```
# -*- coding: utf-8 -*-

import pandas as pd
import math
import random
import numpy as np
import time

# from google.colab import drive
# drive.mount("/content/drive")

path = r"Dataset_project_2.csv"
df = pd.read_csv(path)

# df['demand'][1] for the demand of node 1 etc.
# df['x'][2] for x coordinate of node 2
# df['y'][0] for y coordinate of node 0

# GLOBAL VARIABLES

popSize = 50
noGen = 2000
crossRate = 0.85
mutRate = 0.35
numberNodes = 55
numberVehicles = 9
selectionPressure = 2

from numpy.random import choice


def SelectAParent(currPop, list_fairness):
    sort_pop = [x for _, x in sorted(zip(list_fairness, currPop))]
    rank = [sort_pop.index(x) for x in currPop]
    list_rank = [2 - selectionPressure + 2 * (selectionPressure - 1) * ((r - 1) /
(popSize - 1)) for r in rank]
    total = sum(list_rank)
    selectMed = [x / total for x in list_rank]
    # find cumulative fitness values for roulette wheel selection , saw on youtube
that you need cumulative fitness
    cumulative_fitness = []
    start = 0
    for norm_value in selectMed:
        start += norm_value
        cumulative_fitness.append(start)

    random_number = random.uniform(0, 1)
    n = len(currPop)
    for i in range(n):
        if (random_number < cumulative_fitness[i]):
            return currPop[i]


def GeneratePopulation(PopSize):
    population = []
    index = PopSize
    while index != 0:
        list_nodes_not_sorted = list(range(1, 54 + 1))
        list_lists = [[], [], [], [], [], [], [], [], []]
```

```
        list_capacity = [0] * 9
        list_demand = []
        for elem in list_nodes_not_sorted:
            list_demand.append(df['demand'][elem])
        list_nodes = [x for _, x in sorted(zip(list_demand,
list_nodes_not_sorted))]
        possible = True
        while len(list_nodes) > 0 and possible:
            elem = list_nodes.pop()
            insere = False
            flags = [True] * 8
            while not insere:
                position = random.randint(0, 7)
                if flags[position]:
                    if list_capacity[position] + df['demand'][elem] <= 100:
                        list_lists[position].append(elem)
                        insere = True
                        list_capacity[position] = list_capacity[position] +
df['demand'][elem]
                    else:
                        flags[position] = False
                if not insere and list_capacity[-1] + df['demand'][elem] <= 100:
                    list_lists[8].append(elem)
                    insere = True
                    list_capacity[8] = list_capacity[8] + df['demand'][elem]
                elif not insere:
                    possible = False
                    insere = True
        if possible:
            index += -1
            individual = list_lists[0] + [-1] + list_lists[1] + [-1] +
list_lists[2] + [-1] + list_lists[3] + [-1] + \
                         list_lists[4] + [-1] + list_lists[5] + [-1] +
list_lists[6] + [-1] + list_lists[7] + [-1] + \
                         list_lists[8]
            population.append(individual)
    return population


def next_to(liste):
    flag = False
    for i in range(len(liste) - 1):
        if liste[i] == -1 and liste[i + 1] == -1:
            flag = True
    return flag


def DoCrossover(par1, par2):
    # ordercrossover
    # we choose randomly start and end of the kept part
    start = random.randint(0, len(par1))
    end = random.randint(0, len(par1))
    if start > end:
        start, end = end, start
    # creation of the offspring
    offspring = [0] * start + par1[start:end + 1] + [0] * (len(par1) - end - 1)
    # we create a copy to not modify par2
    par2_bis = par2.copy()
    for elem in offspring:
```

```
        if elem != 0:
            if elem == -1:
                positions = []
                for i, elem in enumerate(par2_bis):
                    if elem == -1:
                        positions.append(i)
                delete = random.choice(positions)
                par2_bis.pop(delete)
            else:
                par2_bis.remove(elem)
    for i, elem in enumerate(offspring):
        if (len(par2_bis) != 0) and elem == 0:
            offspring[i] = par2_bis.pop(0)
    # now we verify if everything is okay
    if next_to(offspring) or offspring[0] == -1 or offspring[-1] == -1 or
(check_capacity(offspring) == 0):
        return DoCrossover(par1, par2)
    return offspring


def check_capacity(node):
    capacity = []
    val = 0
    for i in range(len(node)):
        if node[i] == -1:
            capacity.append(val)
            val = 0
        else:
            val += df['demand'][node[i]]
    capacity.append(val)

    for i in range(len(capacity)):
        if capacity[i] > 100:
            return 0
    return 1


def swapped(liste):
    a = random.randint(0, len(liste) - 1)
    b = random.randint(0, len(liste) - 1)
    liste[a], liste[b] = liste[b], liste[a]
    while next_to(liste) or liste[0] == -1 or liste[-1] == -1 or
(check_capacity(liste) == 0):
        liste[a], liste[b] = liste[b], liste[a]
        a = random.randint(0, len(liste) - 1)
        b = random.randint(0, len(liste) - 1)
        liste[a], liste[b] = liste[b], liste[a]


def shifted(liste):
    a = random.randint(0, len(liste) - 1)
    elem = liste.pop(a)
    b = random.randint(0, len(liste) - 1)
    liste.insert(b, elem)
    while next_to(liste) or liste[0] == -1 or liste[-1] == -1 or
(check_capacity(liste) == 0):
        elem = liste.pop(b)
        liste.insert(a, elem)
        a = random.randint(0, len(liste) - 1)
```

```
        elem = liste.pop(a)
        b = random.randint(0, len(liste) - 1)
        liste.insert(b, elem)


def inverted(liste):
    start = random.randint(0, len(liste))
    end = random.randint(0, len(liste))
    if start > end:
        start, end = end, start
    liste[start:end + 1] = liste[start:end + 1][::-1]
    while next_to(liste) or liste[0] == -1 or liste[-1] == -1 or
(check_capacity(liste) == 0):
        liste[start:end + 1] = liste[start:end + 1][::-1]
        start = random.randint(0, len(liste))
        end = random.randint(0, len(liste))
        if start > end:
            start, end = end, start
        liste[start:end + 1] = liste[start:end + 1][::-1]


def DoMutation(child, mutRate):
    # three types of mutation but muRate for it to happen
    a = random.random()
    if a <= mutRate:
        swapped(child)

    b = random.random()
    if b <= mutRate:
        inverted(child)

    c = random.random()
    if c <= mutRate:
        shifted(child)


def calDis(node1, node2):
    x1 = df['x'][node1]
    y1 = df['y'][node1]
    x2 = df['x'][node2]
    y2 = df['y'][node2]
    return math.sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))


def totalDistance(node):
    totalDis = 0
    routes = []
    flag = True
    for i in range(len(node)):
        if flag == 1:
            totalDis += calDis(0, node[i])
            flag = False
        elif node[i] == -1:
            flag = True
            totalDis += calDis(node[i - 1], 0)
            routes.append(totalDis)
            totalDis = 0
        else:
            totalDis += calDis(node[i - 1], node[i])
```

```
    totalDis += calDis(node[-1], 0)
    routes.append(totalDis)
    return sum(routes)


def Uniform():
    return random.randint(0, 1)


def EvaluateFitness(child):
    # check Total traveling distance
    # if any vehicle bends capacity, then we set fitness 0
    # return value should be list, each element indicates fitness value of i th
chromosome(child)
    totalDis = 0
    totalCap = 0
    capacityFlag = 0
    routes = []
    capacity = []
    flag = True
    for i in range(len(child)):
        if flag == 1:
            totalDis += calDis(0, child[i])
            totalCap += df['demand'][child[i]]
            flag = False
        elif child[i] == -1:
            flag = True
            totalDis += calDis(child[i - 1], 0)
            routes.append(totalDis)
            capacity.append(totalCap)
            totalDis = 0
            totalCap = 0
        else:
            totalDis += calDis(child[i - 1], child[i])
            totalCap += df['demand'][child[i]]
    totalDis += calDis(child[-1], 0)
    routes.append(totalDis)
    capacity.append(totalCap)

    for i in range(len(routes)):
        if capacity[i] > 100:
            capacityFlag = True

    if capacityFlag:
        return (1 / sum(routes)) * (0.001)
    else:
        return (1 / sum(routes))


def BestChromosome(currPop, fitList):
    # return the best chromosome
    bestIdx = fitList.index(max(fitList))
    return currPop[bestIdx]


def GenerateFitList(currPop):
    fitList = []
    for i in range(0, len(currPop)):
        child = currPop[i]
```

```
        fitList.append(EvaluateFitness(child))
    return fitList


def printGen(currPop):
    print('==start print===')
    for i in range(len(currPop)):
        print(currPop[i])
    print('===end print===')


def GASolve(noGen, PopSize, keepBest, crossRate, mutRate):
    currPop = GeneratePopulation(popSize)
    fitList = GenerateFitList(currPop)
    childList = []
    newPop = []
    for i in range(0, noGen):
        #print(i, "th fitList: ", fitList)
        for j in range(0, popSize):
            # printGen(currPop)
            par1 = SelectAParent(currPop, fitList)
            if Uniform() < crossRate:
                par2 = SelectAParent(currPop, fitList)
                child = DoCrossover(par1, par2)
                while check_capacity(child) == 0:
                    child = DoCrossover(par1, par2)
            else:
                child = par1[:]

            DoMutation(child, mutRate)
            childList.append(EvaluateFitness(child))
            newPop.append(child)
        if keepBest == 1:
            best = BestChromosome(currPop, fitList)
            newPop[0] = best[:]
            # print("Best solution of ", i, "th population is : ", best)
            childList[0] = fitList[currPop.index(best)]
            # print("fitness of best chromosome: ", fitList[currPop.index(best)], "
totalDis: ", totalDistance(best), "calDis: ", 1/totalDistance(best))
        currPop = newPop[:]
        newPop = []
        fitList = childList[:]
        childList = []
    solution = BestChromosome(currPop, fitList)
    print("The distance of the solution is :", totalDistance(solution))
    return solution


def routes_from_list(liste):
    route_construct = False
    new_route = []
    count = 0
    while len(liste) != 0:
        if not route_construct:
            new_route.clear()
            route_construct = True
            count += 1
        elem = liste.pop(0)
        if elem != -1:
```

```
            new_route.append(elem)
        else:
            print("Route", count, ":", new_route)
            route_construct = False
    print("Route", count, ":", new_route)


start = time.time()
solution = GASolve(noGen, popSize, 1, crossRate, mutRate)
print('CPU Time:', time.time() - start)
routes_from_list(solution)
```