TSP 问题 GA

1.实验环境：

- OS: Windows 10
- CPU: intel CORE i5
- Python 3.6
- Sckikit_learn 0.18.1
- 其他 Python 科学计算模块
- 编译器：PythonCharm 2017.3.3

2.主要参考文档：

- Sklearn 0.18 官方文档

3.实验过程

3.1.实验问题描述

TSP 问题（Traveling Salesman Problem），是数学领域著名的问题之一，设有一个旅行商人要拜访 n 个城市，他必须选择要走的路径，路径的限制是每一个城市只能拜访 1 次，而且最后要回到原来的城市。路径的选择目标是要求得到的路径为所有路径之中的最小值。

TSP 问题是一个组合优化问题。该问题被证明具有 NPC 计算复杂度。迄今为止，还没有一个有效的算法，所以大家认为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。

本实验使用 GA 算法（genetic algorithm）,即遗传算法来解决 TSP 问题。

3.2.实验资源库：

本实验在测试过程中，使用到了一些数据来自于 TSPLIB 网站，但是这个网站的很多数据来自于实际的生活中，所以坐标点集合有些过于排列整齐，我自己对数据进行选择及修改。

TSPLIB 网站网址：

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

3.3 实验原理与过程：

本实验参考了网上的一些博文，在充分理解的基础上做出了自己的修改和一些必要的补充。首先简短阐述一下 GA 算法的原理：

遗传算法是在 20 世纪六七十年代由美国密歇根大学安娜堡分校的约翰.霍兰德教授及其研究团队提出并发展起来的。遗传算法的主要特点是通过染色体的模拟和模拟染色体上的进化操作，搜索问题的最优解。因此，解被编码为生物个体中的染色体，问题的目标函数对应生物个体的适应度函数。在此基础上，通过在

染色体上进行的生物种群的多代进化，获得最适应环境的染色体作为最优解。

遗传算法最重要的构成要素有：

（1）解的编码方式

解的编码方式有较多种形式，对于问题的表示与编码方式对后续的操作会有较大的影响。编码用于将问题解的原始数据结构变化为用串位形式编码的染色体；译码用于将问题解的染色体表示形式变换为问题的原始数据结构。

基本的遗传算法采用的是二进制的编码方式，但是在本实验中，我采用了直接的十进制数字表示每一个城市节点。

（2）初始群体和群体规模的确定方法。在初始群体上，一般采用随机方法构造，也可以用经验方法构造，以减少进化代数。但是采用经验构造，使得算法陷入局部最优解的概率增大。在一些实际问题中，也可以使得群体规模随着遗传算法代数发生变化，以获得更好的优化效果。

在本实验中，我们经过多次调解，最后选择的规模是 110 个基因。

（3）适应度函数。适应度函数是对生物个体适应性进行度量的函数。通过适应度函数来确定染色体所代表的解的优劣程度，体现了对求解目标的要求。对于优化问题，通常可以直接取目标函数作为适应度函数。

（4）遗传操作。解的搜索通过染色体上的遗传操作来实现。基本的遗传算法的遗传操作包括选择，交叉和突变。

遗传操作的执行顺序是：先选择父代染色体，然后在选出的父代染色体中随机的交叉获得子代染色体。

下面结合具体代码，说明实验过程。

首先是对单个基因中各参数的初始化，这个初始化函数是类 Gene 中的：

```
def __init__(self, Distance, alpha, Path=None):
    '''''
    初始化一个基因的参数，生成一个基因的初始路径，即其代表的解
    :param Distance:
    :param alpha: 一次交换次数
    :param Path:
    '''
    self.Distance = Distance
    self.num_cities = len(Distance)
    self.alpha = alpha
    # 随机生成初始的路径
    if Path is None:
        self.Path = list(range(self.num_cities))
        np.random.shuffle(self.Path)
    else:
        self.Path = Path.copy()

    self.Length = self._Length(self.Path)
    self.score = self._Score(self.Length)
```

需要对初始的各城市距离，城市节点数目等初始化。

接下来还需要对另外一个类进行初始化，是遗传算法的类 GA：

```
def __init__(self, cities, num_genes=90, num_children=90):
    '''''
    初始化 GA 算法的参数
    :param cities: 传入城市节点
    :param num_genes: 父代个数
    :param num_children: 子代个数
    '''

    self.cities = cities
    self.num_cities = len(cities)
    self.Distance = np.empty((self.num_cities, self.num_cities))
    for i in range(self.num_cities):
        for j in range(self.num_cities):
            V = (cities[i][0] - cities[j][0],
                     cities[i][1] - cities[j][1])
            self.Distance[i, j] = np.linalg.norm(V)

    self.num_genes = num_genes
    self.num_children = num_children
    #每一次迭代的交换次数
    self.alpha = 50
    self.recombination_prob = 0.9

    #默认每次都进行突变
    self.mutation_prob = 1

    # 生成了父代的基因序列集合
    self.genes = [Gene(self.Distance, self.alpha) for i in range(self.num_ge
nes)]
```

在这个初始化的函数中，生成初始时刻的解集合，由于 Path 是空的，所以，初始时刻的所有的解都是随机生成的，规模为 110 个。

接下来是适应度函数：

```
def _Score(self, Length):
    '''''
    评估函数，根据路径的总长度进行评估
    :param Length:
    :return:
    '''
    return 1 / np.power(Length, self.alpha)

```

适应度函数思想较为简单，因为 TSP 问题对于结果的评价是能否找到一个解，使得其路径的总长度最小。所以，适应度函数取其路径总长度作为一个参数来进行计算。

接下来是 GA 算法在这个实验中的主要遗传算子的代码分析，这是遗传算法的关键部分。

选择和组合。

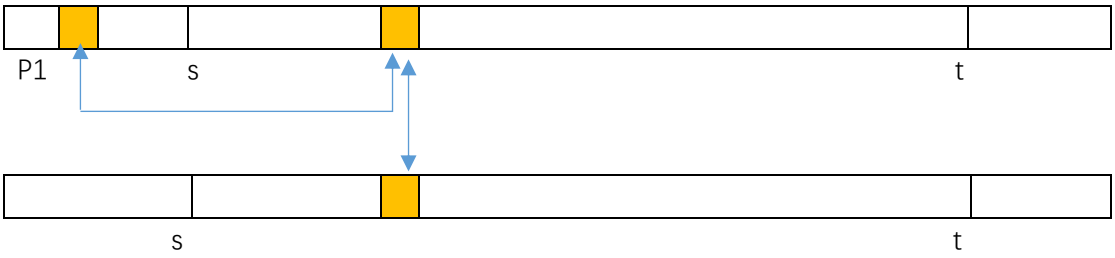
```
def _Combine(self, Path1, Path2):
    '''''
    基因组合函数
    :param Path1:  第一个体对应的路径
    :param Path2:  第二个个体对应的路径
    :return:
    '''
    Path1 = Path1.copy()
    Path2 = Path2.copy()

    if np.random.random() < self.recombination_prob:
        s, t = np.random.choice(range(1, self.num_cities - 1),
                                size=2, replace=False)
        s, t = min(s, t), max(s, t)
        Path1[s:t + 1], Path2[s:t + 1] = Path2[s:t + 1], Path1[s:t + 1]
        self._turePath(Path1, Path2, s, t)
        self._turePath(Path2, Path1, s, t)
    return Path1, Path2

```

选择个体进行组合，交叉遗传得到子代个体。这里选择父代个体使用的是轮盘赌算法。由每个基因的适应度函数，去对应相应的基因个体的选择的概率。

```
def _Select(self, genes, number):
        '''''
        轮盘赌算法选择出相应的父代基因
        :param genes: 传入基因
        :param number: 传入需要得到的子代个体
        :return:
        '''
        weights = [x.score for x in genes]
        sum_weight = sum(weights)
        weights = [x / sum_weight for x in weights]
        samples = np.random.choice(genes, number, p=weights, replace=False)
        return list(samples)
```

在进行组合的时候还有个重要的问题需要解决：由于问题的解是使用十进制

表示的，这就意味着在交换之后肯定会存在在新的子代基因型中存在基因重复的问题。



P2

如上图所示，为了解决这个问题，我们可以在 p1 子代染色体中的 st 之间的交换段和两端的未交换段找到重复的数字，重复了就表明交换过来的染色体段是不合理的，那么就要找到一个合理的数字代替当前的数字，合理的数字得从原来交换到 p2 的那一段来找到，只需要将对应位置的数字交换过来，就可以了，这就是 truePath 函数所实现的过程。

```
def _turePath(self, Path1, Path2, s, t):
    '''''
    对于基因重组之后的个体，需要进行调整，避免出现重复的进行
    :param Path1: 第一个体对应的路径
    :param Path2: 第二个个体对应的路径
    :param s:   开始交换节点
    :param t:   末尾交换节点
    :return:
    '''
    while len(set(Path1)) < self.num_cities:
        for i in list(range(0, s)) + list(range(t + 1, self.num_cities)):
            if Path1.count(Path1[i]) > 1:
                p1 = i
                break
        p2 = Path1.index(Path1[p1], s, t + 1)
        Path1[p1] = Path2[p2]
```

在得到了子代个体之后，还需要对个体以一定的概率进行突变，这个是较为容易实现的。

```
def _Mutate(self, Path):
    '''''
    突变函数
    :param Path:需要突变的基因个体
    :return:
    '''
    if np.random.random() < self.mutation_prob:
```

```
8.          p1, p2 = np.random.choice(range(self.num_cities), 2, replace=False)

9.          Path[p1], Path[p2] = Path[p2], Path[p1]
```

最后需要进行说明的是进化计算的整个过程：

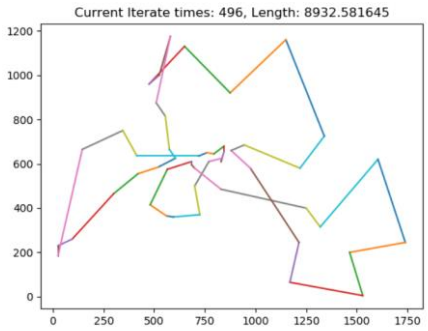
```
1.  def Iter_once(self):
2.          '''''
3.          进化一次的过程
4.          :return:
5.          '''
6.          children = []
7.          for i in range(self.num_children // 2):
8.              parents = self._Select(self.genes, 10)
9.              parents.sort(key=lambda x: x.Length)
10.             #交叉组合
11.             Path1, Path2 = self._Combine(parents[0].Path, parents[1].Path)
12.             self._Mutate(Path1)
13.             self._Mutate(Path2)
14.             children.append(Gene(self.Distance, self.alpha, Path1))
15.             children.append(Gene(self.Distance, self.alpha, Path2))
16.         self.genes += children
17.         self.genes = self._Select(self.genes, self.num_genes)
18.         # 得到此次迭代的最好的基因
19.         bestgene = max(self.genes, key=lambda x: x.Length)
20.         return bestgene.Length, bestgene.Path
```

上边这个函数表示进化一次的过程，就是将前边几个遗传的主要算子进行了整合，并且不断得迭代得到下一代个体，最后输入最大的迭代的次数，进行整个的迭代。

3.4 实验结果与分析

这个实验，我设置为迭代 500 次，子代与父代的个体数目都设置为 110。得到的结果如图所示

```
Iterate times: 496, Length: 8932.581644681275, Path: [3, 12, 2, 24, 25, 10, 22, 17, 23, 6, 7, 5, 33, 19, 36, 45, 4, 42, 21, 38, 1, 44, 9, 16, 15, 14, 34, 49, 48, 29, 4
```



从上图可以看出来，最后的结果其实不是特别理想，在实验过程中，我也多

次调整了各个参数，但是最后的结果都不是特别完美。这和初始的数据有关。但是数据的迭代的优化效果是十分明显的，这一点毋庸置疑。

从这个实验总结以下几点：

遗传算法是从问题解的串集开始搜索的，而不是从单个解开始的。传统的优化算法都是从单个解开始入手的，所以容易陷入局部最优，但是这里不会，GA 算法大大减小了陷入局部最优的概率，，同时算法本身易于实现并行化。另外一点是适应度函数不受连续可微的约束，在定义域内可以任意设定，这使得遗传算法的应用范围大大拓展。

遗传算法具有自组织和自学习性。遗传算法利用了进化过程获取的信息自行组织搜索时，适应度大的个体具有较高的生存概率，并获得更加适应环境的基因结构。

GA 算法存在的一些问题：比如如何选择问题的编码方式将对算法的效率等产生较大的影响，并且必须要承认的是，遗传算法的效率通常较其他优化算法的效率更低，并且容易过早收敛，还有就是在算法的精度，可行度以及计算复杂性等方面，还没有有效的定量分析方法。