CHARLOTTE

Solving N-Queens Using Simulated Annealing

Extra Credit

Submitted by:

Varad Deshpande **801243927**

Summary of the Program

- The simulated annealing program begins with the user entering the no of queens required on board and the no of iterations on which the program should iterate.
- An initial board configuration is selected at random by placing one queen in each row.
- The heuristic value is calculated based on the no of attacks present on the board.
- The initial temperature is set to a high value(4000 in the code)
- A random new state is selected and its heuristic value is calculated.
- The heuristic values of current state and new state are compared and accepted based on the following criteria:
    - The heuristic value of the new state is less than the current state.
    - The heuristic is greater than the current state but is within the bounds of the temperature.
- If these requirements are satisfied, we move to the new state and continue.
- We continue this process till the heuristic of current state is zero, i.e., we have found a solution or the no of iterations are completed.
- For every iteration, we decrease the temperature. The temperature thus decreases exponentially as the algorithm progresses.
- By doing this, we avoid getting trapped in local minima early on in the algorithm but start to hone in on a viable solution by the time the algorithm ends.

Code

```
import copy
from random import Random, randint, random
import numpy as np
import math
import decimal
import random



class Node:
   def __init__(self, board, hvalue) -> None:
       self.board = board
       self.hvalue = hvalue

   def printBoard(self):
       print(np.matrix(self.board))
```

```
class NQueen:
   def __init__(self, size, iterations) -> None:
       self.size = size
       self.iterations = iterations
       self.temperature = 4000
       self.sch = 0.99

   def generateRandomPosition(self):
       numOfQueens = self.size
             mat = [['-' for i in range(numOfQueens)] for j in
range(numOfQueens)]
       for y in range(numOfQueens):
           queenPositionx = randint(0, self.size - 1)
           for x in range(numOfQueens):
               if (x == queenPositionx):
                   mat[x][y] = 'Q'
               else:
                   mat[x][y] = '-'
       return mat

   def heuristicFunction(self, board):
       heuristicValueRows = 0
       size = len(board)
       for x in board:
           count = x.count('Q')
           if(count > 1):
                           heuristicValueRows = heuristicValueRows +
(sum(range(x.count('Q'))))
       heuristicValueDiagonal = 0
       for y in range(size-1, -1, -1):
           numQueen = 0
           col = 0
           row = y
           while(col < size and row < size):
```

```
                value = board[row][col]
                if(value == 'Q'):
                    numQueen = numQueen + 1
                col = col + 1
                row = row + 1
            if(numQueen > 1):
                    heuristicValueDiagonal = heuristicValueDiagonal +
(sum(range(numQueen)))
        for y in range(size-1, 0, -1):
            numQueen = 0
            col = y
            row = 0
            while(col < size and row < size):
                value = board[row][col]
                if(value == 'Q'):
                    numQueen = numQueen + 1
                col = col + 1
                row = row + 1
            if(numQueen > 1):
                    heuristicValueDiagonal = heuristicValueDiagonal +
(sum(range(numQueen)))
        for y in range(size-1, -1, -1):
            numQueen = 0
            col = y
            row = 0
            while(col >= 0 and row < size):
                value = board[row][col]
                if(value == 'Q'):
                    numQueen = numQueen + 1
                col = col - 1
                row = row + 1
            if(numQueen > 1):
                    heuristicValueDiagonal = heuristicValueDiagonal +
(sum(range(numQueen)))
        for y in range(size-1, 0, -1):
```

```
            numQueen = 0
            col = size-1
            row = y
            while(col >= 0 and row < size):
                value = board[row][col]
                if(value == 'Q'):
                    numQueen = numQueen + 1
                col = col - 1
                row = row + 1
            if(numQueen > 1):
                    heuristicValueDiagonal = heuristicValueDiagonal +
(sum(range(numQueen)))
        heuristicValue = heuristicValueRows + heuristicValueDiagonal
        return heuristicValue

    def hillClimb(self):
        board = NQueen.generateRandomPosition(self)
        current = Node(board, 0)
        print("Initial Board Configuration:")
        current.printBoard()
        current.hvalue = self.heuristicFunction(current.board)
        solution = False
        for count in range(0, self.iterations):
            self.temperature = self.temperature * self.sch
            tempBoard = NQueen.generateRandomPosition(self)
            tempHeuristic = self.heuristicFunction(tempBoard)
            tempNode = Node(tempBoard, tempHeuristic)
            difference = current.hvalue - tempNode.hvalue
                exponent = decimal.Decimal(decimal.Decimal(math.e) **
(decimal.Decimal(-difference) * decimal.Decimal(self.temperature)))

            if difference > 0 or random.uniform(0, 1) < exponent:
                current = tempNode

            if current.hvalue == 0:
```

```
                print("Solution found!")
                current.printBoard()
                solution = True
                break

        if solution == False:
                print("The algorithm was unsuccessful in finding any
solution!")



print("Enter number of Queens:")
noOfQueens = int(input())
print("Enter number of iterations:")
iterations = int(input())
nQueen = NQueen(noOfQueens, iterations)
nQueen.hillClimb()
```

Parameters to run the Program

- Download the code NQueen_Simulated_Annealing.py
- Run the code by typing the following command on terminal:
  py NQueen_Simulated_Annealing.py
- The program will run and ask for the no of queens. Enter any value above 3 as the queen problem cannot be solved for boards less than 4.
- Then the program will ask for the no of iterations for which you want to run the algorithm. Simulated annealing will most likely give a solution for 150000 iterations and above.

Output

Case 1: Successful run for 8 Queens

```
Enter number of Queens:
8
Enter number of iterations:
170000
Initial Board Configuration:
[['-' '-' '-' '-' '-' '-' 'Q' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' 'Q' 'Q' '-' 'Q' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' 'Q']
 ['Q' '-' '-' 'Q' '-' 'Q' '-' '-']]
Solution found!
[['-' 'Q' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' 'Q']
 ['-' '-' '-' '-' '-' 'Q' '-' '-']
 ['Q' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' 'Q' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' 'Q' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' 'Q' '-']
 ['-' '-' '-' 'Q' '-' '-' '-' '-']]
```

Case 2: Unsuccessful run for 8 Queens

```
Enter number of Queens:
8
Enter number of iterations:
150000
Initial Board Configuration:
[['-' '-' '-' '-' '-' '-' '-' '-']
 ['Q' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' 'Q' 'Q' '-' '-' '-']
 ['-' 'Q' '-' '-' '-' '-' '-' 'Q']
 ['-' '-' '-' '-' '-' 'Q' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' 'Q' '-' '-' '-' 'Q' '-']]
The algorithm was unsuccessful in finding any solution!
```

Opinion

- The simulated annealing algorithm is an effective algorithm in solving the n-queens problem.
- It tends to give us a solution provided the iterations are high enough for it to converge to a solution.
- The temperature variable introduced helps in avoiding the local minima initially as the probability of moving to the new state is high early in the run even if the successor is slightly worse than the current state.
- As we decrease the temperature exponentially, the program converges to a solution and the probability of moving to a new state which is worse than the current state decreases with the decrease in temperature.
- Codewise, the algorithm is easy to implement and pretty straightforward to understand.