中山大学计算机学院人工智能本科生实验报告

课程：Artificial Intelligence

姓名： 学号：

一、实验题目

用遗传算法求解TSP问题。选择两个数据集进行测试。

二、实验内容

1.算法原理

遗传算法是一种随机算法，原理是模拟自然界中生物进化。每一个个体用一个染色体代表。大量的染色体组成一个种群，种群中优秀的染色体得以生存并产生后代。这样就可以在大量的迭代之后，以更高的概率获得更优秀的个体，即问题的解。

算法的关键是：需要模拟种群与染色体的行为。即——1.种群中挑选优秀的个体得以生存；2.优秀的染色体以一定概率交叉，获得后代；3.种群中的染色体以一定概率变异。

2.伪代码

首先规定类 `GeneticAlgTSP` 中的方法如下：

```python
class GeneticAlgTSP:
    def __init__(self, filename: str) -> None:
        """初始化"""

    def iterate(self) -> np.ndarray:
        """解问题的对外方法，返回所得的最优路径"""

    def DrawPresent(self) -> None:
        """绘制当前迭代所得最优路径图"""
        # 这个函数会导致严重性能问题，故实际并不使用

    def DrawResult(self) -> None:
        """绘制迭代过程中最优路程的变化，以及迭代前的初始路径图、与迭代之后的最优路径图"""

    def __init_poplulation__(self) -> "list[np.ndarray]":
        """初始化种群"""

    def __SinglePointCross__(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
        """两个染色体间单点交叉"""

    def __TwoPointsCross__(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
        """两个染色体间双点交叉"""

    def CrossBetween(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
        """两个染色体间交叉"""

    def Cross(self, selection: "list[int]") -> None:
        """在选择亲本染色体后，按概率生成子代染色体"""

    def __PositionExchangeMeta__(self, gene: np.ndarray) -> None:
        """仅针对两个位的互换变异"""

    def __SegmentExchangeMeta__(self, gene: np.ndarray) -> None:
        """针对两个段的互换变异"""

    def Meta(self) -> None:
        """对种群中所有染色体按概率变异"""

    def Select(self) -> list:
        """选择亲本染色体"""

    def DistanceBetween(self, city1: int, city2: int) -> float:
        """计算两个城市间的距离"""

    def DistanceTotal(self, path: np.ndarray) -> float:
        """计算一个路径中的总距离"""

    def TotalFitness(self) -> tuple:
        """计算当前种群的各染色体适应度与总适应度"""

    def Fitness(self, path: np.ndarray) -> float:
```

```
        """计算一个路径的适应度"""

    @staticmethod
    def Fitness(distance: float) -> float:
        """计算一个路径的适应度"""
```

它们之间的调用层级如下所示：

$$
iterate \begin{cases} Select \begin{cases} DistanceTotal \rightarrow DistanceBetween \\ TotalFitness \rightarrow Fitness \rightarrow DistanceBetween \end{cases} \\ Cross \rightarrow CrossBetween \begin{cases} SinglePointCross \\ TwoPointsCross \end{cases} \\ Meta \begin{cases} PositionExchangeMeta \\ SegmentExchangeMeta \end{cases} \\ DrawResult \end{cases}
$$

其中关键的解问题的函数非常简单，就像在算法原理中所说的，不过三步而已：

```
def iterate(self):
    for 最大迭代次数内迭代:
        选择优秀个体
        交叉产生后代
        后代产生变异
    return 结果
```

现在比较详细地解析这三步的实现。首先是选择优秀个体 `Select` 。这一步的原理是按照适应度，加权作为概率，从中尽可能地选择优秀个体，或者说淘汰适应度不高的个体。总体思路是：

```
def Select(self) -> list:
    按路程升序排列(即按适应度降序排列)
    计算每一个染色体的适应度与总适应度
    按(染色体适应度/总适应度)的概率选择亲本(的序号)
    return 所选亲本序号
```

接下来是交叉。交叉是对两个亲本染色体操作，使其产生与亲本不同的后代。在我的程序中， `Cross` 只是一个入口函数，它只负责按概率从被选亲本中挑选两个亲本，作为 `CrossBetween` 的参数：

```
def Cross(self, selection: "list[int]") -> None:
    while (种群未满):
        在selection中选择两个亲本
        CrossBetween(亲本1, 亲本2)
```

`CrossBetween` 也只是一个入口函数，它只负责按概率决定两个亲本是单点交叉还是双点交叉：

```
def CrossBetween(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
    if (概率): return (self.__SinglePointCross__(gene1, gene2))
    else: return (self.__TwoPointsCross__(gene1, gene2))
```

关键在于交叉的具体操作。双点交叉和单点交叉基本类似，这里只说明单点交叉。由于 TSP 问题的特殊性，染色体交叉绝非是直接相互替换。首先选定交叉点。之后，在亲本1的保留部分中剔除掉亲本2替换部分中含有的城市，同理对亲本2类似操作。之后再相互拼接。这样才能保证对每一个城市都至少且至多途经一次。

```
def __SinglePointCross__(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
    选定crosspoint
    child1, child2 = gene1[0:crosspoint], gene2[0:crosspoint]
    for city in gene2[crosspoint:]: child1.remove(city)
    for city in gene1[crosspoint:]: child2.remove(city)
    child1.append(gene2[crosspoint:])
    child2.append(gene1[crosspoint:])
    return (child1, child2)
```

类似地，接下来的变异操作 Meta 也只是一个入口函数。它只负责按概率决定是否变异，并按概率决定选择是单位交换变异还是段交换变异：

```
def Meta(self) -> None:
    for 新生子代:
        if (小于变异概率): continue
        if (概率): self.__PositionExchangeMeta__(新生子代)
        else: self.__SegmentExchangeMeta__(新生子代)
```

还是只介绍单位交换变异：

```
def __PositionExchangeMeta__(self, gene: np.ndarray) -> None:
    选择point1, point2
    gene[point1], gene[point2] = gene[point2], gene[point1]
```

我的适应度函数很简单，将路程取倒数即可。不过考虑到大部分路程的量级都是万以上，因此还再放大 1000~10000 倍：

```
def Fitness(self, path: np.ndarray) -> float:
        return (1000 / self.DistanceTotal(path))
```

3.关键代码展示

以下展示 Select，Cross，Meta 函数(模块)。首先是 Select ：

```
def Select(self) -> list:
        """选择亲本染色体"""
        self.population.sort(key = lambda x: self.DistanceTotal(x)) # 按照路程大小升序排序
        self.best_gene_per_generation.append(self.population[0]) # 因此排序之后第一个染色体必是最优者，记录之
        fitness, totalfitness = self.TotalFitness() # 为选择做准备。fitness是各路径的适应度的数组。
        return np.random.choice(np.arange(0, self.max_population), size = (int)(self.max_population * self.survive
```

交叉函数 Cross：

```
def __SinglePointCross__(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
    """两个染色体间单点交叉"""
    crosspoint = np.random.randint(0, self.cities_amount) # 选择交叉点
    gene1_list, gene2_list = gene1.tolist(), gene2.tolist()
    child1, child2 = copy.deepcopy(gene1_list), copy.deepcopy(gene2_list)
    for city in gene1_list[crosspoint:]: child2.remove(city) # 剔除保留部分中在交换部分中已有的基因
    for city in gene2_list[crosspoint:]: child1.remove(city) # 剔除保留部分中在交换部分中已有的基因
    child1.extend(gene2_list[crosspoint:])
    child2.extend(gene1_list[crosspoint:])
    return np.array(child1), np.array(child2)

def __TwoPointsCross__(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
    """两个染色体间双点交叉"""
    crosspoint1 = np.random.randint(0, self.cities_amount // 2)
    crosspoint2 = np.random.randint(self.cities_amount // 2, self.cities_amount)
    gene1_list, gene2_list = gene1.tolist(), gene2.tolist()
    child1_front, child2_front = copy.deepcopy(gene1_list[0: crosspoint1]), copy.deepcopy(gene2_list[0: crossp
    child1_back, child2_back = copy.deepcopy(gene1_list[crosspoint1:]), copy.deepcopy(gene2_list[crosspoint1:]
    for city in gene1_list[crosspoint1: crosspoint2]:
        if city in child2_front: child2_front.remove(city) # 剔除保留部分中在交换部分中已有的基因
        if city in child2_back: child2_back.remove(city)
    for city in gene2_list[crosspoint1: crosspoint2]:
        if city in child1_front: child1_front.remove(city) # 剔除保留部分中在交换部分中已有的基因
        if city in child1_back: child1_back.remove(city)
    child1_front.extend(gene2_list[crosspoint1: crosspoint2])
    child1_front.extend(child1_back)
    child2_front.extend(gene1_list[crosspoint1: crosspoint2])
    child2_front.extend(child2_back)
    return np.array(child1_front), np.array(child2_front)

def CrossBetween(self, gene1: np.ndarray, gene2: np.ndarray) -> "tuple[np.ndarray]":
    """两个染色体间交叉"""
    if (np.random.rand() < 0.5): return self.__SinglePointCross__(gene1, gene2) # 五成概率单点交叉
    else: return self.__TwoPointsCross__(gene1, gene2) # 五成概率双点交叉

def Cross(self, selection: "list[int]") -> None:
    """在选择亲本染色体后，按概率生成子代染色体"""
    # 这里是直接将被淘汰者直接替换为新的子代，直到全部替换为止
    replace_index = len(selection) + 1
    for path_index in selection:
        if (np.random.rand() > self.cross_prob): continue
        if (replace_index >= self.max_population): break
        other = np.random.randint(0, len(selection))
        self.population[replace_index - 1], self.population[replace_index] = self.CrossBetween(self.populatior
        replace_index = replace_index + 2
```

变异函数 Meta ：

```
def __PositionExchangeMeta__(self, gene: np.ndarray) -> None:
    """仅针对两个位的互换变异"""
    point1, point2 = np.random.randint(0, self.cities_amount), np.random.randint(0, self.cities_amount)
    gene[point1], gene[point2] = gene[point2], gene[point1]

def __SegmentExchangeMeta__(self, gene: np.ndarray) -> None:
    """针对两个段的互换变异"""
    point1, point2 = np.random.randint(0, self.cities_amount // 2), np.random.randint(self.cities_amount // 2,
    lenth = np.random.randint(0, min(point2 - point1, self.cities_amount - point2))
    for i in range(0, lenth): gene[point1 + i], gene[point2 + i] = gene[point2 + i], gene[point1 + i]

def Meta(self) -> None:
    """对种群中所有子代按概率变异"""
    for i in range((int)(self.max_population * self.survive_rate), self.max_population): # 只对子代部分变异
        if (np.random.rand() > self.meta_prob): continue
        else:
            if (np.random.rand() < 0.5): self.__PositionExchangeMeta__(self.population[i]) # 五成概率单位交换变
            else: self.__SegmentExchangeMeta__(self.population[i]) # 五成概率段交换变异
```

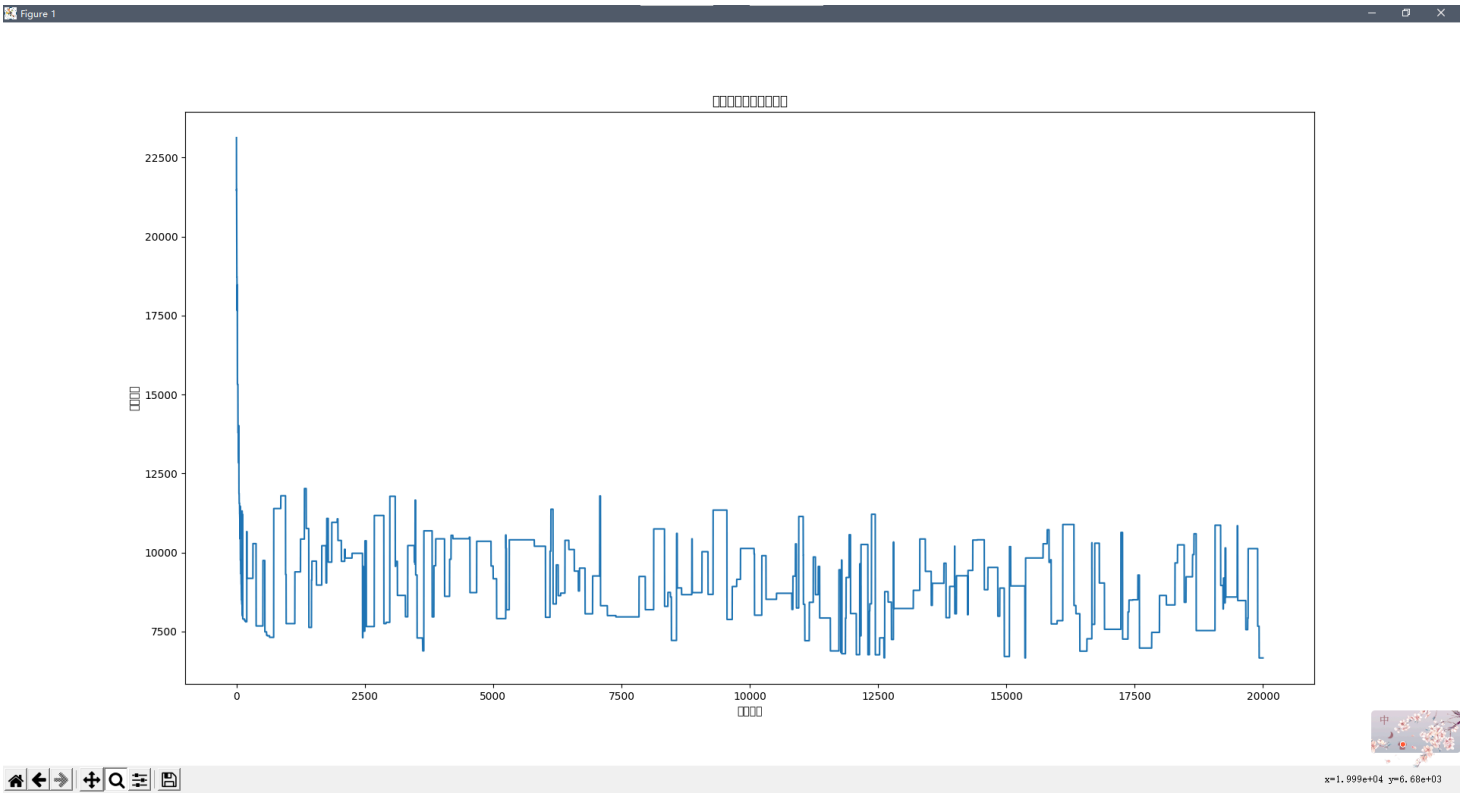
4.创新优化

本次实验的创新之处在于，对于交叉和变异，均提供了两种具体方法。即对于交叉操作，既有可能是单点交叉，也有可能是双点交叉；对于变异操作，既有可能是单位交换变异，也有可能是段交换变异。这样的操作为算法提供了更多样的选择与更大的可能性。

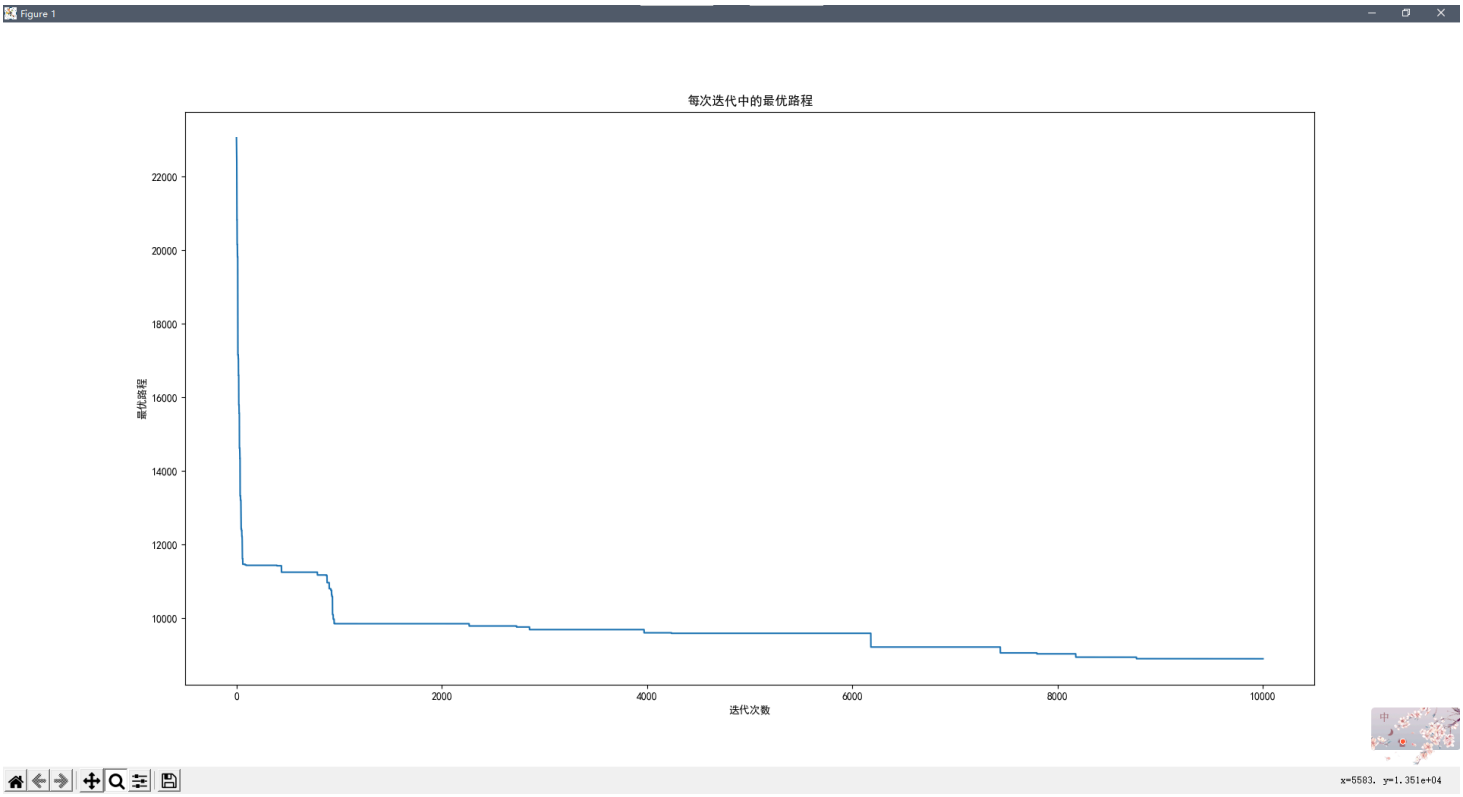
三、实验结果分析

1.实验结果展示

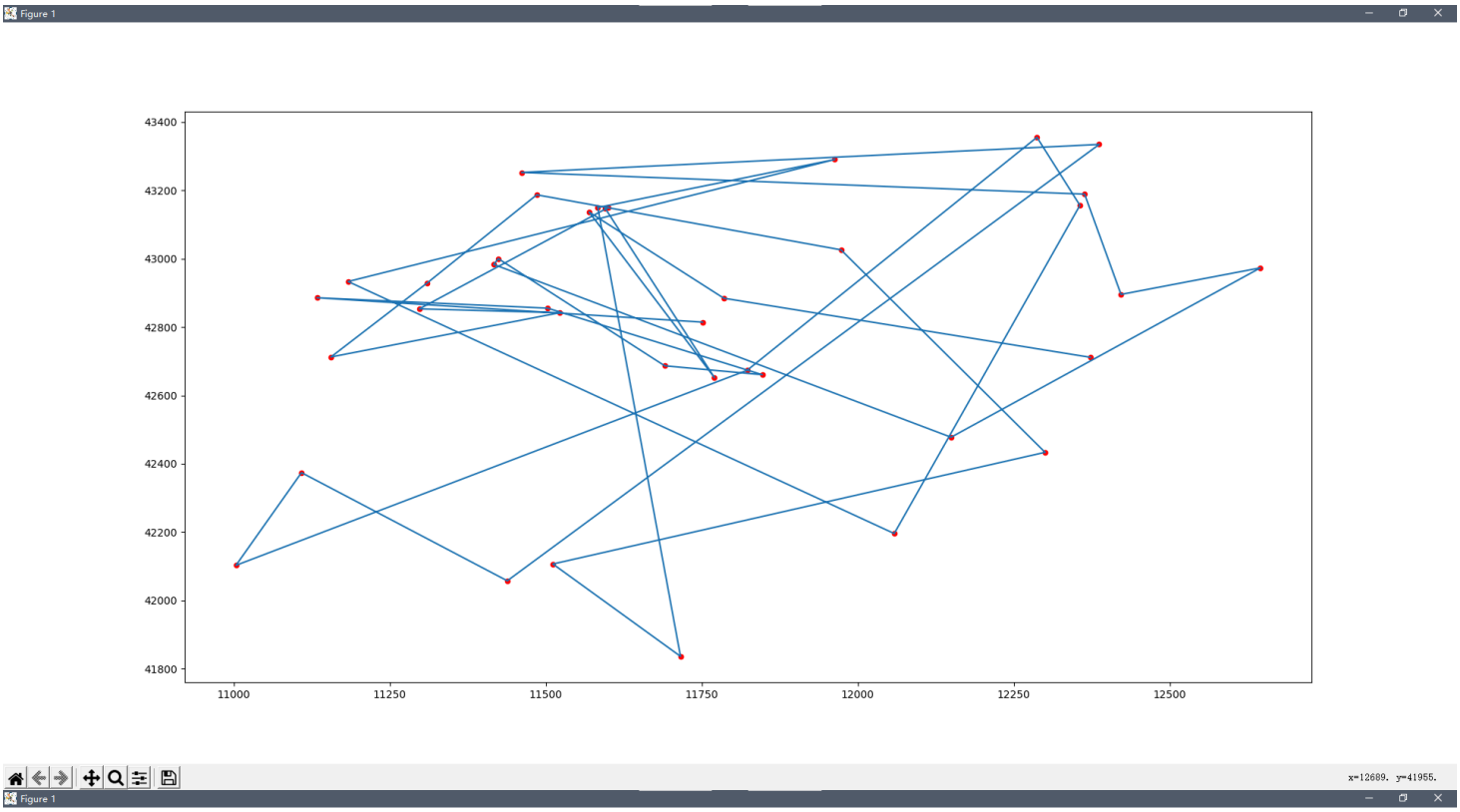
首先测试了38个城市的数据集。一开始，我采用了对种群中所有染色体均概率变异的策略，得到每代最优路程的下降如图：

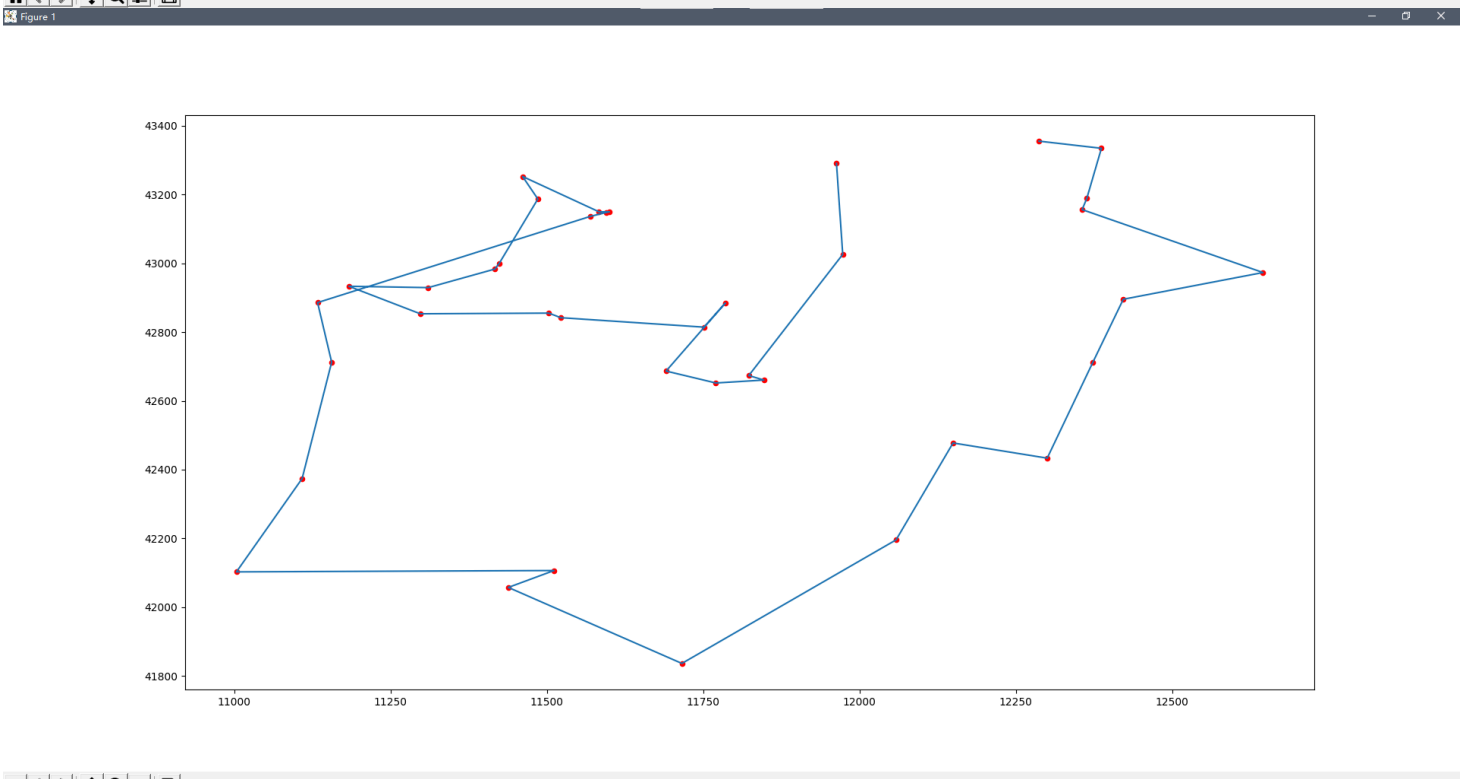


可以看到这样的策略会导致极大的波动，最后一次迭代得到的不一定会是最优解。采用只对子代变异的策略之后：

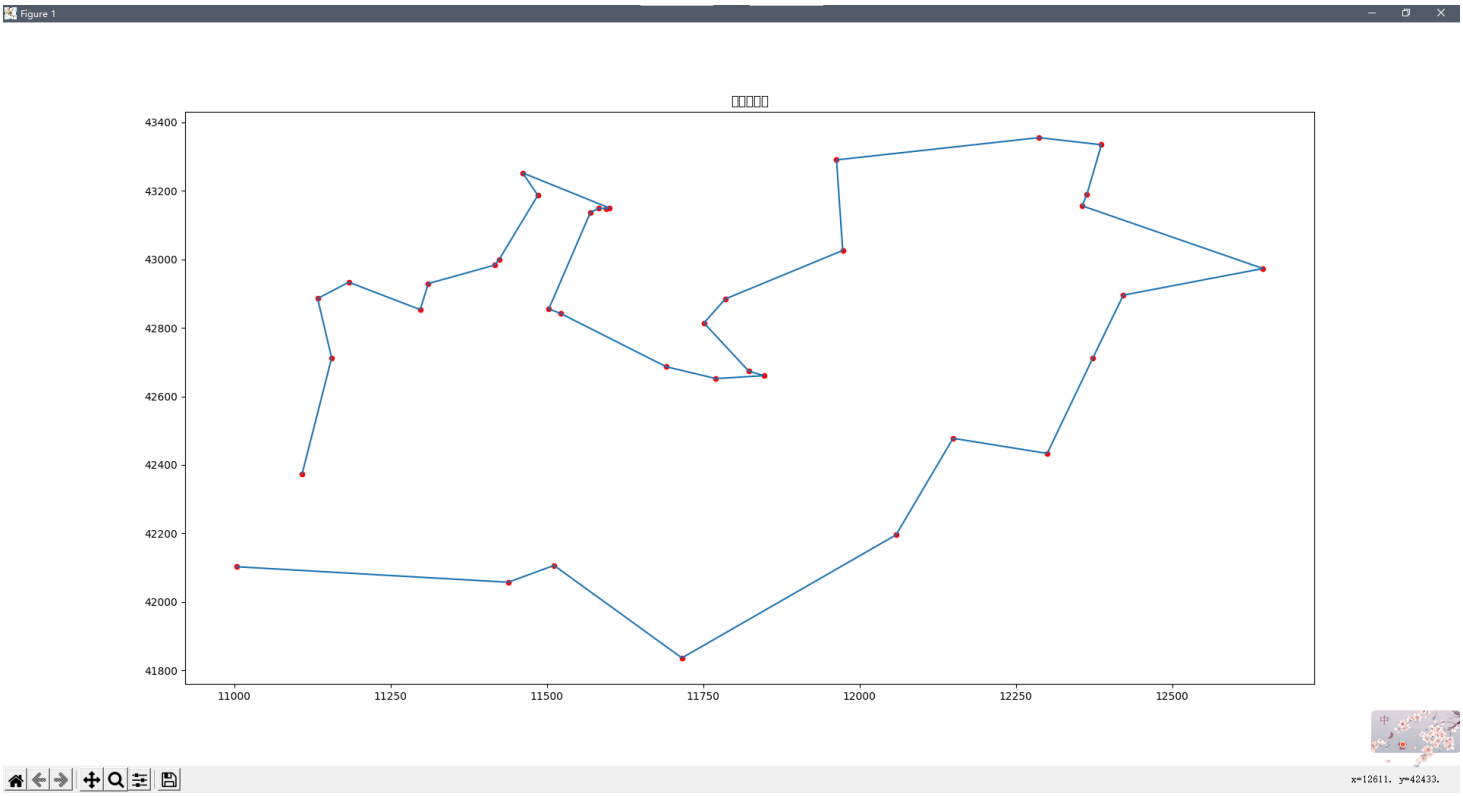


趋势图就是完全单调下降了，可以保证最后一代一定比之前的优秀。以下给出 dj38 数据集在算法运行之前和之后给出的路径图：

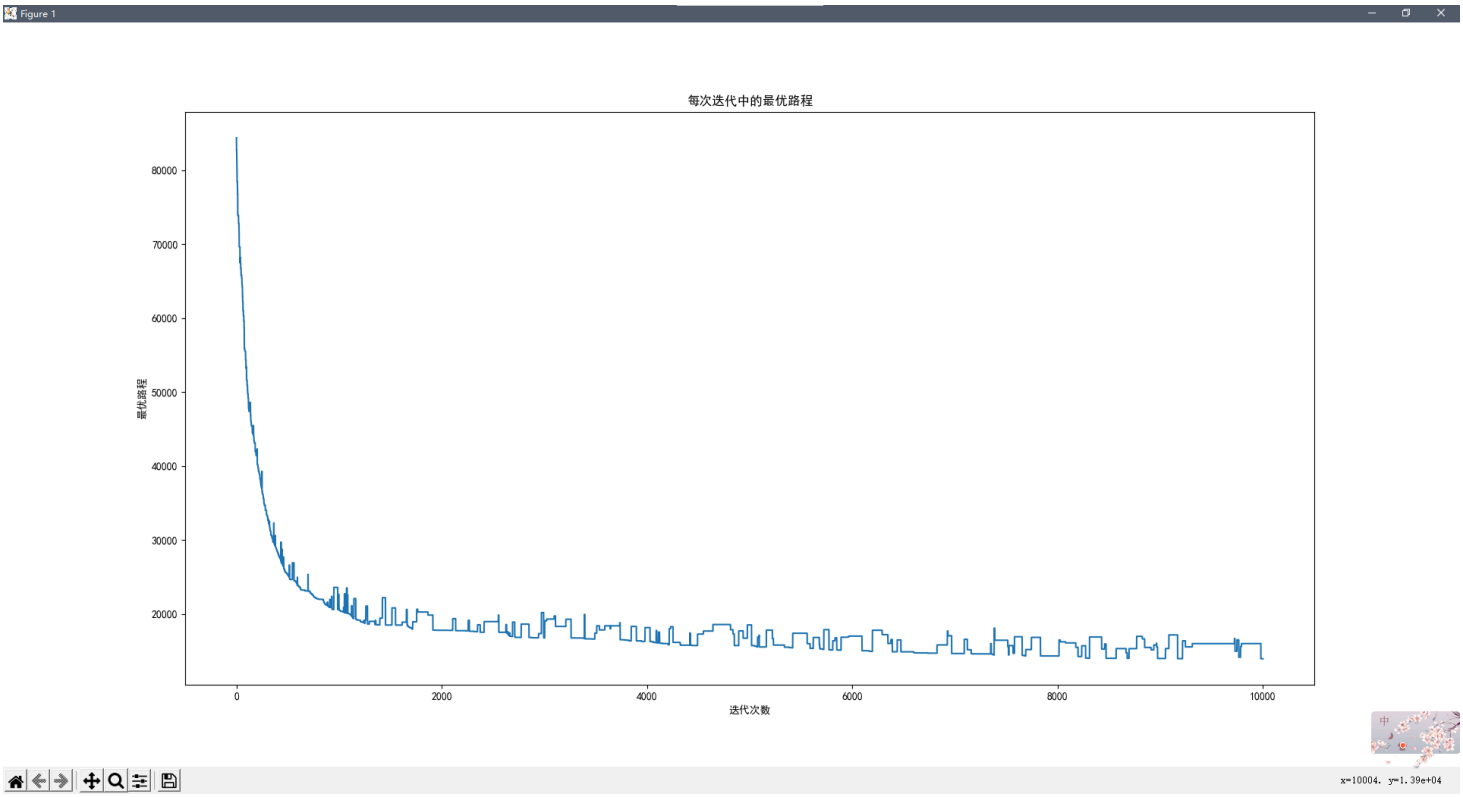


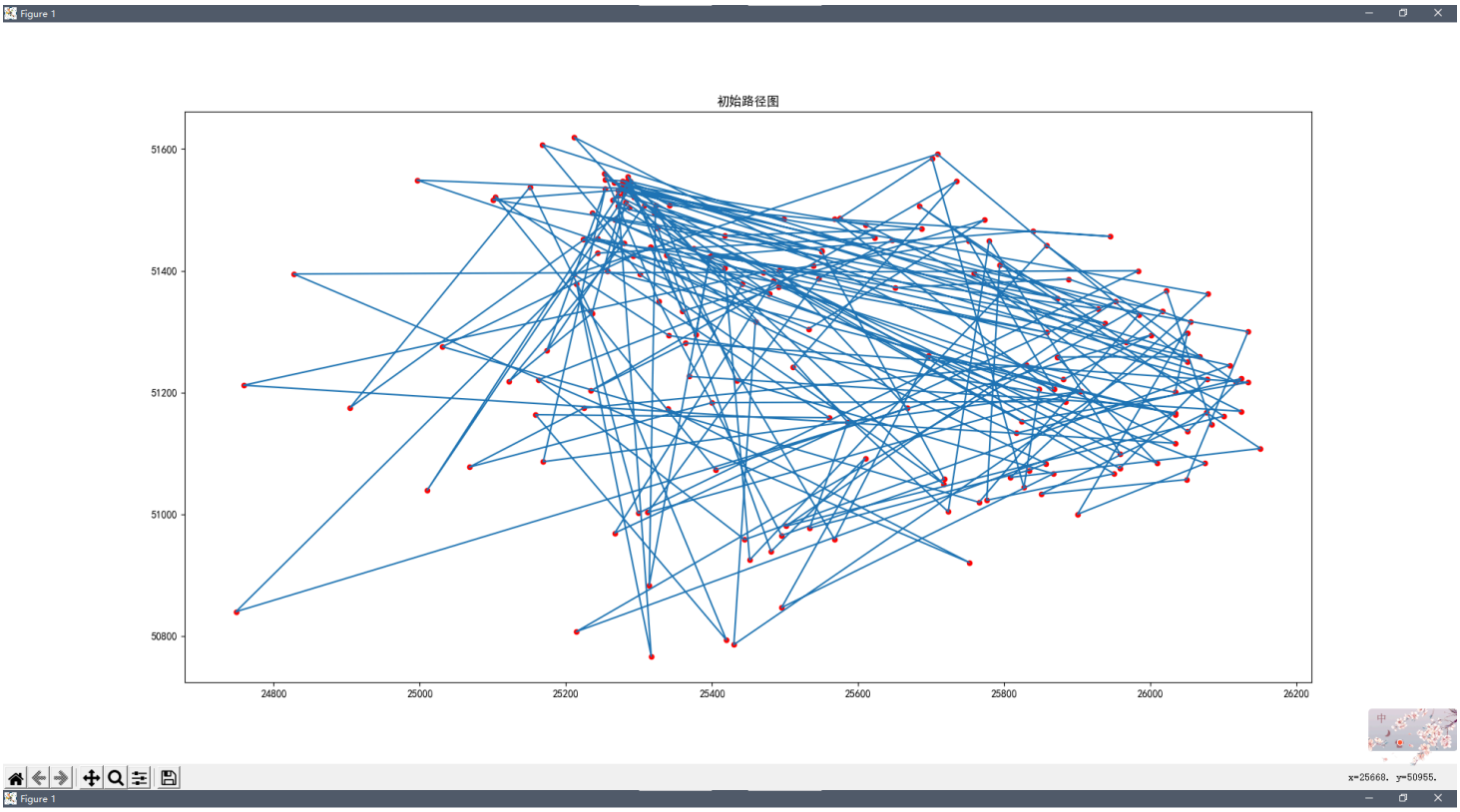


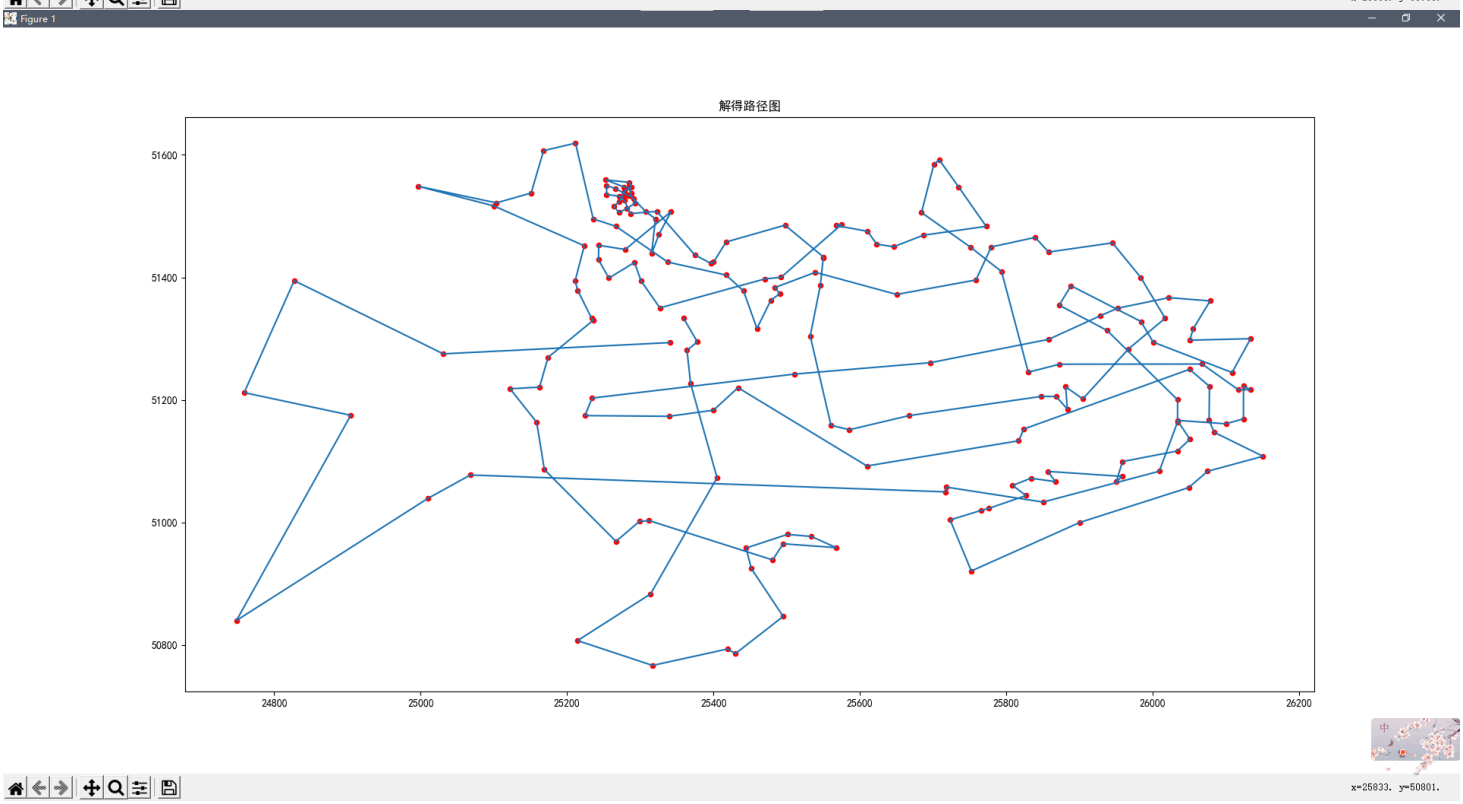
在运气好的情况下，有可能可以得到最优解：



现在给出 qa194 即194个城市的每代最佳路程下降图与算法运行前后的路径图：







虽然获得的不是最优解，但所得结果显然比初始状态优秀得多。

现在给出 `mu1979` 即1979个城市的每代最佳路程下降图与算法运行前后的路径图：

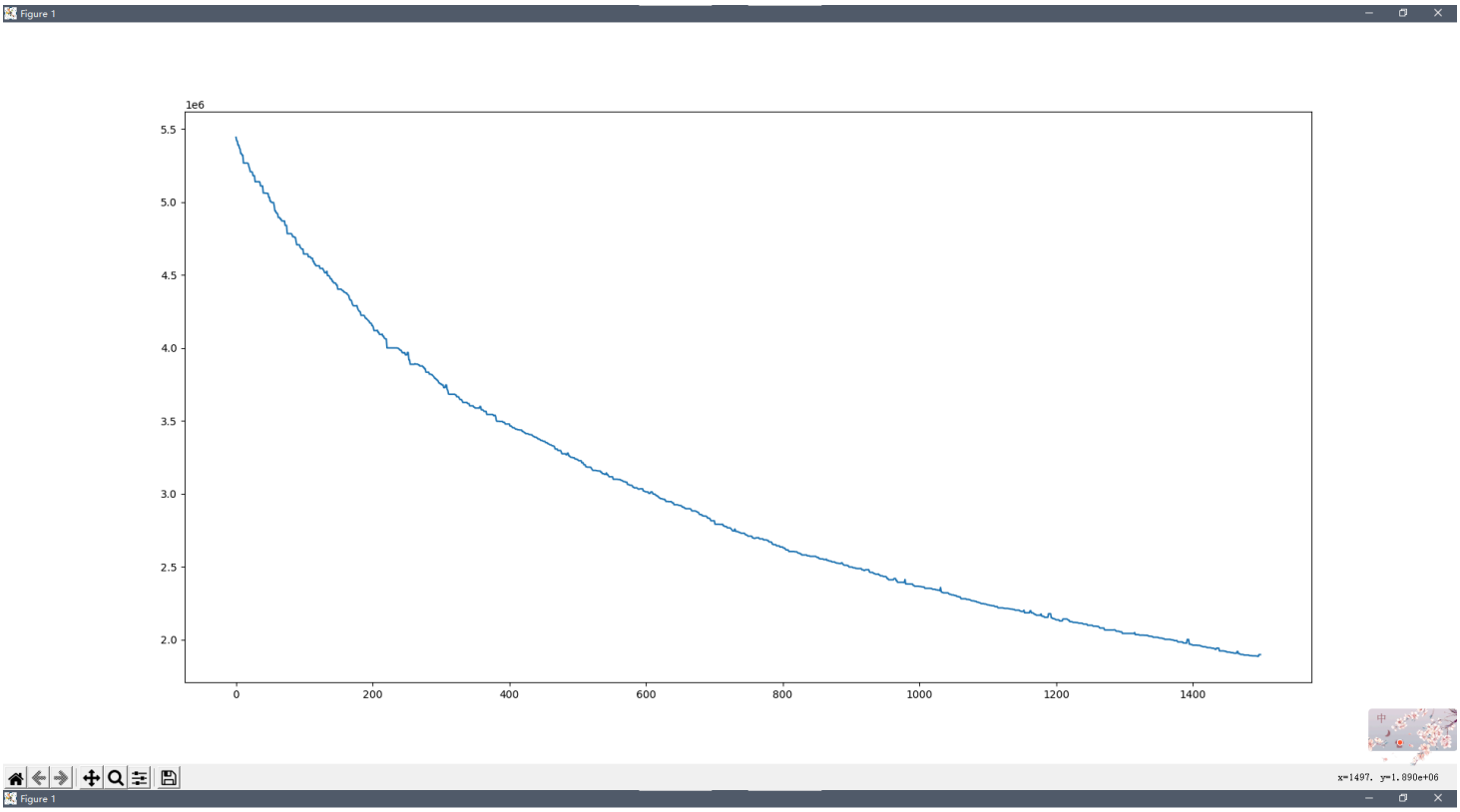


Figure 1
1e6
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
0
200
400
600
800
1000
1200
1400
x=1497. y=1.890e+06

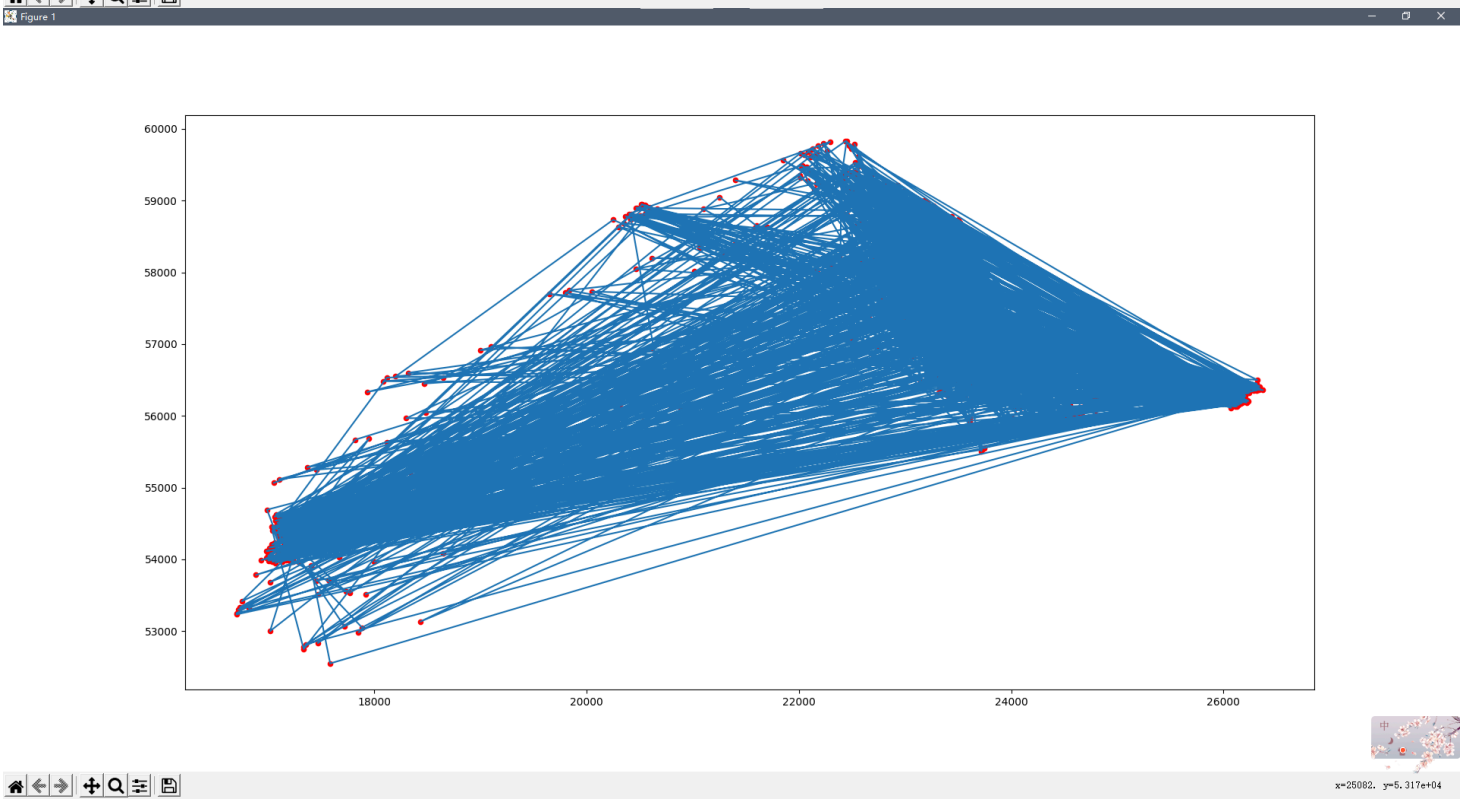
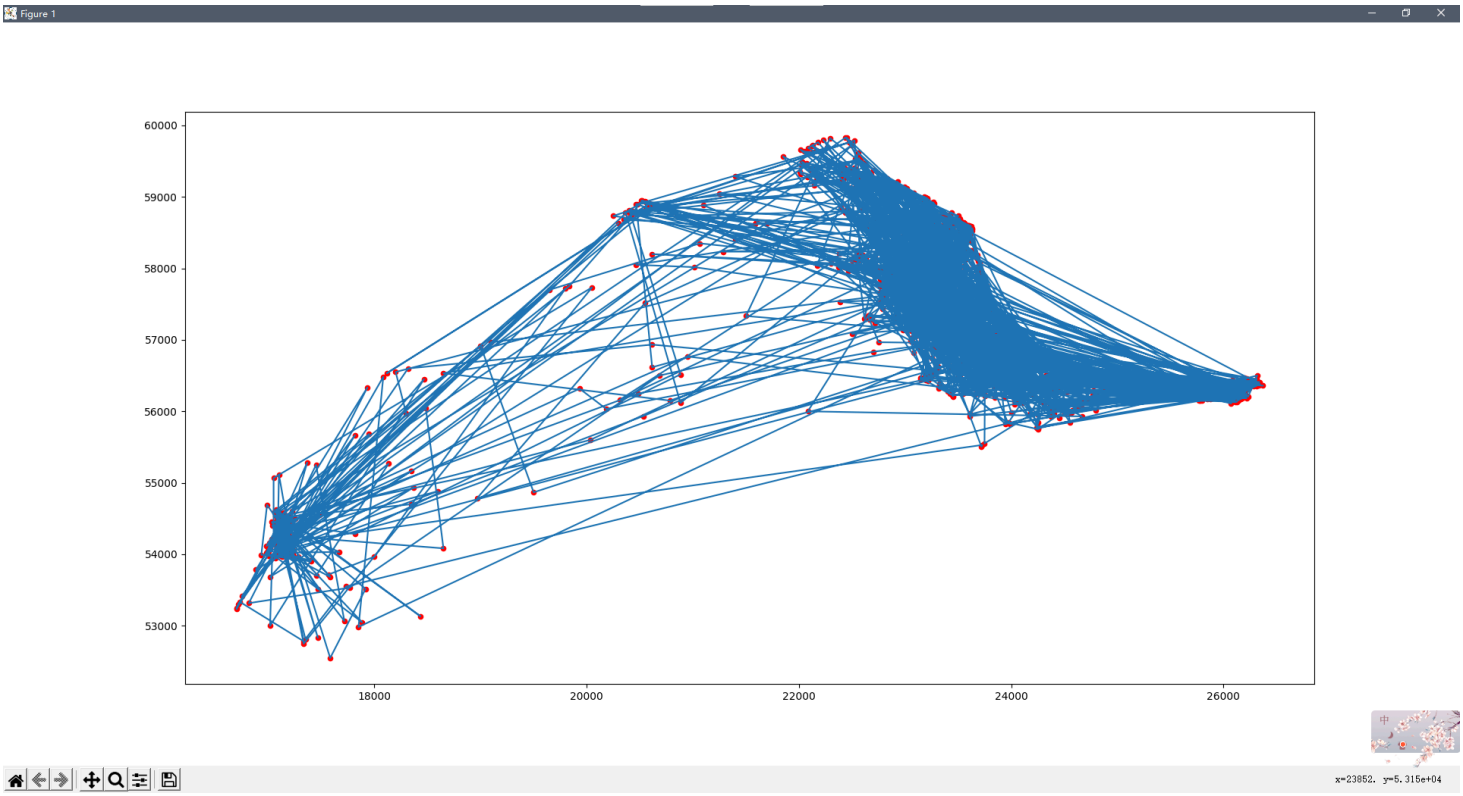
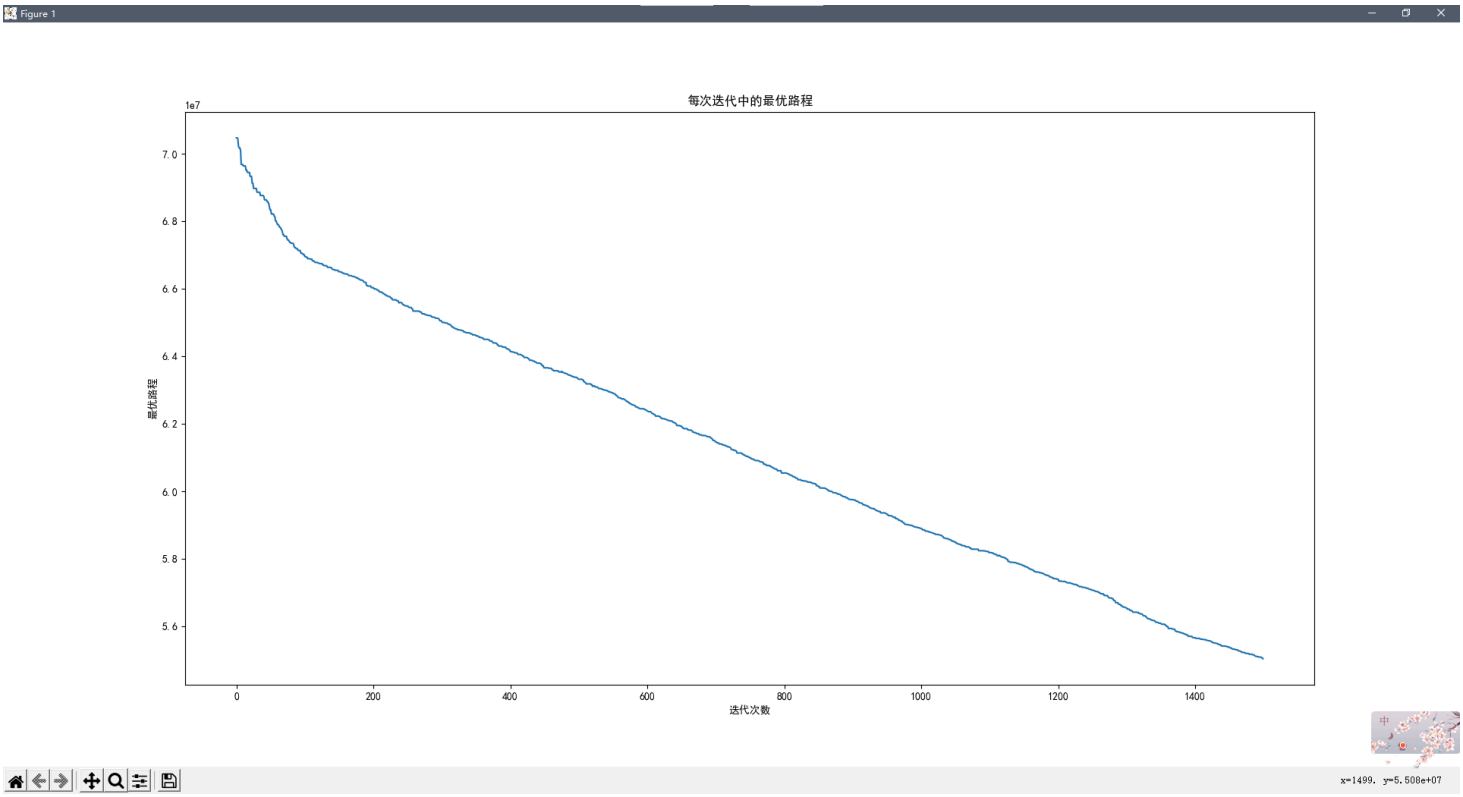


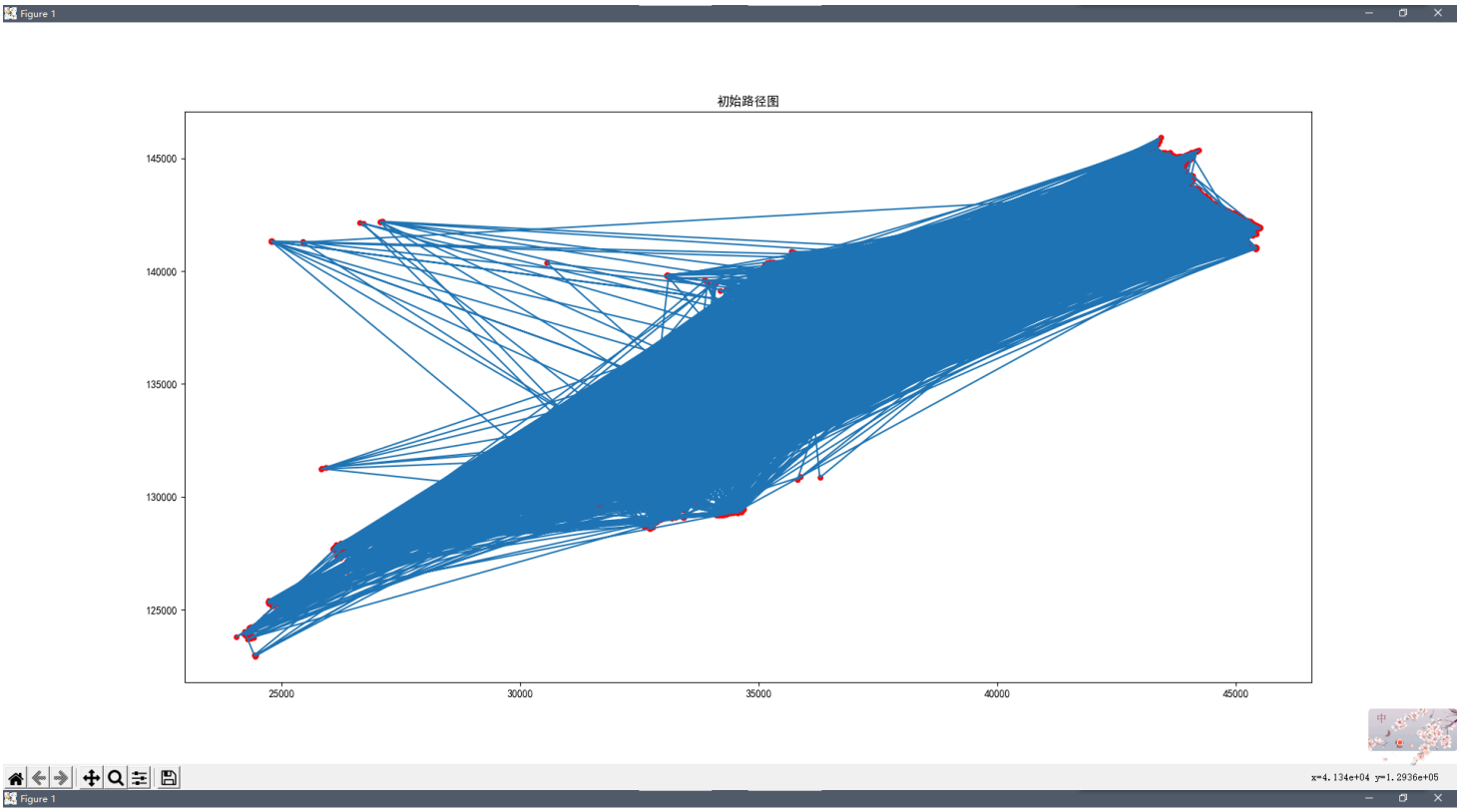
Figure 1
60000
59000
58000
57000
56000
55000
54000
53000
18000
20000
22000
24000
26000
x=25082. y=5.317e+04

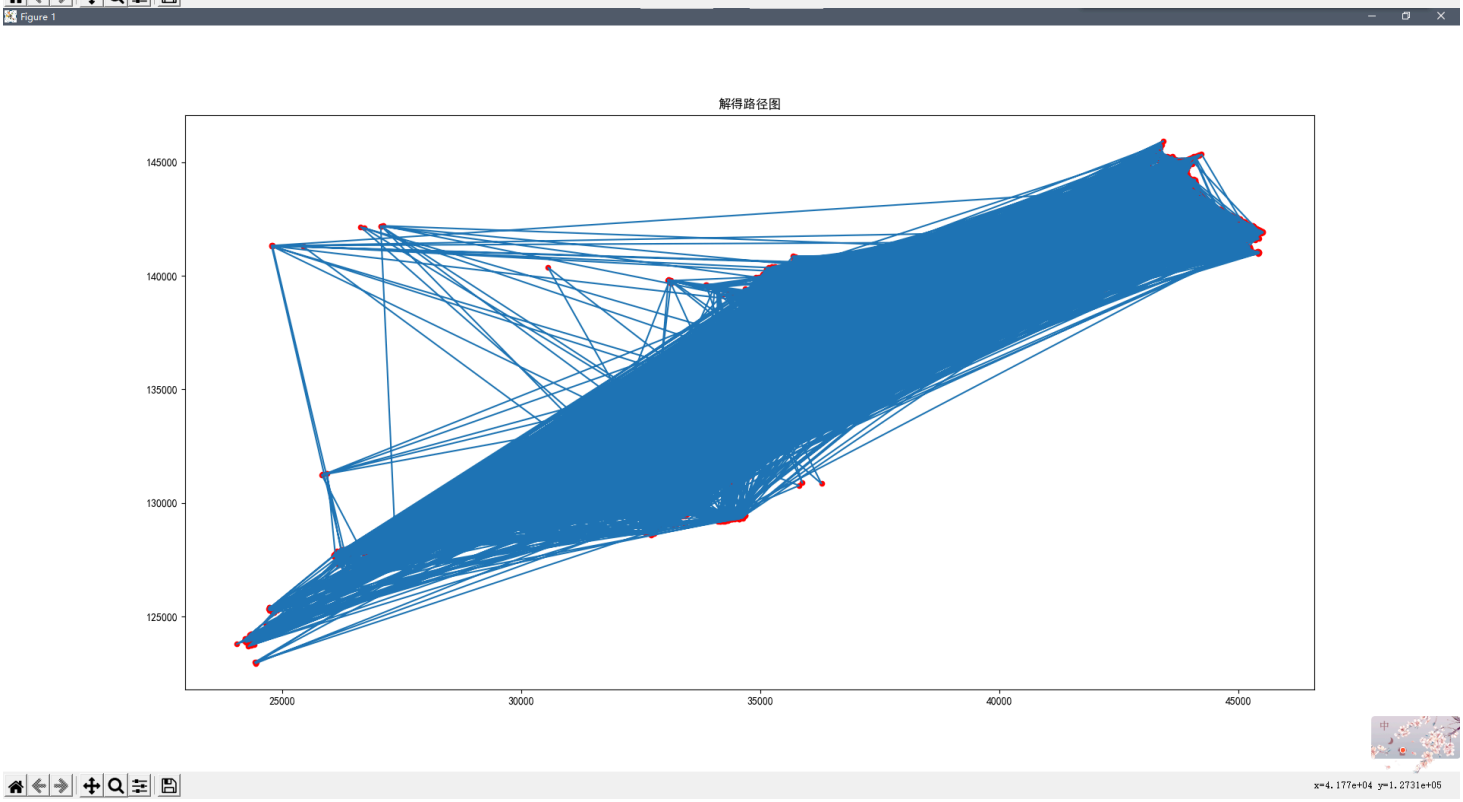


当城市数数量级再次上升，所需要的时间也成倍增长，万次的迭代已经不可接受。

现在给出 `ja9847` 即9847个城市的每代最佳路程下降图与算法运行前后的路径图：







对于万级别的城市数，千次的迭代已经不能为路径带来本质的不同，尽管每代的最佳路程数依然是稳定下降的。可见对于大规模问题而言，使用遗传算法必须要有高算力的支持，或者在遗传算法内部结合使用其它搜索算法，以提高搜索效率。

2.评测指标展示分析

以下给出几个条件下的性能指标分析：

城市数	种群容量	生存率	交叉率	变异率	迭代次数	运行时间	所得最短路程	推荐最短路程
38	100	0.5	0.6	0.01	10000	3:55	8893	6656
38	100	0.5	0.6	0.2	10000	3:23	7285	6656
38	500	0.5	0.6	0.01	20000	33:09	6661	6656
197	100	0.2	0.6	0.01	10000	11:33	23548	9352
197	500	0.5	0.6	0.01	10000	58:35	13926	9352
1979	100	0.5	0.6	0.01	3000	58:22	1.9e6	8.7e4
1979	100	0.1	0.6	0.01	3000	47:58	4.4e6	8.7e4
1979	500	0.5	0.6	0.01	1500	2:24:36	1.9e6	8.7e4
9847	100	0.5	0.6	0.01	1500	7:06:52	5.5e7	4.9e5
9897	100	0.5	0.6	0.2	1500	8:36:21	5.4e7	4.9e5

从上表分析可知，对算法运行时间有显著影响的因素是：城市数、种群容量、迭代次数。基本上对于每一因素而言，都是正比关系。其中，由于在“选择”步骤中涉及到对种群排序的操作，因此当种群容量从n放大至an时，时间应放大$\frac{an\ log\ an}{nlogn} = a(log_n a + 1)$倍。在以上测试例中，放大倍数都为$5$，相对于原种群容量$100$极小，$log_n a$可忽略不计，因此可以直接认为时间与种群容量成正比。

四、思考题

无。

五、参考资料

无。