BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

WORK INTEGRATED LEARNING PROGRAMMES

DSECL ZG557, Artificial and Computational Intelligence

Familiarize with the working of Local Search algorithms: Genetic Algorithm

Tool: Python

Libraries Used: numpy, sys

Sample Problem: The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.
Queens can attack either on the same row, on the same column or across the diagonal.
If none of the queens are located on the same row, same column or across the diagonals for each other then we call the positioning/configuration of Queens to be a solution.

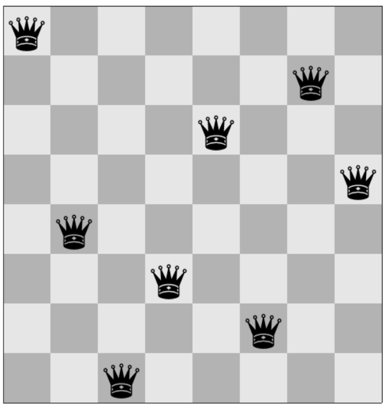
Input: Population with multiple Board Configurations of N-Queens

Example: [1,2,3,7,5,0,4,6]

This represents 0^{th} queen lies in 1^{st} row, 1^{st} queen lies in 2^{nd} row, 2^{nd} queen lies in 3^{rd} row, 3^{rd} queen lies in 7^{th} row, 4^{th} queen lies in 5^{th} row, 5^{th} queen lies in 0^{th} row, 6^{th} queen lies in 4^{th} row and 7^{th} queen lies in 6^{th} row.

Output: A possible configuration of queens such that none of the attack each other.

Example (8-Queens):



[0,4,7,5,2,6,1,3]

Explanation: If 0^{th} queen lies in 0^{th} row, 1^{st} queen lies in 4^{th} row, 2^{nd} queen lies in 7^{th} row, 3^{rd} queen lies in 5^{th} row, 4^{th} queen lies in 2^{nd} row, 5^{th} queen lies in 6^{th} row, 6^{th} queen lies in 1^{st} row and 7^{th} queen lies in 3^{rd} row, then none can attack each other.

Implementation:

```python
import numpy as np
import sys

nQueens = 8
STOP_CTR = 28
MUTATE = 0.01
MUTATE_FLAG = True
# MAX_ITER = 100000
MAX_ITER = 1000000
POPULATION = None


class BoardPosition:
    def __init__(self):
        self.sequence = None
        self.fitness = None
        self.survival = None

    def setSequence(self, val):
        self.sequence = val

    def setFitness(self, fitness):
        self.fitness = fitness

    def setSurvival(self, val):
        self.survival = val

    def getAttr(self):
        return {'sequence': self.sequence, 'fitness': self.fitness, 'survival':
self.survival}


def fitness(chromosome=None):
    """
    returns 28 - <number of conflicts>
    to test for conflicts, we check for
     -> row conflicts
     -> columnar conflicts
     -> diagonal conflicts

    The ideal case can yield upton 28 arrangements of non attacking pairs.
    for iteration 0 -> there are 7 non attacking queens
    for iteration 1 -> there are 6 no attacking queens ..... and so on

    Therefore max fitness = 7 + 6+ 5+4 +3 +2 +1 = 28

    hence fitness val returned will be 28 - <number of clashes>

    """
    # calculate row and column clashes
    # just subtract the unique length of array from total length of array
    # [1,1,1,2,2,2] - [1,2] => 4 clashes
    clashes = 0
    row_col_clashes = abs(len(chromosome) - len(np.unique(chromosome)))
    clashes += row_col_clashes
```

```python
    # calculate diagonal clashes
    for i in range(len(chromosome)):
        for j in range(len(chromosome)):
            if (i != j):
                dx = abs(i - j)
                dy = abs(chromosome[i] - chromosome[j])
                if (dx == dy):
                    clashes += 1

    return 28 - clashes

def generateChromosome():
    # randomly generates a sequence of board states.
    global nQueens
    init_distribution = np.arange(nQueens)
    np.random.shuffle(init_distribution)
    return init_distribution


def generatePopulation(population_size=100):
    global POPULATION

    POPULATION = population_size

    population = [BoardPosition() for i in range(population_size)]
    for i in range(population_size):
        population[i].setSequence(generateChromosome())
        population[i].setFitness(fitness(population[i].sequence))

    summation_fitness = np.sum([x.fitness for x in population])
    for each in population:
        each.survival = each.fitness / (summation_fitness * 1.0)

    return population


def getParent():
    globals()
    parent1, parent2 = None, None
    # parent is decided by random probability of survival.
    # since the fitness of each board position is an integer >0,
    # we need to normaliza the fitness in order to find the solution



    while True:
        parent1_random = np.random.rand()
        parent1_rn = [x for x in population if x.survival <= parent1_random]
        try:
            parent1 = parent1_rn[0]
            break
        except:
            pass

    while True:
        parent2_random = np.random.rand()
        parent2_rn = [x for x in population if x.survival <= parent2_random]
```

```python
        try:
            t = np.random.randint(len(parent2_rn))
            parent2 = parent2_rn[t]
            if parent2 != parent1:
                break
            else:
                continue
        except:
            continue

    if parent1 is not None and parent2 is not None:
        return parent1, parent2
    else:
        sys.exit(-1)


def reproduce_crossover(parent1, parent2):
    globals()
    n = len(parent1.sequence)
    c = np.random.randint(0, n)
    child = BoardPosition()
    child.sequence = []
    child.sequence.extend(parent1.sequence[0:c])
    child.sequence.extend(parent2.sequence[c:])
    child.setFitness(fitness(child.sequence))
    return child


def mutate(child):
    """
    - according to genetic theory, a mutation will take place
    when there is an anomaly during cross over state

    - since a computer cannot determine such anomaly, we can define
    the probability of developing such a mutation

    """
    if child.survival < MUTATE:
        c = np.random.randint(8)
        child.sequence[c] = np.random.randint(8)
    return child.sequence


def GA(iteration):
    print(" #" * 10, "Executing Genetic  generation : ", iteration, " #" * 10)
    globals()
    newpopulation = []
    for i in range(len(population)):
        parent1, parent2 = getParent()
        # print "Parents generated : ", parent1, parent2

        child = reproduce_crossover(parent1, parent2)
        newpopulation.append(child)

    summation_fitness = np.sum([x.fitness for x in newpopulation])
    for each in newpopulation:
        each.survival = each.fitness / (summation_fitness * 1.0)
```

```
    if (MUTATE_FLAG):
        for each in newpopulation:
            presentVal = each.sequence
            mightBeChangedVal = mutate(each)
            if presentVal!=mightBeChangedVal:
                each.sequence = presentVal
                each.fitness = each.setFitness(fitness(each.sequence))

    summation_fitness = np.sum([x.fitness for x in newpopulation])
    for each in newpopulation:
        each.survival = each.fitness / (summation_fitness * 1.0)

    return newpopulation

def stop():
    globals()

    fitnessvals = [pos.fitness for pos in population]
    if STOP_CTR in fitnessvals:
        return True
    if MAX_ITER == iteration:
        return True
    return False

population = generatePopulation(100)

iteration = 0
while not stop():
    # keep iteratin till  you find the best position
    population = GA(iteration)
    iteration += 1

print("Iteration Number: ", iteration)
for each in population:
    if each.fitness == 28:
        print(each.sequence)
```

Output :

[5, 2, 0, 7, 4, 1, 3, 6]

Output Explanation: The output shows which queen would lie in which row so that there are no clashes. 0^{th} queen in 5^{th} row, 1^{st} queen in 2^{nd} row and so on. Refer table below

Column	Row
0	5
1	2
2	0
3	7
4	4
5	1
6	3
7	6

Screenshot:

```
 # # # # # # # # # # Executing Genetic  generation :  2420  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2421  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2422  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2423  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2424  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2425  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2426  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2427  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2428  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2429  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2430  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2431  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2432  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2433  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2434  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2435  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2436  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2437  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2438  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2439  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2440  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2441  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2442  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2443  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2444  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2445  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2446  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2447  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2448  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2449  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2450  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2451  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2452  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2453  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2454  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2455  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2456  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2457  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2458  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2459  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2460  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2461  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2462  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2463  # # # # # # # # # #
 # # # # # # # # # # Executing Genetic  generation :  2464  # # # # # # # # # #
Iteration Number:  2465
[5, 2, 0, 7, 4, 1, 3, 6]
```

Lab Exercises:

1. Experiment with number of queens as 16
2. Experiment with population size of 500, 1000
3. Experiment with different MUTATE values.
4. Implement cryptarithmetic using Genetic Algorithm