # Get new conflicts

                if temp_conflicts[queen_to_move] < min_conflict:  # If conflict reduces
                    min_conflict = temp_conflicts[queen_to_move]  # Update min conflict
                    best_position = col  # Update best position

        board[queen_to_move] = best_position  # Move queen to best position

        # If no improvement, restart with a new random board
        if min_conflict == conflicts[queen_to_move]:
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]

    return board  # Return final board

# Function to solve the N-Queens problem
def solve_n_queens(n):
    board = [random.randint(0, n-1) for _ in range(n)]  # Randomly place queens
    return hill_climb(board)  # Solve using hill climbing

# Taking user input for the number of queens
n = int(input("Enter the number of queens: "))
solution = solve_n_queens(n)  # Solve the problem
print("Solution:")
print_board(solution)  # Print the final board
```

5. Results & Outputs

The program successfully finds all valid solutions for a given N. For example, for N=7, one of the possible solutions is:

```
Enter the number of queens: 7
Solution:
. . . . . Q .
Q . . . . . .
. . Q . . . .
. . . . Q . .
. . . . . . Q
. Q . . . . .
. . . Q . . .
```

for N= 10, one of the possible solutions is:

```
Enter the number of queens: 10
Solution:
. . . . . . . . Q .
. . . . . Q . . . .
. . Q . . . . . . .
. . . . Q . . . . .
. Q . . . . . . . .
. . . . . . . Q . .
. . . . . . . . . Q
. . . . . . Q . . .
. . . Q . . . . . .
Q . . . . . . . . .
```

for N= 5, one of the possible solutions is:

```
Enter the number of queens: 5
Solution:
. . . . Q
. Q . . .
. . . Q .
Q . . . .
. . Q . .
```

6. Conclusion

The N-Queens problem demonstrates the power of backtracking in solving complex constraint satisfaction problems. This implementation successfully finds solutions for various N values and can be extended further for optimization or graphical visualization.