# 中山大学计算机学院
# 人工智能
# 本科生实验报告

课程名称：Artificial Intelligence

| | | | |
|---|---|---|---|
| 学号 | 22336259 | 姓名 | **谢宇桐** |

## 一、 实验题目

**遗传算法解决 TSP 问题**

## 二、 实验内容

### 1. 算法原理

### 在此次解决旅行商问题我们用到了遗传算法，其原理如下：

遗传算法是一种基于自然选择和群体遗传机理的搜索算法,它模拟了自然选择和自然遗传过程中的繁殖、杂交和突变现象。求解问题时,问题的每一个可能解都被编码成一个“染色体”,即个体,若干个个体构成了群体(所有可能解)。它反复修改一种候选解的群体。在每一代中，对群体中的个体评估适应度， 然后从当前种群中选择更适合的个体，并使用交叉、变异操作生成新的生命种群， 引导种群朝着最优解进化。

其关键步骤如下：

### 1.编码：

利用遗传算法求解问题时,首先要确定问题的目标函数和变量,然后对变量进行编码。考虑 n 个城市，我们定义染色体是一个整数排列Π，表示访问城市的顺序，其中Π(i)∈{1,2,3,…,n}是不同城市的编号。

### 2.遗传操作

遗传操作是模拟生物基因的操作，他的任务就是根据个体适应度对其施加一定的操作，从而实现优胜劣汰的进化过程。我们取适应度函数为路径总长度的倒数：

$$Fitness(\pi) = \frac{1}{L(\pi)}$$

其中：

$$L(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$$

d 表示两个城市间的距离。这样，路径长度越短，适应度值越高。

### 2.1 选择

选择是指从群体中选择优良个体并淘汰劣质个体的操作.它建立在适应度评估的基础上.适应度越大的个体,被选中上的可能性就越大,他的“子孙”在下一代中的个数就越多,选择出来的个体就被放入配对库中。在这里，我们使用轮盘赌注方法进行选择。对于种群中的某个体 i，其被选择的概率定义为：

$$P(i) = \frac{Fitness(i)}{\sum_{j=1}^{N} Fitness(j)}$$

其中 N 是种群的大小。

### 2.2 交叉

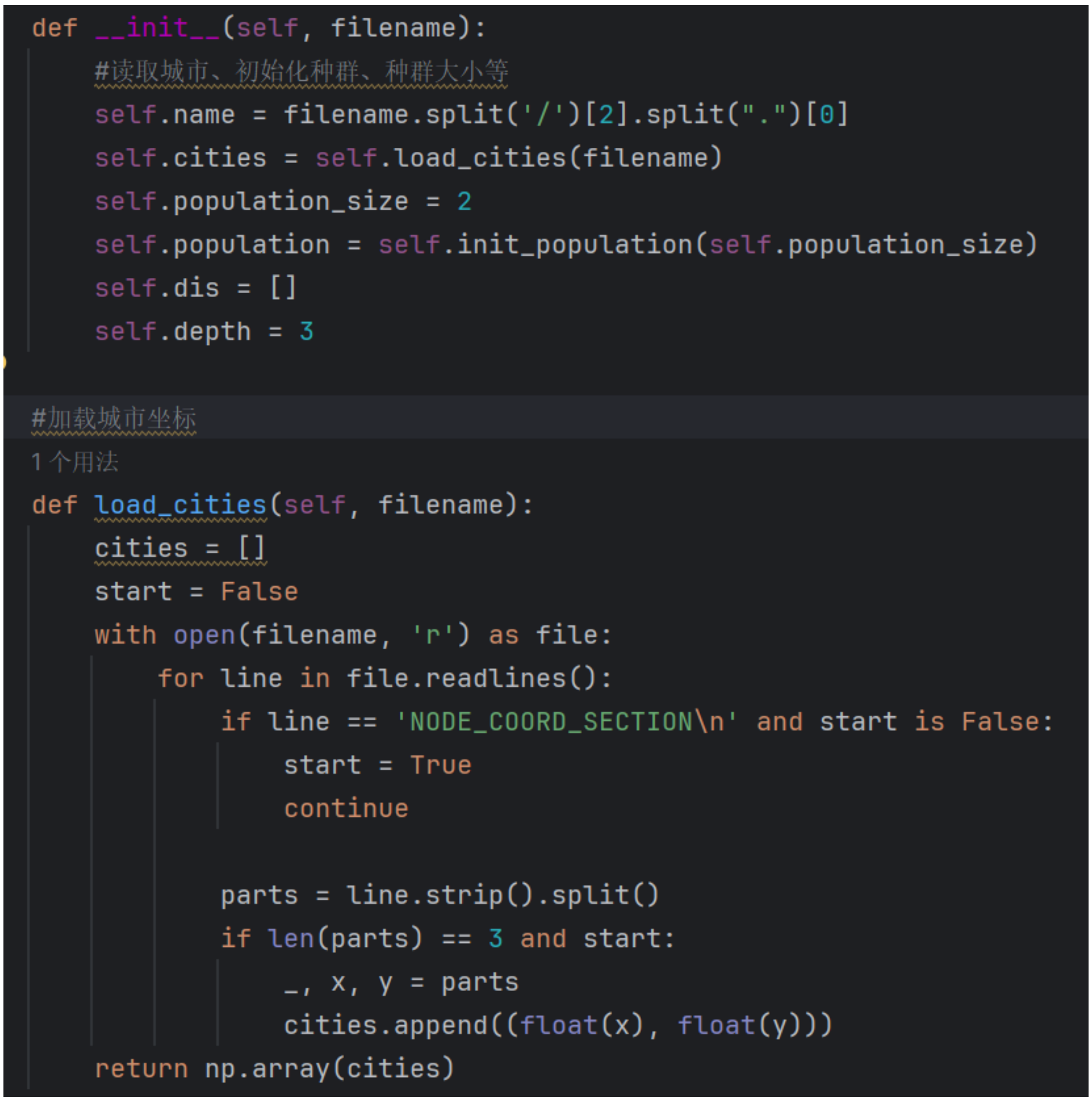
交叉就是指把两个父代个体的部分结构加以替换重组而生成新的个体的操作,交叉的目的是为了在下一代产生新的个体。交叉操作是按照一定的交叉概率在匹配库中随机的选取两个个体进行的,交叉位置也是随机的，例如随机选择两个交叉点 s,t。再将 c1 的 s 到 t 段直接复制自 p1，c2 的 s 到 t 段直接复制自 p2。最后根据第二步的映射关系，修改 c1 和 c2 的剩余部分，保证每个城市只出现一次。

### 2.3 变异

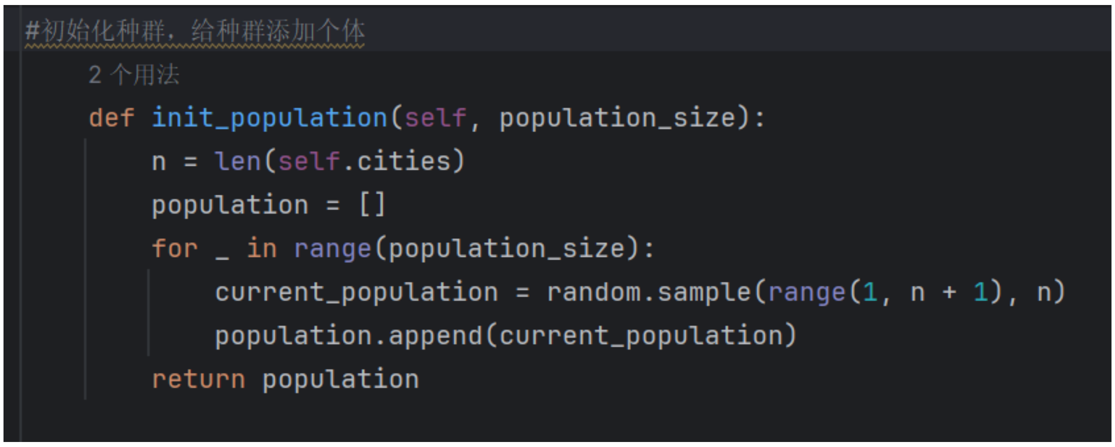
变异就是以很小的变异概率 Pm 随机地改变种群中个体的某些基因的值，操作如下：产生一个[0,1]之间的随机数 rand,如果 rand<Pm,则进行变异操作。我们考虑在子代个体上进行倒置变异，随机改变其部分染色体。对于子代个体的染色体Π，我们随机选取Π上的两个点 i 和 j，把Π(i)～ Π(j)倒置。

遗传算法通过以上核心步骤，不断迭代进化，逐步逼近最优解，从而解得最优 TSP 问题。

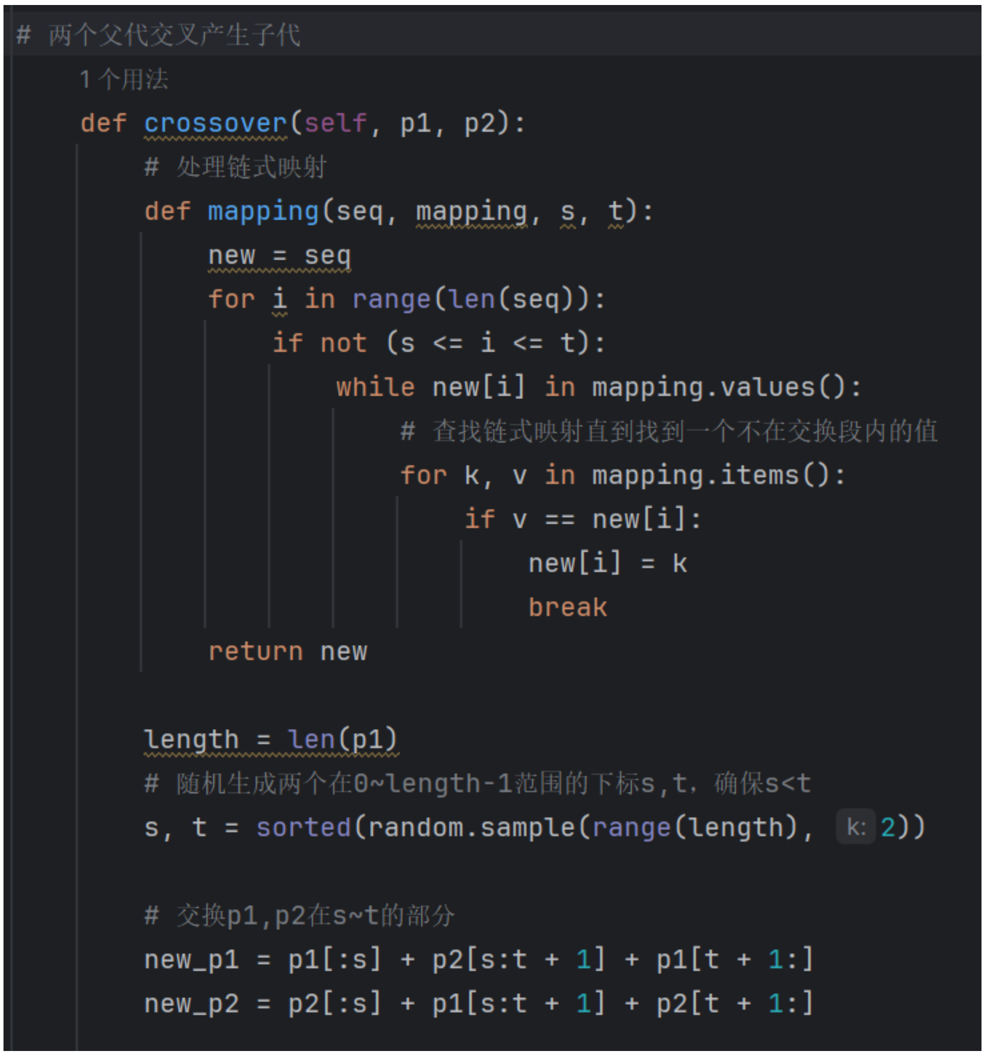
## 2. 关键代码展示（可选）



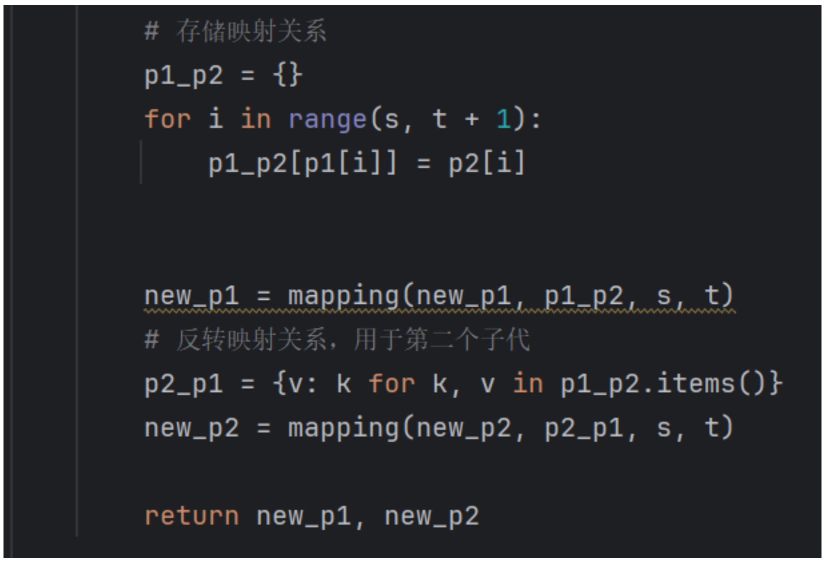
```
def __init__(self, filename):
    #读取城市、初始化种群、种群大小等
    self.name = filename.split('/')[2].split(".")[0]
    self.cities = self.load_cities(filename)
    self.population_size = 2
    self.population = self.init_population(self.population_size)
    self.dis = []
    self.depth = 3

#加载城市坐标
1 个用法
def load_cities(self, filename):
    cities = []
    start = False
    with open(filename, 'r') as file:
        for line in file.readlines():
            if line == 'NODE_COORD_SECTION\n' and start is False:
                start = True
                continue

            parts = line.strip().split()
            if len(parts) == 3 and start:
                _, x, y = parts
                cities.append((float(x), float(y)))
    return np.array(cities)
```



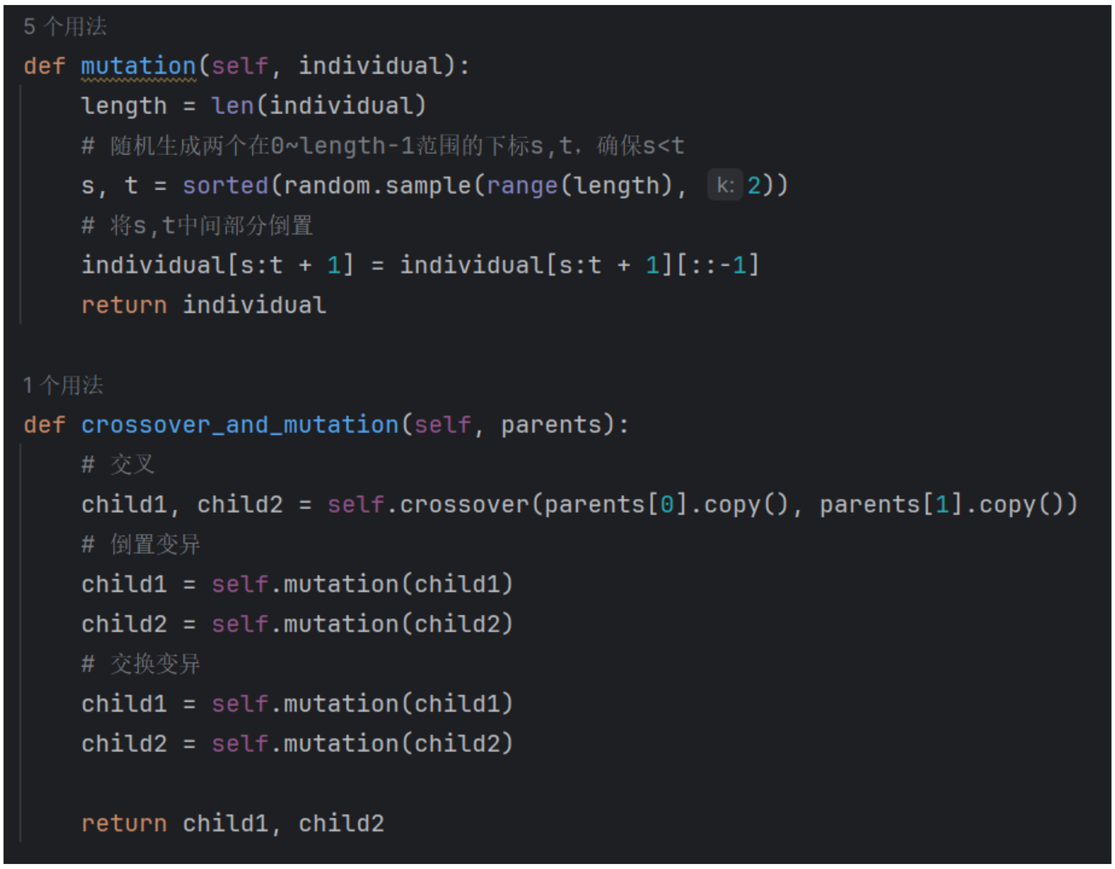
```
#初始化种群，给种群添加个体
    2 个用法
    def init_population(self, population_size):
        n = len(self.cities)
        population = []
        for _ in range(population_size):
            current_population = random.sample(range(1, n + 1), n)
            population.append(current_population)
        return population
```



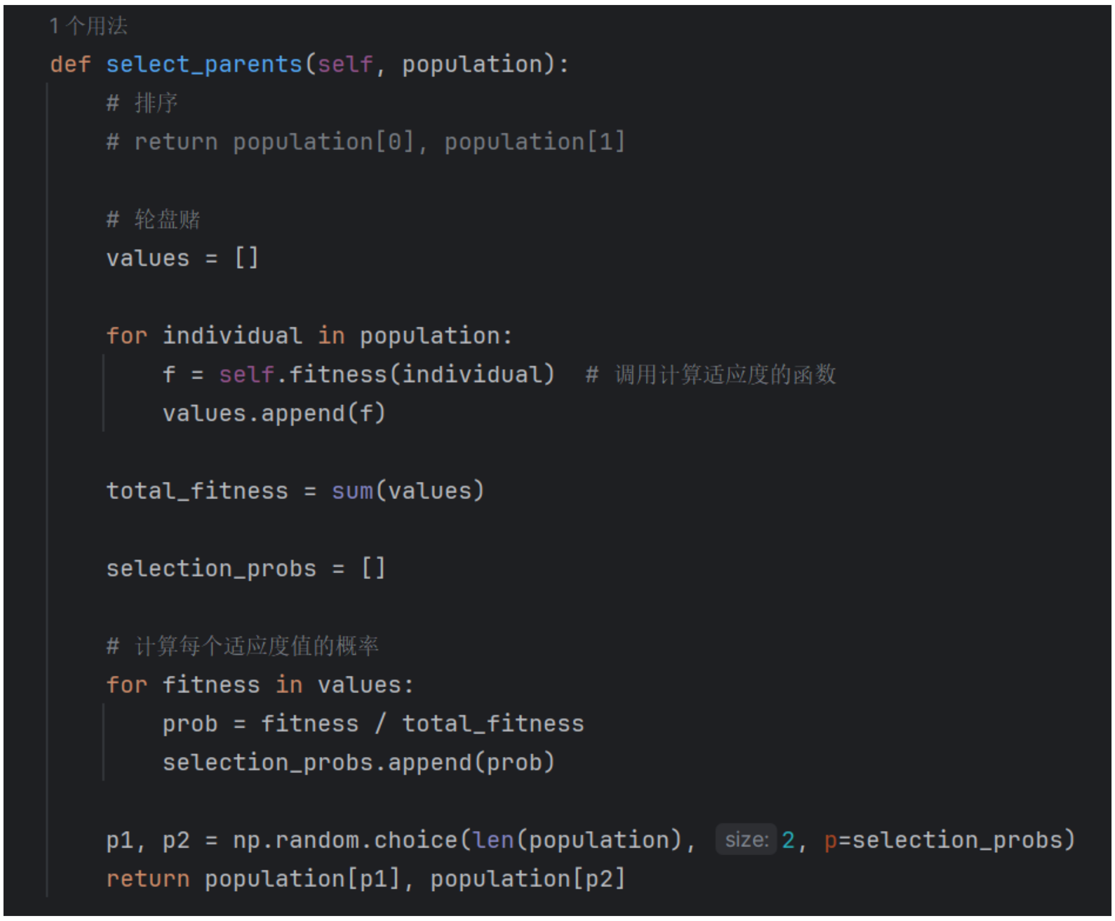
```python
# 两个父代交叉产生子代
    1 个用法
    def crossover(self, p1, p2):
        # 处理链式映射
        def mapping(seq, mapping, s, t):
            new = seq
            for i in range(len(seq)):
                if not (s <= i <= t):
                    while new[i] in mapping.values():
                        # 查找链式映射直到找到一个不在交换段内的值
                        for k, v in mapping.items():
                            if v == new[i]:
                                new[i] = k
                                break
            return new

        length = len(p1)
        # 随机生成两个在0~length-1范围的下标s,t, 确保s<t
        s, t = sorted(random.sample(range(length), k: 2))

        # 交换p1,p2在s~t的部分
        new_p1 = p1[:s] + p2[s:t + 1] + p1[t + 1:]
        new_p2 = p2[:s] + p1[s:t + 1] + p2[t + 1:]
```



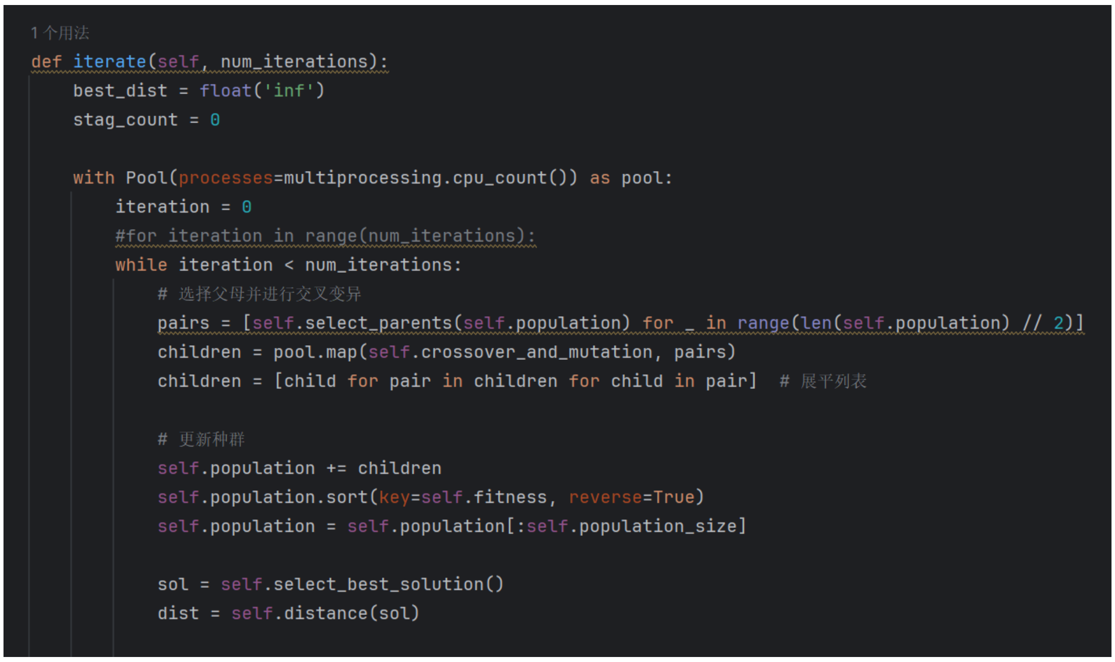
```python
        # 存储映射关系
        p1_p2 = {}
        for i in range(s, t + 1):
            p1_p2[p1[i]] = p2[i]


        new_p1 = mapping(new_p1, p1_p2, s, t)
        # 反转映射关系, 用于第二个子代
        p2_p1 = {v: k for k, v in p1_p2.items()}
        new_p2 = mapping(new_p2, p2_p1, s, t)

        return new_p1, new_p2
```



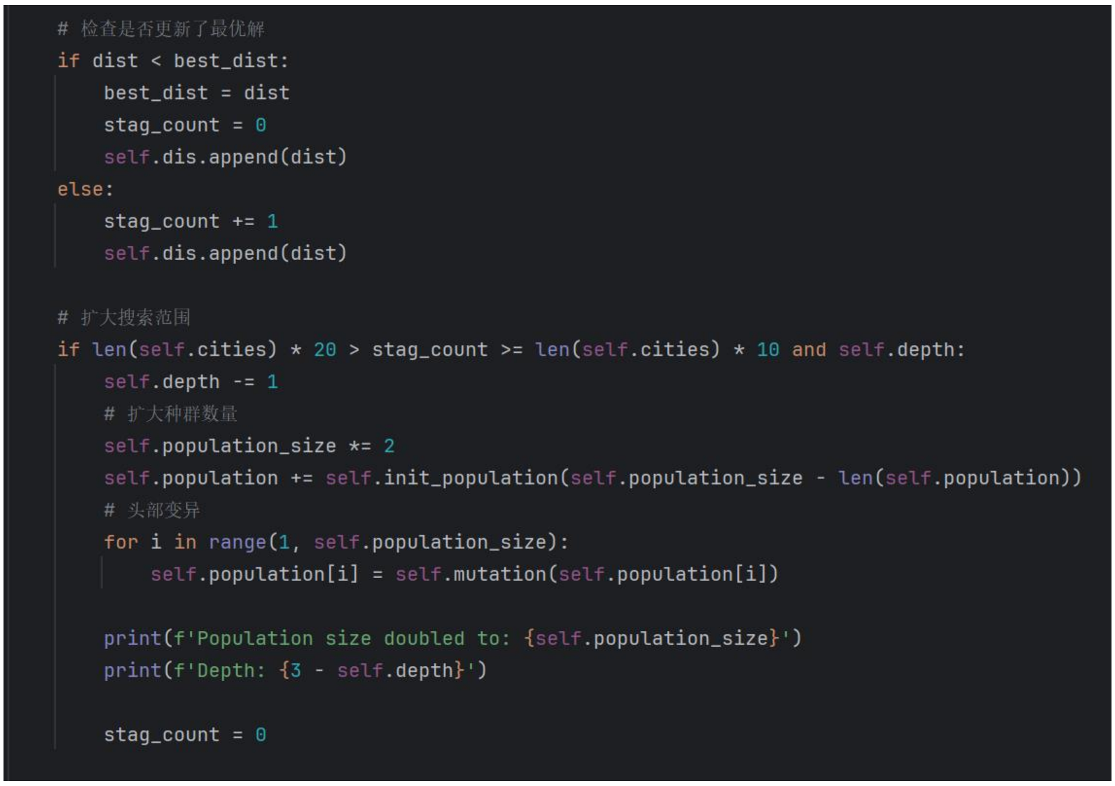
```
5 个用法
def mutation(self, individual):
    length = len(individual)
    # 随机生成两个在0~length-1范围的下标s,t, 确保s<t
    s, t = sorted(random.sample(range(length),  k: 2))
    # 将s,t中间部分倒置
    individual[s:t + 1] = individual[s:t + 1][::-1]
    return individual

1 个用法
def crossover_and_mutation(self, parents):
    # 交叉
    child1, child2 = self.crossover(parents[0].copy(), parents[1].copy())
    # 倒置变异
    child1 = self.mutation(child1)
    child2 = self.mutation(child2)
    # 交换变异
    child1 = self.mutation(child1)
    child2 = self.mutation(child2)

    return child1, child2
```



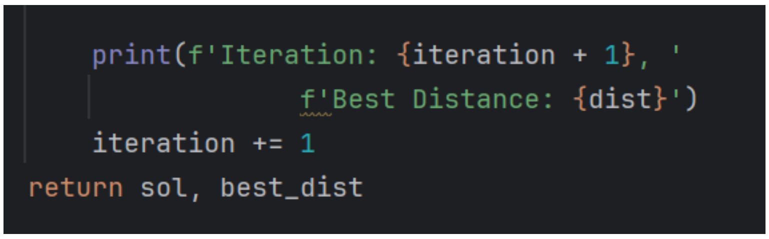
```
1 个用法
def select_parents(self, population):
    # 排序
    # return population[0], population[1]

    # 轮盘赌
    values = []

    for individual in population:
        f = self.fitness(individual)  # 调用计算适应度的函数
        values.append(f)

    total_fitness = sum(values)

    selection_probs = []

    # 计算每个适应度值的概率
    for fitness in values:
        prob = fitness / total_fitness
        selection_probs.append(prob)

    p1, p2 = np.random.choice(len(population),  size: 2, p=selection_probs)
    return population[p1], population[p2]
```



```
1 个用法
def iterate(self, num_iterations):
    best_dist = float('inf')
    stag_count = 0

    with Pool(processes=multiprocessing.cpu_count()) as pool:
        iteration = 0
        #for iteration in range(num_iterations):
        while iteration < num_iterations:
            # 选择父母并进行交叉变异
            pairs = [self.select_parents(self.population) for _ in range(len(self.population) // 2)]
            children = pool.map(self.crossover_and_mutation, pairs)
            children = [child for pair in children for child in pair]  # 展平列表

            # 更新种群
            self.population += children
            self.population.sort(key=self.fitness, reverse=True)
            self.population = self.population[:self.population_size]

            sol = self.select_best_solution()
            dist = self.distance(sol)
```



```
            # 检查是否更新了最优解
            if dist < best_dist:
                best_dist = dist
                stag_count = 0
                self.dis.append(dist)
            else:
                stag_count += 1
                self.dis.append(dist)

            # 扩大搜索范围
            if len(self.cities) * 20 > stag_count >= len(self.cities) * 10 and self.depth:
                self.depth -= 1
                # 扩大种群数量
                self.population_size *= 2
                self.population += self.init_population(self.population_size - len(self.population))
                # 头部变异
                for i in range(1, self.population_size):
                    self.population[i] = self.mutation(self.population[i])

                print(f'Population size doubled to: {self.population_size}')
                print(f'Depth: {3 - self.depth}')

                stag_count = 0
```

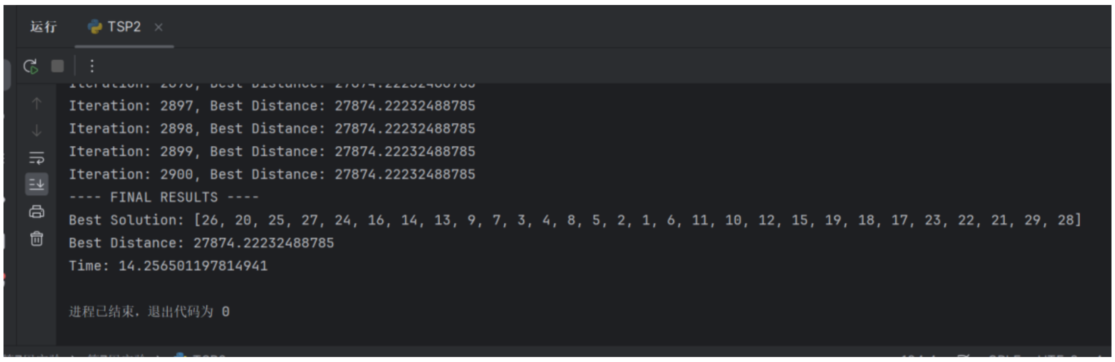


```
            print(f'Iteration: {iteration + 1}, '
                  f'Best Distance: {dist}')
            iteration += 1
    return sol, best_dist
```

## 三、 实验结果及分析

### 1. 实验结果展示示例（可图可表可文字，尽量可视化）

**由于时间问题，我们使用城市最少的 wi29 进行测试，下面为最接近最优解 27601 的结果：**



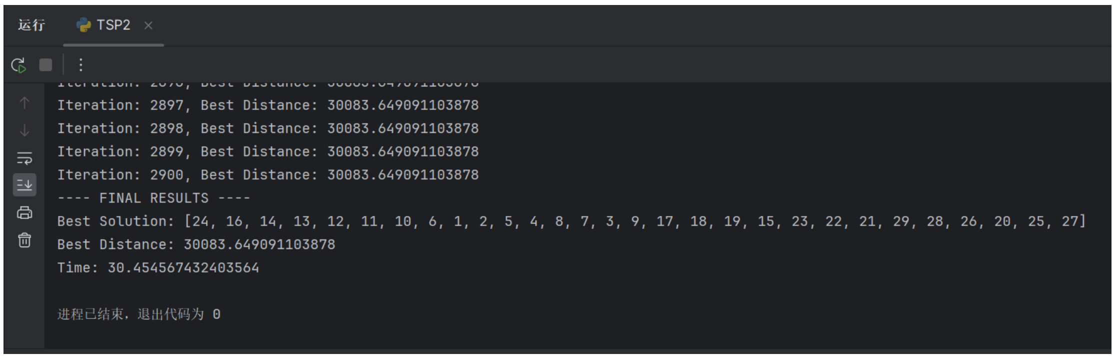
```
Iteration: 2897, Best Distance: 27874.22232488785
Iteration: 2898, Best Distance: 27874.22232488785
Iteration: 2899, Best Distance: 27874.22232488785
Iteration: 2900, Best Distance: 27874.22232488785
---- FINAL RESULTS ----
Best Solution: [26, 20, 25, 27, 24, 16, 14, 13, 9, 7, 3, 4, 8, 5, 2, 1, 6, 11, 10, 12, 15, 19, 18, 17, 23, 22, 21, 29, 28]
Best Distance: 27874.22232488785
Time: 14.256501197814941

进程已结束，退出代码为 0
```

### 2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

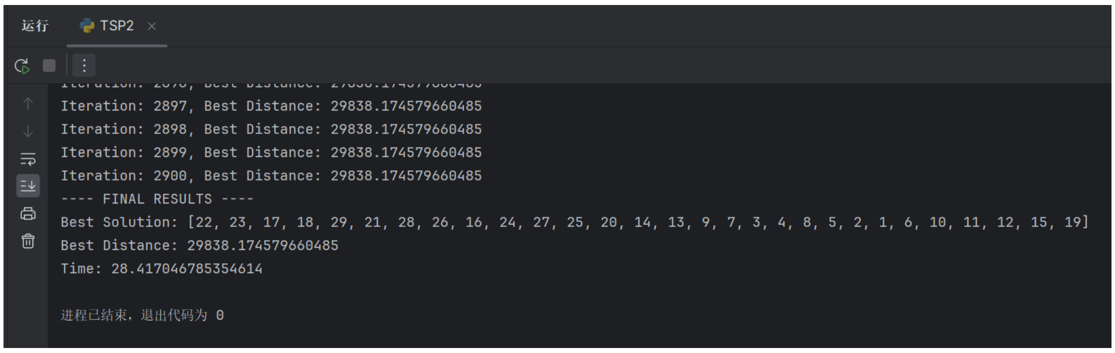
根据实验要求，由于遗传算法是基于随机搜索的算法，只运行一次算法的结果并不能反映算法的性能，所以我们进行多次评测。而对于城市数量较大的数据集，用朴素遗传算法的搜索空间过于巨大，跑完迭代次数时间花费巨大，所以我们只评测数据少的城市。

我们这里使用 wi29 进行评测，为了更好地分析遗传算法的性能，我们修改初始种群大小进行评测（在本次代码中，变异概率默认为 1，我们不作修改）：

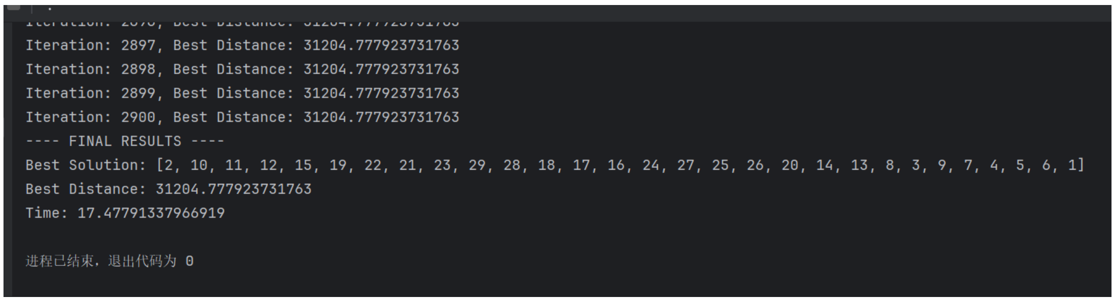
当 population_size = 2 时，评测三次，结果如下：



```
Iteration: 2897, Best Distance: 30083.649091103878
Iteration: 2898, Best Distance: 30083.649091103878
Iteration: 2899, Best Distance: 30083.649091103878
Iteration: 2900, Best Distance: 30083.649091103878
---- FINAL RESULTS ----
Best Solution: [24, 16, 14, 13, 12, 11, 10, 6, 1, 2, 5, 4, 8, 7, 3, 9, 17, 18, 19, 15, 23, 22, 21, 29, 28, 26, 20, 25, 27]
Best Distance: 30083.649091103878
Time: 30.454567432403564

进程已结束，退出代码为 0
```

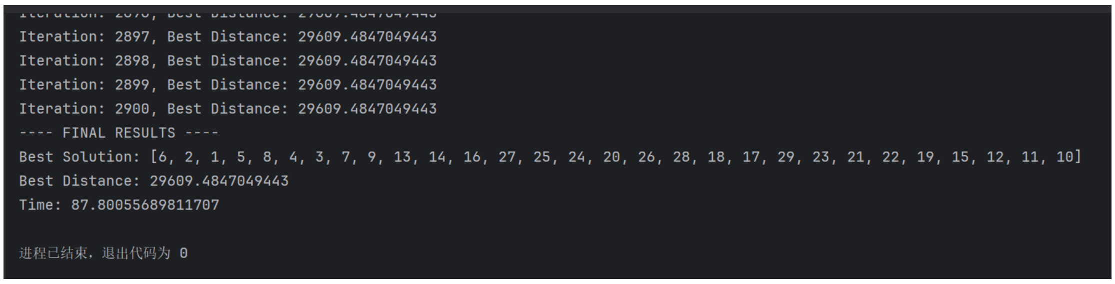


```
Iteration: 2897, Best Distance: 29838.174579660485
Iteration: 2898, Best Distance: 29838.174579660485
Iteration: 2899, Best Distance: 29838.174579660485
Iteration: 2900, Best Distance: 29838.174579660485
---- FINAL RESULTS ----
Best Solution: [22, 23, 17, 18, 29, 21, 28, 26, 16, 24, 27, 25, 20, 14, 13, 9, 7, 3, 4, 8, 5, 2, 1, 6, 10, 11, 12, 15, 19]
Best Distance: 29838.174579660485
Time: 28.417046785354614

进程已结束，退出代码为 0
```

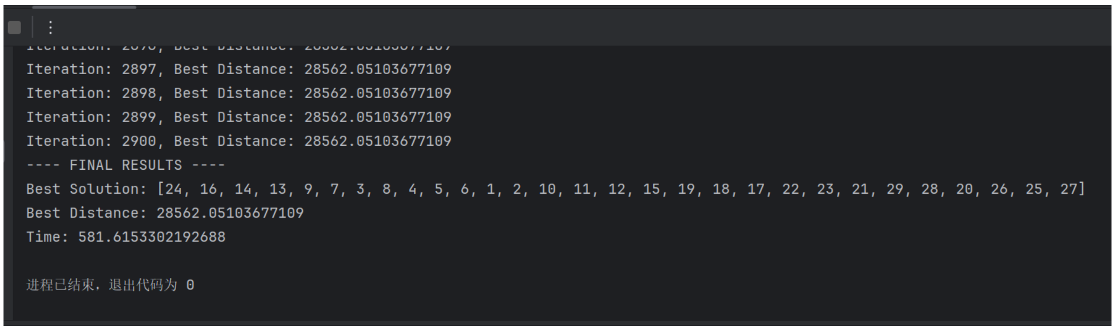


```
Iteration: 2897, Best Distance: 31204.777923731763
Iteration: 2898, Best Distance: 31204.777923731763
Iteration: 2899, Best Distance: 31204.777923731763
Iteration: 2900, Best Distance: 31204.777923731763
---- FINAL RESULTS ----
Best Solution: [2, 10, 11, 12, 15, 19, 22, 21, 23, 29, 28, 18, 17, 16, 24, 27, 25, 26, 20, 14, 13, 8, 3, 9, 7, 4, 5, 6, 1]
Best Distance: 31204.777923731763
Time: 17.47791337966919

进程已结束，退出代码为 0
```

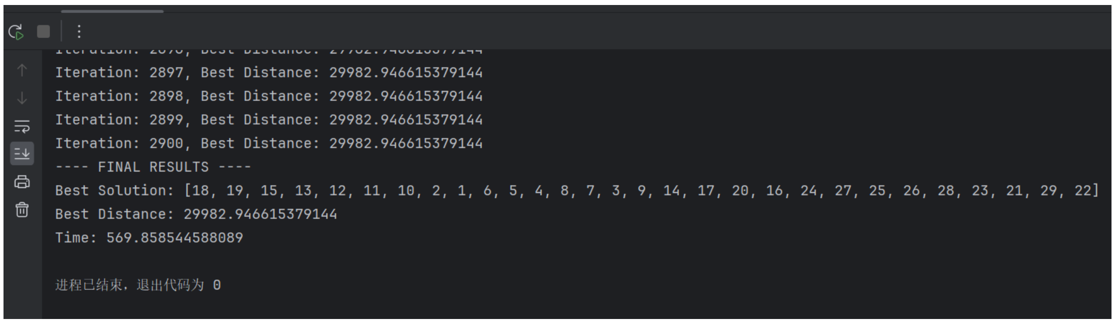
当 population_size = 10 时：



```
Iteration: 2897, Best Distance: 29609.4847049443
Iteration: 2898, Best Distance: 29609.4847049443
Iteration: 2899, Best Distance: 29609.4847049443
Iteration: 2900, Best Distance: 29609.4847049443
---- FINAL RESULTS ----
Best Solution: [6, 2, 1, 5, 8, 4, 3, 7, 9, 13, 14, 16, 27, 25, 24, 20, 26, 28, 18, 17, 29, 23, 21, 22, 19, 15, 12, 11, 10]
Best Distance: 29609.4847049443
Time: 87.80055689811707

进程已结束，退出代码为 0
```

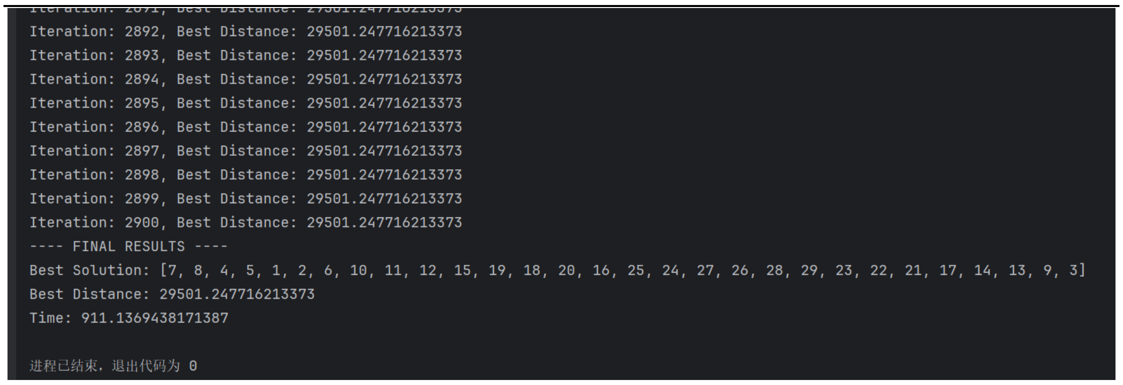


```
Iteration: 2897, Best Distance: 28562.05103677109
Iteration: 2898, Best Distance: 28562.05103677109
Iteration: 2899, Best Distance: 28562.05103677109
Iteration: 2900, Best Distance: 28562.05103677109
---- FINAL RESULTS ----
Best Solution: [24, 16, 14, 13, 9, 7, 3, 8, 4, 5, 6, 1, 2, 10, 11, 12, 15, 19, 18, 17, 22, 23, 21, 29, 28, 20, 26, 25, 27]
Best Distance: 28562.05103677109
Time: 581.6153302192688

进程已结束，退出代码为 0
```

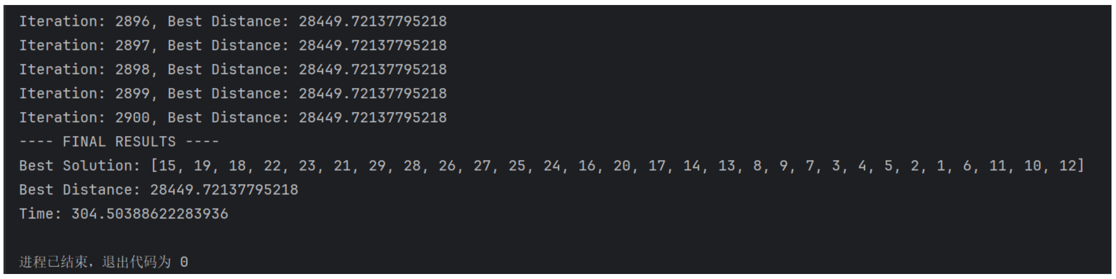


```
Iteration: 2897, Best Distance: 29982.946615379144
Iteration: 2898, Best Distance: 29982.946615379144
Iteration: 2899, Best Distance: 29982.946615379144
Iteration: 2900, Best Distance: 29982.946615379144
---- FINAL RESULTS ----
Best Solution: [18, 19, 15, 13, 12, 11, 10, 2, 1, 6, 5, 4, 8, 7, 3, 9, 14, 17, 20, 16, 24, 27, 25, 26, 28, 23, 21, 29, 22]
Best Distance: 29982.946615379144
Time: 569.858544588089

进程已结束，退出代码为 0
```

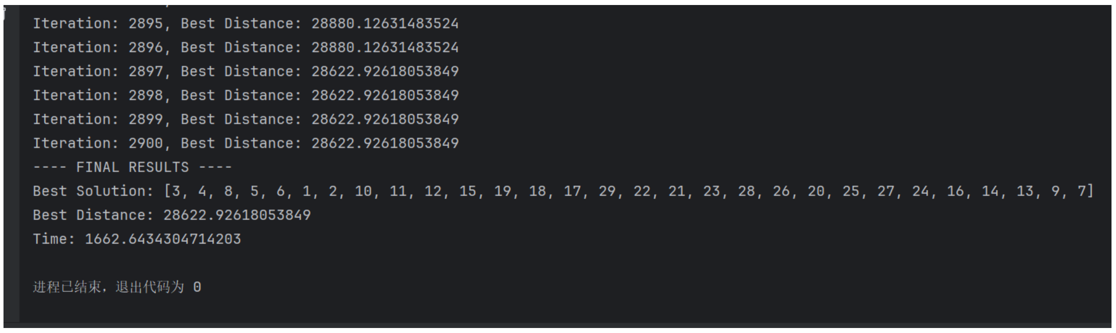
population_size = 20



```
Iteration: 2892, Best Distance: 29501.247716213373
Iteration: 2893, Best Distance: 29501.247716213373
Iteration: 2894, Best Distance: 29501.247716213373
Iteration: 2895, Best Distance: 29501.247716213373
Iteration: 2896, Best Distance: 29501.247716213373
Iteration: 2897, Best Distance: 29501.247716213373
Iteration: 2898, Best Distance: 29501.247716213373
Iteration: 2899, Best Distance: 29501.247716213373
Iteration: 2900, Best Distance: 29501.247716213373
---- FINAL RESULTS ----
Best Solution: [7, 8, 4, 5, 1, 2, 6, 10, 11, 12, 15, 19, 18, 20, 16, 25, 24, 27, 26, 28, 29, 23, 22, 21, 17, 14, 13, 9, 3]
Best Distance: 29501.247716213373
Time: 911.1369438171387

进程已结束，退出代码为 0
```



```
Iteration: 2896, Best Distance: 28449.72137795218
Iteration: 2897, Best Distance: 28449.72137795218
Iteration: 2898, Best Distance: 28449.72137795218
Iteration: 2899, Best Distance: 28449.72137795218
Iteration: 2900, Best Distance: 28449.72137795218
---- FINAL RESULTS ----
Best Solution: [15, 19, 18, 22, 23, 21, 29, 28, 26, 27, 25, 24, 16, 20, 17, 14, 13, 8, 9, 7, 3, 4, 5, 2, 1, 6, 11, 10, 12]
Best Distance: 28449.72137795218
Time: 304.50388622283936

进程已结束，退出代码为 0
```



```
Iteration: 2895, Best Distance: 28880.12631483524
Iteration: 2896, Best Distance: 28880.12631483524
Iteration: 2897, Best Distance: 28622.92618053849
Iteration: 2898, Best Distance: 28622.92618053849
Iteration: 2899, Best Distance: 28622.92618053849
Iteration: 2900, Best Distance: 28622.92618053849
---- FINAL RESULTS ----
Best Solution: [3, 4, 8, 5, 6, 1, 2, 10, 11, 12, 15, 19, 18, 17, 29, 22, 21, 23, 28, 26, 20, 25, 27, 24, 16, 14, 13, 9, 7]
Best Distance: 28622.92618053849
Time: 1662.6434304714203

进程已结束，退出代码为 0
```

由上面可以看到，通过修改种群大小，评测指标有变化，当种群越大，算法平均消耗时间越长，但平均结果更加接近最优解。

总而言之，遗传算法随机性较大，一旦搜索空间大到一定程度，算法收敛会变得异常缓慢，在上述的测评中同样，哪怕城市只有 29 个，刚开始速度很快，但迭代到 2000 轮以后，速度依然变得很慢，这也是属于遗传算法自身的局限性。

## 四、 参考资料

遗传算法的基本原理——CSDN