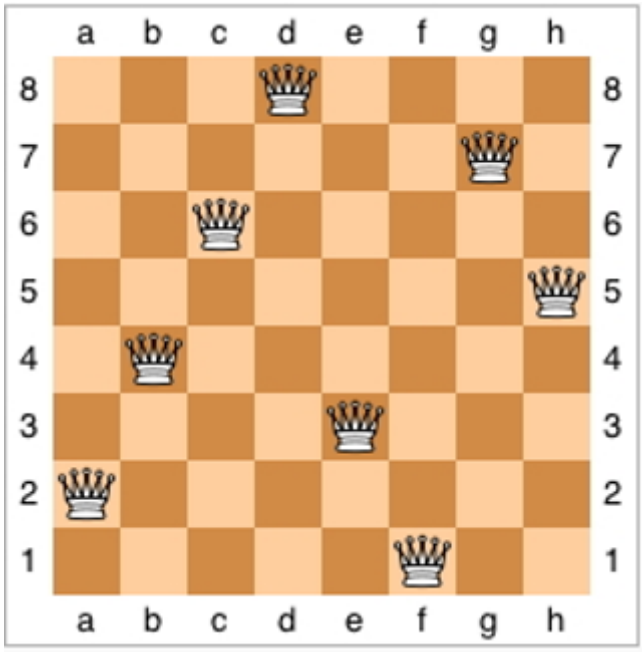
Lab 09 - Task

8-Queen Problem using Hill Climbing

The objective of the 8-queens problem is to place eight queens on a chessboard in such a way that no two queens can capture each other, which means no two queens can be on the same row, column, or diagonal.



The first observation is that, in any solution, no two queens can occupy the same column, and consequently no column can be empty. At the start we can therefore assign a column to each queen and reduce the problem to the simpler task of finding an appropriate row.

Statement of the Problem:

Consider an 8 × 8 chessboard. Place In queens on the board such that no two queens are attacking each other. The queen is the most powerful piece in chess and can attack from any distance horizontally, vertically, or diagonally. Thus, a solution must place the queens such that no two queens are in the same row, the same column, or along the same diagonal.

Implementation Steps

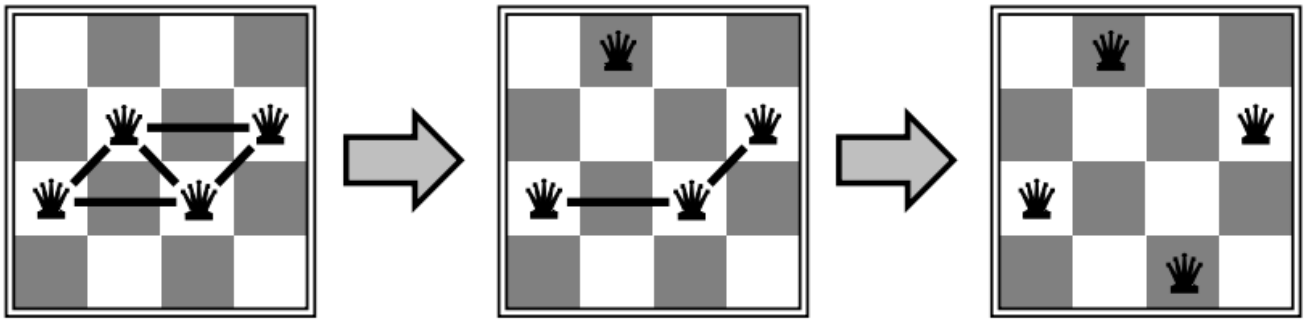
1. Create a 2D array to represent the chessboard and initialize it with zeros to represent no queen on the board.
2. Write a function that randomly places a queen in the array by setting the position index to 1.
3. Write a function to calculate the objective function, which is the number of queens attacking each other.
4. Write a function to search for the solution using the hill climbing algorithm.

```
 a. Initializing the starting solution by randomly placing queens on the bo
ard.

 b. Evaluating the objective function for the starting solution.

 c. Generating a set of candidate solutions by applying some neighborhood f
unctions to the current solution.

 d. Evaluating the objective function for each candidate solution.

 e. Choosing the best candidate solution as the new current solution.

 f. Continuing this process until a stopping criterion is met.
```

5. Return the best solution found during the search, which represents a placement of 8 queens on the chessboard such that no queen attacks another.

Note:

The heuristic cost function “h” is the number of pairs of queens that are attacking each other, either directly or indirectly.



Hill Climbing

- Start with a random configuration.
- Calculate h(board) as the number of pairs of attacking queens.
- Consider all 56 (for N = 8) possible next boards formed by moving any one queen to any other row in its column.
- Evaluate h(board) for each of these possibilities.
- If none is better than the current h, quit (local min). Else, take the best as the new current board and repeat.

This will fail 86% of the time for N = 8.

Task Solution

In [1]:

```
import random

N = 8 # number of queens
MAX_STEPS = 1000  # maximum number of steps to search for solution


def initialize_board():
    """Initialize an NxN board with zeros."""
    board = [[0 for _ in range(N)] for _ in range(N)]
    return board


def random_queen_placement(board):
    """Randomly place a queen on the board."""
    col = random.randint(0, N-1)
    row = random.randint(0, N-1)
    board[row][col] = 1
    return board


def calculate_obj(board):
    attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 1:
                # check horizontal attacks
                for k in range(n):
                    if k != j and board[i][k] == 1:
                        attacks += 1
                # check vertical attacks
                for k in range(n):
                    if k != i and board[k][j] == 1:
                        attacks += 1
                # check diagonal attacks
                for k in range(1, n):
                    if i-k >= 0 and j-k >= 0 and board[i-k][j-k] == 1:
                        attacks += 1
                    if i-k >= 0 and j+k < n and board[i-k][j+k] == 1:
                        attacks += 1
                    if i+k < n and j-k >= 0 and board[i+k][j-k] == 1:
                        attacks += 1
                    if i+k < n and j+k < n and board[i+k][j+k] == 1:
                        attacks += 1
    return attacks // 2  # divide by 2 to avoid double-counting attacks


def get_neighbors(board):
    """Generate all possible neighboring boards by moving one queen per column."""
    neighbors = []
    for col in range(N):
        row = [i for i in range(N) if board[i][col] == 1]
        if row:
            row = row[0]
            for r in range(N):
                if r != row:
                    neighbor = [row[:] for row in board]
                    neighbor[row][col] = 0
                    neighbor[r][col] = 1
```

```
                    neighbors.append(neighbor)
    return neighbors
def hill_climbing(board):
    """Search for the solution using hill climbing algorithm."""
    current_obj = calculate_obj(board)
    for i in range(MAX_STEPS):
        if current_obj == 0:
            return board

        neighbors = get_neighbors(board)
        neighbor_objs = [calculate_obj(neighbor) for neighbor in neighbors]
        min_obj = min(neighbor_objs)

        if min_obj >= current_obj:
            return board

        min_boards = [board for board, obj in zip(neighbors, neighbor_objs) if obj == mi
        board = random.choice(min_boards)
        current_obj = min_obj

    return board


# test the algorithm
board = initialize_board()

for i in range(N):
    board = random_queen_placement(board)
    print("Initial board:")
    for row in board:
        print(row)
    print("Initial objective function value:", calculate_obj(board))

    board = hill_climbing(board)


print("Solution board:")
for row in board:
    print(row)
print("Solution objective function value:", calculate_obj(board))
```

```
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
Initial objective function value: 0
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
Initial objective function value: 1
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
Initial objective function value: 1
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
Initial objective function value: 0
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
Initial objective function value: 1
Initial board:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
Initial objective function value: 0
Initial board:
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
```