```python
N = 4  # Size of the board

# A utility function that prints the board representation based on the state
def printBoard(state):
    # Create a 2D array initialized to zeros to represent the board
    board = [[0] * N for _ in range(N)]

    # Place a queen on the board based on the current state
    for col in range(N):
        board[state[col]][col] = 1  # Place a queen in the specified row for each column

    # Print the board row by row
    for row in board:
        print(*row)  # Unpack the row list for clean output
    print()  # Print a blank line for better readability

# A utility function that calculates the number of attacking queens
def calculateObjective(state):
    attacking = 0  # Initialize the count of attacking queens to zero

    # Check each pair of queens to see if they attack each other
    for i in range(N):
        for j in range(i + 1, N):  # Only check queens to the right to avoid double counting
            if state[i] == state[j]:  # Check if they are in the same row
                attacking += 1  # Increment the count if they are in the same row
            # Check if they are on the same diagonal
            if abs(state[i] - state[j]) == abs(i - j):
                attacking += 1  # Increment the count for diagonal attack

    return attacking  # Return the total number of attacking pairs

# This function finds the neighbor of the current state with the least objective value
def getNeighbour(state):
    best_state = state[:]  # Start with the current state as the best state
    best_objective = calculateObjective(state)  # Calculate the current objective value (attacks)

    # Loop through each queen to find a better configuration
    for i in range(N):
        original_row = state[i]  # Store the original row position of the queen

        # Try moving the queen to each possible row in the current column
        for j in range(N):
            if j != original_row:  # Skip the current row
                state[i] = j  # Move the queen to the new row
                current_objective = calculateObjective(state)  # Calculate attacks for the new state

                # Check if this new state has fewer attacks
                if current_objective < best_objective:
                    best_objective = current_objective  # Update best objective
                    best_state = state[:]  # Update best state to the new configuration

        state[i] = original_row  # Restore the original position of the queen

    return best_state  # Return the best state found

# The main function that implements the hill climbing algorithm
def hillClimbing(state):
    while True:  # Loop indefinitely until a solution is found or no better state exists
        current_objective = calculateObjective(state)  # Calculate attacks for the current state

        # Check if a solution has been found (no attacks)
        if current_objective == 0:
            print("Final board configuration:")  # Indicate that a solution has been found
            printBoard(state)  # Print the final configuration
            break  # Exit the loop

        next_state = getNeighbour(state)  # Find the best neighboring state

        # Check if no better state was found (stuck in a local minimum)
        if next_state == state:
            print("Stuck in local minimum.")  # Indicate that no improvement is possible
            printBoard(state)  # Print the current state
            break  # Exit the loop
        else:
            state = next_state  # Update the current state to the better neighbor

# Driver code
state = [1, 3, 2, 4]  # Initial position of queens, specifying their row positions

# Run the hill climbing algorithm on the initial state
hillClimbing(state)
```

```
Final board configuration:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```