# 遗传算法实验

## 题目内容：

用遗传算法求解TSP问题（问题规模等和模拟退火求解TSP实验同），要求：

1. 设计较好的交叉操作，并且引入多种局部搜索操作（可替换通常遗传算法的变异操作）
2. 和之前的模拟退火算法（采用相同的局部搜索操作）进行比较
3. 得出设计高效遗传算法的一些经验，并比较单点搜索和多点搜索的优缺点。

## 实现环境：

该实验在Windows下使用python3编码实现

## 数据测例：

本次实验使用的测例主要是tsplib上面的ch130问题，是130个城市的旅行商问题，该问题最优tsp解路径长度为6110

另外还使用了ch150和a280进行测试，用于对比得出一些关于调参的结论，这两个问题分别是150个城市和280个城市的旅行商问题

## 算法概述：

**遗传算法：**

1. **设置初始种群**，我是随机产生的初始种群，个体100个。

尝试过设置一定量贪心解，效果并不好，虽然收敛很快，但是因为种群多样性受限，最终的结果并不好（哪怕仅仅设置一个贪心解，也会导致种群收敛很快，多样性不足，最终生成优解的潜力不足）

1. **计算出当前种群的适应度**，计算f（i）求和，f（i）是每个个体的路径总长度的倒数，即 1/path\_length
2. **选择**：使用轮盘赌从种群中随机选择两个到三个个体
3. **交叉**：一开始的时候自己使用的交叉操作是随机的顺序交叉操作，没有启发式任何信息，后来发现效果极差（反正基本上达不到10%以内），收敛的很慢甚至无法收敛，所以自己在网上查找比较好的交叉操作：两交换/三交换启发式交叉算子，使用两个/三个父代生成一个子代：

比如有如下三条路径作为父代：

A->B->C->D->E->F->A

A->D->C->F->E->B->A

A->C->F->E->B->D->A

1. 这三条路径都是**从A点出发**（所以子代也从A出发）
2. 然后分别前往B、D、C三个城市，**我们从A->B、A->D、A->C三条路中选择最短的**（比如A->B是最短的，经过城市，可以**将剩余路径循环移位使得B出现在队首**：比如第二条路径剩余D->C->F->E->B，循环移位得到B->D->C->F->E，最后这个父代更新为A->B->D->C->F->E->A
3. 不断执行（i）（ii），得到完整子代
4. 操作（4）之后执行**变异**操作：

变异操作尝试了爬山法中（i）（ii）的操作，**最终选择了第二种，逆转一段路径，因为这种变异操作表现更好，但是其实逆转节点的操作也能够接受**

因为变异操作的作用就是增加种群的多样性

在遗传算法执行代数很多之后会收敛，这时候种群里面的个体基本上相同，交叉操作已经无法产生不同于父代的子代了。（所以没办法进一步优化种群）

这时候就通过变异操作获得不同于父代的个体，从而有小概率能够优化种群

1. **反复执行（3）-（5）**生成100\*0.9个后代，另外100\*（1-0.9）个后代从父代中选择（保留父代中优秀的个体），注意不能选太多的父代个体放到下一代，这会导致多样性不足
2. **循环执行（2）到（6）**10000代，将最后种群的最优后代输出作为结果

## 代码框架简要说明

generate\_random\_list函数产生初始随机解，输入问题规模（有多少个城市），返回一种随机的走法：

def generate\_random\_list(point\_num):

point\_list = [0]\*(point\_num - 1)  
 for i in range(0, point\_num - 1):  
 point\_list[i] = i+1  
 random.shuffle(point\_list)  
 point\_list.insert(0, 0)  
 point\_list.append(0)  
 return point\_list

get\_distance获取两点间的距离，将两个城市的位置作为参数，返回距离，用于计算整个tsp路径长度

def get\_distance(point\_a, point\_b):

temp1 = math.pow(coordinate\_x[point\_a] - coordinate\_x[point\_b], 2)  
 temp2 = math.pow(coordinate\_y[point\_a] - coordinate\_y[point\_b], 2)  
 return math.pow(temp1 + temp2, 0.5)

get\_path\_length函数使用路径数组作为参数，用get\_distance计算路径距离，得出整个tsp路径的长度

def get\_path\_length(point\_list):

path\_length = 0  
 for i in range(0, len(point\_list)-1):  
 path\_length += get\_distance(point\_list[i], point\_list[i+1])  
 path\_length += get\_distance(point\_list[len(point\_list)-1], point\_list[0])  
 return path\_length

GenericAlgorithm类，其solve函数遗传算法求解tsp最短路

class GenericAlgorithm:

GenericAlgorithm类的show\_figure函数用于将tsp问题可视化，显示出；路径求解的具体过程

GenericAlgorithm类的store\_result函数用于将求解的结果存入文件中

def show\_figure(self):

def store\_result(self):

GenericAlgorithm类的solve函数，用于求解tsp问题，在这个函数中，循环1000代，不断产生后代（使用稍后介绍到的generate\_next\_generation函数产生一代后代），然后将100代繁衍之后的种群最优解进行存储

def solve(self):

while self.generation\_count < MAX\_GENERATION:  
 self.generate\_next\_generation()  
 self.store\_result()

GenericAlgorithm类的generate\_next\_generation函数，该函数使用当前的种群产生下一代子代，具体过程是：（1）对当前种群估值（2）循环使用generate\_children函数产生一个后代个体，直到达到一定数量（3）将生成的后代和一定比例的父代最有个体合并为新的种群（4）代数计数加一

def generate\_next\_generation(self):

self.evaluate()  
  
 new\_generation = queue.PriorityQueue()  
  
 # generate child  
 while len(new\_generation.queue) < self.population\_size\*(1 - self.preserve\_rate):  
 temp = self.get\_children()  
 new\_generation.put(Individual(get\_path\_length(temp), temp))  
  
 # add children individual and parent individual:  
 for i in range(0, int(self.population\_size\*self.preserve\_rate)):  
 new\_generation.put(self.population.get())  
  
 # change generation  
 self.population = new\_generation  
  
 self.generation\_count += 1

GenericAlgorithm类的generate\_children函数，即产生一个后代的具体过程：（1）选择，根据轮盘赌策略获取两个父本，（2）交叉，使用之前提到过的二交叉启发式算子，利用两个父本交叉产生孩子（3）变异，生成的孩子以一定概率变异（这里的变异操作与之前模拟退火种局部搜索操作一样，有交换两个城市（交换基因）/逆转子路径（逆转基因片段）两种）

def get\_children(self):

new\_children = []  
 # selection  
 parent1, parent2 = self.get\_two\_parent()  
 # crossover  
 new\_children = self.crossover\_1(parent1, parent2)  
 # mutation  
 if random.random() < self.mutation\_rate:  
 new\_children = self.mutation(new\_children)  
 return new\_children

GenericAlgorithm类的evaluate函数，即**估值函数**，使用1/path\_length作为个体适应度，对所有的适应度求和，使用轮盘赌的策略选择父本（这里仅仅是计算适应度、求和，供在get\_two\_parent中轮盘赌策略使用）

def evaluate(self):

self.population\_roulette = 0  
 for i in range(0, self.population\_size):  
 self.population\_roulette += 1/self.population.queue[i].score  
 # for i in range(0, self.population\_size):  
 # print(i, 'th sol:', self.population.queue[i].score, 'in ', self.generation\_count, 'generation')  
 if IF\_SHOW\_FIGURE and self.population.queue[0].score < self.min\_cost:  
 self.min\_cost = self.population.queue[0].score  
 self.min\_cost\_set.append(self.min\_cost)  
 self.best\_solution = self.population.queue[0].point\_list  
 self.show\_figure()  
 elif self.population.queue[0].score < self.min\_cost:  
 self.min\_cost = self.population.queue[0].score  
 self.min\_cost\_set.append(self.min\_cost)  
 self.best\_solution = self.population.queue[0].point\_list  
 print('best sol:', self.population.queue[0].score, 'in ', self.generation\_count, 'generation')

GenericAlgorithm类的get\_two\_parent函数，即**选择函数**，使用轮盘赌策略获取两个父本并返回

# selection

def get\_two\_parent(self):  
 parent1 = -1  
 parent2 = -1  
 while parent1 == parent2:  
 roulette\_value1 = random.uniform(0, self.population\_roulette)  
 for i in range(0, self.population\_size):  
 roulette\_value1 -= 1/self.population.queue[i].score  
 if roulette\_value1 < 0:  
 parent1 = i  
 break  
  
 roulette\_value2 = random.uniform(0, self.population\_roulette)  
 for i in range(0, self.population\_size):  
 roulette\_value2 -= 1/self.population.queue[i].score  
 if roulette\_value2 < 0:  
 parent2 = i  
 break  
 if parent1 == -1 or parent2 == -1:  
 print('error')  
 exit(0)  
 return parent1, parent2

GenericAlgorithm类的crossover函数，即**交叉**操作，使用之前选择的父本，使用算法概述中提到的二交叉启发式算子得到一个子代tsp解

def crossover\_1(self, parent1, parent2):

temp\_child1 = []  
 temp\_child2 = []  
 for i in range(0, self.problem\_size+1):  
 temp\_child1.append(self.population.queue[parent1].point\_list[i])  
 temp\_child2.append(self.population.queue[parent2].point\_list[i])  
  
 for i in range(1, self.problem\_size):  
 temp\_dist = min(get\_distance(temp\_child1[i - 1], temp\_child1[i]),  
 get\_distance(temp\_child2[i - 1], temp\_child2[i]))  
 if temp\_dist == get\_distance(temp\_child1[i-1], temp\_child1[i]):  
 temp\_child2[i:self.problem\_size] = cycle\_shift(temp\_child2[i:self.problem\_size], temp\_child1[i])  
 elif temp\_dist == get\_distance(temp\_child2[i-1], temp\_child2[i]):  
 temp\_child1[i:self.problem\_size] = cycle\_shift(temp\_child1[i:self.problem\_size], temp\_child2[i])  
  
 return temp\_child1

GenericAlgorithm类的mutation函数，即**变异**操作，对交叉产生的后代以一定概率进行变异操作，产生变异后代，这里自己实现了一下两种变异操作，分别对应于（1）交换两个城市（交换基因）（2）逆转tsp路径的子路径（逆转基因片段）两种局部搜索操作：

def mutation(self, new\_child):

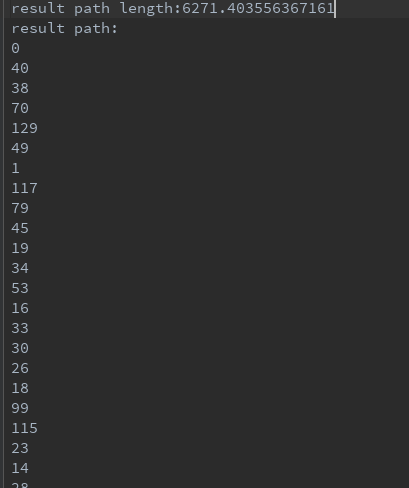
index1 = random.randint(1, self.problem\_size - 1)  
 index2 = random.randint(1, self.problem\_size - 1)  
 for i in range(index1, math.floor((index1+index2)/2) + 1):  
 temp = new\_child[i]  
 new\_child[i] = new\_child[index1+index2-i]  
 new\_child[index1+index2-i] = temp  
 return new\_child

# different mutation strategy by swap two point

def mutation\_1(self, new\_child):  
 index1 = random.randint(1, self.problem\_size - 1)  
 index2 = random.randint(1, self.problem\_size - 1)  
 temp = new\_child[index1]  
 new\_child[index1] = new\_child[index2]  
 new\_child[index2] = temp  
 return new\_child

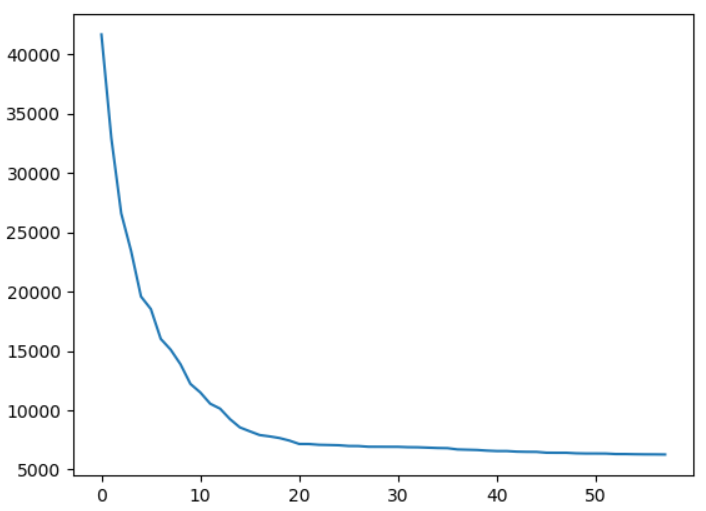
## 实验结果展示：

使用遗传算法求解tsp的最终解决方案：

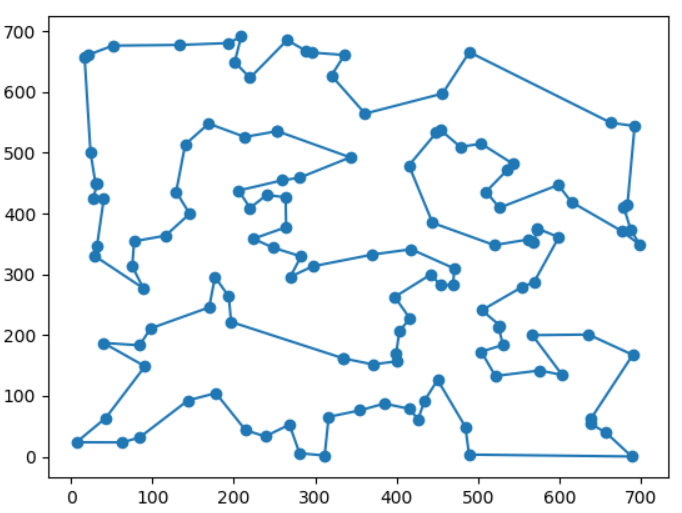


一共100行以上，所以只能部分截取，更多内容见res文件夹下的result.txt

该方法求解问题时随迭代次数增加，tsp解对应路径长度变化曲线：

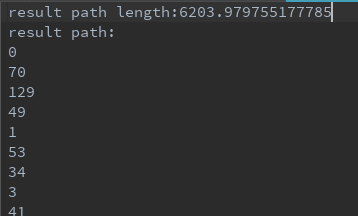


最终tsp路径连线图：

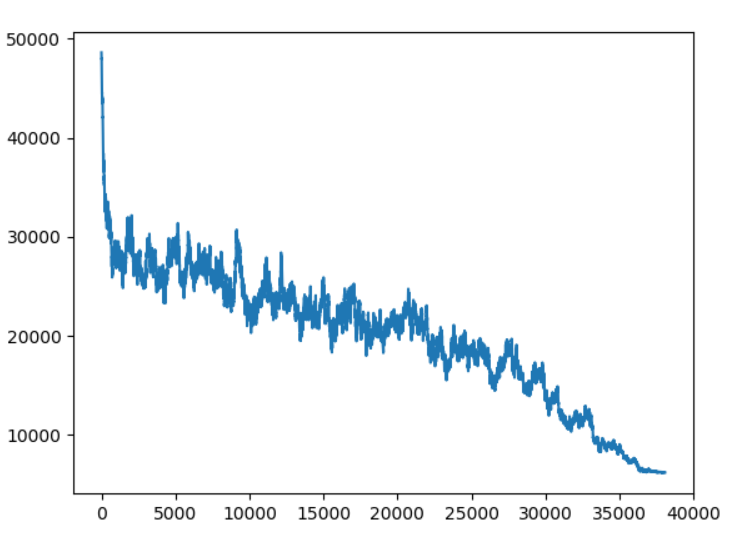


**对比使用SimulateAnneal类的solve函数求解tsp所得结果：**

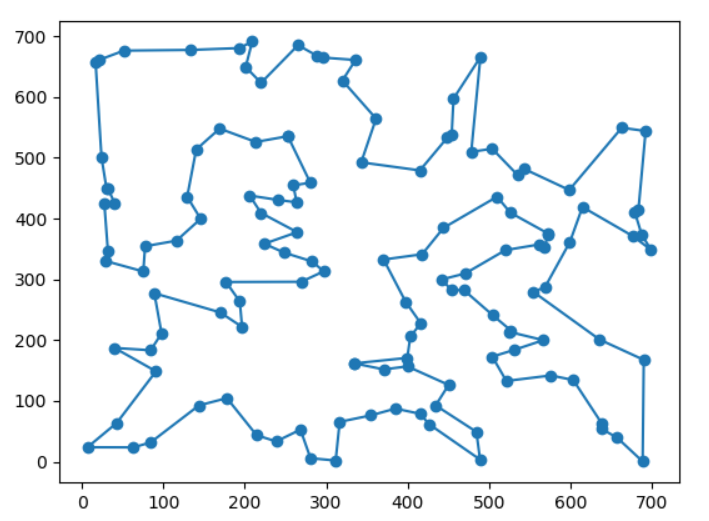
使用模拟退火求解的最终解决方案：



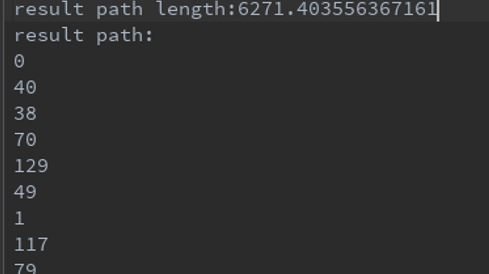
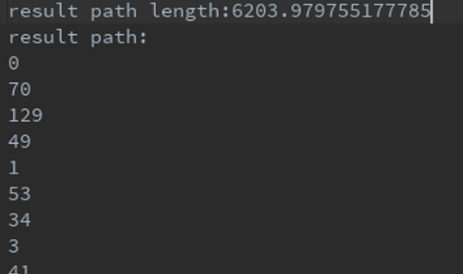
该方法求解问题时随迭代次数增加，tsp解对应路径长度变化曲线：

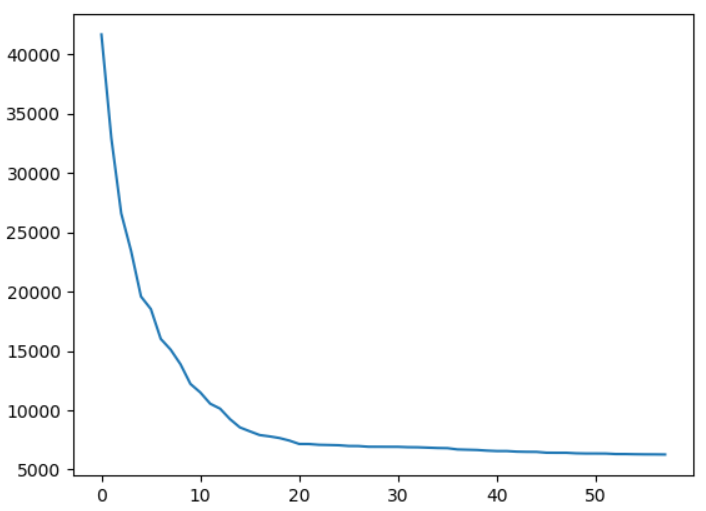
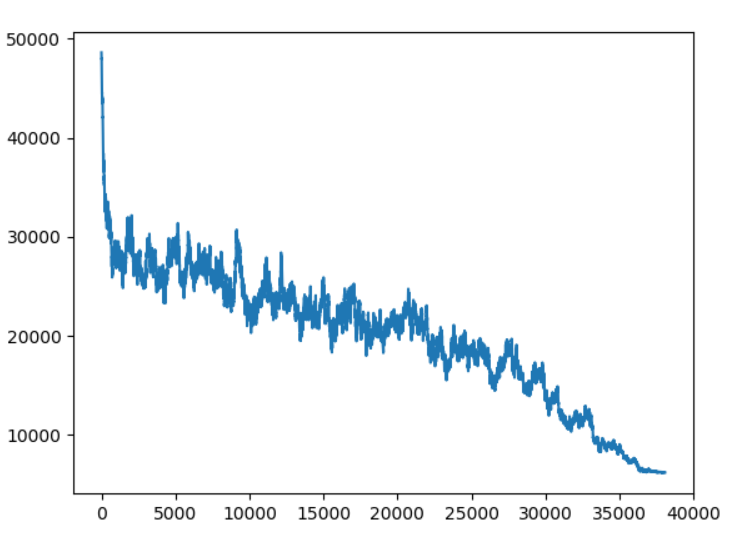


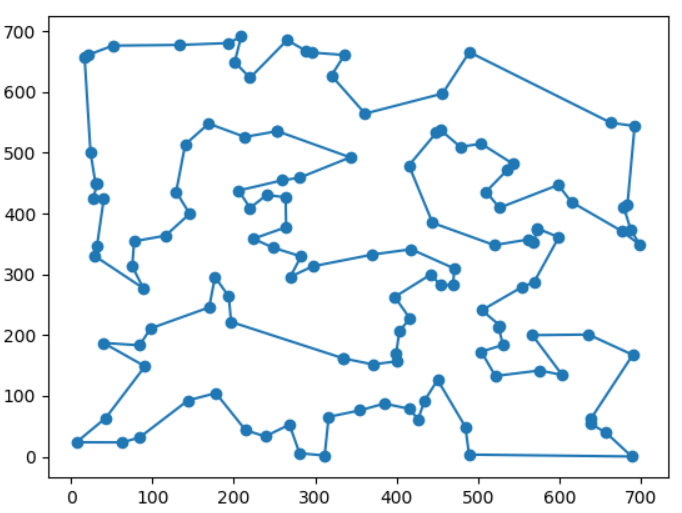
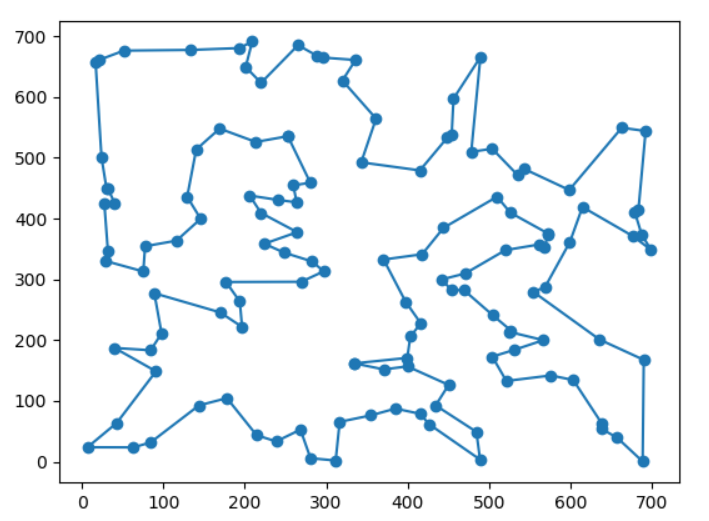
最终tsp路径连线图：



**放在一起对比：（左边GA，右边SA算法）**

对比上面的结果中我们看到：

1. 使用遗传算法求解tsp问题用时很长，虽然看上去只有几百上千代，但是每一代中有100个个体，所以总的计算量非常大，**用时长，这是缺点，遗传算法用时甚至比模拟退火长的多，更不用说局部搜索了**
2. 从迭代层数（代数）-tsp路径长度变化曲线来看，在开始的时候收敛的非常快（这是因为自己使用的是启发式交叉算子，所以收敛很快），到代数很大时收敛很慢，但是仍在缓慢变化，即它不会陷入局部最优，**这是优点，因为交叉和变异总能够产生新的子代个体，种群总是在动态变化的，所以不会陷入局部最优，这一点不同于局部搜索和模拟退火，局部搜索和模拟退火会都陷入局部最优**（即使模拟退火有一定跳出局部最优的能力，但是仍然会无法跳出一些局部最优）；
3. **遗传算法的最优解个体不会波动（不会变差），而模拟退火的解有可能变差**，因此遗传算法得到的最终解是整个搜索过程中的最佳答案，而模拟退火算法得到的最终解有可能不是；
4. 在这一次的遗传算法和求解vs这一次模拟退火中，**遗传算法得到的解法tsp路径长度为6271，模拟退火算法得到的tsp路径长度是6203，距离最优解6110分别在3%和2%的偏差之内，都达到了老师的要求**，从结果看来模拟退火要好一点点（好1%），这一方面得益于自己模拟退火参数设置非常好，另一方面在于遗传算法的种群多样性不足（主要是启发式交叉导致种群丧失了一定的多样性，使种群过于快速的收敛）。**实际上相比模拟退火，遗传算法有更大的潜力，只要种群多样性足够好，它能够一直向全局最优解靠近而不会陷入局部最优，而模拟退火尽管有一定跳出局部最优的能力，但是仍然是会陷入局部最优的**，尤其是在求解的末期，温度接近末温，接受差解能力很差，因而难以跳出局部最优。
5. 对比两个tsp解的路径连线图，都没有任何路径的交叉，只是在一些城市走法先后选择有所不同，这两个解都非常接近最优解了，从直观视觉效果看起来也都是不错的。
6. 经过多次计算求解，所得的解路径长度均限制在最优解6110的5%之内，即不会比最优解差5%以上

**关于路径变化过程的更多展示截图可以到我的github相关项目上看：**

https://github.com/ousuixin/TSP-/tree/master

## 总结经验&单点搜索vs多点搜索

**关于设计高效的遗传算法：**

遗传算法的核心包括（1）适应度估值；（2）选择；（3）交叉；（4）变异；

以上四个核心步骤对于高效的遗传算法设计都十分重要

1. 选择**合适的适应度估价算法**能够使得算法正确筛选出合适的父本，从而结合产生优秀的子代
2. 选择算子，根据每个个体的适应度选择父本，**选择算子的改进空间不大**，基本上常用的选择算子就是轮盘赌和锦标赛
3. 交叉算子，**交叉的设计非常重要，它决定了整个算法的收敛速度、算法潜力**，在完成本次实验的时候，一开始自己用的交叉算子是随即交叉（无启发式意义），结果整个种群收敛极慢，甚至无法收敛到50%之内，最后的结果很差，后来自己在网上查阅资料，了解到三交换启发式算子对tsp解进行交叉，从而极大程度改进了算法，算法不仅能够收敛，并且能够稳定收敛到5%内。但是这种启发式算子也带来了一些缺点----使得种群多样性不足，潜力下降，所以算法其实还能进一步改进，如果有很好的交叉操作，理论上能够兼顾收敛速度和算法潜力（种群多样性）
4. **变异操作也很重要，它影响算法的潜力（和种群多样性有关）**，较好的变异操作能够增加种群多样性，使得在算法后期也能够不断产生不同的个体，增加种群多样性，帮助种群向更加优的方向前进，而如果变异操作不够好，那么生成的子代总是和父代相似，这样就陷入一个局部无法跳出了。所以理论上很好的变异操作能够避免增加种群多样性，陷入局部最优，从而提升算法潜力

**单点搜索vs多点搜索：**

单点搜索（诸如模拟退火、局部搜索）都能够**比较快的得到比较优秀的解、可能具有一定跳过局部最优的能力但是仍然有可能陷入局部最优无法自拔**。至少模拟退火和局部搜索都是如此。

**多点搜索则能够避免陷入局部最优**（遗传算法），除非整个种群所有个体全部一样，那么就处在一个局部最优点，即使如此该种群也能够通过变异跳出局部最优，**总之只要给足够的计算时间，多点搜索具有很大的潜力**，能够一直向着最优解前进，**求解过程用时长是它的缺点**

单点搜索和多点搜索都需要设置好参数才能够有较好的表现，**多点搜索能够利用整体种群的力量求解问题，因此能够稳定得到好的解**，而**单点搜索使用个体求解问题，具有偶然性，因此单点搜索的最终解的好坏与初始解的选择有很大关系**（因此只有很小概率得到比较好的解）