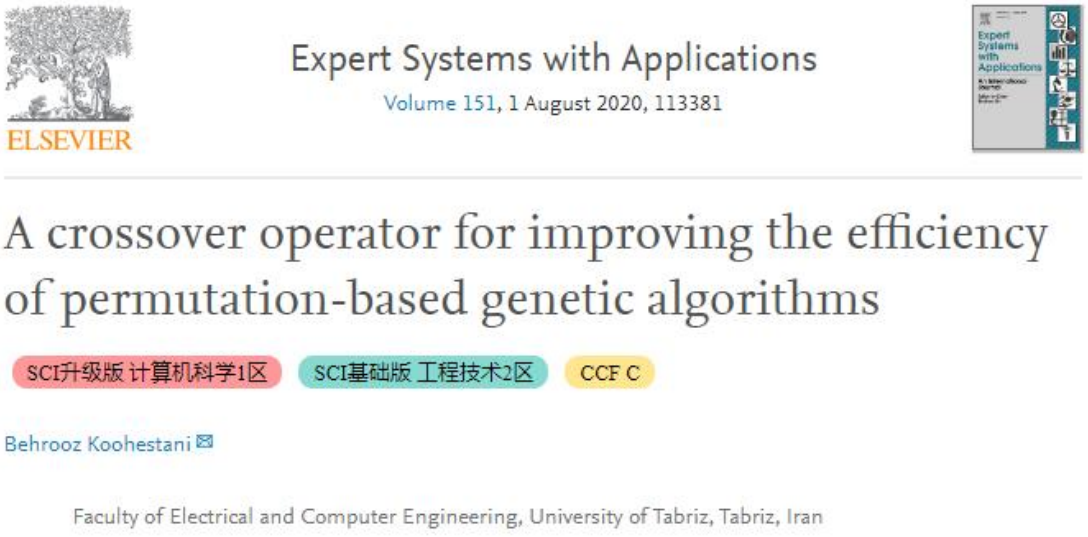
1 论文调研对象

选择精读的论文基本信息如下图所示：



论文标题：一种提高基于置换的遗传算法的效率的交叉算子
刊载期刊：Expert Systems with Applications，CCF-C 类、SCI 中科院 1 区期刊
关 键 词：遗传算法 染色体表示 交叉算子 组合优化问题 旅行商问题
发表时间：2020 年
引用次数：33 次

文献来源：Google 学术搜索
（该论文与实验代码压缩包同附在邮件附件中）

2 论文主要贡献

2.1 提出问题

遗传算法(GA)是高度并行的搜索算法，受达尔文进化论和自然选择理论的启发，它使用遗传操作（即选择，交叉和突变）。

用于组合优化问题的染色体表示方法通常可以分为五种主要类型，包括路径表示、邻接表示、二进制表示、矩阵表示和序数表示。基于这些表示，各位研究人员已经设计了许多交叉和变异算子来满足不同的需求。

对基于排列的组合优化问题而言，路径表示被认为是最重要和最广泛应用的染色体表示方法，其中解决方案被编码为一组整数的排列。同时，应用 GA 解决置换问题的几项关键研究都采用了路径表示，并为此表示定义了合适的交叉和变异算子。著名的一种是，Goldberg 和 Lingle (1985) 提出了部分映射交叉 (PMX)，它是最广泛使用的基于置换的交叉算子。在 PMX 中，第一个父项的子字符串映射到第二个父项的子字符串，并交换剩余信息。该算法的伪代码如下所示。

算法 1 PMX 的伪代码

Step 1 随机选择父染色体上的两个切点
Step 2 交换切割点之间的子串以产生原始后代

Step 3 确定关于所选子串的映射关系
Step 4 使用映射关系使原始后代合法化

遗传算法中的交叉操作，旨在解决基于排列的组合优化问题，与其他情况相比，计算成本更高。这主要是由于染色体中不允许重复数字，因此在每次子串交换后都需要后代合法化。在这些条件下，应用交叉算子 PMX，执行交叉操作所需的时间与染色体大小的**平方**成正比（具体证明详见论文第 4 节开头）。这可能会严重影响 GA 的效率，并大大限制了 GA 求解更大规模问题的能力，因此 PMX 有待改进。

2.2 改进方法

在本文中，提出了一种用于解决基于排列的组合优化问题的 GA。该 GA 的主要特点是它结合了 PMX 的改进版本，称为 IPMX。 IPMX 的设计目的是在不影响解决方案质量的前提下，提高基于排列的 GA 的效率。该算法的伪代码如下表所示。

算法 2 IPMX 的伪代码

Step 1 在亲本染色体 parent1 和 parent2 上随机选择两个切割点 cp1 和 cp2
Step 2 通过交换 cp1 和 cp2 之间的子串来产生原始后代 1 和 2
Step 3 定义关于所选子串的交换列表
Step 4 生成交换列表的有向图
Step 5 找到图中节点之间的所有不同路径
Step 6 对于包含两个以上节点的每条路径，在两个端点之间添加一条边，然后删除所有中间节点
Step 7 更新交换列表，考虑简化后的路径
Step 8 将更新后的交换列表应用到原始后代 1 以产生后代 1
Step 9 生成 F，一个与父染色体长度相同的列表，初始化为 0
Step 10 通过执行以下操作产生后代 2，其中 i = 1 , 2 , ..., n 表示父染色体的长度：
（**a**）F[offspring1[i]] = parent1[i].
（**b**）offspring2[i] = F[parent2[i]].

在论文第 4 节末尾处，证明了 IPMX 的时间复杂度为 O(n)。与 PMX 的 $O(n^2)$ 复杂度相比，IPMX 显著地提高了 GA 的效率。

2.3 实验结果

为了检验 IPMX 的能力，进行了两组实验。

首先设置 GA 的初始参数如下表。

Table 1
Parameters of the runs for experiment set 1.

Parameter	Value
Population size	500
Tournament size	3
Crossover rate	100%
Mutation rate	0%
Number of generations	50
Number of runs	30

在实验集 1 中，测试了 IPMX 对基于置换的 GA 速度的影响，将本文提出的包含 IPMX 的 GA 与具有标准 PMX（Goldberg & Lingle，1985）、MX（Kumar 等人，2013 年）和 BRKX（Andrade 等人，2019 年）的相同 GA 进行比较。其中 IPMX 在 36 个基准问题上表现出卓越的性能，在所有实例的 CPU 时间总和上，比标准 PMX 快约 9 倍，同时比 MX 和 BRKX 快。这也与 IPMX 和 PMX 计算所得的时间复杂度一致。新的跨界车 IPMX 似乎将其成功很大程度上归功于能够显着减少消除重复元素和产生第二个后代所需的操作数量。 IPMX 在 15 个基准实例上也产生了最低的目标函数值，这比 PMX、MX 和 BRKX 获得的结果要好得多。具体实验结果如下表所示：

Table 2
Computing times of the PMX, BRKX, MX and IPMX for a set of 36 TSP instances, shown with the following format: Best (Mean, Standard Deviation) and their associated best objective function values obtained, shown in italics.

TSP instance	Description	PMX	BRKX	MX	IPMX
eil51	51-city problem	0.1411 (0.1771, 0.0184) *600*	0.3144 (0.3312, 0.0100) *613*	0.1321 (0.1542, 0.0141) *631*	**0.0797** (0.0862, 0.0077) *593*
st70	70-city problem	0.2544 (0.3128, 0.0305) *1453*	0.5853 (0.6194, 0.0175) *1432*	0.2436 (0.2796, 0.0178) *1383*	**0.1139** (0.1226, 0.0098) ***1369***
pr76	76-city problem	0.3268 (0.3628, 0.0191) *233147*	0.6608 (0.7146, 0.0293) *227716*	0.2873 (0.3331, 0.0241) *237314*	**0.1265** (0.1309, 0.0032) ***222364***
lin105	105-city problem	0.5463 (0.6860, 0.0607) ***49488***	1.2467 (1.3484, 0.0493) *51253*	0.5678 (0.6444, 0.0454) *50727*	**0.1945** (0.2033, 0.0051) *51445*
d198	Drilling problem	2.2850 (2.6419, 0.1788) *85073*	4.4844 (4.7541, 0.1274) *81744*	2.0897 (2.3418, 0.1810) ***81269***	**0.4723** (0.5022, 0.0152) *81961*
pr226	226-city problem	2.9222 (3.3151, 0.2004) *909430*	6.0260 (6.1863, 0.0999) *919885*	2.6860 (3.1657, 0.2182) *919953*	**0.5769** (0.6261, 0.0232) ***900831***
pr264	264-city problem	3.8152 (4.6160, 0.3749) *565232*	8.3725 (8.6692, 0.1431) *562542*	3.8977 (4.3731, 0.2108) ***552146***	**0.7693** (0.8096, 0.0290) *563014*
pr299	299-city problem	4.9015 (5.7650, 0.4283) ***434147***	10.8229 (11.2586, 0.4082) *446306*	5.0920 (5.9693, 0.4941) *438905*	**0.9257** (0.9951, 0.0346) *436505*
lin318	318-city problem	5.7605 (6.7991, 0.4942) ***370552***	12.4209 (12.5161, 0.0784) *380286*	5.7462 (6.5982, 0.4933) *371027*	**1.0505** (1.1131, 0.0436) *372610*
pr439	439-city problem	10.6604 (13.1100, 1.0372) *1266361*	23.4947 (24.2322, 0.4230) ***1260566***	11.5645 (13.3609, 0.9854) *1273484*	**1.7406** (1.9024, 0.0728) *1264850*
d493	Drilling problem	14.7093 (18.6065, 2.1312) ***304539***	30.5771 (30.9398, 0.2718) *304956*	14.9678 (16.9331, 1.2598) *307721*	**2.1850** (2.3448, 0.0831) *305414*
rat575	Rattled grid problem	21.0062 (25.0886, 2.1291) *79061*	41.7143 (42.4207, 0.5770) *79420*	20.8135 (24.3785, 1.8185) *80459*	**2.8137** (3.0763, 0.1129) ***76815***
d657	Drilling problem	27.2169 (30.2025, 2.2969) ***622511***	55.0861 (55.5647, 0.6634) *622571*	28.8551 (33.2867, 2.7608) *639037*	**3.5992** (3.9104, 0.1471) *635306*
u724	Drilling problem	32.3592 (38.7305, 2.4248) *632404*	67.7509 (68.0499, 0.4041) *639199*	33.1371 (39.7687, 4.5737) *649106*	**4.4974** (4.8571, 0.2721) ***624561***
rat783	Rattled grid problem	38.8181 (45.8914, 3.3537) *131816*	77.9358 (79.0037, 0.8691) *130895*	40.9188 (47.9978, 4.4779) *133665*	**5.0821** (5.7679, 0.7627) ***130817***
pr1002	1002-city problem	62.9191 (75.5716, 5.2763) *5024038*	130.0221 (130.7902, 0.5706) *4970672*	68.4253 (80.8361, 8.4179) *5086670*	**7.9063** (8.6990, 0.3113) ***4964965***
d1291	Drilling problem	97.7342 (125.1184, 10.3539) *1404879*	220.1945 (220.4408, 0.24630) ***1401986***	125.0393 (139.9825, 11.8363) *1428906*	**12.6199** (13.8249, 0.6251) *1404833*
d1655	Drilling problem	190.7995 (215.8063, 14.2441) ***1785635***	365.0797 (365.8307, 0.75103) *1808874*	216.9726 (247.8239, 20.0794) *1824611*	**20.0846** (22.4201, 0.9105) *1807110*
u1817	Drilling problem	225.1480 (262.2252, 18.9452) *1739232*	439.6913 (443.5588, 3.8675) ***1718856***	261.9256 (297.5908, 23.7747) *1753633*	**23.3181** (26.9595, 1.2496) *1760303*
pcb3038	Drilling problem	655.5607 (769.6929, 58.5211) *4724293*	1244.5353 (1257.4957, 8.6771) *4723976*	770.7431 (920.7725, 73.4113) *4762984*	**65.5933** (73.8239, 3.1729) ***4666769***

在实验集 2 中，评估了 IPMX 对基于置换的 GA 产生的解决方案质量的影响。获得的结果表明，在解决方案质量方面，IPMX 在选择测试的 5 个 TSP 实例上仅比标准 PMX 得出的解高 2.49-9.82%。在某些情况下，IPMX 还产生了比 MPPMX（即高效的交叉算子）和大多数情况下 BRKX 更好的解决方案。此外，从结果中出 IPMX 比 MPPMX 快得多。具体实验结果如下表所示：

Table 4
Mean, standard deviation (in parenthesis), and mutation rate (marked with an asterisk) related to the best tour length over 30 runs of the PMX, MPPMX, BRKX and IPMX.

	eil51	st70	pr76	lin105	d198
PMX	533.52 (31.56)	1015.90 (68.79)	162172.2 (9898.4)	28409.2 (2527.5)	37275.8 (4472.8)
	0.04*	0.05*	0.03*	0.02*	0.005*
MPPMX (n = 2)	541.71 (30.74)	1039.51 (72.28)	163586.6 (10820.0)	28759.3 (2652.8)	37838.7 (4518.4)
	0.06*	0.03*	0.03*	0.02*	0.006*
MPPMX (n = 3)	527.18 (28.93)	1002.48 (73.47)	161547.9 (11214.7)	27049.8 (2545.3)	36049.1 (2803.7)
	0.05*	0.05*	0.03*	0.01*	0.01*
MPPMX (n = 4)	522.28 (28.93)	973.89 (67.54)	157436.3 (11099.9)	25768.0 (2267.6)	34812.9 (3669.4)
	0.06*	0.04*	0.04*	0.02*	0.01*
MPPMX (n = 5)	**513.86** (28.31)	962.79 (66.38)	154115.7 (10660.5)	25958.1 (1776.2)	34220.0 (3371.5)
	0.04*	0.04*	0.04*	0.005*	0.01*
MPPMX (n = 6)	520.19 (32.03)	959.57 (63.46)	156310.9 (11335.8)	25111.6 (2636.9)	34163.2 (3004.0)
	0.05*	0.05*	0.04*	0.03*	0.01*
MPPMX (n = 7)	517.77 (26.58)	956.28 (81.60)	155252.7 (10066.9)	25045.3 (2310.7)	33398.7 (3253.1)
	0.06*	0.05*	0.04*	0.02*	0.005*
MPPMX (n = 8)	516.41 (28.27)	951.51 (66.56)	155301.8 (10731.1)	25176.9 (2045.2)	33371.0 (3240.0)
	0.06*	0.05*	0.04*	0.02*	0.01*
MPPMX (n = 9)	519.09 (35.58)	**939.41** (63.71)	**152737.9** (9697.3)	24566.2 (1561.9)	**32309.6** (3459.1)
	0.05*	0.03*	0.04*	0.02*	0.004*
MPPMX (n = 10)	514.01 (30.48)	947.07 (59.42)	153161.7 (11058.6)	**24444.9** (2013.7)	33365.0 (3912.9)
	0.06*	0.04*	0.04*	0.005*	0.004*
BRKX	532.6 (23.52)	989.2 (49.85)	154850.1 (9022.38)	24596.7 (1312.22)	37112.0 (4834.002)
	0.04*	0.05*	0.03*	0.02*	0.005*
IPMX	518.13 (22.36)	987.36 (71.35)	158228.9 (10651.1)	25867.86 (2200.38)	36110.33 (3546.13)
	0.04*	0.05*	0.03*	0.02*	0.005*

PMX: Standard PMX operator. MPPMX: Multi-parent PMX operator. BRKX: Biased random-key crossover. IPMX: Improved PMX operator (proposed crossover operator). Numbers in bold face are the best results.

3 精读感想

本文提出的新遗传算法交叉算子 IPMX，除了具有更低的时间复杂度外，还能够产生同质量、甚至更短的路径解决方案。这份工作是一次有益且创新的尝试。

4 编程实现遗传算法

本节编程实现遗传算法以解决 TSP 问题，并尝试了多种选择算子、交叉算子和变异算子。具体代码可以查看所附压缩包，程序均可运行出解并可视化结果。

4.1 准备数据集

从 http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/ 中选择了 rd100、lin105、gr120、ch130 这 4 个数据集（散点图）作为 TSP 问题的输入数据，详见 data 文件夹中的.tsp 文件。同时，为了检验遗传算法的解题效果，也从该网站中下载了这 4 个数据集对应的最优路径，详见 data 文件夹中的.tour 文件。

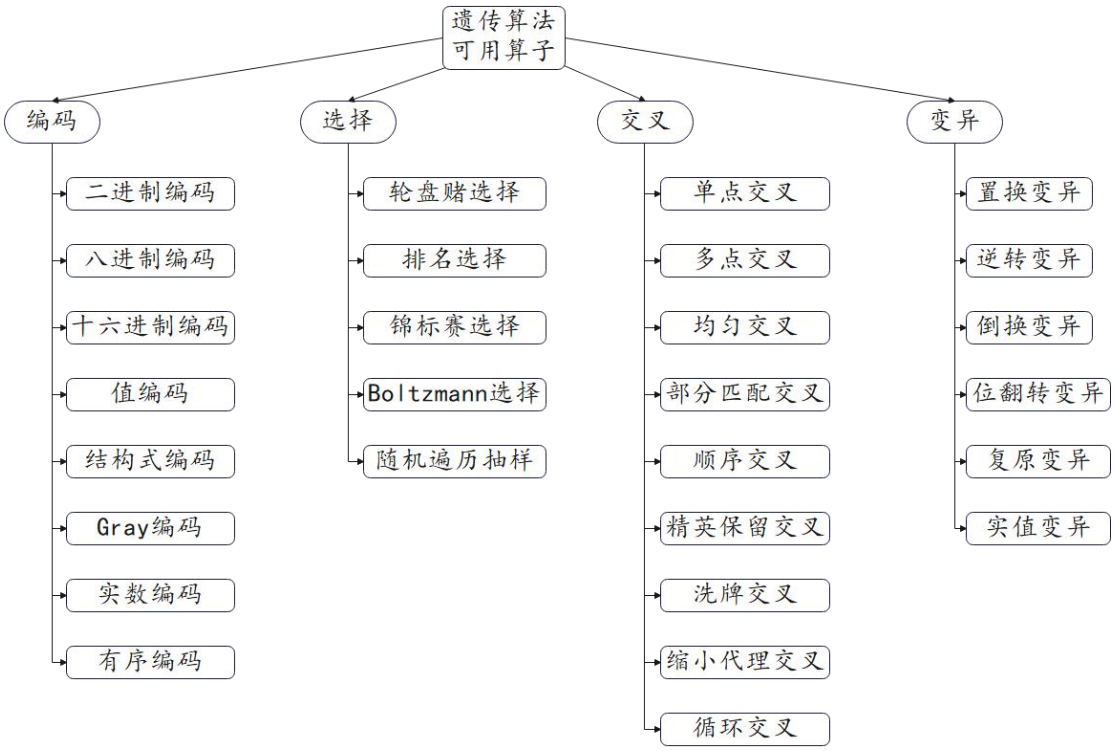
4.2 构思核心步骤

遗传算法的核心步骤如下：

遗传算法求解过程
Step 1 设定初始状态，参数以及编码等，导入数据
Step 2 评估每条染色体所对应个体的适应度
Step 3 遵照适应度越高，选择概率越大的原则，从种群中选择两个个体作为父方和母方
Step 4 抽取父母双方的染色体，进行交叉，产生子代

Step 5 对子代的染色体进行变异
Step 6 重复 3、4、5 三个步骤，直到进化到符合最优或达到最大遗传代数
Step 7 结果解码输出，展示最短路径可视化图和进化过程示意图

在经过国内外多年的研究后，GA 已经发展出了多种编码、交叉、变异和选择的方案。在翻阅了多篇教材和文献后，笔者总结遗传算法的最新进展如下图所示。



根据此图，笔者得出了通过实现并对比不同选择/交叉/变异算子的效果，再选用效果最优的一种，来优化遗传算法的思路。时间所限，笔者只选择了数种算子实现。

4.3 关键代码展示

4.3.1 初始种群产生方式

设种群内个体规模为 n，则随机产生 50 个$^{0\sim n-1}$的排列作为初始种群。

```
1.  # 产生初始种群
2.      def createOriginalGroup(self):
3.          origin=[i for i in range(self.dim)]
4.          for i in range(self.og):
5.              person=np.random.permutation(origin)
6.              if person.tostring():
7.                  self.group.append(np.random.permutation(origin))  # 产生随机个体
```

4.3.2 适应度函数

设该种群的排列所对应的路径长度为 d(x)，则适应度函数为

$$f(x)=\frac{1000}{d(x)}$$

```
# 根据距离矩阵计算距离值
def calDistance(self,status):
    dis=0
    for i in range(self.dim-1):
        dis+=self.distmat[status[i],status[i+1]]
    dis+=self.distmat[status[0],status[-1]]
    return dis

# 个体适应值函数
def fitness(self,status):
    return 1000/self.calDistance(status)
```

4.3.3 选择算子

- 轮盘赌策略：个体适应度越高，被选择的概率越大

```
def nextStatus_swap(self):
    temp=deepcopy(self.path)
    left = np.random.randint(0, self.dim - 1)
    right = np.random.randint(left + 1, self.dim)
    # 交换位置
    temp[left],temp[right]=temp[right],temp[left]
    return temp
```

- 精英保留策略：依某种策略生成新种群后，用当代最优个体随机替换新种群的某个体，或者直接替换最差个体

```
def select_optimal(self):
    fits=self.groupFitness()
    temp_group=np.array(self.group)
    new_group_index=np.random.choice(self.og,size=self.mc,replace=True,p=fits/fits.sum())
    # 先进行轮盘赌策略筛选个体
    new_group=temp_group[new_group_index]
    new_fits=fits[new_group_index]
    max_fit=fits.argmax()
    min_fit=new_fits.argmin()
    new_group[min_fit]=temp_group[max_fit]  # 最优个体替换
    self.group=new_group.tolist()
    self.og=self.mc
```

- 截断选择策略：保留所有适应值在平均水平之上的个体

```
def select_truncation(self):
    fits=self.groupFitness()
    temp_index=[i for i,j in enumerate(fits>np.average(fits)) if j]
    temp_group=np.array(self.group)
    temp_fits=fits[temp_index]
    temp_group=temp_group[temp_index]  # temp_group 中是全部大于平均值的个体
      new_group_index=np.random.choice(len(temp_group),size=self.mc,replace=True,p=temp_fits/temp_fits.sum())
    new_group=temp_group[new_group_index]
    new_fits=temp_fits[new_group_index]
    max_fit=temp_fits.argmax()  # 同时保留精英
    min_fit=new_fits.argmin()
    new_group[min_fit]=temp_group[max_fit]
    self.group=new_group.tolist()
    self.og=self.mc
```

4.3.4 交叉算子

- OX：单点交叉

```
def crossOver_OX(self):
    temp=self.og
    for i in range(temp // 2):
        if np.random.random()>self.pc:  # 判断是否交叉
            continue
        parent1 = self.group[i]
        parent2 = self.group[i + temp // 2]
        oops = np.random.randint(1, self.dim)
        child1, child2 = list(parent1[:oops].copy()), list(parent2[:oops].copy())
        for j in range(len(parent1)):  # 解决冲突
            if parent1[j] not in child2:
                child2.append(parent1[j])
            if parent2[j] not in child1:
                child1.append(parent2[j])
        self.group.append(child1.copy())
        self.group.append(child2.copy())
        self.og+=2
```

- PMX：两点交叉

```
def crossOver_PMX(self):
    temp=self.og
    for n in range(temp//2):
        if np.random.random()>self.pc:  # 判断是否交叉
            continue
        parent1 = np.array(self.group[n])
        parent2 = np.array(self.group[n + temp // 2])
```

```
        child1,child2=copy.deepcopy(parent1),copy.deepcopy(parent2)
        oops0=np.random.randint(0,self.dim-1)
        oops1=np.random.randint(oops0+1,self.dim)
        cross_area=range(oops0,oops1)  # 交叉区索引
        keep_area=np.delete(range(self.dim),cross_area)  # 非交叉区索引
        keep1=parent1[keep_area]
        keep2=parent2[keep_area]
        cross1=parent1[cross_area]
        cross2=parent2[cross_area]
        child1[cross_area],child2[cross_area]=cross2,cross1  # 先对交叉区进行交换
        mapping=[[],[]]  # 映射表
        # 生成映射表
        for i, j in zip(cross1, cross2):
            if j in cross1 and i not in cross2:
                index = np.argwhere(cross1 == j)[0, 0]
                value = cross2[index]
                while value in cross1:
                    index = np.argwhere(cross1 == value)[0, 0]
                    value = cross2[index]
                mapping[0].append(i)
                mapping[1].append(value)
            elif j not in cross1 and i not in cross2:
                mapping[0].append(i)
                mapping[1].append(j)
        # 根据映射表解决冲突
        for i, j in zip(mapping[0], mapping[1]):
            if i in keep1:
                keep1[np.argwhere(keep1 == i)[0, 0]] = j
            elif i in keep2:
                keep2[np.argwhere(keep2 == i)[0, 0]] = j
            if j in keep1:
                keep1[np.argwhere(keep1 == j)[0, 0]] = i
            elif j in keep2:
                keep2[np.argwhere(keep2 == j)[0, 0]] = i
        child1[keep_area], child2[keep_area] = keep1, keep2
        self.group.append(child1)
        self.group.append(child2)
        self.og+=2
```

- IPMX：论文中提出的 PMX 改进版

```
    def crossOver_IPMX(self):
        temp=self.og
        for n in range(temp//2):
            if np.random.random()>self.pc:  # 判断是否交叉
                continue
```

```
            parent1 = np.array(self.group[n])
            parent2 = np.array(self.group[n + temp // 2])
            child1,child2=copy.deepcopy(parent1),copy.deepcopy(parent2)
            oops0=np.random.randint(0,self.dim-1)
            oops1=np.random.randint(oops0+1,self.dim)
            cross_area=range(oops0,oops1)  # 交叉区索引
            keep_area=np.delete(range(self.dim),cross_area)  # 非交叉区索引
            keep1=parent1[keep_area]
            keep2=parent2[keep_area]
            cross1=parent1[cross_area]
            cross2=parent2[cross_area]
            child1[cross_area],child2[cross_area]=cross2,cross1  # 先对交叉区进行交换
            # 生成映射表
            L1=[0 for i in range(self.dim)]
            L2=[0 for i in range(self.dim)]
            dict = {}
            newDict = {}
            for i,j in zip(cross2,cross1):
                dict[i] = j
                L2[i] = 1
                L1[j] = 1
            L1 += L2
            for i in range(self.dim):
                if L1[i] <= 1 and i in dict:
                    pre,cur = i,dict[i]
                    while cur in dict and cur != dict[cur]:
                        cur = dict[cur]
                        if cur == pre: break
                    newDict[pre] = cur
            #根据映射表解决原始后代 1 的冲突
            for i in keep1:
                if i in newDict:
                    keep1[np.argwhere(keep1 == i)[0, 0]] = newDict[i]
            child1[keep_area], child2[keep_area] = keep1, keep2
            #处理原始后代 2 的冲突
            F = {}
            for i in range(self.dim):
                F[child1[i]] = parent1[i]
            for i in range(self.dim):
                #if parent2[i] in F:
                    child2[i] = F[parent2[i]]
            self.group.append(child1)
            self.group.append(child2)
            self.og+=2
```

4.3.5 变异算子

- 简单交换两个城市

```
def mutate_swap(self):
    for i in range(self.og):
        if np.random.random()<self.pm:
            random_site0 = np.random.randint(0, self.dim-1)
            random_site1 = np.random.randint(random_site0,self.dim)
            # 交换两个城市
             self.group[i][random_site0], self.group[i][random_site1] = self.group[i][random_site1], self.group[i][random_site0]
```

- 交换城市并倒置

```
def mutate_inversion(self):
    for i in range(self.og):
        if np.random.random()<self.pm:
            random_site0 = np.random.randint(0, self.dim-1)
            random_site1 = np.random.randint(random_site0,self.dim)
            # 交换两个城市并将之间的城市倒置
            while random_site0 < random_site1:
                 self.group[i][random_site0],self.group[i][random_site1]=self.group[i][random_site1],self.group[i][random_site0]
                random_site0+=1
                random_site1-=1
```

4.4 对遗传算法的优化 1.0

在优化方面主要是通过反复测试训练集来选出最好的算子和相应的参数，下面给出测试结果。

各参数初始值为：测试集 ch130，进化数 10000，交叉概率 0.9，交叉算子:OX，选择算子“精英保留策略”，变异概率 0.03，变异算子“交换并倒置城市”。在选择算子/参数时应用控制变量法，仅改变自变量，其余算子/参数保持初始值不变。

变异概率优化：

变异概率	0.01	0.02	0.025	0.03	0.035	0.04	0.05
最短路径	8309.7	7214.9	7047.3	**6737.8**	7366.5	7169.4	6756.5
最优解	6110.9						
运行时间	50.7s	62.5s	50.2s	49.0s	50.8s	49.0s	49.7s

通过多方面的比较加上算法具有一定随机性的考量，这里选择变异概率 0.03。

交叉概率优化：

概率	0.85	0.88	0.9	0.92	0.95	1
最短路径	7993.0	7237.2	**6737.8**	7120.6	7169.0	7148.3
运行时间	47.1s	49.2s	49.0s	50.5s	53.6s	55.0s

从表中数据可以看出，当交叉概率为 0.9 时，遗传算法的效果和效率都表现优秀，故采用 0.9。

交叉算子优化：

交叉算子	PMX	OX	IPMX
最短路径	6953.9	**6731.8**	6953.9
运行时间	89.8s	54.7s	57.8s

可以看到交叉算子 OX 的性能明显优于 PMX，故这里采用 OX 交叉算子进行下一步测试。

同时可以看出，IPMX 的运行时间与 PMX 相比有了大幅度的减小，这印证了论文结论。

选择算子优化：

选择算子	轮盘赌策略	精英保留策略
最短路径	41190.0	**6590.5**
运行时间	99.0s	98.1s

由于截断选择需要大量的新个体产生，与此时的测试环境不符，故不对截断选择进行测试。

从表中数据可以看出轮盘赌策略效果很差，故采用精英保留策略。

变异算子优化：

变异算子	简单交换城市	交换并倒置城市
最短路径	11296.7	**6591.0**
运行时间	98.8s	100.0s

从表中数据可以看出交换并倒置城市的策略效果很好，故采用后一策略。

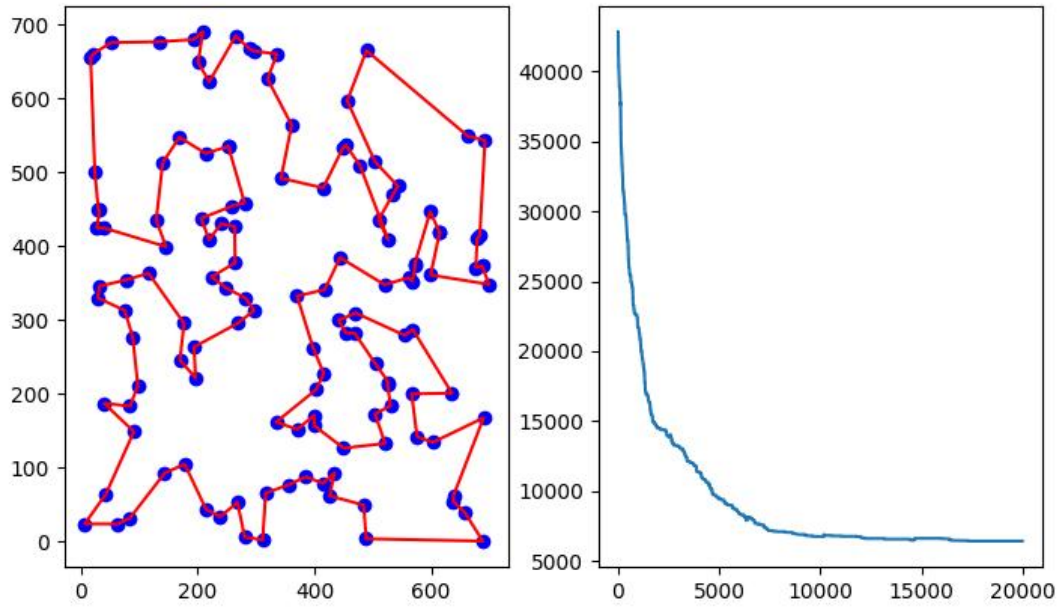
4.5 测试结果

为得到尽可能更优的结果，测试时将进化代数设为 20000,并多次调整随机种子、运行程序，取运行结果中的最小值为最短路径。测试所得的图表如下所示：

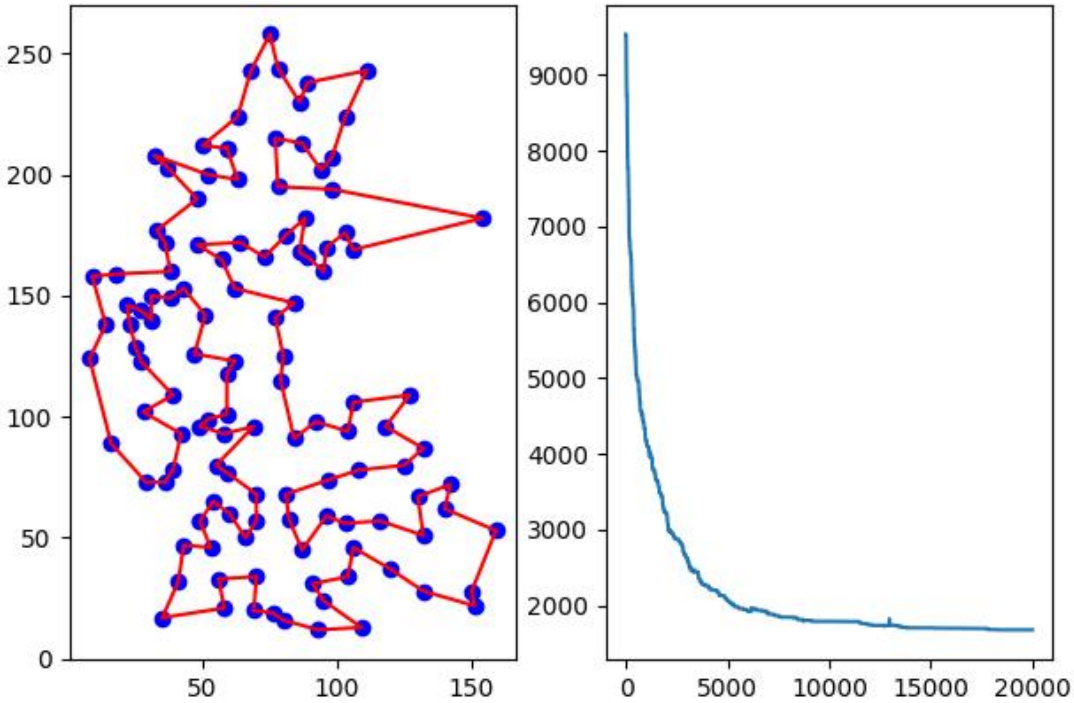
测试集	运行时间	最短路径	官方最优解	与最优解误差
ch130	101.1s	6411.1	6110.9	4.91%
gr120	92.5s	1678.8	1666.5	0.74%
lin105	81.6s	14893.0	14383.0	3.55%

rd100	70.9s	8260.0	7910.4	4.42%

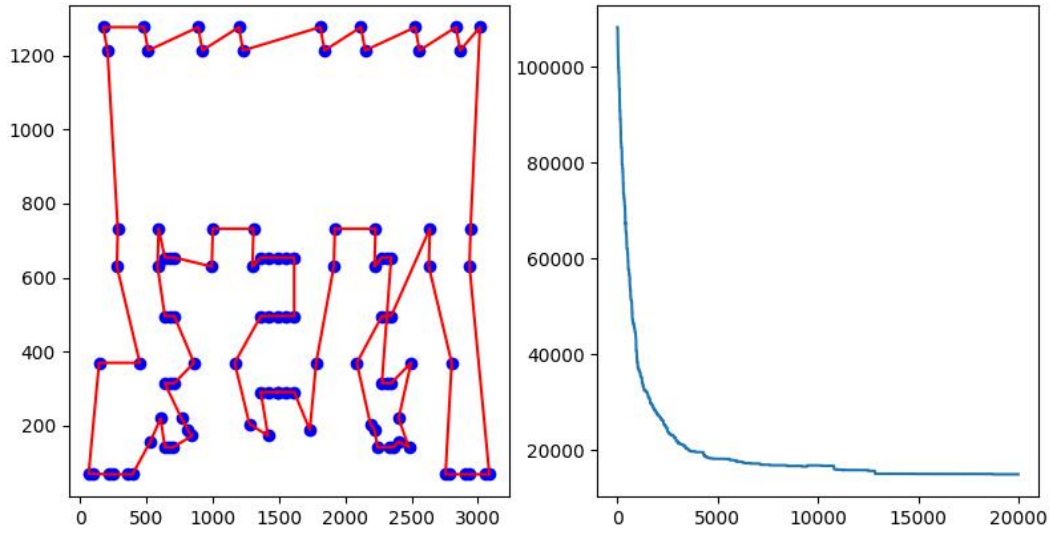
测试集 ch130 的最短路径图及遗传进化图：



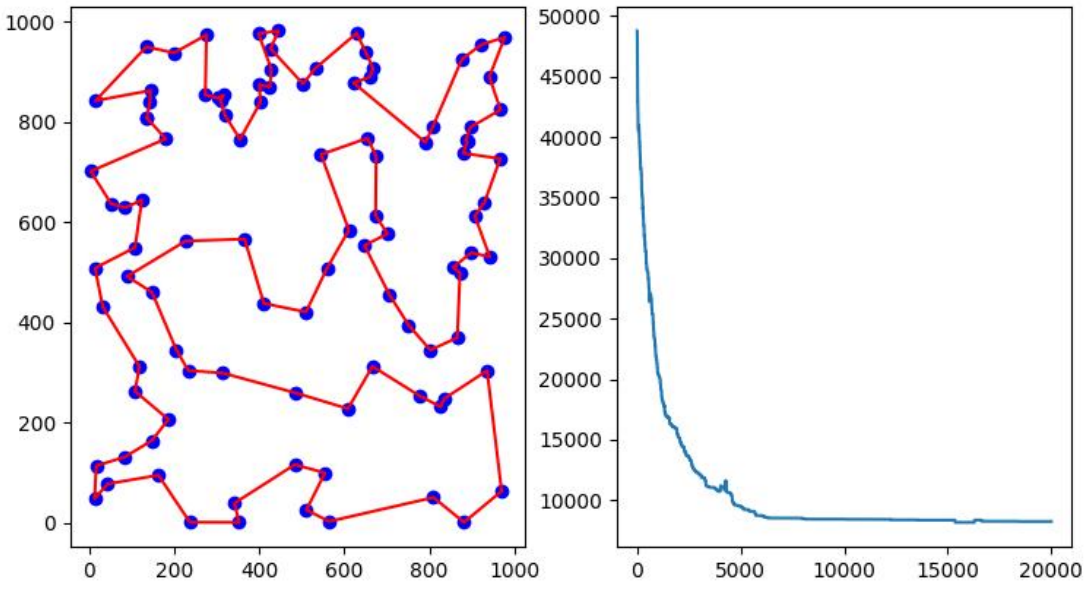
测试集 gr120 的最短路径图及遗传进化图：



测试集 ln105 的最短路径图及遗传进化图：



测试集 rd100 的最短路径图及遗传进化图：



通过测试结果可以发现，自主选择算子优化后的遗传算法表现优良，保证了最短路径长在最优解的 5%以内。

4.6 PMX 与 IPMX 对比测试

为验证论文结论，证明 IPMX 比 PMX 更优。笔者按照与上一节相同的条件，做了重复对比测试。测试结果如下所示：

测试集	PMX 时间	IPMX 时间	PMX 最短路径	IPMX 最短路径
ch130	174.2s	**112.0s**	**6584.7**	6629.8
gr120	168.8s	**107.1s**	1742.3	**1738.4**
lin105	149.7s	**99.2s**	15753.7	**15395.7**

rd100	133.1s	89.6s	8691.2	8575.1

由测试结果对比可得，论文结论正确，对论文的复现取得圆满成功。

唯一可惜的是，在实验测试中，使用交叉算子 PMX 和 IPMX 的 GA 所得最短路径，均比使用交叉算子 OX 的 GA 所得最短路径长，求解所用时间也更长。从直觉上也可以理解，因为单点交叉实际上比双点交叉（段交叉）提供了更多可能的排列，使得 GA 更有可能得到更优解。

4.7 对遗传算法的优化 2.0

通过学习翻转课堂上同学的想法，我了解到，要优化遗传算法，还可以从优化初始种群入手。

传统的随机生成法虽然足够简单，但得到的初始种群不够优秀，收敛速度慢。因此考虑将凸包法和随机插入法相结合，可以得到优秀的初始种群，具体算法如下：

初始种群优化算法求解过程

Step 1 根据输入数据中每个点的坐标，采用 Graham's Scan 法生成二维凸包，将凸包中的点按照顺序加入到初始路径中

Step 2 随机选择还没有在路径中的点，遍历枚举插入的位置。定义将点 x 插入到点 a 和点 b 之间的路径增量为 dis[a][x]+dis[x][b]-dis[a][b]，选择距离增量最小的位置插入

Step 3 重复 Step 2 直到所有点都插入到路径中，就得到了一个初始染色体

Step 4 重复以上过程，直到生成既定数目的染色体构成初始种群

具体实现如以下函数 newCreate()所示。

```
# 计算两个向量之间的叉积。返回三点之间的关系：
    def ccw(self,a, b, c):
        return ((b[1] - a[1]) * (c[0] - b[0]))   -    ((c[1] - b[1]) * (b[0] - a[0]) )

# 分别求出后面 n-1 个点与出发点的斜率，借助 sorted，按斜率完成从小到大排序
    def compute(self,next):
        start = self.points[0]  # 第一个点
        if start[0] == next[0]:
            return 99999
        slope = (start[1] - next[1]) / (start[0] - next[0])
        return slope

    def Graham_Scan(self,points):
    # # 找到最左边且最下面的点作为出发点，和第一位互换
        Min=999999
        for i in range(len(points)):
        # 寻找最左边的点
            if points[i][0]<Min:
                Min = points[i][0]
```

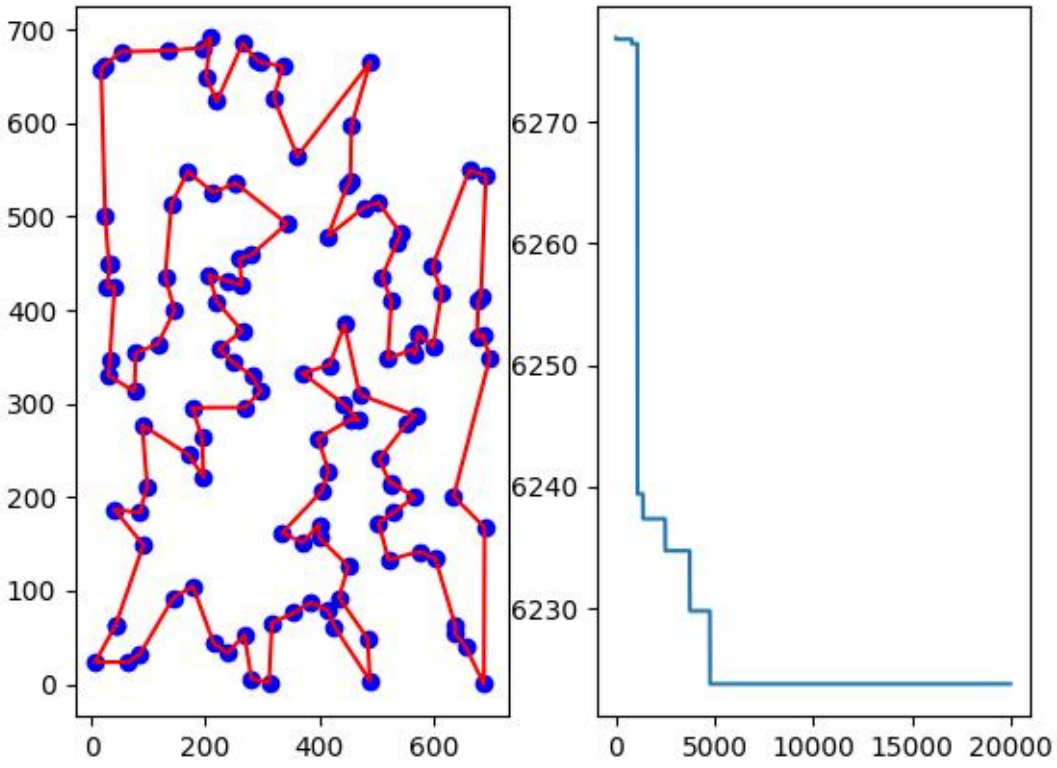
```
                index = i
        # 如果同在最左边，可取 y 值更小的
            elif points[i][0]==Min:
                if points[i][1]<=points[index][1]:
                    Min = points[i][0]
                    index = i
        # 和第一位互换位置
        temp = points[0]
        points[0] = points[index]
        points[index] = temp
        # 排序：从第二个元素开始，按与第一个元素的斜率排序
        points = points[:1] + sorted(points[1:], key=self.compute)   # 前半部分是出发点;后半部分是经过按斜率排序之后的 n-1 个坐标点
        #注意： “+”是拼接的含义，不是数值相加
        # 用列表模拟一个栈。（最先加入的是前两个点，前两次 while 必定不成立，从而将点加进去）
        convex_hull = []
        for p in points:
            while len(convex_hull) > 1 and self.ccw(convex_hull[-2], convex_hull[-1], p) >= 0:
                convex_hull.pop()
            convex_hull.append(p)
        person=[x[2] for x in convex_hull]
        return person

    #优化后的初始种群产生方法
    def newCreate(self):
        stp=self.Graham_Scan(self.points)
        origin=[i for i in range(self.dim)]
        for x in stp:
                origin.remove(x)
        sto=origin
        for i in range(self.og):
            person=stp.copy()#不加.copy()是赋了指针！！！
            #print("st",person)
            origin=sto.copy()
            toto,totp=len(sto),len(stp)
            #print(toto,totp)
            #print(origin)
            while toto>0:
                x=origin[random.randint(0,toto-1)]
                mn,posmn=999999,0
                for j in range(totp):
                    if j<totp-1:Dis=self.distmat[person[j],x]+self.distmat[x,person[j+1]]-self.distmat[person[j],person[j+1]]
```

```
                        else:Dis=self.distmat[person[j],x]+self.distmat[x,person[0]]-self.distmat[person[j],person[0]]
                    if Dis<mn:
                        mn=Dis
                        posmn=j
                person.insert(posmn+1,x)
                totp+=1
                origin.remove(x)
                toto-=1
            self.group.append(person)  # 产生随机个体
```

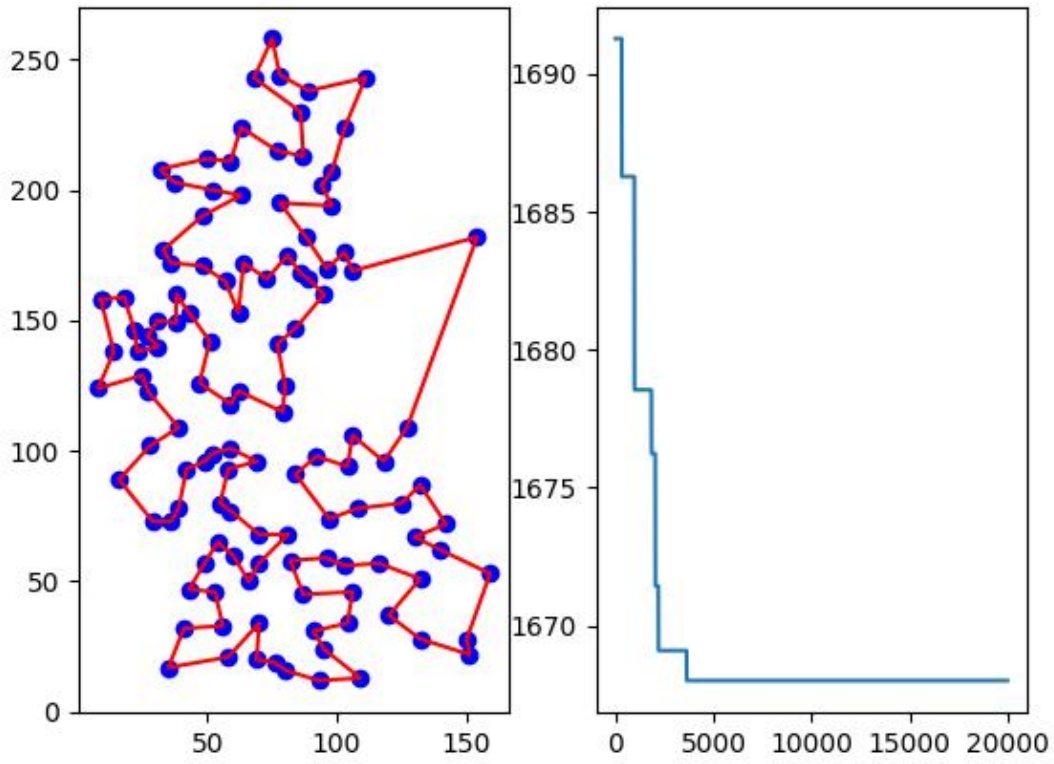
用这种方法生成的初始种群，在少量时间的额外开销下，能够得到非常接近最优解的值，甚至比经过前面所有优化后求得的解更优。同时，求得的最短路径与官方最优解的误差也大大缩小了，由原来的 5%缩小到现在的 2%;**甚至在 gr120 数据集中能够求出较官方最优解更优的解**。测试所得的图表如下所示：

测试集	优化 2.0 前	优化 2.0 后	官方最优解	与最优解误差
ch130	6411.1	**6223.7**	6110.9	1.84%
gr120	1678.8	**1650.7**	1666.5	**-0.95%**
lin105	14893.0	**14428.7**	14383.0	0.31%
rd100	8260.0	**7957.4**	7910.4	0.59%

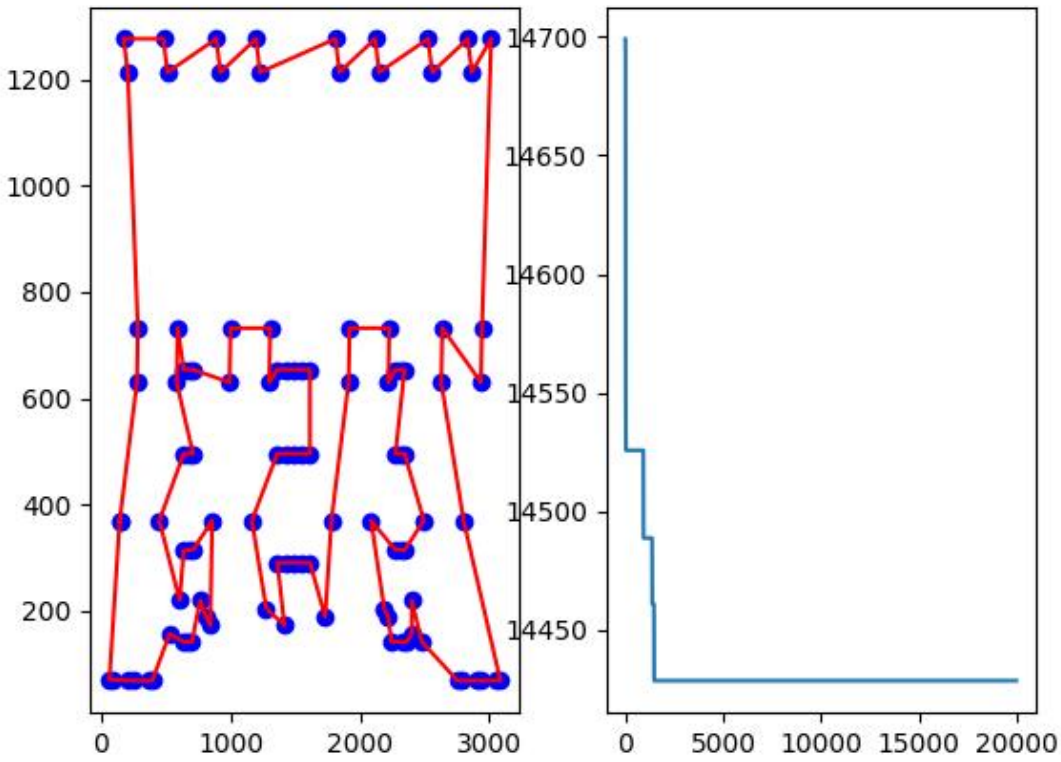
测试集 ch130 的最短路径图及遗传进化图：



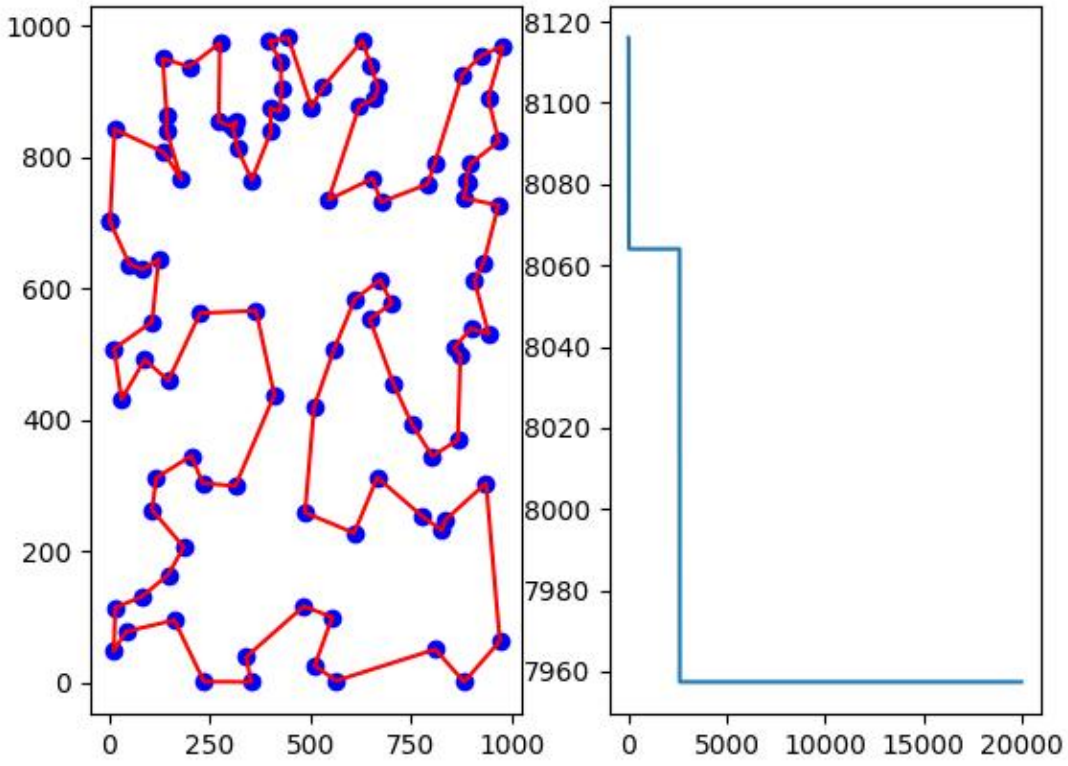
测试集 gr120 的最短路径图及遗传进化图：



测试集 ln105 的最短路径图及遗传进化图：

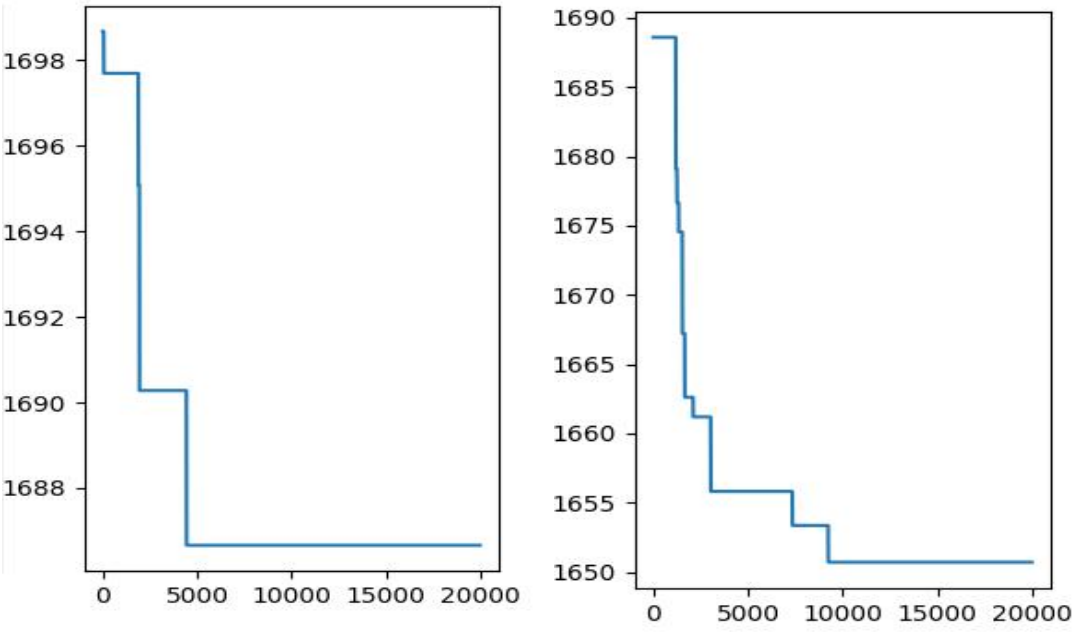


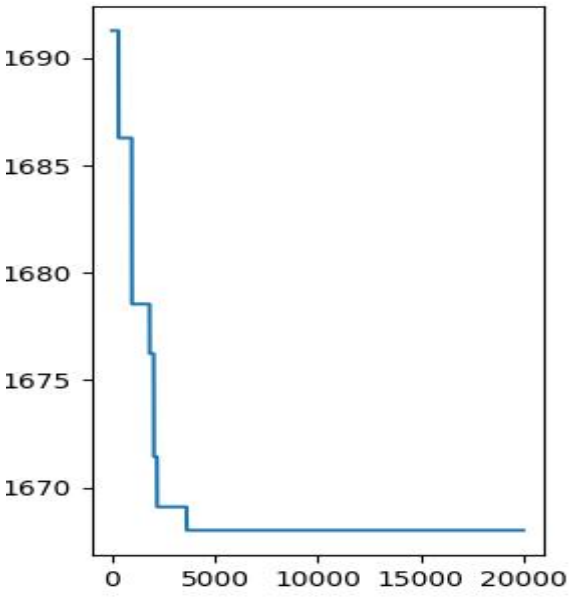
测试集 rd100 的最短路径图及遗传进化图：



同时，与前面的遗传进化图对比可以发现，最终解与初始种群的差距大大减小，并且收敛所需代数也由近 20000 代缩至 5000 代以内。

在测试过程中，也发现了一个普遍的现象：初始种群越优的，最终解的质量往往越好。由此可见初始种群优化在遗传算法优化中的重要性。以下三图是测试集 gr120 的三次运行结果，可佐证这一结论。





5 总结与感想

通过对教材、博客和中外论文的研读，我充分了解了遗传算法及其原理，以及其选择/交叉/变异算子的多样性。这一科研前沿的算法，初学时以为很难，但是编程实现后感觉也不过如此，多实现几种算子后也仅有数百行代码。但是，遗传算法的超参数和算子种类较多，编写完多种算子较为困难；不断跑代码选出大致的最合适参数与算子耗时较多，也难以判断选出的这些参数是局部最优还是全局最优。在这方面模拟退火算法就具有相对较大的优势。

同时，在改进遗传算法的过程中，我深刻体会到学术交流的重要性。原本我并没有想到初始种群优化的较好方法，直到翻转课堂上周佬提到二维凸包才意识到，由此实现遗传算法优化 2.0,达到了现在初始种群比原先优化得到的最优解更优的震撼效果。同学们对遗传算法的各种魔改也让我脑洞打开，比如将遗传算法与模拟退火算法结合和改进适应值函数等。大家的灵感启发着我对自己的工作精益求精，以求至善。

无论如何，模拟退火算法、遗传算法、人工免疫算法和蚁群算法，都只是平等的智能算法。每个算法都有一片最适合于自己的一片一亩三分地，不可相互替代。遗传算法里的各种编码方式、选择、交叉和变异算子，以及遗传算法的各种变体，也是同理。科研领域的百花齐放、百家争鸣、多样性与包容性令人欣慰。它们并存的最大意义，就是为求得复杂问题的搜索解，提供了更多可供选择的工具，为人类科学的进步提供了更多的探索方向。