

人工智能第四次实验报告

遗传算法求TSP问题

课程：人工智能原理	年级专业：19级软件工程
姓名：郑有为	学号：19335286

目录

目录

一、问题背景

- 1.1 遗传算法简介
- 1.2 遗传算法基本要素
- 1.3 遗传算法一般步骤
- 1.4 TSP问题简介
- 1.5 遗传算法求解TSP问题流程图

二、程序说明

- 2.1 控制参数
- 2.2 编码规则
- 2.3 选择初始群体
- 2.4 适应度函数
- 2.5 遗传操作
- 2.6 迭代过程

三、程序测试

- 3.1 求解不同规模的TSP问题的算法性能
- 3.2 种群规模对算法结果的影响
- 3.3 交叉概率对算法结果的影响
- 3.4 变异概率对算法结果的影响
- 3.5 交叉概率和变异概率对算法结果的影响

四、算法改进

- 4.1 块逆转变异策略
- 4.2 锦标赛选择法

五、实验总结

附录

- 附录 1 - 改进和测试代码

一、问题背景

1.1 遗传算法简介

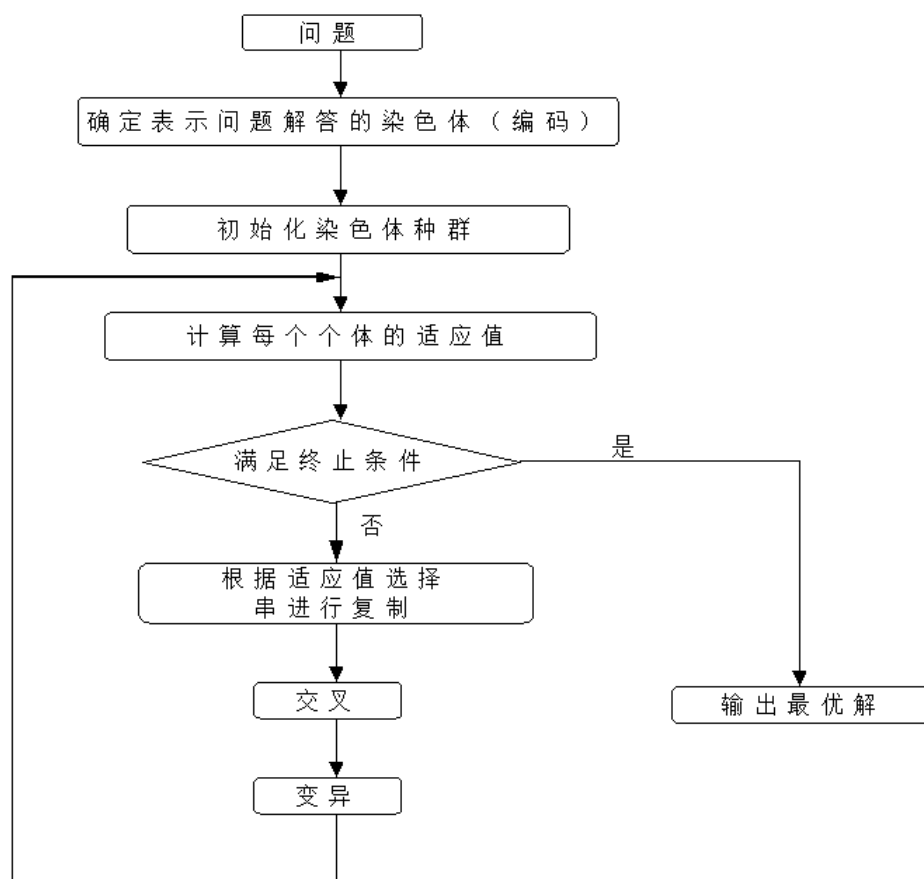
遗传算法是一种进化算法，基于自然选择和生物遗传等生物进化机制的一种搜索算法，其通过选择、重组和变异三种操作实现优化问题的求解。它的本质是从原问题的一组解出发改进到另一组较好的解，再从这组改进的解出发进一步改进。在搜索过程中，它利用结构和随机的信息，是满足目标的决策获得最大的生存可能，是一种概率型算法。

遗传算法主要借用生物中“适者生存”的原则，在遗传算法中，染色体对应的是数据或数组，通常由一维的串结构数据来表示。串上的各个位置对应一个基因座，而各个位置上所取的值对等位基因。遗传算法处理的是基因型个体，一定数量的个体组成了群体。群体的规模就是个体的数目。不同个体对环境的适应度不同，适应度高的个体被选择进行遗传操作产生新个体。每次选择两个染色体进行产生一组新染色体，染色体也可能发生变异，得到下一代群体。

1.2 遗传算法基本要素

1. **参数编码**：可以采用位串编码、实数编码、多参数级联编码等
2. **设定初始群体**：
 1. 启发 / 非启发给定一组解作为初始群体
 2. 确定初始群体的规模
3. **设定适应度函数**：将目标函数映射为适应度函数，可以进行尺度变换来保证非负、归一等特性
4. **设定遗传操作**：
 1. 选择：从当前群体选出一系列优良个体，让他们产生后代个体
 2. 交叉：两个个体的基因进行交叉重组来获得新个体
 3. 变异：随机变动个体串基因座上的某些基因
5. **设定控制参数**：例如变异概率、交叉程度、迭代上限等

1.3 遗传算法一般步骤



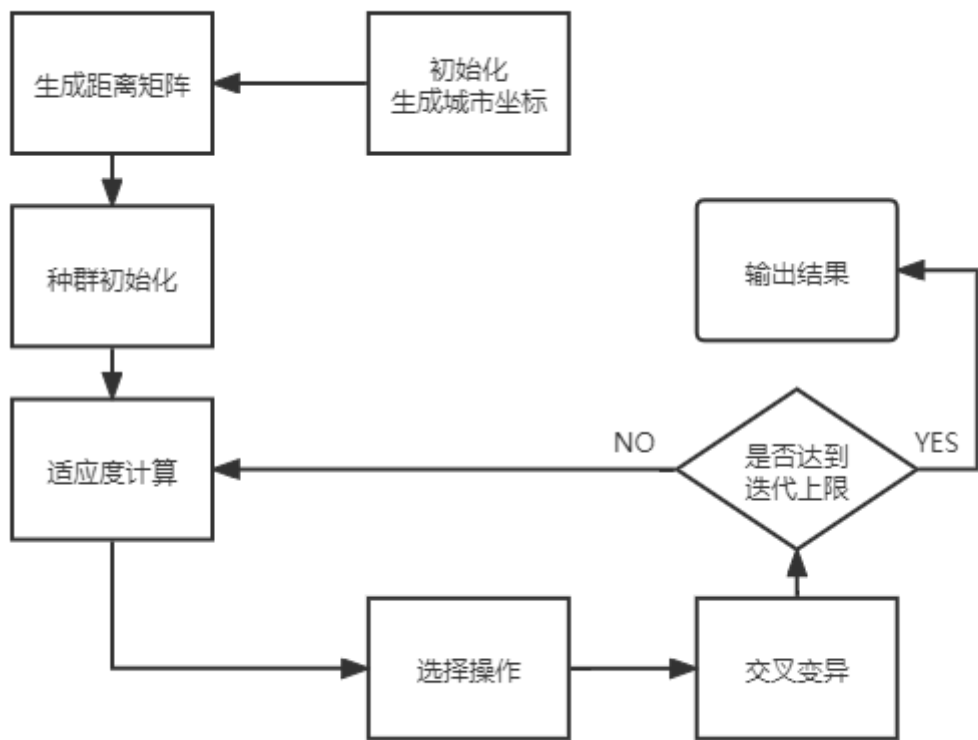
1.4 TSP问题简介

TSP问题 (Traveling Salesman Problem)：假设有一个商人要拜访 n 个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

一般可以用一个无向加权图来对TSP问题建模，为了简化编程，我们将若干个城市表示为平面上的若干个点，两个城市的距离即两个点的欧氏距离。

TSP问题是一个组合优化问题，暴力求解的时间复杂度为 $2^n n^2$ 。该问题是NP完全问题，这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。本实验所采用遗传算法就是一种近似算法。

1.5 遗传算法求解TSP问题流程图



二、程序说明

2.1 控制参数

变量	默认值	含义
DNA_SIZE	20	编码长度，城市个数
POP_SIZE	200	种群大小
CROSS_RATE	0.6	交叉率
MUTA_RATE	0.2	变异率
Iterations	1000	迭代次数

2.2 编码规则

对TSP问题，遗传算法首先对每一个城市编号，编码为一个长度为城市数目的一维数组 A，每个元素为一个城市编号，数组决定了城市的顺序，即从第 A[i] 个城市走到第 A[i+1] 个城市。

在实现中，使用 shuffle 函数来打乱一个数组：

```
1 list = list(range(DNA_SIZE))
2 random.shuffle(list)
```

2.3 选择初始群体

不依靠任何经验，直接随机生成 POP_SIZE 个随机 DNA。

```
1 pop = []
2 list = list(range(DNA_SIZE))
3 for i in range(POP_SIZE):
4     random.shuffle(list)
5     l = list.copy()
6     pop.append(l)
```

2.4 适应度函数

适应度与该 DNA 对应的旅行距离有关，TSP问题又是一个最小值问题，故旅行距离越小的 DNA 适应度越大，故采用旅行距离的倒数来作为适应度函数。

```
1 fitness = 1/(distance(DNA))
```

程序中 getfitness 函数返回的是种群旅行距离的倒数减去旅行距离最小值 + 0.000001，加上一个很小的数是为了避免为 0 时取不了概率。

```
1 fitness = 1/(distance(DNA)) - min_fitness + 0.000001
```

2.5 遗传操作

- **选择**：根据适应度选择，以赌轮盘的形式，适应度越大的个体被选中的概率越大，每个DNA的选择概率为：

```
1 p = (fitness/fitness.sum())
```

- **交叉变异**：对于每一个DNA
 - 有 **(1 - CROSS_RATE)** 的概率不发生交叉变异，直接进入下一代
 - 有 **CROSS_RATE** 的概率发生交叉和变异：
 - **交叉**具体指：选取种群中另一个个体进行交叉，产生2个不相等的节点，中间部分作为交叉段，采用部分匹配交叉，在这个过程中，需要排除不合法的个体，将不合法的个体转换为合法个体。
 - **不合法指**：一个DNA中有两个相同的城市编号。
 - **非法个体转换为合法个体**的方法：遍历原DNA除去交叉片段后剩下的DNA片段，如果其中有与待插入的新片段相同的城市编号，则找出这个相同的城市编号在原DNA同位置编号的位置的城市编号，循环查找，直至找到的城市编号不再在插入的片段中，最后修改原DNA片段中该位置的城市编号为这个新城市编号。
 - **变异**：以 **MUTA_RATE** 的概率发生变异，变异行为为随机选取两个不同的位置，将DNA上这两个位置的城市编号交换。

2.6 迭代过程

```

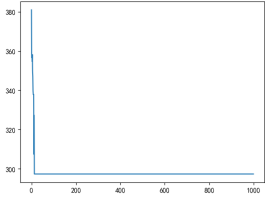
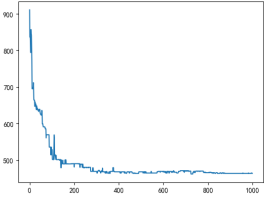
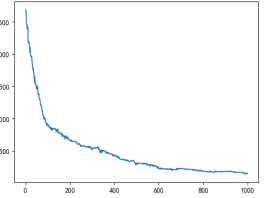
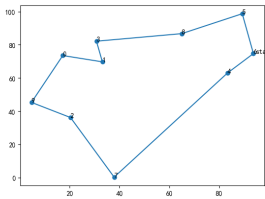
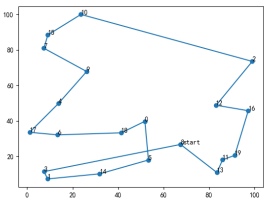
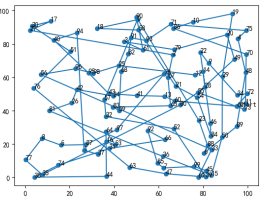
1 for i in range(Iterations):      # 迭代 N 代
2     pop = crossmuta(pop, CROSS_RATE)  # 交叉变异
3     fitness = getfitness(pop)        # 计算适应度
4     maxfitness = np.argmax(fitness)  # 选取最佳适应度DNA
5     best_dis.append(distance(pop[maxfitness])) # 保存结果
6     pop = select(pop, fitness)      # 选择生成新的种群

```

三、程序测试

3.1 求解不同规模的TSP问题的算法性能

用遗传算法求解不同规模（如10个城市，20个城市，100个城市）的TSP问题

城市规模	10个城市	20个城市	100个城市
迭代效果			
最佳路线			
运行时间	8.5秒	14.5秒	65.5秒
最短距离	297.35091447158914	463.9137406306509	2150.158898442909

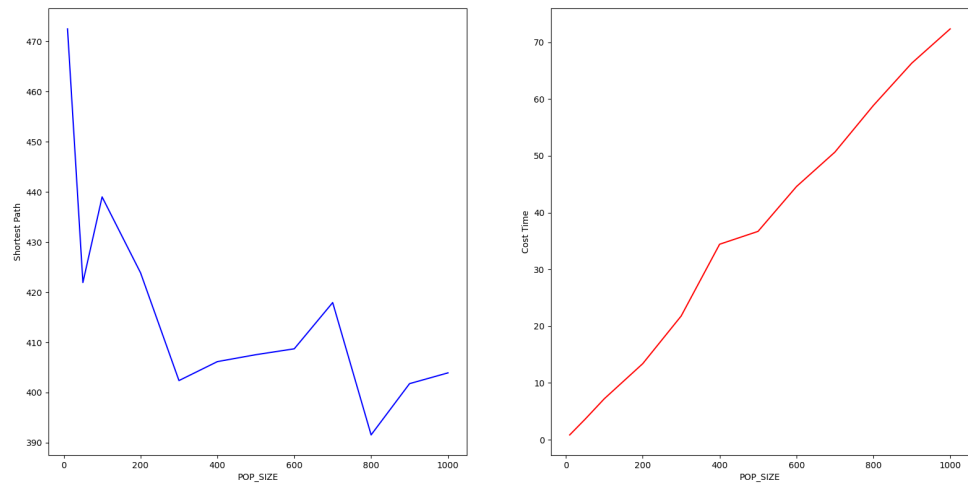
实验分析：对于迭代效果，可以看到随着城市数目的增加，收敛速度放缓。粗略统计：规模为10时，第10次迭代就开始收敛，而规模为20时到第300次迭代才开始收敛，而规模为1000时，第800次才开始有收敛的迹象；运行时间和最短距离随城市规模的增加而呈线性增长。

3.2 种群规模对算法结果的影响

测试参数：对于每一次测试，City_Map都是相同的，DNA_SIZE = 20, Iterations = 1000, CROSS_RATE = 0.6, MUTA_RATE = 0.2。

变量	值	备注
POP_SIZE	[10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]	每个值测试3次求平均数以降低随机误差

测试结果：如图所示。



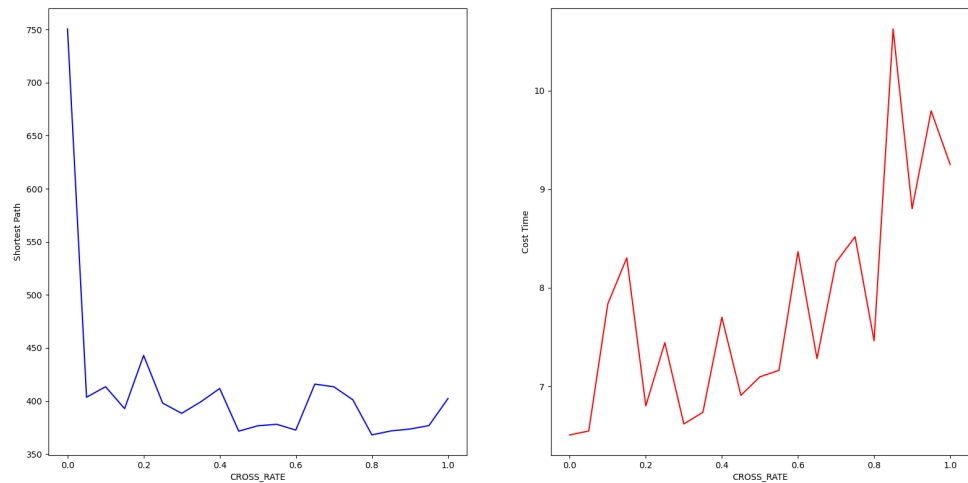
测试分析：随着种群数目的增加，算法求解效果越好，在种群数目大于500后，求出的最短路径变化不大；同时，程序耗时随种群规模的增加呈线性增长。

3.3 交叉概率对算法结果的影响

测试参数：对于每一次测试，City_Map都是相同的，DNA_SIZE = 20, POP_SIZE = 100, Iterations = 1000, MUTA_RATE = 0.2。

变量	值	备注
CROSS_RATE	[0,1], 间隔0.05	每个值测试3次求平均数以降低随机误差

测试结果：如图所示。



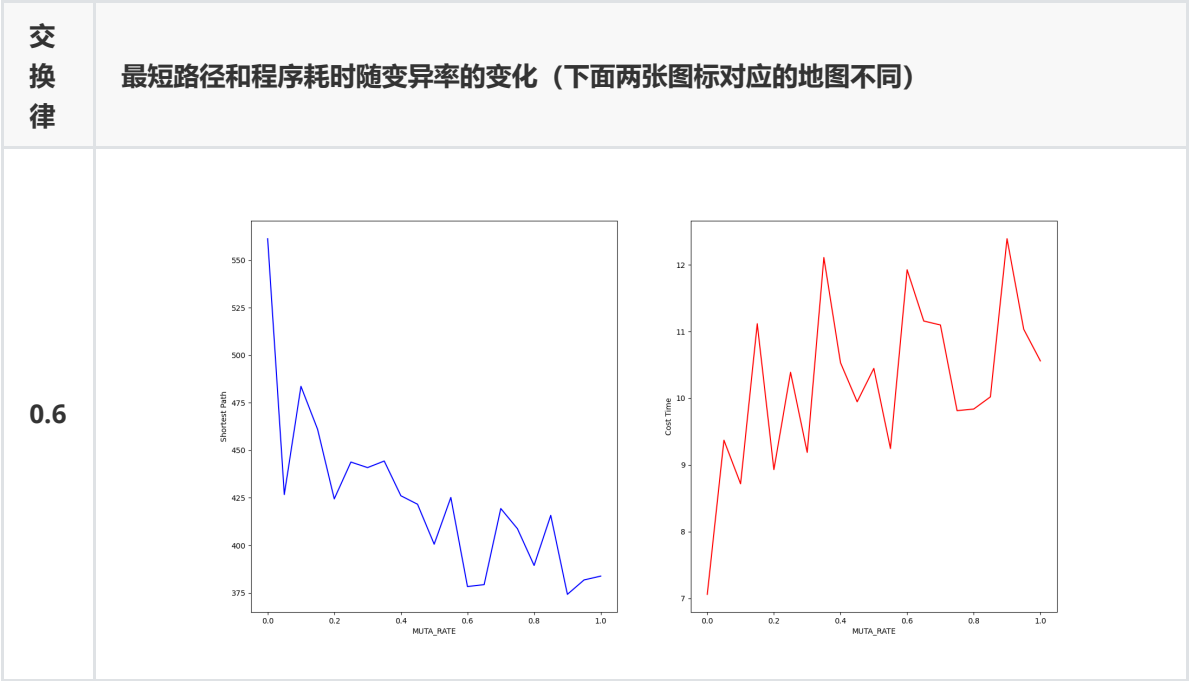
测试分析：由于种群数目大，当交叉率大于0.05后，算法都能得到比较好的结果；在耗时上，受随机性影响，耗时不稳定，但总体随交叉率的增大而增多。

3.4 变异概率对算法结果的影响

测试参数：对于每一次测试，City_Map都是相同的，DNA_SIZE = 20, POP_SIZE = 100, Iterations = 1000, CROSS_RATE = 0.6 或 0。

变量	值	备注
MUTA_RATE	[0,1], 间隔0.05	每个值测试3次求平均数以降低随机误差

测试结果：如图所示。



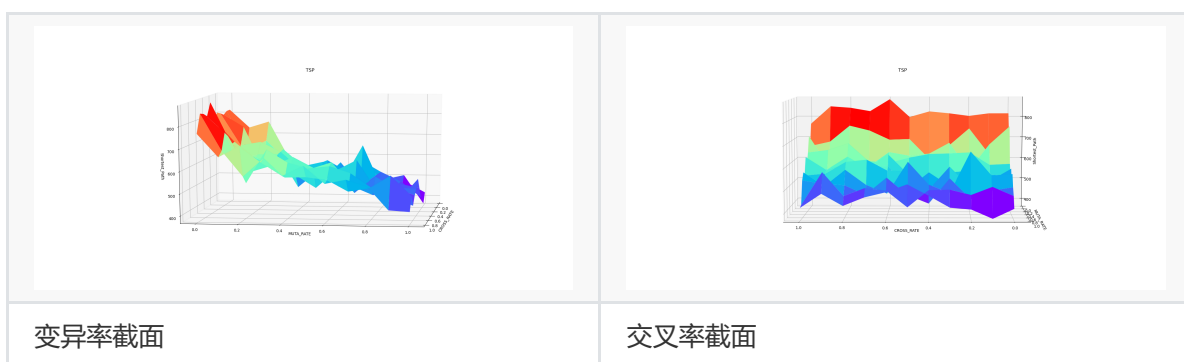
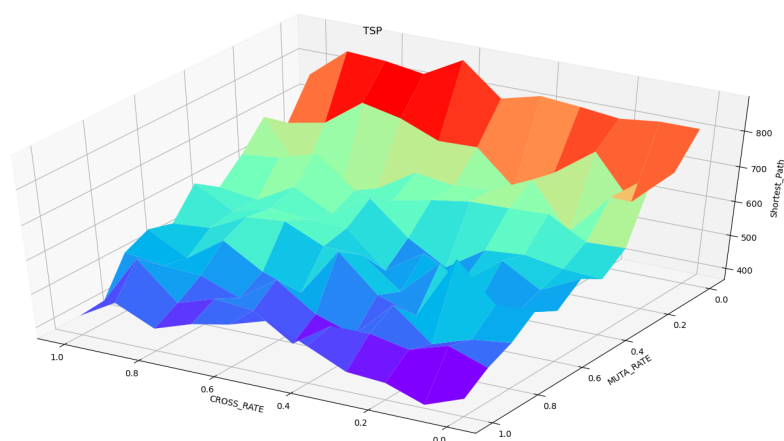
测试分析：从上图可以看出，变异率越大，求出来的最短路径越短，因此高变异率对与本问题是有有利的；在时间损耗上，耗时随变异率的升高有上升趋势。

3.5 交叉概率和变异概率对算法结果的影响

测试参数：下面尝试同时考虑交叉率和变异率对算法结果的影响。对于每一次测试，City_Map都是相同的，DNA_SIZE = 20, POP_SIZE = 100, Iterations = 1000。

变量	值
CROSS_RATE	[0,1], 间隔0.1
MUTA_RATE	[0,1], 间隔0.1

测试结果：如下图所示，红色代表最短路径长，即效果不佳，蓝色代表求出来的最短路径越优。



测试分析：我们可以看到总体而言，高变异率的效果优于低变异率，而交换律的带来的效果在本图中并不明显，与3.3和3.4的测试结果一致。

四、算法改进

要求：增加一种变异策略和一种个体选择概率分配策略，比较求解同一TSP问题时不同变异策略及不同个体选择分配策略对算法结果的影响。

4.1 块逆转变异策略

采用块逆转变异 Block-reversal 作为TSP的变异策略：在父代中随机选择两个点，然后反转之间的部分，这种变异方法特别适合像TSP这样的问题，即邻接关系。

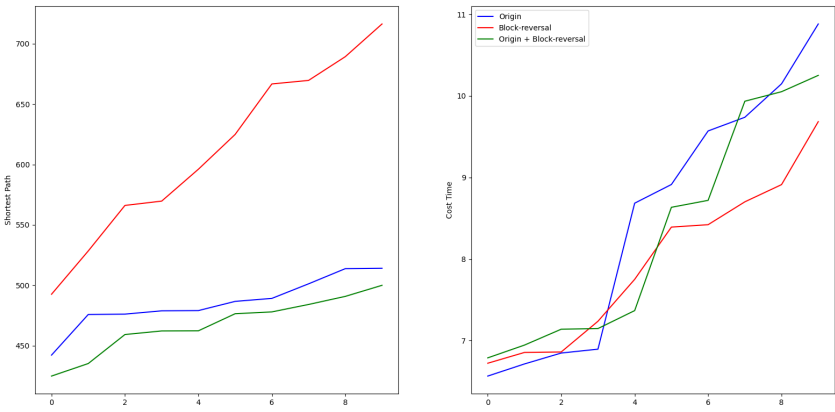
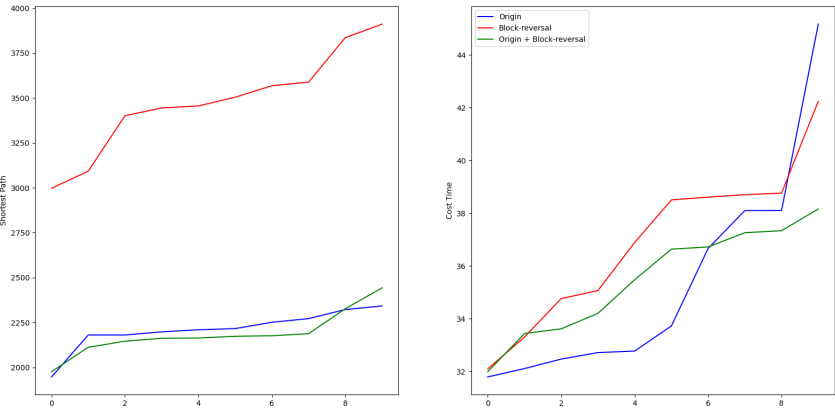
使用这种方法的一个好处是，这种方法只会破坏与两个点直接相连的边，而不会破坏两点之间和两点之外的部分，可以保留中间部分和外部部分的特征，确保优秀的基因不会被严重破坏。

实现非常简单，只需在源代码上添加一行代码即可实现：

```
1 DNA[mutate_point1:mutate_point2].reverse()
```

测试结果：（曲线上升是做了排序处理，为了更好比较不同策略的结果）

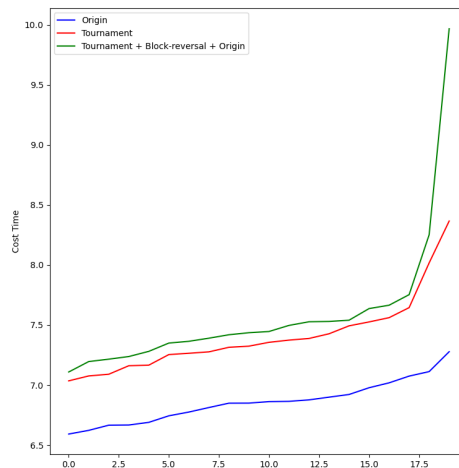
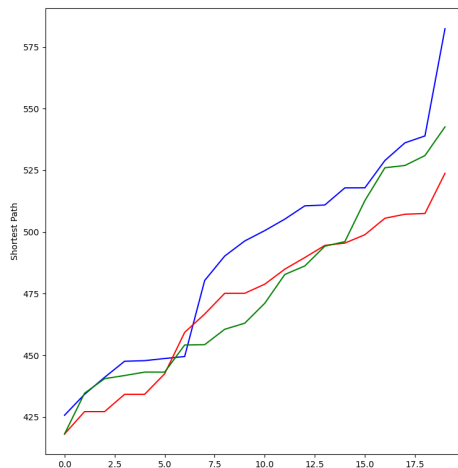
测试结果如图所示。其中，红色曲线是采用块逆转变异的结果，而蓝色是采用原程序的变异策略的结果，可以看到原程序的变异策略明显在性能上更优；值得注意的是绿色曲线，这是同时使用原程序变异策略和块逆转变异策略的结果。从结果上看，同时使用两种变异策略所达到的效果优于单独使用其中一种。

问题规模	运行结果
20个城市	 <p>Figure 1: Performance comparison for 20 cities. The left graph shows the shortest path length, and the right graph shows the cost time. The 'Origin + Block-reversal' strategy (green line) consistently achieves the lowest shortest path length and the lowest cost time across all iterations.</p>
100个城市	 <p>Figure 2: Performance comparison for 100 cities. The left graph shows the shortest path length, and the right graph shows the cost time. The 'Origin + Block-reversal' strategy (green line) consistently achieves the lowest shortest path length and the lowest cost time across all iterations.</p>

4.2 锦标赛选择法

锦标赛选择法 (tournament selection)：每次从种群中取出一定数量个体（成为竞赛规模），然后选择其中最好的一个进入子代种群。重复该操作，直到新的种群规模达到原来的种群规模。

测试结果：20次测试结果如图所示，（曲线上升是做了排序处理，为了更好比较不同策略的结果）其中，红色曲线为使用锦标赛选择法的运行结果，绿色曲线为使用锦标赛选择法和块逆转变异策略的运行结果，蓝色曲线为原程序运行结果。观察左图，可以看到锦标赛选择法的性能要优于原程序所采用的赌轮盘选择法；观察右图，可以看到锦标赛选择法的耗时要高于原程序所采用的赌轮盘选择法。



五、实验总结

通过本次实验，我们进一步掌握遗传算法的原理、流程和编码策略，重点理解求解TSP问题的流程并测试主要参数对结果的影响，掌握遗传算法的基本实现方法，在原程序的基础上添加了块逆转变异的变异策略和锦标赛选择法。

附录

电子版报告和代码地址：<https://gitee.com/WondrousWisdomcard/ai-homework>

附录 1 - 改进和测试代码

```

1  import numpy as np
2  import random
3  import matplotlib.pyplot as plt
4  import copy
5  import time
6
7  from matplotlib.ticker import MultipleLocator
8  from scipy.interpolate import interpolate
9
10 CITY_NUM = 20
11 City_Map = 100 * np.random.rand(CITY_NUM, 2)
12
13 DNA_SIZE = CITY_NUM      #编码长度
14 POP_SIZE = 100           #种群大小
15 CROSS_RATE = 0.6         #交叉率
16 MUTA_RATE = 0.2          #变异率
17 Iterations = 1000        #迭代次数
18
19 # 根据DNA的路线计算距离
20 def distance(DNA):
21     dis = 0
22     temp = City_Map[DNA[0]]
23     for i in DNA[1:]:
24         dis = dis + ((City_Map[i][0]-temp[0])**2+(City_Map[i][1]-
25 temp[1])**2)**0.5
26         temp = City_Map[i]
27     return dis+((temp[0]-City_Map[DNA[0]][0])**2+(temp[1]-City_Map[DNA[0]]
28 [1])**2)**0.5

```

```

27
28 # 计算种群适应度，这里适应度用距离的倒数表示
29 def getfitness(pop):
30     temp = []
31     for i in range(len(pop)):
32         temp.append(1/(distance(pop[i])))
33     return temp-np.min(temp) + 0.000001
34
35 # 选择：根据适应度选择，以赌轮盘的形式，适应度越大的个体被选中的概率越大
36 def select(pop, fitness):
37     s = fitness.sum()
38     temp = np.random.choice(np.arange(len(pop)), size=POP_SIZE,
39                             replace=True,p=(fitness/s))
40     p = []
41     for i in temp:
42         p.append(pop[i])
43     return p
44
45 # 4.2 选择：锦标赛选择法
46 def selectII(pop, fitness):
47     p = []
48     for i in range(POP_SIZE):
49         temp1 = np.random.randint(POP_SIZE)
50         temp2 = np.random.randint(POP_SIZE)
51         DNA1 = pop[temp1]
52         DNA2 = pop[temp2]
53         if fitness[temp1] > fitness[temp2]:
54             p.append(DNA1)
55         else:
56             p.append(DNA2)
57     return p
58
59 # 变异：选择两个位置互换其中的城市编号
60 def mutation(DNA, MUTA_RATE):
61     if np.random.rand() < MUTA_RATE: # 以MUTA_RATE的概率进行变异
62         # 随机产生两个实数，代表要变异基因的位置，确保两个位置不同，将2个所选位置进行互
63         换
64         mutate_point1 = np.random.randint(0, DNA_SIZE)
65         mutate_point2 = np.random.randint(0,DNA_SIZE)
66         while(mutate_point1 == mutate_point2):
67             mutate_point2 = np.random.randint(0,DNA_SIZE)
68         DNA[mutate_point1],DNA[mutate_point2] =
69         DNA[mutate_point2],DNA[mutate_point1]
70
71 # 4.1 变异：在父代中随机选择两个点，然后反转之间的部分
72 def mutationII(DNA, MUTA_RATE):
73     if np.random.rand() < MUTA_RATE:
74         mutate_point1 = np.random.randint(0, DNA_SIZE)
75         mutate_point2 = np.random.randint(0, DNA_SIZE)
76         while (mutate_point1 == mutate_point2):
77             mutate_point2 = np.random.randint(0, DNA_SIZE)
78         if(mutate_point1 > mutate_point2):
79             mutate_point1, mutate_point2 = mutate_point2, mutate_point1
80         DNA[mutate_point1:mutate_point2].reverse()
81
82 # 4.1 变异：调用 I 和 II
83 def mutationIII(DNA, MUTA_RATE):
84     mutationII(DNA, MUTA_RATE)

```

```

82     mutation(DNA, MUTA_RATE)
83
84     # 交叉变异
85     # muta = 1时变异调用 mutation;
86     # muta = 2时变异调用 mutationII;
87     # muta = 3时变异调用 mutationIII
88     def crossmuta(pop, CROSS_RATE, muta=1):
89         new_pop = []
90         for i in range(len(pop)): # 遍历种群中的每一个个体，将该个体作为父代
91             n = np.random.rand()
92             if n >= CROSS_RATE: # 大于交叉概率时不发生变异，该子代直接进入下一代
93                 temp = pop[i].copy()
94                 new_pop.append(temp)
95             # 小于交叉概率时发生变异
96             if n < CROSS_RATE:
97                 # 选取种群中另一个个体进行交叉
98                 list1 = pop[i].copy()
99                 list2 = pop[np.random.randint(POP_SIZE)].copy()
100                 status = True
101                 # 产生2个不相等的节点，中间部分作为交叉段，采用部分匹配交叉
102                 while status:
103                     k1 = random.randint(0, len(list1) - 1)
104                     k2 = random.randint(0, len(list2) - 1)
105                     if k1 < k2:
106                         status = False
107
108                 k11 = k1
109
110                 # 两个DNA中待交叉的片段
111                 fragment1 = list1[k1: k2]
112                 fragment2 = list2[k1: k2]
113
114                 # 交换片段后的DNA
115                 list1[k1: k2] = fragment2
116                 list2[k1: k2] = fragment1
117
118                 # left1就是 list1除去交叉片段后剩下的DNA片段
119                 del list1[k1: k2]
120                 left1 = list1
121
122                 offspring1 = []
123                 for pos in left1:
124                     # 如果 left1 中有与待插入的新片段相同的城市编号
125                     if pos in fragment2:
126                         # 找出这个相同的城市编号在原DNA同位置编号的位置的城市编号
127                         # 循环查找，直至这个城市编号不再待插入的片段中
128                         pos = fragment1[fragment2.index(pos)]
129                         while pos in fragment2:
130                             pos = fragment1[fragment2.index(pos)]
131                         # 修改原DNA片段中该位置的城市编号为这个新城市编号
132                         offspring1.append(pos)
133                         continue
134                 offspring1.append(pos)
135                 for i in range(0, len(fragment2)):
136                     offspring1.insert(k11, fragment2[i])
137                     k11 += 1
138                 temp = offspring1.copy()
139                 # 根据 type 的值选择一种变异策略

```

```

140         if muta == 1:
141             mutation(temp, MUTA_RATE)
142         elif muta == 2:
143             mutationII(temp, MUTA_RATE)
144         elif muta == 3:
145             mutationIII(temp, MUTA_RATE)
146         # 把部分匹配交叉后形成的合法个体加入到下一代种群
147         new_pop.append(temp)
148
149     return new_pop
150
151 def print_info(pop):
152     fitness = getfitness(pop)
153     maxfitness = np.argmax(fitness)      # 得到种群中最大适应度个体的索引
154     print("最优的基因型: ", pop[maxfitness])
155     print("最短距离: ", distance(pop[maxfitness]))
156     # 按最优结果顺序把地图上的点加入到best_map列表中
157     best_map = []
158     for i in pop[maxfitness]:
159         best_map.append(City_Map[i])
160     best_map.append(City_Map[pop[maxfitness][0]])
161     x = np.array((best_map))[:,0]
162     y = np.array((best_map))[:,1]
163     # 绘制地图以及路线
164     plt.figure()
165     plt.rcParams['font.sans-serif'] = ['SimHei']
166     plt.scatter(x,y)
167     for dot in range(len(x)-1):
168         plt.annotate(pop[maxfitness][dot],xy=(x[dot],y[dot]),xytext =
169 (x[dot],y[dot]))
170         plt.annotate('start',xy=(x[0],y[0]),xytext = (x[0]+1,y[0]))
171     plt.plot(x,y)
172
173 # 3.2 种群规模对算法结果的影响
174 def pop_size_test():
175     global POP_SIZE
176     ITE = 3 # 每个值测试多次求平均数以降低随机误差
177     i_list = [10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
178     b_list = []
179     t_list = []
180     for i in i_list:
181         print(i)
182         POP_SIZE = i
183         time_cost = 0
184         min_path = 0
185         for j in range(ITE):
186             time_start = time.time()
187             ans = tsp_solve()
188             min_path += min(ans)
189             time_end = time.time()
190             time_cost += time_end - time_start
191
192         b_list.append(min_path / ITE)
193         t_list.append(time_cost / ITE)
194     show_test_result(i_list, b_list, t_list, "POP_SIZE")
195
196 # 3.3 交叉概率对算法结果的影响
197 def cross_rate_test():

```

```

197     global CROSS_RATE
198     ITE = 3 # 每个值测试多次求平均数以降低随机误差
199     i_list = range(0, 21)
200     b_list = []
201     t_list = []
202     ii_list = [] # [0, 0.05, 0.1, ... 0.95, 1]
203     for i in i_list:
204         print(i)
205         CROSS_RATE = 0.05 * i
206         ii_list.append(CROSS_RATE)
207         time_cost = 0
208         min_path = 0
209         for j in range(ITE):
210             time_start = time.time()
211             ans = tsp_solve()
212             min_path += min(ans)
213             time_end = time.time()
214             time_cost += time_end - time_start
215
216         b_list.append(min_path / ITE)
217         t_list.append(time_cost / ITE)
218     show_test_result(ii_list, b_list, t_list, "CROSS_RATE")
219
220 # 3.4 变异概率对算法结果的影响
221 def muta_rate_test():
222     global MUTA_RATE
223     ITE = 3 # 每个值测试多次求平均数以降低随机误差
224     i_list = range(0, 21)
225     b_list = []
226     t_list = []
227     ii_list = [] # [0, 0.05, 0.1, ... 0.95, 1]
228     for i in i_list:
229         print(i)
230         MUTA_RATE = 0.05 * i
231         ii_list.append(MUTA_RATE)
232         time_cost = 0
233         min_path = 0
234         for j in range(ITE):
235             time_start = time.time()
236             ans = tsp_solve()
237             min_path += min(ans)
238             time_end = time.time()
239             time_cost += time_end - time_start
240
241         b_list.append(min_path / ITE)
242         t_list.append(time_cost / ITE)
243     show_test_result(ii_list, b_list, t_list, "MUTA_RATE")
244
245 # 3.5 交叉概率和变异概率对算法结果的影响
246 def cross_muta_test():
247     s = np.array([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
248     x, y = np.meshgrid(s,s)
249     z = np.zeros(shape=(11, 11))
250
251     global MUTA_RATE
252     global CROSS_RATE
253     for i in range(11):
254         for j in range(11):

```

```

255         print(str(i) + ":" + str(j))
256         CROSS_RATE = X[0,i]
257         MUTA_RATE = Y[0,j]
258         ans = tsp_solve()
259         Z[i, j] = min(ans)
260
261     ax = plt.axes(projection='3d')
262     ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='rainbow',
edgecolor='none')
263     ax.set_xlabel("CROSS_RATE")
264     ax.set_ylabel("MUTA_RATE")
265     ax.set_zlabel("Shortest_Path")
266     ax.set_title('TSP')
267     plt.show()
268
269 # 3.2-3.4 生成参数测试结果的可视化图表
270 def show_test_result(i_list, b_list, t_list, msg):
271     ax1 = plt.subplot(121)
272     ax1.plot(i_list, b_list, 'b')
273     ax1.set_xlabel(msg)
274     ax1.set_ylabel("Shortest Path")
275
276     ax2 = plt.subplot(122)
277     ax2.plot(i_list, t_list, 'r')
278     ax2.set_xlabel(msg)
279     ax2.set_ylabel("Cost Time")
280     plt.show()
281
282 # 求解TSP问题并返回最大值
283 # muta 指定变异方式, sel 指定选择方式
284 def tsp_solve(muta=1, sel=1):
285     pop = []
286     li = list(range(DNA_SIZE))
287     for i in range(POP_SIZE):
288         random.shuffle(li)
289         l = li.copy()
290         pop.append(l)
291     best_dis = []
292     # 进行选择, 交叉, 变异, 并把每代的最优个体保存在best_dis中
293     for i in range(Iterations): # 迭代N代
294         pop = crossmuta(pop, CROSS_RATE, muta=muta)
295         fitness = getfitness(pop)
296         maxfitness = np.argmax(fitness)
297         best_dis.append(distance(pop[maxfitness]))
298         if sel == 1:
299             pop = select(pop, fitness) # 选择生成新的种群
300         elif sel == 2:
301             pop = selectII(pop, fitness) # 选择生成新的种群
302
303     return best_dis
304
305 # 4.1 块逆转变异策略对比测试
306 def opt1_test():
307     ITE = 20 # 测试次数
308     i_list = range(ITE)
309     b_list = [] # 每次求出的最短路径
310     t_list = [] # 每次求解的耗时
311     b_listII = []

```

```

312     t_listII = []
313     b_listIII = []
314     t_listIII = []
315
316     for i in i_list:
317         print(i)
318         # I. 原两点互换策略
319         time_start = time.time()
320         b_list.append(min(tsp_solve(muta=1)))
321         time_end = time.time()
322         t_list.append(time_end - time_start)
323         # II. 块逆转变异策略
324         time_startII = time.time()
325         b_listII.append(min(tsp_solve(muta=2)))
326         time_endII = time.time()
327         t_listII.append(time_endII - time_startII)
328         # III. 同时使用上述两种变异策略
329         time_startIII = time.time()
330         b_listIII.append(min(tsp_solve(muta=3)))
331         time_endIII = time.time()
332         t_listIII.append(time_endIII - time_startIII)
333
334     # 做排序处理，方便比较
335     b_list.sort()
336     t_list.sort()
337     b_listII.sort()
338     t_listII.sort()
339     b_listIII.sort()
340     t_listIII.sort()
341
342     ax1 = plt.subplot(121)
343     ax1.plot(i_list, b_list, 'b', label="Origin")
344     ax1.plot(i_list, b_listII, 'r', label="Block-reversal")
345     ax1.plot(i_list, b_listIII, 'g', label="Origin + Block-reversal")
346     ax1.set_ylabel("Shortest Path")
347     ax2 = plt.subplot(122)
348     ax2.plot(i_list, t_list, 'b', label="Origin")
349     ax2.plot(i_list, t_listII, 'r', label="Block-reversal")
350     ax2.plot(i_list, t_listIII, 'g', label="Origin + Block-reversal")
351     ax2.set_ylabel("Cost Time")
352     plt.legend()
353     plt.show()
354
355 # 4.2 锦标赛选择策略对比测试
356 def opt2_test():
357     ITE = 20 # 测试次数
358     i_list = range(ITE)
359     b_list = [] # 每次求出的最短路径
360     t_list = [] # 每次求解的耗时
361     b_listII = []
362     t_listII = []
363     b_listIII = []
364     t_listIII = []
365
366     for i in i_list:
367         print(i)
368         # I. 原赌轮盘选择策略
369         time_start = time.time()

```



```

370     b_list.append(min(tsp_solve(sel=1)))
371     time_end = time.time()
372     t_list.append(time_end - time_start)
373     # II. 锦标赛选择策略
374     time_startII = time.time()
375     b_listII.append(min(tsp_solve(sel=2)))
376     time_endII = time.time()
377     t_listII.append(time_endII - time_startII)
378     # III. 锦标赛选择策略 + 两点互换变异 + 块逆转变异策略
379     time_startIII = time.time()
380     b_listIII.append(min(tsp_solve(sel=2,muta=3)))
381     time_endIII = time.time()
382     t_listIII.append(time_endIII - time_startIII)
383
384     # 做排序处理，方便比较
385     b_list.sort()
386     t_list.sort()
387     b_listII.sort()
388     t_listII.sort()
389     b_listIII.sort()
390     t_listIII.sort()
391
392     ax1 = plt.subplot(121)
393     ax1.plot(i_list, b_list, 'b', label="Origin")
394     ax1.plot(i_list, b_listII, 'r', label="Tournament")
395     ax1.plot(i_list, b_listIII, 'g', label="Tournament + Block-reversal +
Origin")
396     ax1.set_ylabel("Shortest Path")
397     ax2 = plt.subplot(122)
398     ax2.plot(i_list, t_list, 'b', label="Origin")
399     ax2.plot(i_list, t_listII, 'r', label="Tournament")
400     ax2.plot(i_list, t_listIII, 'g', label="Tournament + Block-reversal +
Origin")
401     ax2.set_ylabel("Cost Time")
402     plt.legend()
403     plt.show()
404
405     # 3.1 原程序的主函数 - 求解不同规模的TSP问题的算法性能
406     def ori_main():
407         time_start = time.time()
408         pop = [] # 生成初代种群pop
409         li = list(range(DNA_SIZE))
410         for i in range(POP_SIZE):
411             random.shuffle(li)
412             l = li.copy()
413             pop.append(l)
414         best_dis= []
415         # 进行选择，交叉，变异，并把每代的最优个体保存在best_dis中
416         for i in range(Iterations): # 迭代N代
417             pop = crossmuta(pop, CROSS_RATE)
418             fitness = getfitness(pop)
419             maxfitness = np.argmax(fitness)
420             best_dis.append(distance(pop[maxfitness]))
421             pop = select(pop, fitness) # 选择生成新的种群
422
423         time_end = time.time()
424         print_info(pop)
425         print('逐代的最小距离: ',best_dis)

```

```

426     print('Totally cost is', time_end - time_start, "s")
427     plt.figure()
428     plt.plot(range(Iterations),best_dis)
429
430 # 4.1 块逆转变异策略运行效果展示
431 def opt1_main():
432     time_start = time.time()
433     pop = []      # 生成初代种群pop
434     li = list(range(DNA_SIZE))
435     for i in range(POP_SIZE):
436         random.shuffle(li)
437         l = li.copy()
438         pop.append(l)
439     best_dis= []
440     # 进行选择, 交叉, 变异, 并把每代的最优个体保存在best_dis中
441     for i in range(Iterations): # 迭代N代
442         pop = crossmuta(pop, CROSS_RATE, muta=3)
443         fitness = getfitness(pop)
444         maxfitness = np.argmax(fitness)
445         best_dis.append(distance(pop[maxfitness]))
446         pop = select(pop, fitness) # 选择生成新的种群
447
448     time_end = time.time()
449     print_info(pop)
450     print('逐代的最小距离: ',best_dis)
451     print('Totally cost is', time_end - time_start, "s")
452     plt.figure()
453     plt.plot(range(Iterations),best_dis)
454
455 if __name__ == "__main__":
456
457     ori_main()      # 原程序的主函数
458     opt1_main()     # 块逆转变异策略运行效果展示
459     plt.show()
460     plt.close()
461
462     # opt1_test()    # 块逆转变异策略对比测试
463     # opt2_test()    # 锦标赛选择策略对比测试
464
465     # pop_size_test()      # POP_SIZE 种群规模参数测试
466     # cross_rate_test()    # CROSS_RATE 交叉率参数测试
467     # muta_rate_test()     # MUTA_RATE 变异率参数测试
468     # cross_muta_test()    # 交叉率和变异率双参数测试

```