N-Queens Problem Solver

Using Hill Climbing with Random Restarts

Title Page

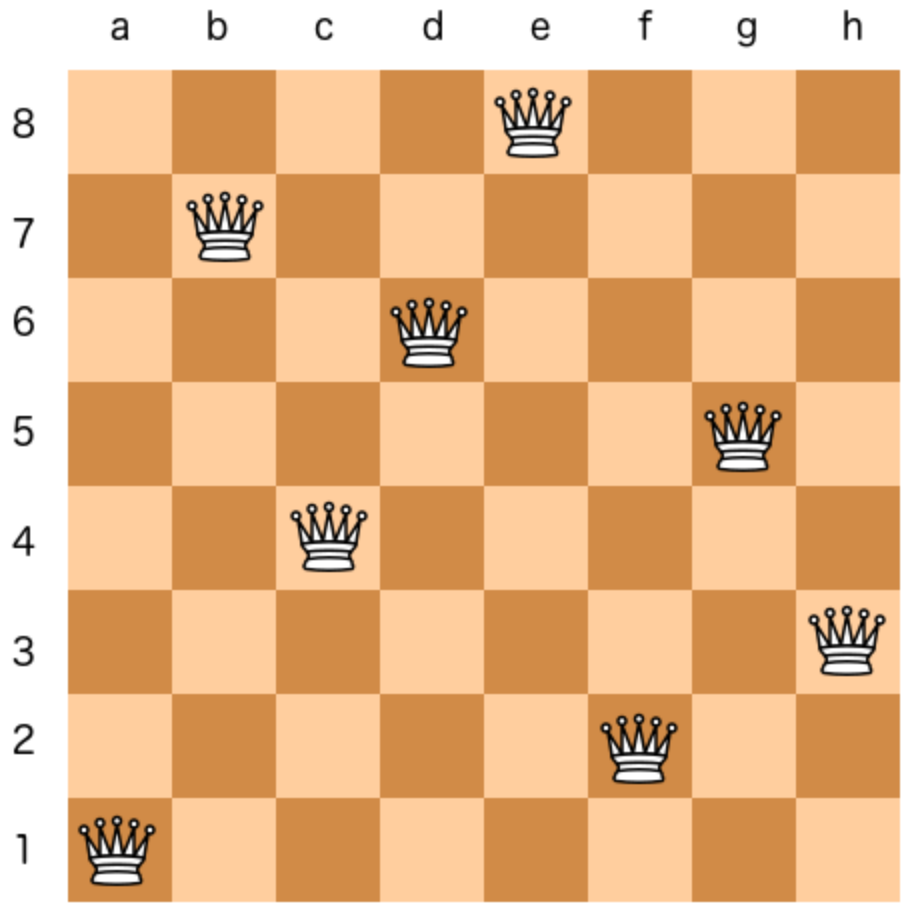
Problem Statement: Solve the N-Queens problem using Hill Climbing with Random Restarts algorithm.

Name:Nishchay Agarwal
Branch&Sec:CSEAI C
Roll No.: 202401100300163

Introduction

The N-Queens problem is a classic combinatorial problem where N chess queens must be placed on an N×N chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal.

This problem becomes increasingly complex as N grows. For N=4, there are only 2 solutions, but for N=8, there are 92 solutions. The problem is known to have solutions for all N≥4.



Methodology

The approach used in this solution is **Hill Climbing with Random Restarts**, which is a local search algorithm:

1. **Representation**:
    - The board is represented as a one-dimensional array of length N.
    - Each index represents a column, and the value at that index represents the row where a queen is placed.
    - This representation automatically ensures that no two queens share the same column.
2. **Conflict Detection**:
    - Queens conflict if they share the same row or diagonal.
    - The `calculate_conflicts()` function counts the number of attacking pairs.

3. **Hill Climbing**:
    - Start with a random initial state (random placement of queens).
    - Evaluate all possible moves by moving a queen within its column.
    - Select the move that results in the lowest number of conflicts.
    - If no better state can be found, we've reached a local minimum.
4. **Random Restart**:
    - To escape local minima, the algorithm restarts with a new random initial state.
    - A maximum number of restarts is specified (50 in the code).
5. **Solution Validation**:
    - A solution is valid when there are zero conflicts.

Code

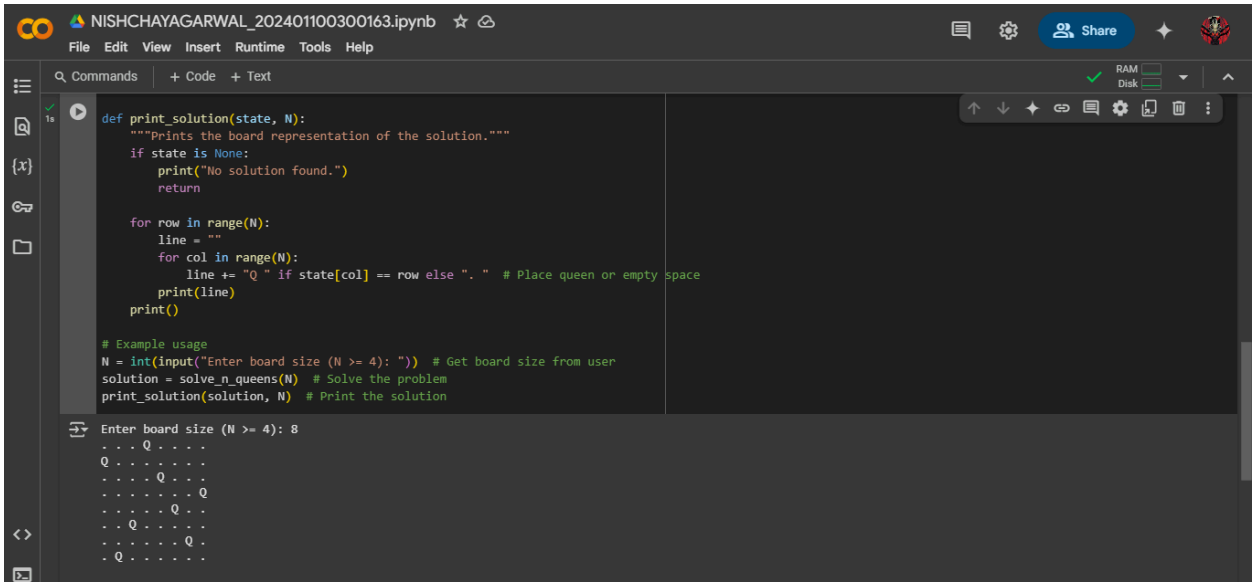
```
import random


def calculate_conflicts(state, N):
    """Returns the number of attacking queen pairs in the current state."""
    conflicts = 0
    for i in range(N):
        for j in range(i + 1, N):
```

```
            # Check if queens are in the same row or on the same diagonal
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts


def get_best_successor(state, N):
    """Finds the best successor state with the least conflicts."""
    min_conflicts = float('inf')  # Initialize with a large number
    best_state = state[:]

    for col in range(N):  # Iterate over each column
        original_row = state[col]  # Save the current row position of the queen
        for row in range(N):  # Try moving the queen to each row in the column
            if row == original_row:
                continue  # Skip if it's the current position

            new_state = state[:]
            new_state[col] = row  # Move the queen to the new row
            conflicts = calculate_conflicts(new_state, N)  # Calculate conflicts for new state

            if conflicts < min_conflicts:  # Update best state if conflicts are reduced
                min_conflicts = conflicts
                best_state = new_state
```

```
    return best_state, min_conflicts


def solve_n_queens(N, max_restarts=50):
    """Solves the N-Queens problem using Hill Climbing with Random Restarts."""
    for _ in range(max_restarts):  # Allow multiple restarts to escape local minima
        state = [random.randint(0, N - 1) for _ in range(N)]  # Generate a random initial state


        while True:
            new_state, new_conflicts = get_best_successor(state, N)  # Get the best possible
successor


            if calculate_conflicts(state, N) <= new_conflicts:
                break  # Stop if no better state is found (local minimum)


            state = new_state  # Move to the better state


        if calculate_conflicts(state, N) == 0:
            return state  # Found a valid solution with zero conflicts


    return None  # No solution found within the given restarts


def print_solution(state, N):
    """Prints the board representation of the solution."""
    if state is None:
        print("No solution found.")
```

```
        return


    for row in range(N):
        line = ""
        for col in range(N):
            line += "Q " if state[col] == row else ". "  # Place queen or empty space
        print(line)
    print()


# Example usage
N = int(input("Enter board size (N >= 4): "))  # Get board size from user
solution = solve_n_queens(N)  # Solve the problem
print_solution(solution, N)  # Print the solution
```

Output/Result



Example output for N=8:

References/Credits

1. GeeksforGeeks
2. Image credit: Wikipedia Commons - Eight Queens Animation
3. N-Queens problem description: https://en.wikipedia.org/wiki/Eight_queens_puzzle