# 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.

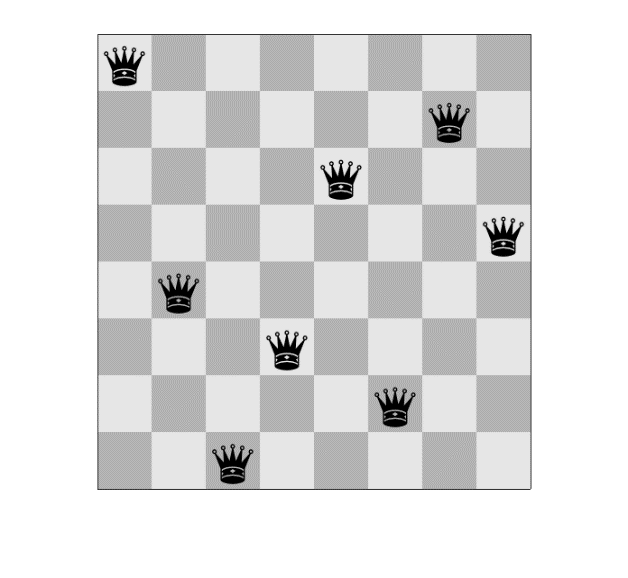
If none of the queens are located on the same row, same column or across the diagonals for each other then we call the positioning/configuration of Queens to be a solution.

**Input:** Population with multiple Board Configurations of N-Queens

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

This represents 0th queen lies in 1st row, 1st queen lies in 2nd row, 2nd queen lies in 3rd row, 3rd queen lies in 7th row, 4th queen lies in 5th row, 5th queen lies in 0th row, 6th queen lies in 4th row and 7th queen lies in 6th 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 0th queen lies in 0th row, 1st queen lies in 4th row, 2nd queen lies in 7th row, 3rd queen lies in 5th row, 4th queen lies in 2nd row, 5th queen lies in 6th row, 6th queen lies in 1st row and 7th queen lies in 3rd row, then none can attack each other.

**Implementation:**

**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  
  
 *# 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]  
 **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)

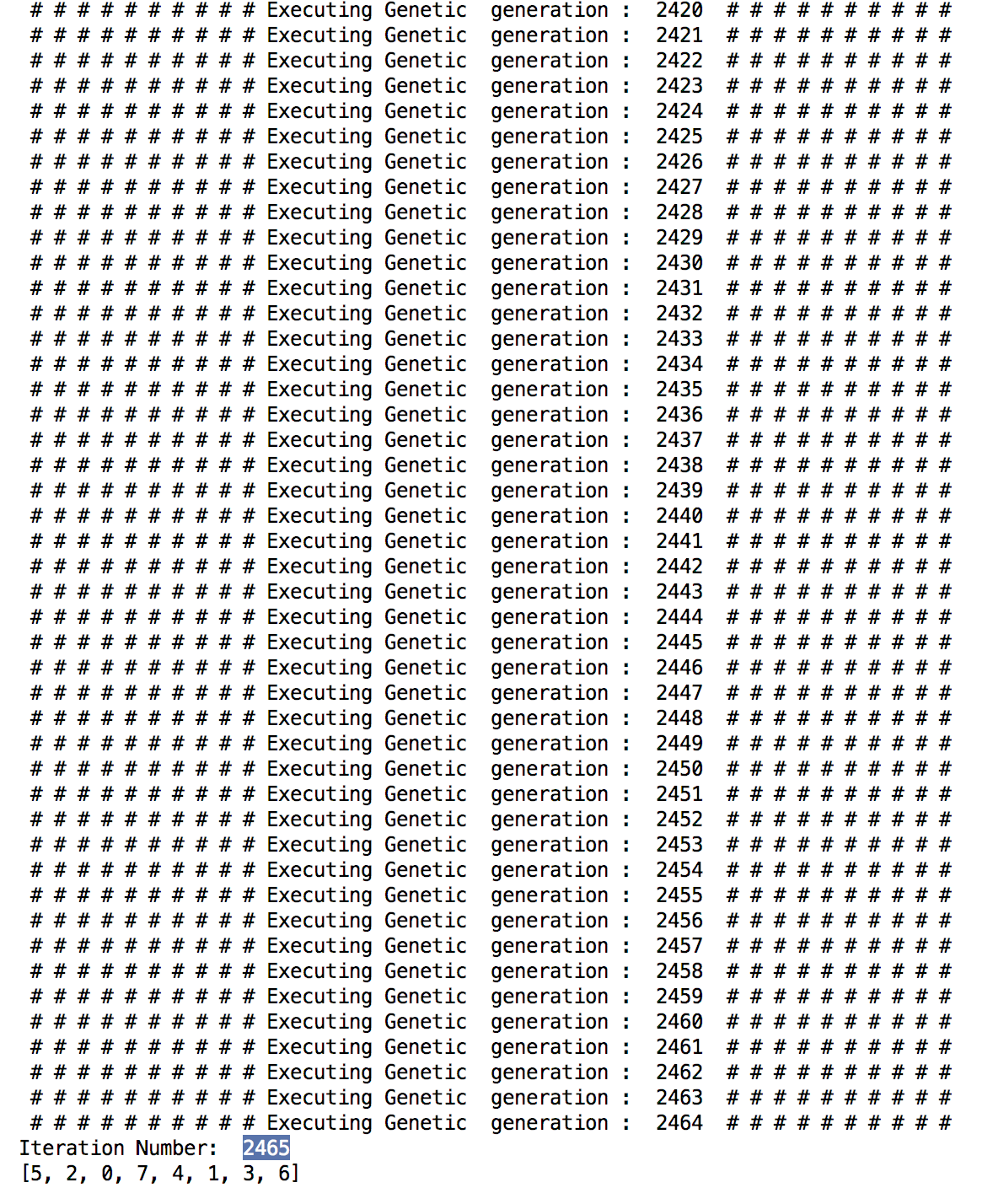
**Output:**

[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. 0th queen in 5th row, 1st queen in 2nd 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:**



**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