**Practical 10B**

import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

class City:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def distance(self, city):

        xDis = abs(self.x - city.x)

        yDis = abs(self.y - city.y)

        distance = np.sqrt((xDis \*\* 2) + (yDis \*\* 2))

        return distance

    def \_\_repr\_\_(self):

        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:

    def \_\_init\_\_(self, route):

        self.route = route

        self.distance = 0

        self.fitness= 0.0

    def routeDistance(self):

        if self.distance ==0:

            pathDistance = 0

            for i in range(0, len(self.route)):

                fromCity = self.route[i]

                toCity = None

                if i + 1 < len(self.route):

                    toCity = self.route[i + 1]

                else:

                    toCity = self.route[0]

                pathDistance += fromCity.distance(toCity)

            self.distance = pathDistance

        return self.distance

    def routeFitness(self):

        if self.fitness == 0:

            self.fitness = 1 / float(self.routeDistance())

        return self.fitness

def createRoute(cityList):

    route = random.sample(cityList, len(cityList))

    return route

def initialPopulation(popSize, cityList):

    population = []

    for i in range(0, popSize):

        population.append(createRoute(cityList))

    return population

def rankRoutes(population):

    fitnessResults = {}

    for i in range(0,len(population)):

        fitnessResults[i] = Fitness(population[i]).routeFitness()

    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)

def selection(popRanked, eliteSize):

    selectionResults = []

    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])

    df['cum\_sum'] = df.Fitness.cumsum()

    df['cum\_perc'] = 100\*df.cum\_sum/df.Fitness.sum()

    for i in range(0, eliteSize):

        selectionResults.append(popRanked[i][0])

    for i in range(0, len(popRanked) - eliteSize):

        pick = 100\*random.random()

        for i in range(0, len(popRanked)):

            if pick <= df.iat[i,3]:

                selectionResults.append(popRanked[i][0])

                break

    return selectionResults

def matingPool(population, selectionResults):

    matingpool = []

    for i in range(0, len(selectionResults)):

        index = selectionResults[i]

        matingpool.append(population[index])

    return matingpool

def breed(parent1, parent2):

    child = []

    childP1 = []

    childP2 = []

    geneA = int(random.random() \* len(parent1))

    geneB = int(random.random() \* len(parent1))

    startGene = min(geneA, geneB)

    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):

        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2

    return child

def breedPopulation(matingpool, eliteSize):

    children = []

    length = len(matingpool) - eliteSize

    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):

        children.append(matingpool[i])

    for i in range(0, length):

        child = breed(pool[i], pool[len(matingpool)-i-1])

        children.append(child)

    return children

def mutate(individual, mutationRate):

    for swapped in range(len(individual)):

        if(random.random() < mutationRate):

            swapWith = int(random.random() \* len(individual))

            city1 = individual[swapped]

            city2 = individual[swapWith]

            individual[swapped] = city2

            individual[swapWith] = city1

    return individual

def mutatePopulation(population, mutationRate):

    mutatedPop = []

    for ind in range(0, len(population)):

        mutatedInd = mutate(population[ind], mutationRate)

        mutatedPop.append(mutatedInd)

    return mutatedPop

def nextGeneration(currentGen, eliteSize, mutationRate):

    popRanked = rankRoutes(currentGen)

    selectionResults = selection(popRanked, eliteSize)

    matingpool = matingPool(currentGen, selectionResults)

    children = breedPopulation(matingpool, eliteSize)

    nextGeneration = mutatePopulation(children, mutationRate)

    return nextGeneration

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):

    pop = initialPopulation(popSize, population)

    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):

        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))

    bestRouteIndex = rankRoutes(pop)[0][0]

    bestRoute = pop[bestRouteIndex]

    return bestRoute

cityList = []

for i in range(0,25):

    cityList.append(City(x=int(random.random() \* 200), y=int(random.random() \* 200)))

geneticAlgorithm(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)

def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):

    pop = initialPopulation(popSize, population)

    progress = []

    progress.append(1 / rankRoutes(pop)[0][1])

    for i in range(0, generations):

        pop = nextGeneration(pop, eliteSize, mutationRate)

        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)

    plt.ylabel('Distance')

    plt.xlabel('Generation')

    plt.show()

geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)

Output:

Initial distance: 2073.9131784861547

Final distance: 831.2750685598561

