**Genetic Algorithm**

import numpy as np

def fitness(population, n):

# list of fitness function which will be converted to numpy array

fit = []

# total possible non-attacking pairs

total\_nap = int((n \* (n - 1)) / 2)

# list of Non-attacking pairs of each individual in population

list\_of\_nap = []

for p in population:

# count of horizontal attacking pairs

horizontal\_ap = 0

# count of diagonal attacking pairs

diagonal\_ap = 0

# count of non-attacking pairs

non\_ap = 0

# calculating horizontal attacking pairs

if len(set(p)) < len(p):

horizontal\_ap += (len(p) - len(set(p)))

# calculating diagonal attacking pairs

for j in range(len(p)):

for k in range(j):

diff = j - k

if p[k] == (p[j] - diff):

diagonal\_ap += 1

elif p[k] == (p[j] + diff):

diagonal\_ap += 1

# calculating non-attacking pairs and adding to the list

non\_ap += total\_nap - (horizontal\_ap + diagonal\_ap)

list\_of\_nap.append(non\_ap)

# sum of all Non-attacking pairs of the individuals

sum = 0

for pop in range(len(population)):

sum += list\_of\_nap[pop]

# fitness calculation

for l in range(len(list\_of\_nap)):

fit.append(list\_of\_nap[l]/100)

fit = np.array(fit, float)

return fit, sum

def select(population, fit, sum):

m = int(len(population))

a = np.empty(m, int)

for i in range(m):

a[i] = i

size = 1

replace = True

# list of probabilities

probab = []

for p in range(len(fit)):

prob = (fit[p]\*100)/sum

probab.append(prob)

probab = np.array(probab)

# getting the index for one random individual in the population

c = int(np.random.choice(a, size, replace, probab))

# the random individual in the population

selection = population[c]

return selection

def crossover(x, y):

# getting first half from parent X

child = np.array(n)

slice1shape = int((n / 2) - 1)

slice1 = []

s1 = 0

while s1 < slice1shape:

slice1.append(s1)

s1 += 1

slice1 = np.array(x[slice1])

# getting second half from parent Y

slice2 = []

s2 = slice1shape

while s2 < len(y):

slice2.append(s2)

s2 += 1

slice2 = np.array(y[slice2])

# child created from the crossover

child = np.concatenate((slice1, slice2))

return child

def mutate(child):

# count of times to mutate genes

count = int(len(child)/2)

for c in range(count):

# mutating a random gene in the individual

random1 = np.random.randint(0, len(child))

random2 = np.random.randint(0, len(child))

child[random1] = random2

return child

def genetic\_algo(population, n, mutation\_threshold):

print("Solving...")

# the goal fitness to reach

goal\_fit = ((n \* (n - 1)) / 2)/100

# generation count

gen\_count = 1

# the number of times the algo loop will run until it finds a solution

n\_max = 10000

# the solution as an 1d array

solution = np.zeros(n, int)

while n\_max > 0:

fit, sum = fitness(population, n)

if np.max(fit) == goal\_fit:

index = np.where(fit == goal\_fit)

solution = population[index[0][0]]

break

else:

new\_population = []

for p in population:

x = select(population, fit, sum)

y = select(population, fit, sum)

child = crossover(x, y)

if np.random.uniform(0, 1) < mutation\_threshold:

child = mutate(child)

new\_population.append(child)

population = np.array(new\_population)

gen\_count += 1

n\_max -= 1

# prompts to try again if solution not found after trying n\_max times

if np.max(solution) == 0:

print("Please try again")

else:

print("Solution found in generation -", gen\_count, ":\n", solution)

'''for 8 queen problem, n = 8'''

n = 8

'''start\_population denotes how many individuals/chromosomes are there

in the initial population n = 8'''

start\_population = 10

'''if you want you can set mutation\_threshold to a higher value,

to increase the chances of mutation'''

mutation\_threshold = 0.5

'''creating the population with random integers between 0 to 7 inclusive

for n = 8 queen problem'''

population = np.random.randint(0, n, (start\_population, n))

print("Population:\n", population)

'''calling the genetic\_algo function'''

genetic\_algo(population, n, mutation\_threshold)