**实 验 报 告**

**学 院：** 计算机科学与技术学院

**班 级：** 计非211

**学 号：** 202131990522

**姓 名：** 嵇纹垲

**课 程：** 人工智能基础

**指导教师：** 周昌军

**完成时间：** 2024 **年** 4 **月** 3 **日**

**浙 江 师 范 大 学 制**

**实验二：遗传算法实验要求**

**实验题目**

用遗传算法解决20个城市的TSP问题。

20个城市的坐标

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | 53 | 94 | 62 | 50 | 160 | 35 | 87 | 25 | 40 | 130 |
| y | 20 | 50 | 95 | 120 | 102 | 190 | 61 | 82 | 118 | 47 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | 30 | 48 | 76 | 35 | 152 | 162 | 60 | 76 | 83 | 179 |
| y | 58 | 68 | 39 | 67 | 110 | 175 | 95 | 102 | 79 | 58 |

**问题分析**

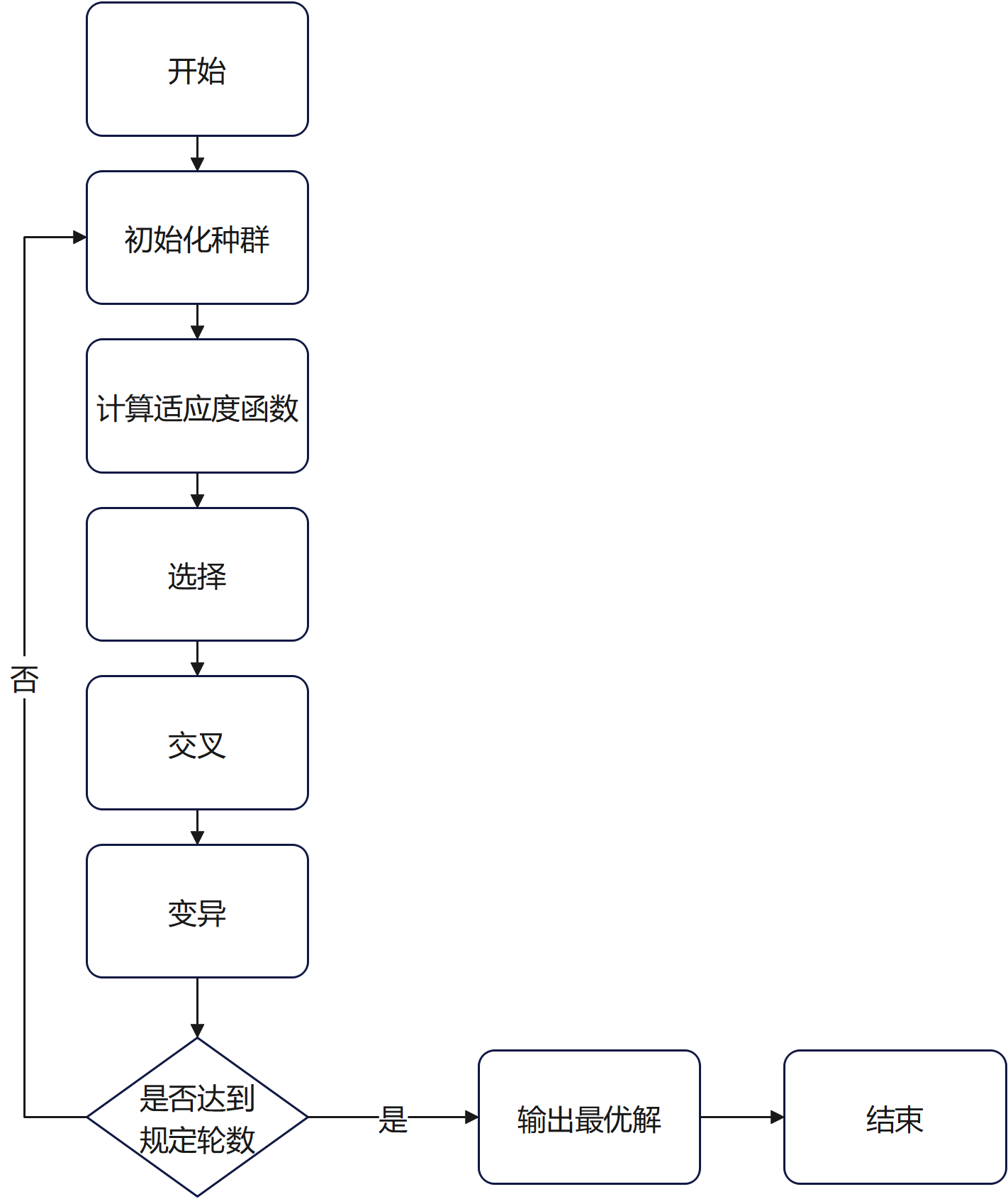
TSP问题是一个经典的问题，需要我们在一张图上找到一条经过所有点的最短点回路。由于其精确解被证明无法在多项式时间内得到，所以我们可以采用遗传算法来逼近精确解。

遗传算法作为一种基于生物进化原理的优化算法，具有较好的鲁棒性和全局搜索能力。通过模拟生物进化过程中的遗传、突变和自然选择等现象，遗传算法能够在搜索空间中自适应地寻找最优解。

在该问题中，我们可以将适应度函数定义为回路中所有相邻节点的距离之和，这样我们的目标就从求最短的总路径之和变成了求最小的适应度函数了。从而可以通过改变路径中的随机两个点的位置等操作进行变异，从而可以用遗传算法来处理该问题。

**算法设计**

首先给出算法运行的流程图



初始化种群：包括设定种群的数目的的大小，以及不同城市之间的距离等

求适应度函数：直接采用两个城市之间的欧式距离作为它们之间的代价，整个适应度就是相邻城市之间的距离的总和了

选择：在上一代中，通过优胜劣汰，使得适应性更强的解得以保留。具体来说，我们通过锦标赛算法，以适应度函数为比较标准，每次保留适应度最好的个体，并且淘汰其余的个体，最后得到整个种群中剩下的个体。

交叉：

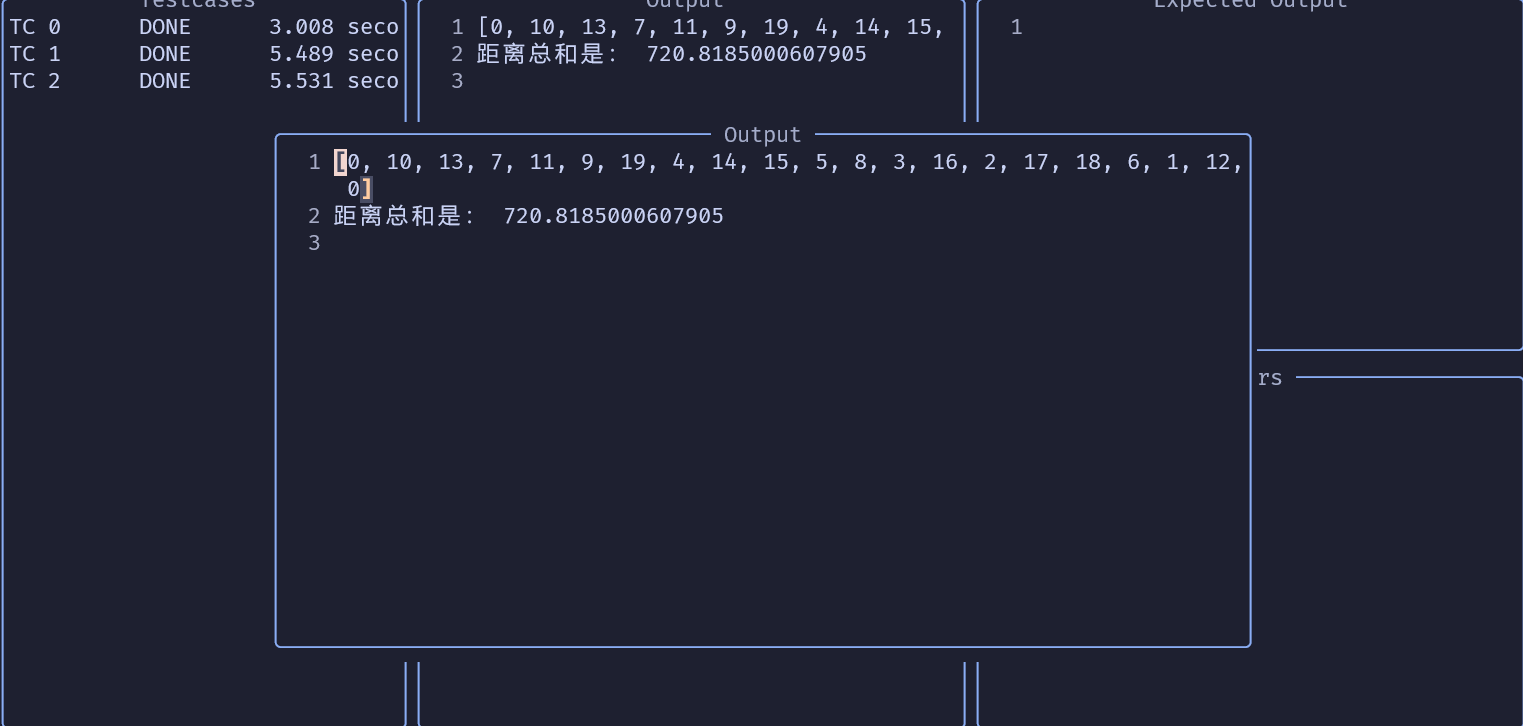
交叉就是改变个体的基因，随机选取一个节点，将交叉双方该节点后的部分进行交换。在交换后，单个物种可能会出现有重复城市的情况，因此我们进行了去重操作，即记录下重复的位置，使交叉双方重复的节点进行交换。

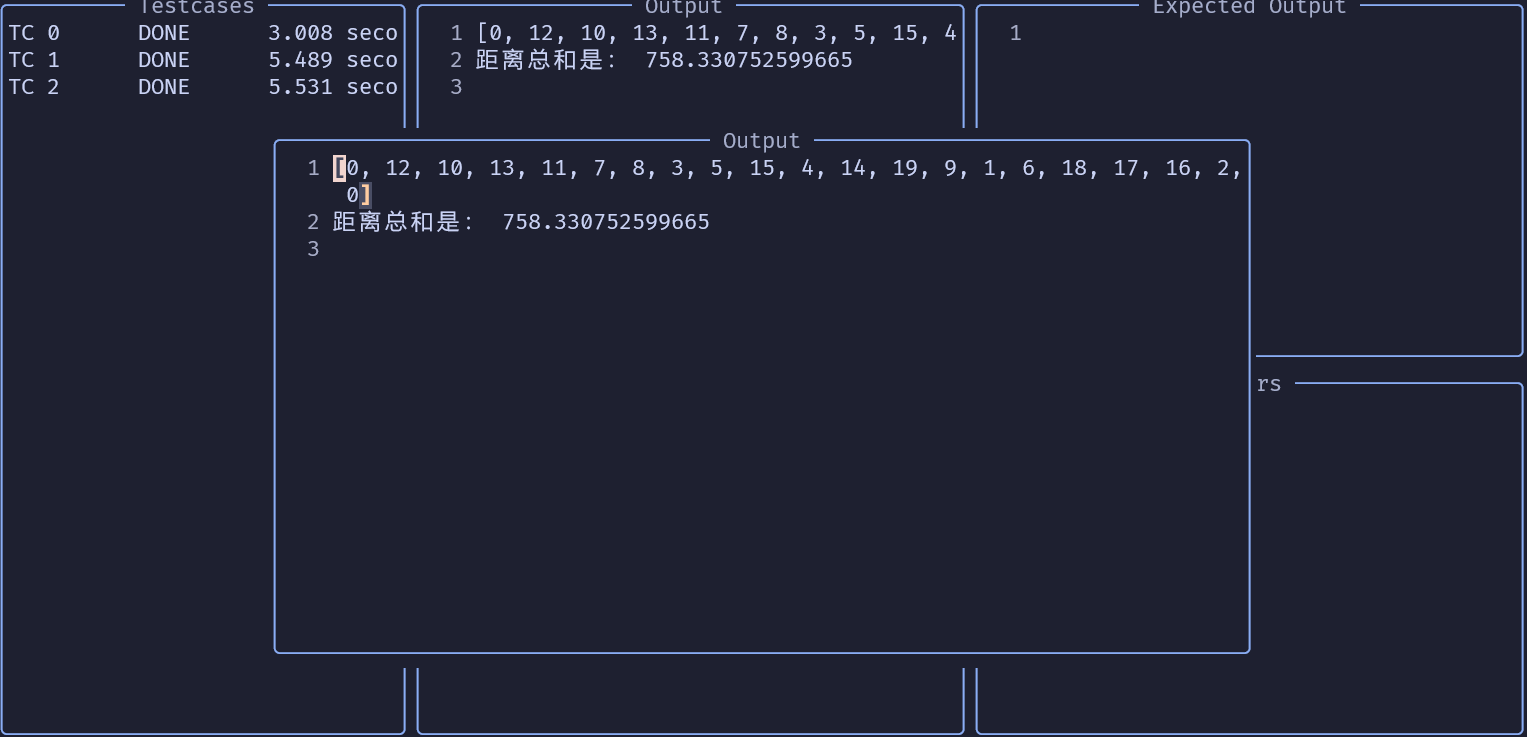
变异：

该操作是遗传算法跳出局部最优解的一个重要操作。对于每一代，我们设定一个变异的概率，对个体的节点中的城市进行随机的顺序颠倒，实现变异。

基于此，整个遗传算法就设计完毕了。

**运算结果**







多次运行之后可以看到结果如上显示，最优路径的值基本是在720左右，但是每一次运行的结果的波动较大，说明该算法的稳定性较差，还不够逼近最优解。这也与实现过程中的参数的设置有关。

**实验总结**

本次实验中成功地使用遗传算法解决了TSP问题。实验结果表明，遗传算法是一种有效的解决TSP问题的工具，利用其随机性以及与遗传变异相类比的思想，能够在很短的时间内得到一个交到的解。但是其问题在于稳定性不足，这与参数的设置以及概率都有关系，并且很难验证得到的解是否一定是全局最优解，也可能是某个局部最优解。这一点很难说明。

**代码**

import numpy as np

import copy

import math

import random

city\_axis = [[53, 20], [94, 50], [62, 95], [50, 120], [160, 102], [35, 190], [87, 61], [25, 82], [40, 118], [

130, 47], [30, 58], [48, 68], [76, 39], [35, 67], [152, 110], [162, 175], [60, 95], [76, 102], [83, 79], [179, 58]]

def norm(x):

return x \* x

def gt\_dis(a, b):

return math.sqrt(norm(a[0]-b[0]) + norm(a[1]-b[1]))

# 准备好距离矩阵

city\_num = 20

city\_dist\_mat = np.zeros([city\_num, city\_num])

for i in range(0, city\_num):

for j in range(i + 1, city\_num):

city\_dist\_mat[i][j] = city\_dist\_mat[j][i] = gt\_dis(

city\_axis[i], city\_axis[j])

# 1.定义个体类，包括基因（城市路线）和适应度

num\_person\_idx = 0

num\_person = 0

dis\_list = []

ans = 1e9

class Individual:

def \_\_init\_\_(self, genes=None):

global num\_person

global dis\_list

global num\_person\_idx

num\_person\_idx += 1

if num\_person\_idx % 20 == 0:

num\_person += 1

self.genes = genes

if self.genes == None:

genes = [0] \* 5

temp = [0] \* 4

temp = [i for i in range(1, city\_num)]

random.shuffle(temp)

genes[1:] = temp

genes[0] = 0

self.genes = genes

# print("init\_self.genes = ",self.genes)

self.fitness = self.evaluate\_fitness()

# self.fitness = fitness

# dis\_list.append(-1.0)

else:

self.fitness = float(self.evaluate\_fitness())

# print('self.fitness', self.fitness)

# 2. #计算个体的适应度

def evaluate\_fitness(self):

dis = 0

for i in range(city\_num - 1):

dis += city\_dist\_mat[self.genes[i]][self.genes[i + 1]]

if i == city\_num - 2:

dis += city\_dist\_mat[self.genes[i + 1]][0] # 回到0

if num\_person\_idx % 20 == 0:

dis\_list.append(dis)

return 1 / dis

def copy\_list(old):

new = []

for element in old:

new.append(element)

return new

def sort\_win\_num(group):

for i in range(len(group)):

for j in range(len(group) - i - 1):

# print('group[j].fitness\_type', type(group[j].fitness))

if group[j].fitness < group[j + 1].fitness:

temp = group[j]

group[j] = group[j + 1]

group[j + 1] = temp

return group

# 定义Ga类

# 3~5，交叉、变异、更新种群，全部在Ga类中实现

class Ga:

# input\_为城市间的距离矩阵

def \_\_init\_\_(self, input\_):

# 声明一个全局变量

global city\_dist\_mat

city\_dist\_mat = input\_

# self.best = None

self.best = Individual(None)

# print("BBBBBBbest.fitness",self.best.fitness)

# 种群

self.individual\_list = []

# 每一代的最佳个体

self.result\_list = []

# 每一代个体对应的最佳适应度

self.fitness\_list = []

# 交叉,这里采用交叉变异

def cross(self):

new\_gen = []

# 随机选取一段，含有num\_cross个数字（城市）

num\_cross = 3 # 后期可能需要调试的参数,考虑到实际问题里只有5个城市，所以认为3较为合适

for i in range(0, len(self.individual\_list) - 1, 2):

parent\_gen1 = copy\_list(self.individual\_list[i].genes)

parent\_gen2 = copy\_list(self.individual\_list[i + 1].genes)

# print("parent\_gen1",parent\_gen1)

# print("parent\_gen2",parent\_gen2)

index1\_1 = 0

index1\_2 = 0

index2\_1 = 0

index2\_2 = 0

# 定义一个下表列表

index\_list = [0] \* 30

for i in range(city\_num - 3): # 就是2，即0，1

index\_list[i] = i + 1

index1\_1 = random.choice(index\_list)

index1\_2 = index1\_1 + 2

index2\_1 = random.choice(index\_list)

index2\_2 = index2\_1 + 2

choice\_list1 = parent\_gen1[index1\_1:index1\_2 + 1]

choice\_list2 = parent\_gen2[index2\_1:index2\_2 + 1]

# 利用这一段生成两个子代,下面的赋值只是为了获取长度，所以用哪个父代能可以

# 也可以直接用city\_num直接代替

son\_gen1 = [0] \* city\_num

son\_gen2 = [0] \* city\_num

# 找到之后进行交叉，分别得到son\_gen1,son\_gen2

# 先把选中的段复制进去

son\_gen1[index1\_1: index1\_2 + 1] = choice\_list1

son\_gen2[index2\_1: index2\_2 + 1] = choice\_list2

# 然后左、右“查漏补缺”

temp1 = choice\_list1

temp2 = choice\_list2

if index1\_1 == 0:

pass

else:

for i in range(index1\_1):

for j in range(city\_num):

# 如果父代2里面的这个当初没被选中，那就加入son\_gene1

if parent\_gen2[j] not in choice\_list1:

son\_gen1[i] = parent\_gen2[j]

# 这个时候要扩增choice\_list1, 这样parent\_gen2里面未被选中的元素才会一个个被遍历到#1

choice\_list1.append(parent\_gen2[j])

# 找到之后马上break，防止被覆盖

break

choice\_list1 = temp1

if index1\_2 == city\_num - 1:

pass

else:

for i in range(index1\_2 + 1, city\_num):

for j in range(city\_num):

if parent\_gen2[j] not in choice\_list1:

son\_gen1[i] = parent\_gen2[j]

# 这个时候要扩增choice\_list1, 这样parent\_gen2里面未被选中的元素才会一个个被遍历到#2

choice\_list1.append(parent\_gen2[j])

# 找到之后马上break，防止被覆盖

break

# son\_gen2亦是如此

if index2\_1 == 0:

pass

else:

for i in range(index2\_1):

for j in range(city\_num):

# 如果父代1里面的这个当初没被选中，那就加入son\_gen2

if parent\_gen1[j] not in choice\_list2:

son\_gen2[i] = parent\_gen1[j]

# 这个时候要扩增choice\_list2, 这样parent\_gen1里面未被选中的元素才会一个个被遍历到#3

choice\_list2.append(parent\_gen1[j])

# 找到之后马上break，防止被覆盖

break

choice\_list2 = temp2

if index2\_2 == city\_num - 1:

pass

else:

for i in range(index2\_2 + 1, city\_num):

for j in range(city\_num):

if parent\_gen1[j] not in choice\_list2:

# print("i == ", i)

son\_gen2[i] = parent\_gen1[j]

# 这个时候要扩增choice\_list2, 这样parent\_gen1里面未被选中的元素才会一个个被遍历到#4

choice\_list2.append(parent\_gen1[j])

# 找到之后马上break，防止被覆盖

break

# 新生成的子代基因加入new\_gene列表

# print('son\_gen1 = ',son\_gen1)

# print('son\_gen2 = ',son\_gen2)

new\_gen.append(Individual(son\_gen1))

# print('new\_gen[-1].genes', new\_gen[-1].genes)

new\_gen.append(Individual(son\_gen2))

return new\_gen

# 变异

def mutate(self, new\_gen):

change = 0

mutate\_p = 0.02 # 待调参数

index\_list = [0] \* (city\_num - 1)

index\_1 = 1

index\_2 = 1

for i in range(city\_num - 1):

index\_list[i] = i + 1

for individual in new\_gen:

if random.random() < mutate\_p:

change += 1

# 如果变异，采用基于位置的变异,方便起见，直接使用上面定义的index列表

# index\_2 = (index\_1 + 2) % city\_num#这里让间隔为2的两个城市进行交换

index\_2 = random.choice(index\_list)

while index\_1 == index\_2:

index\_2 = random.choice(index\_list)

# 交换

temp = individual.genes[index\_1]

individual.genes[index\_1] = individual.genes[index\_2]

individual.genes[index\_2] = temp

# 变异结束，与老一代的进行合并

self.individual\_list += new\_gen

# 选择

def select(self):

# 在此选用锦标赛算法

group\_num = 100 # 待调参数

group\_size = 10

win\_num = 3

# 锦标赛的胜 者列表

winners = []

for i in range(group\_num):

# 定义临时列表，存储够一组为止

group = []

for j in range(group\_size):

gen\_player = random.choice(self.individual\_list)

gen\_player = Individual(gen\_player.genes)

group.append(gen\_player)

# 存储完一组之后选出适应度最大的前4个

group = sort\_win\_num(group)

winners += group[: win\_num]

# 选择结束，生成全新一代,赋值给self.individual\_list

self.individual\_list = winners

# print('selcetselect')

# 更新种群

def next\_gen(self):

# 交叉

new\_gene = self.cross()

# 变异

self.mutate(new\_gene)

# 选择

self.select()

for individual in self.individual\_list:

if individual.fitness > self.best.fitness:

self.best = individual

def train(self):

# 随机出初代种群#

individual\_num = 300

self.individual\_list = [Individual() for \_ in range(individual\_num)]

# 迭代

gen\_num = 200

for i in range(gen\_num):

# 从当代种群中交叉、变异、选择出适应度最佳的个体，获得子代产生新的种群

self.next\_gen()

# 连接首位

# print("i = ", i)

result = copy.deepcopy(self.best.genes)

result.append(result[0])

self.result\_list.append(result)

self.fitness\_list.append(self.best.fitness)

print(self.result\_list[-1])

print('距离总和是：', 1 / self.fitness\_list[-1])

if \_\_name\_\_ == '\_\_main\_\_':

# for i in city\_dist\_mat:

# print(i)

route = Ga(city\_dist\_mat)

route.train()