8 Queens problem

Team Members:

Pietro Cicciari,
Mario Choto,
Israel Sanchez,
Raj Kumar,
Armon Lee,
Venkat Sairam Ravala

Content:

Problem Statement and Requirements

Definition of the Problem:

The 8 Queens problem is a classic puzzle in computer science and mathematics. The challenge is to place eight queens on an 8x8 chessboard such that no two queens can attack each other. In chess, a queen can move any number of squares horizontally, vertically, or diagonally. Therefore, the placement must ensure that no two queens share the same row, column, or diagonal.

Specific Requirements: We select a **goal-based agent** for solving this problem. The agent is designed to achieve a specific goal: placing all eight queens on the board without conflicts. This agent can use search algorithms to explore different configurations and find a solution that meets the goal.

PEAS Framework:

- **Performance:** Successfully place all eight queens on the board with no conflicts.
- **Environment:** The environment is the 8x8 chessboard.
- **Actuators:** Move queens to different positions on the board.
- **Sensors:** Detect the current positions of all queens and identify conflicts.

Properties of the Task Environment:

- **Fully Observable:** The agent can see the entire board and the positions of all queens at any time.

- **Deterministic:** The result of placing a queen in a particular position is predictable.
- **Sequential:** The placement of queens occurs one after another, with each step depending on the previous ones.
- **Static:** The board does not change unless the agent moves a queen.
- **Discrete:** The board has discrete positions (64 squares) because of the size of the board.
- **Known:** The rules and layout of the board are known to the agent.

Formal Specifications of the Problem:

- **Possible States:** Each state represents a configuration of queens on the board.
- **Initial State:** No queens on the board.
- **Goal State(s):** Eight queens placed on the board without conflicts.
- **Actions Available to the Agent:** Placing a queen in an empty square.
- **Transition Model:** Moving from one state to another by placing a queen.
- **Action Cost Function:** Each move has a uniform cost.
- **Heuristic Function:** Number of conflicts (queens attacking each other) or potential conflicts based on the current board state.

Defining a heuristics function.

In the 8 Queens problem, a heuristic function is used to estimate how close a given configuration of queens is to the goal state, but also where no queens can attack each other. The purpose of the heuristic is to be a basic guide to search the process by taking in account the current state and predicting the quality of potential in future states.

The heuristic function we'll define for this project here is based on counting the number of conflicts between queens. Specifically, the heuristic value (h) for a given board configuration is equal to the total number of pairs of queens that are attacking each other.

- h(state) = Number of attacking pairs of queens

How It Works:

1. **Row Conflicts:** Since our approach puts one queen per row, there are no conflicts between queens in the same row.
2. **Column Conflicts:** The heuristic checks if any two queens are placed in the same column. If two queens are in the same column, it increments the conflict count.
3. **Diagonal Conflicts**: The heuristic also checks for conflicts along both major diagonals (from top-left to bottom-right and from top-right to bottom-left). If the difference in row indices equals the difference in column indices (or their negatives), the queens are on the same diagonal and thus in conflict.

Main Idea of the Heuristic :

- The lower the value of h(state), the closer the configuration is to the goal state.
- An h(state) value of 0 means there are no conflicts, indicating a valid solution.

- This heuristic function helps algorithms like Hill-Climbing or Simulated Annealing to decide which board configurations to explore further by favoring configurations with less and less conflicts.

Approach and Design

Explanation of Our Chosen Approach Using the Algorithms Suggested in the Homework:

1. **Hill-Climb Algorithm:** This algorithm starts with a random arrangement of queens and tries to improve it by making small changes. The goal is to reduce the number of conflicts until no conflicts remain.
2. **Genetic Algorithm:** This algorithm generates lots of possible solutions. The best solutions are chosen and combined to create new solutions until a good solution is found.(sort of natural selection style)
3. **Simulated Annealing:** This algorithm starts with a random solution and makes random changes to it, sometimes accepting worse solutions to avoid getting stuck in a bad arrangement. Over time, it starts focusing more on improving the arrangement.

PSEUDOCODES:

Genetic Algorithm

```
BEGIN
FUNCTION printSolution(solution):
SET board_str = "\n"
FOR i FROM 0 TO 7 DO:
SET x = solution[i] - 1
FOR j FROM 0 TO 7 DO:
IF j == x:
APPEND '[Q]' TO board_str
ELSE:
APPEND '[ ]' TO board_str
APPEND '\n' TO board_str
PRINT board_str

FUNCTION getHeuristic(instance):
SET heuristic = EMPTY LIST
FOR i FROM 0 TO LENGTH(instance) - 1 DO:
SET j = i - 1
APPEND 0 TO heuristic
WHILE j >= 0 DO:
IF instance[i] == instance[j] OR ABS(instance[i] - instance[j]) == ABS(i - j):
INCREMENT heuristic[i]
DECREMENT j
SET j = i + 1
WHILE j < LENGTH(instance) DO:
IF instance[i] == instance[j] OR ABS(instance[i] - instance[j]) == ABS(i - j):
INCREMENT heuristic[i]
INCREMENT j
RETURN heuristic
```

```
FUNCTION getFitness(instance):
SET clashes = 0
FOR i FROM 0 TO LENGTH(instance) - 2 DO:
FOR j FROM i + 1 TO LENGTH(instance) - 1 DO:
IF instance[i] == instance[j]:
INCREMENT clashes
IF ABS(instance[j] - instance[i]) == ABS(j - i):
INCREMENT clashes
RETURN 28 - clashes

FUNCTION crossover(parent1, parent2, crossOverPoint):
SET offspring = EMPTY LIST
FOR i FROM 0 TO crossOverPoint - 1 DO:
APPEND parent1[i] TO offspring
FOR i FROM crossOverPoint TO 7 DO:
APPEND parent2[i] TO offspring
RETURN offspring

FUNCTION crossOverParents(crossOverPoint):
SET child1 = CALL crossover(parent1, parent2, crossOverPoint)
SET child2 = CALL crossover(parent2, parent1, crossOverPoint)

FUNCTION mutate(instance):
SET newChange = -1
WHILE newChange != 0 DO:
SET newChange = 0
SET tmpInstance = COPY(instance)
```

```
SET heuristics = CALL getHeuristic(tmpInstance)
SET index = INDEX OF MAX(heuristics)
SET maxFitness = CALL getFitness(tmpInstance)
FOR i FROM 1 TO 8 DO:
tmpInstance[index] = i
IF CALL getFitness(tmpInstance) > maxFitness:
SET maxFitness = CALL getFitness(tmpInstance)
SET newChange = i
SET tmpInstance = COPY(instance)
IF newChange == 0:
FOR i FROM 0 TO LENGTH(instance) - 2 DO:
FOR j FROM i + 1 TO LENGTH(instance) - 1 DO:
IF instance[i] == instance[j]:
instance[j] = RANDOM(1, 8)
ELSE:
instance[index] = newChange

FUNCTION tournament_selection(population, k = 3):
SET selected = RANDOM SAMPLE(population, k)
SORT selected BY getFitness IN ASCENDING ORDER
RETURN selected[0]

MAIN:
SET crossOverPoint = 4
SET parent1 = CALL tournament_selection(population)
SET parent2 = CALL tournament_selection(population)
SET fitnessParent1 = CALL getFitness(parent1)
SET fitnessParent2 = CALL getFitness(parent2)
```

PSEUDOCODE

- ### Simulated Annealing (SA)

```
FUNCTION SimulatedAnnealing(board, initial_temperature, cooling_rate):

    # Initialize the current state with the given board
    current_board = board
    current_cost = CalculateCost(current_board)

    # Repeat the process while the temperature is greater than 0 and the current cost is not 0
    WHILE temperature > 0 AND current_cost != 0:

        # Generate a neighboring solution by altering the position of one queen
        new_board = GenerateRandomNeighbour(current_board)
        new_cost = CalculateCost(new_board)

        # Calculate the change in cost between the new board and the current board
        delta_cost = new_cost - current_cost

        # If the new board is better (lower cost) or probabilistically acceptable, move to the new board
        IF delta_cost < 0 OR RandomValue(0, 1) < exp(-delta_cost / temperature):
            current_board = new_board
            current_cost = new_cost

        # Cool down the system by reducing the temperature
        temperature = temperature * cooling_rate

    RETURN current_board, current_cost


FUNCTION CalculateCost(board):

    # Initialize the cost to 0
    cost = 0
    n = length(board)

    # Calculate the number of conflicting pairs of queens
    FOR i = 0 TO n-1:
        FOR j = i+1 TO n-1:

            # If two queens are in the same row, increment the cost
            IF board[i] == board[j]:
                cost = cost + 1

            # If two queens are on the same diagonal, increment the cost
            IF abs(board[i] - board[j]) == abs(i - j):
                cost = cost + 1

    RETURN cost
```

```
FUNCTION GenerateRandomNeighbour(board):

    # Copy the current board to a new board (neighbor)
    neighbour = Copy(board)

    # Select a random column and a new random row for the queen in that column
    i = RandomInteger(0, length(board) - 1)
    new_position = RandomInteger(0, length(board) - 1)

    # Move the queen to the new row
    neighbour[i] = new_position

    RETURN neighbour


FUNCTION PrintBoard(board):

    n = length(board)

    # For each row in the board
    FOR i = 0 TO n-1:
        row = ["." * n]

        # Place the queen in the correct column
        row[board[i]] = "Q"

        # Print the row
        PRINT row


# Main Program

# Initialize parameters
n = 8
initial_temperature = 10000
cooling_rate = 0.995

# Create an initial random configuration of queens on the board
initial_board = [RandomInteger(0, n-1) FOR _ IN range(n)]

# Solve the 8-Queens problem using Simulated Annealing
solution_board, solution_cost = SimulatedAnnealing(initial_board, initial_temperature, cooling_rate)

# Print the initial and solution configurations
PRINT "Initial board configuration:"
CALL PrintBoard(initial_board)
PRINT "Initial cost (number of conflicts):", CalculateCost(initial_board)

PRINT "Solution board configuration:"
CALL PrintBoard(solution_board)
PRINT "Solution cost (number of conflicts):", solution_cost
```

PSEUDOCODE

Hill-climb algorithm

```
#init number of queens/size of board (8x8) --> N
#init heuristic value (goal value based on number of queens attacking each other on the board) --> H

#init row positions of queens for each column --> queenPositions
#init board

#Randomize board

#start algo Hill Climbing Algo
    #init neighbourQueenPositions
    #init neighbour of current state

        #while True:
                #copy neighbourQueenPositions to queenPositions
                #generate board

                #find the H of neighbors of current state

        #if neighbor's H is < current's H
            #optimal H = neighbor's H

                #check if board has reached global H
                        #print board
                        #break
                #check if state has hit local min
                    #Randomize state to get out of it

#print board
```

Code Implementation

- **Generic Algorithm**

```
import random

def printSolution(solution, log_file):
    log_to_file(log_file, f"Final Solution: {solution}")
    board_str = "\n"

    for i in range(8):
        x = solution[i] - 1
        for j in range(8):
            if j == x:
                board_str += '[Q]'
            else:
                board_str += '[ ]'
        board_str += '\n'

    log_to_file(log_file, board_str)

def getHeuristic(instance):
    heuristic = []
    for i in range(len(instance)):
        j = i - 1
        heuristic.append(0)
        while j >= 0:
            if instance[i] == instance[j] or (abs(instance[i] - instance[j]) == abs(i - j)):
                heuristic[i] += 1
            j -= 1
        j = i + 1
        while j < len(instance):
            if instance[i] == instance[j] or (abs(instance[i] - instance[j]) == abs(i - j)):
                heuristic[i] += 1
            j += 1
    return heuristic

def getFitness(instance):
    clashes = 0
    for i in range(len(instance) - 1):
        for j in range(i + 1, len(instance)):
            if instance[i] == instance[j]:
                clashes += 1
    for i in range(len(instance) - 1):
        for j in range(i + 1, len(instance)):
            if abs(instance[j] - instance[i]) == abs(j - i):
                clashes += 1
    return 28 - clashes
```

```
def crossover(parent1, parent2, crossOverPoint):
    offspring = []
    for i in range(crossOverPoint):
        offspring.append(parent1[i])
    for i in range(crossOverPoint, 8):
        offspring.append(parent2[i])
    return offspring

def crossOverParents(crossOverPoint):
    global parent1, parent2, child1, child2
    child1 = crossover(parent1, parent2, crossOverPoint)
    child2 = crossover(parent2, parent1, crossOverPoint)

def mutate(instance):
    newChange = -1
    while newChange != 0:
        newChange = 0
        tmpInstance = instance[:]
        heuristics = getHeuristic(tmpInstance)
        index = heuristics.index(max(heuristics))
        maxFitness = getFitness(tmpInstance)
        for i in range(1, 9):
            tmpInstance[index] = i
            if getFitness(tmpInstance) > maxFitness:
                maxFitness = getFitness(tmpInstance)
                newChange = i
            tmpInstance = instance[:]
        if newChange == 0:
            for i in range(len(instance) - 1):
                for j in range(i + 1, len(instance)):
                    if instance[i] == instance[j]:
                        instance[j] = random.randint(1, 8)
        else:
            instance[index] = newChange

def log_to_file(filename, message):
    with open(filename, 'a') as f:  # 'a' mode appends to the file
        f.write(message + '\n')
# Tournament selection
def tournament_selection(population, k=3):
    selected = random.sample(population, k)
    selected.sort(key=getFitness, reverse=False)
    #log_to_file(log_file, f"Tournament selection sorted {selected}")
    return selected[0]
```

```
if __name__ == "__main__":
    log_file = "genetic_algorithm_log.txt"
    log_to_file(log_file, "*** 8 Queens Problem using Genetic Algorithm ***")

    crossOverPoint = 4
    log_to_file(log_file, f"Initial Crossover Point: {crossOverPoint}")
    #population = [random_chromosome(8) for _ in range(50)]

    #parent1 = [random.randint(1, 8) for _ in range(8)]
    #parent2 = [random.randint(1, 8) for _ in range(8)]
    parent1 = tournament_selection(population)

    #log_to_file(log_file, f"Parent 1: {parent1}")
    parent2 = tournament_selection(population)
    log_to_file(log_file, f"Initial Parent 1: {parent1}")
    log_to_file(log_file, f"Initial Parent 2: {parent2}")

    fitnessParent1 = getFitness(parent1)
    fitnessParent2 = getFitness(parent2)

    while fitnessParent1 != 28 and fitnessParent2 != 28:
        crossOverParents(crossOverPoint)
        log_to_file(log_file, f"CrossOvered Child 1: {child1}")
        log_to_file(log_file, f"CrossOvered Child 2: {child2}")

        mutate(child1)
        mutate(child2)

        fitnessParent1 = getFitness(child1)
        fitnessParent2 = getFitness(child2)

        log_to_file(log_file, f"Fitness of new Parent 1: {fitnessParent1}")
        log_to_file(log_file, f"Fitness new Parent  2: {fitnessParent2}")

        if fitnessParent1 >= fitnessParent2:
            parent1 = child1
        else:
            parent2 = child2

        log_to_file(log_file, f"Selected Parent 1: {parent1}")
        log_to_file(log_file, f"Selected Parent 2: {parent2}")

    solution = parent1 if getFitness(parent1) == 28 else parent2
    printSolution(solution, log_file)
```

Code Implementation

Simulated Annealing (SA)

```
import random
import math

def calculate_cost(board):
    """Calculate the number of conflicting pairs of queens."""
    cost = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                cost += 1
            if abs(board[i] - board[j]) == abs(i - j):
                cost += 1
    return cost

def get_random_neighbour(board):
    """Generate a neighbouring solution by altering the position of one queen."""
    neighbour = board[:]
    i = random.randint(0, len(board) - 1)
    new_position = random.randint(0, len(board) - 1)
    neighbour[i] = new_position
    return neighbour

def simulated_annealing(board, temperature, cooling_rate):
    """Perform the Simulated Annealing algorithm."""
    current_board = board
    current_cost = calculate_cost(current_board)

    while temperature > 0 and current_cost != 0:
        new_board = get_random_neighbour(current_board)
        new_cost = calculate_cost(new_board)
        delta_cost = new_cost - current_cost

        if delta_cost < 0 or random.random() < math.exp(-delta_cost / temperature):
            current_board = new_board
            current_cost = new_cost

        temperature *= cooling_rate

    return current_board, current_cost

def print_board(board):
    """Print the chessboard with queens."""
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'
        print(' '.join(row))
    print()
```

```
# Parameters
n = 8
initial_temperature = 10000
cooling_rate = 0.995

# Initialize board with a random configuration
initial_board = [random.randint(0, n-1) for _ in range(n)]

# Solve the 8-Queens problem using Simulated Annealing
solution_board, solution_cost = simulated_annealing(initial_board, initial_temperature, cooling_rate)

# Print the result
print("Initial board configuration:")
print_board(initial_board)
print(f"Initial cost (number of conflicts): {calculate_cost(initial_board)}")

print("Solution board configuration:")
print_board(solution_board)
print(f"Solution cost (number of conflicts): {solution_cost}")
```

CODE IMPLEMENTATION

Hill-climb algorithm

```
from random import randint

N = 8 #number of queens/size of board (8x8)
H = 0 #heuristic value (goal value based on number of queens attack

#runs program
def run():
queenPositions = [0] * N #row positions of queens for each column.
board = [[0] * N for i in range(N)] #init board

#starting state
randomizeBoard(board, queenPositions)

#print init state
printBoard(board, 0)

#start algo
hillClimbingAlgo(board, queenPositions)

#generate new boards
def generateBoard(board, queenPositions):
for row in board:
for j in range(N):
row[j] = 0

for i in range(N):
board[queenPositions[i]][i] = 1

#randomizes board for init state
def randomizeBoard(board, queenPositions):
for i in range(N):
queenPositions[i] = randint(0, N-1)
board[queenPositions[i]][i] = 1

#check if board has reached global H
def checkIfSolved(position1, position2):
for i in range(N):
if (position1[i] != position2[i]):
return False

return True

#gets the heuristic value of the board
def getH(board, queenPositions):
H = 0 #heuristic value H

# Search left in the current row for a Queen
for i in range(N):
row = queenPositions[i]
col = i - 1
while col >= 0:
if board[row][col] == 1:
H += 1
break
col -= 1
```

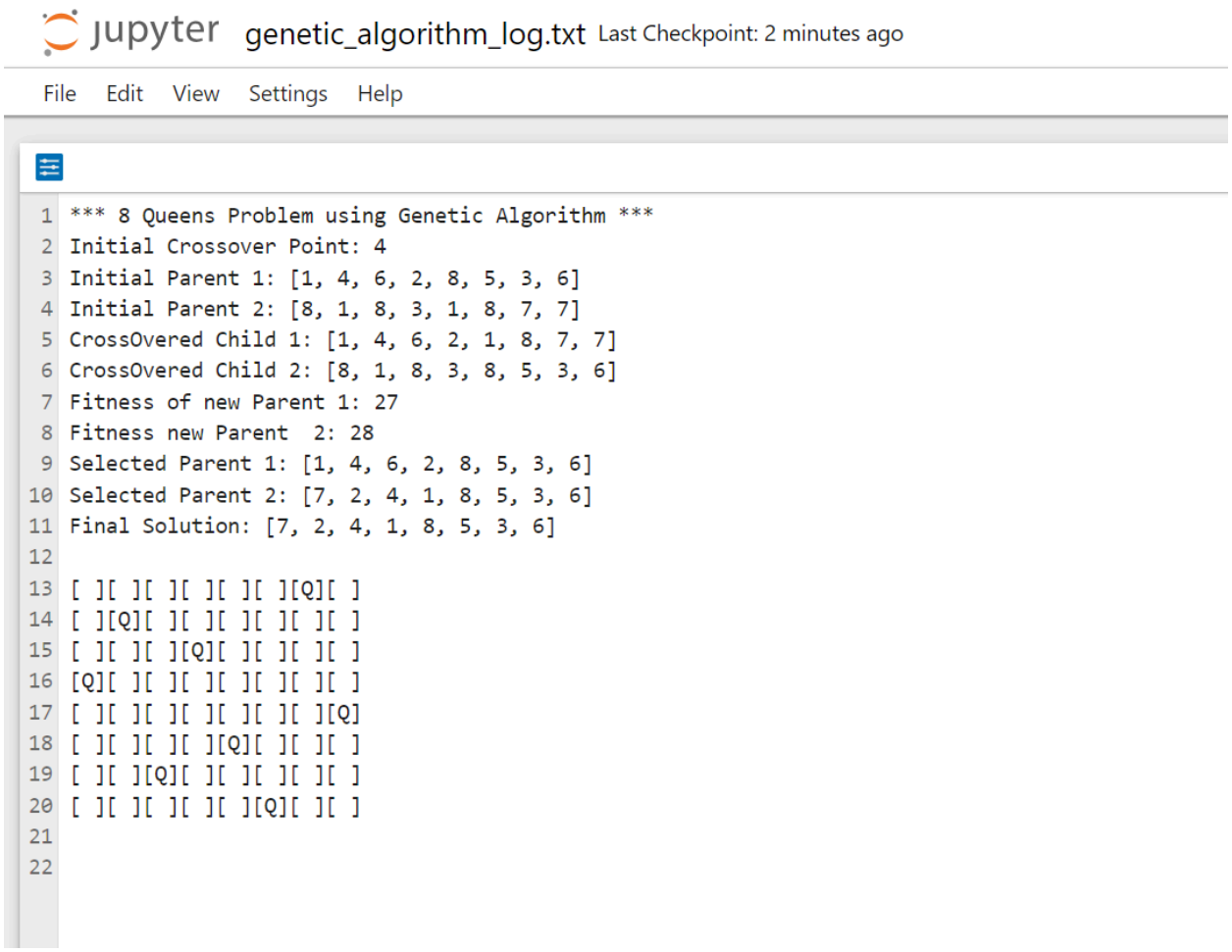
```
        # Search right in the current row for a Queen
        row = queenPositions[i]
        col = i + 1
        while col < N:
                if board[row][col] == 1:
                        H += 1
                        break
                col += 1

        # Search diagonally up-left for a Queen
        row = queenPositions[i] - 1
        col = i - 1
        while col >= 0 and row >= 0:
                if board[row][col] == 1:
                        H += 1
                        break
                col -= 1
                row -= 1

        # Search diagonally down-right for a Queen
        row = queenPositions[i] + 1
        col = i + 1
        while col < N and row < N:
                if board[row][col] == 1:
                        H += 1
                        break
                col += 1
                row += 1

        # Search diagonally down-left for a Queen
        row = queenPositions[i] + 1
        col = i - 1
        while col >= 0 and row < N:
                if board[row][col] == 1:
                        H += 1
                        break
                col -= 1
                row += 1

        # Search diagonally up-right for a Queen
        row = queenPositions[i] - 1
        col = i + 1
        while col < N and row >= 0:
                if board[row][col] == 1:
                        H += 1
                        break
                col += 1
                row -= 1

:urn int(H / 2)  #return heuristic value H
```

```
#get neighbor of current state
def getNeighbor(board, queenPositions):

        #init current board as min H
        minBoard = [[0] * N for i in range(N)]
        bestQueenPositions = [0] * N

        #make board and save queen positions
        bestQueenPositions = queenPositions.copy()
        generateBoard(minBoard, bestQueenPositions)

        #calculate heuristic
        H = getH(minBoard, bestQueenPositions)

        #temp board to get H from neighbors
        tempBoard = [[0] * N for i in range(N)]
        tempQueenPositions = [0] * N

        #make board and save queen positions
        tempQueenPositions = queenPositions.copy()
        generateBoard(tempBoard, tempQueenPositions)

        #loop through to find optimal H
        for i in range(N):
                for j in range(N):
                        #do not check current state
                        if (j != queenPositions[i]) :
                                # Move queen's position on board
                                tempQueenPositions[i] = j
                                tempBoard[j][i] = 1
                                tempBoard[queenPositions[i]][i] = 0

                                # Calculating H for new position
                                temp = getH(tempBoard, tempQueenPositions)

                                #check if temp H is smaller than current H
                                #if true then update H
                                if (temp <= H) :
                                        H = temp
                                        bestQueenPositions = tempQueenPositions.copy()
                                        generateBoard(minBoard, bestQueenPositions)

                                #reset back to current
                                tempBoard[tempQueenPositions[i]][i] = 0
                                tempQueenPositions[i] = queenPositions[i]
                                tempBoard[queenPositions[i]][i] = 1
```

```
        #copy board with best H
        for i in range(N):
                queenPositions[i] = bestQueenPositions[i]

        #generate board with best H
        generateBoard(board, queenPositions)

def hillClimbingAlgo(board, queenPositions):

        neighbourQueenPositions = [0] * N #init neighbourQueenPositions
        neighbourBoard = [[0] * N for i in range(N)] #init neighbour of current state

        while True:
                queenPositions = neighbourQueenPositions.copy()
                generateBoard(board, queenPositions)

                #find neighbors of current state
                getNeighbor(neighbourBoard, neighbourQueenPositions)

                #State has hit global min
                if (checkIfSolved(queenPositions, neighbourQueenPositions)):
                        printBoard(board, 1)
                        print("H:", H)
                        break
                #State has hit local min
                #Randomize state to get out of it
                elif (getH(board, queenPositions) == getH( neighbourBoard,neighbourQueenPositions)):
                        neighbourQueenPositions[randint(0, N-1)] = randint(0, N-1)
                        generateBoard(neighbourBoard, neighbourQueenPositions)

#prints board
def printBoard(board, val):

        #if val is 0 then board is not solved
        if (val == 0):
                print("\n")
                print("Unsolved Board")
        #if val is 1 then board is solved
        else:
                print("\n")
                print("Solved Board")

        for i in range(N):
                print(*board[i])

run()
```

Input and Output

- **Generic Algorithm**



Input and Output

- **Hill Climb Algorithm**

```
Unsolved Board
1 0 1 0 0 0 0 0
0 0 0 1 0 0 0 1
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 1 0 0 0 1 0 0
0 0 0 0 0 0 1 0


Solved Board
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
H: 0
```

- **Simulated Annealing**

```
Initial board configuration:
. . . . . . Q .
. Q . . . . . .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . Q . . . . .
. Q . . . . . .
. . . . . Q . .

Initial cost (number of conflicts): 6
Solution board configuration:
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .

Solution cost (number of conflicts): 0
```

Test Cases

Different Kinds of Tests:

- **Standard Tests:**

  For this test, we use the normal 8x8 chessboard. The goal is to place all eight queens on the board so they don't fight each other. This is the regular way to solve the problem. We hope the algorithms will find a way to put all the queens on the board safely.

- **Edge Tests:**

  Edge tests are special and not like the regular ones. For example, we could, if needed, use smaller boards, like a 4x4 board, or bigger ones, like a 16x16 board. Even though

the problem is usually with an 8x8 board, using different sizes helps us see if the algorithms can still work well. On smaller boards, we might find answers faster, and on bigger boards, it might take more time obviously.

- **Tricky Tests:**

  These tests are harder because we started with some queens already on the board, and it's not easy to add the rest without problems. We want to see if the algorithms can still find a way to put all the queens without them fighting (being on each other sight in any way). This helps us see which algorithm is the best when things get tough.

What We Expect to See:

- **Standard Tests:**

  On the regular 8x8 board, we think the algorithms will find a way to place all the queens so they don't fight. The Hill-Climb Algorithm might get stuck sometimes, but the other two (Genetic Algorithm and Simulated Annealing) should find good solutions if they are set up well.

- **Edge Tests:**

  On small boards like 4x4, we think the algorithms will find answers quickly, but if the board is too small, they might not find a solution. On big boards like 16x16, it might take longer to find answers, and we might need to change how the algorithms work a bit. But with good settings, they should still work.

- **Tricky Tests:**

  In these hard tests, where some queens are already placed in a tough way, we think the Hill-Climb Algorithm might have a hard time and need to start over a few times. The Genetic Algorithm might take longer if the first solutions aren't different enough, but it

should find an answer if it keeps going. Simulated Annealing might do well here if it can explore different ideas before deciding on the final solution.

Performance Analysis

Completeness:

Completeness means if the algorithm can always find a solution when one exists.

- **Hill-Climb Algorithm:** Not complete - This algorithm is not complete because sometimes it gets stuck in a place where it can't find a better solution, even though there is one.
- **Genetic Algorithm:** Not complete - This algorithm usually finds a solution, but it's not 100% sure. It depends on how long it runs and how different the starting solutions are.
- **Simulated Annealing:** Not complete - This one is better at finding a solution than Hill-Climbing because it sometimes accepts worse moves to try to find a better solution later. But it's still not guaranteed to always work.

Cost Optimality:

Cost optimality is about how good the algorithm is at finding a solution without wasting too many resources like time and memory.

- **Hill-Climb Algorithm:** Not Optimal - It's quick and doesn't use that much memory, but sometimes it has to start over if it gets stuck.
- **Genetic Algorithm:** Not Optimal - It uses more time and memory because it works with many possible solutions at once, but it usually finds good solutions.

- **Simulated Annealing:** Not Optimal - This algorithm is somewhere in the middle. It uses more resources than Hill-Climbing but less than the Genetic Algorithm.

Time Complexity:

Time complexity tells us how long the algorithm takes as the problem gets bigger.

- **Hill-Climb Algorithm:** O(n) - n being the number of evaluations or steps required
- **Genetic Algorithm:**O(gpe) – g being the number of generations, p being the population size, and e the evaluation time.
- **Simulated Annealing:**O(nm) – n the number of iterations and m being the average time to evaluate and generate neighboring solutions.

Space Complexity:

Space complexity means how much memory the algorithm uses.

- **Hill-Climb Algorithm:** O(1) - It uses little memory because it only keeps track of a few positions at a time.
- **Genetic Algorithm:** O(pn) – Where p is population size and n is the size of the solution representation.
- **Simulated Annealing:** O(1) - It uses a bit more memory than Hill-Climbing because it needs to keep track of temperature and other settings.

Discussion of Trade-offs:

- **Hill-Climb Algorithm:** It's fast and uses little memory, but it's not always reliable.
- **Genetic Algorithm:** It's strong at finding solutions but needs more time and memory.

- **Simulated Annealing:** It's a good balance between finding solutions and using resources, but it can be slow if not set up correctly.

Error Handling

Explanation of Error Handling:

For the Simulated Annealing algorithm a simple try except statement was added to encompass many sections of the code. Error messages were implemented for calculating cost, printing the board, generating neighbors, and for function in main. This ensured that an error message would be provided if there was a critical error in code execution. As for the Hill-Climbing and Genetic algorithms, error handling was not implemented as the algorithm's implementation required no input to be provided by the user. As this was the case any errors could be sufficiently addressed while implementing the algorithm and additional checks would not be necessary.

Optimization

Optimization Techniques Applied:

- **Hill-Climb Algorithm:** We can add random restarts or allow small backward steps to avoid getting stuck in a bad position.
- **Genetic Algorithm:** We can use techniques like keeping the best solutions, mixing solutions differently, or changing mutation rates to make it work better.

- **Simulated Annealing:** We can adjust how quickly the temperature drops or change how it decides to accept worse solutions to make it find better results.

Comparison with Other Possible Approaches: Other ways to solve the 8 Queens problem, like Depth-First Search with backtracking, can also work. These methods may be easier to understand and always find a solution, but they might not be as fast or efficient as the algorithms we have used here in this exercise.

Assumptions and Limitations

Assumptions Made During Implementation:

- We assumed that the chessboard is always 8x8, and the problem is only about this size and our code only represents such dimensions.
- We also assume we are only looking for one solution, even though there are many possible solutions.

Known Limitations of the Solution:

- **Hill-Climb Algorithm:** It can get stuck and not find a solution.
- **Genetic Algorithm:** It needs a lot of resources, and its performance depends on the settings we use.
- **Simulated Annealing:** It can get stuck if the cooling process is not done correctly.

Results and Conclusion

Summary of Results: We successfully tested the three algorithms on the 8 Queens problem. The Hill-Climb Algorithm was quick but not always successful because sometimes it got stuck. The Genetic Algorithm found good solutions but needed more resources and someshow heated up our CPU's. Simulated Annealing was in between, offering a good balance of finding solutions without using too much time or memory.

Final Thoughts on Effectiveness and Efficiency: We conclude that each algorithm used to solve the 8 Queens problem has its pros and cons. The ideal algorithm to use depends on what's more important for the problem like speed, reliability, or resource use. For general use, Simulated Annealing might be the best choice because it balances everything without risking unwanted behavior.