编号：_____


# 高级数据结构与算法设计

## 作业 3：TSP 问题

学　　院：计算机与通信技术学院

学　　号：G20188751

姓　　名：潘秋实

指 导 老 师：皇甫伟

得　　分：_____

2019 年 03 月 30 日

# 目录

# 摘要

学习旅行商问题是算法研究必经的一步，所以本文即通过遗传算法研究旅行商问题。本文在遗传算法求得近似解的基础上，使用 2 交换的方法求得全局最优解，并将整个过程通过 OpenCV 展示出来。为了调整遗传算法的参数，本文由进行了多次试验，最终得出结论：在一定范围内停止进化的代数设置越小，算法整体速度越快。

# 1 引言

旅行商问题（英语：Travelling salesman problem, TSP）是一个组合优化问题，其经典的描述为：已知一定数量的城市，任意两座城市之间互通并已知路径长度，求解访问每座城市一次并回到原点的访问顺序。

求解 TSP 问题的算法有很多，早期的研究大多采用精确算法，例如线性规划、动态规划等，这些算法可以求得精确解，但是面对问题规模的增大，却显得无能为力；现代的研究中，主要有贪婪算法、模拟退火法、遗传算法和蚁群算法等，这些算法各有优势，都属于近似算法，可以求得一个局部最优解（近似解），而不一定是全局最优解。

本文为了研究遗传算法求解 TSP 问题，主要进行了以下工作：

1）本文的实验中使用 Python 语言，实现遗传算法求 TSP 问题的近似解，又通过二交换算法逐步优化，求全局最优解。

2）探索出遗传算法中交换点的顺序、而交换算法中交换边的顺序的算法思路。

3）算法整体运行过程和测试结果可视化，收集后续实验的数据。

4）尝试使用贪婪算法和随机算法为遗传算法初始化，得出贪婪算法初始化不需要经过遗传算法优化的结论。

5）在不同长度的随机数组上测试遗传算法加二交换优化算法的整体运行时间，求平均值得到算法表现性能。

6）控制其他变量，针对遗传算法中的停止进化代数，分两次设置多组实验，最终得出在一定范围内越早停止进化，执行速度越快。

# 2 相关理论

## 2.1 遗传算法

遗传算法是一种启发式算法，模仿了达尔文生物进化论，网上有比较多的解释，本节就结合我自己的理解简单谈一下本文中使用的遗传算法。

模拟算法运行前，要设置变异概率 M、交配概率 C 和每代生存的数量 N。对于本文研究的 TSP 算法来说，程序运行的开始，会随机生成 GN 数量的路线，然后每条路线会直接遗传，也就是放入下一代路线的候选集 NG 中；每条路线会按照 M 的概率，随机产生或不产生一条变异路线，如果产生的话就放入 NG 中；每两条路线会根据 C 的概率，随机交配与否，如果交配的话就将交配产生的两条新路线放入 NG 中。

收集完 NG，可以确定 NG 中最少有 N 条路线（所有的路线都不变异也不交配），最多有 2N+N(N-1)/2 条路线（所有的路线都变异并两两交配）。而对于 TSP 算法来说，目标是找最短路径，所以“自然选择”阶段就从 NG 中收集路线最短的 N 条路线作为下一代路线，其余的全部删除掉。而对于下一代而言，就由这新挑选的 N 条路线遗传、变异、交配和自然选择。

每一代最短的路线和其距离都要被记录下来。当连续几代都没有产生比原来最短距离还要短的路线的时候，就说明已经求得了局部最优解，从而输出前边求到的路线。

具体的实现过程，本文的第三章会做详细介绍。

## 2.2 贪婪算法

在本文中，一方面贪婪算法是作为遗传算法的对照，另一方面也尝试过取代随机生成路线，使用贪婪算法为遗传算法初始化。所以本节中简单介绍本文中贪婪算法的理论。

本文中的贪婪算法，首先求出图中所有城市之间的距离，用矩阵的形式存储下来。从第 1 个点，找离它距离最短的点，也就是第 1 行中数值最小的元素的位置；假设找到第一行的第 i 个元素最小，则继续从第 i 行找数值最小的元素。加入找到的最小的元素所对应的城市已经加入到路线中，则继续找第二小、第三小的元素，一直按照总城市数求出一条路线为止。

实际上，在城市数量多的情况下，贪婪算法求近似解的速度是比遗传算法快几个数量级的，而且求解效果也好一些。不过本文着重研究遗传算法，所以贪婪算法不再过多介绍。

# 3 TSP 求解算法

本文求解 TSP 算法分为 2 个阶段：第一阶段使用遗传算法求得近似解，第二阶段使用二交换的方法寻找最优解。在这两个阶段的实现过程中，我都参考过网上的代码，又加入了一些优化的方法，所以本章就分两个部分讲解。

## 3.1 遗传算法原理

遗传算法的整体思路是模仿生物进化，核心就是遗传、变异、交配和自然选择，连续几代没有更优解，就停止计算，输出当前最优解，所以本节先讲

解遗传算法的这几个重要的操作。

遗传（Preserve）比较简单，就是直接将上一代的路径放入下一代的候选路径集合NG中,等待自然选择筛选。而一同被加入候选路径集NG的还有通过变异（Mutation）、交配（CrossOver）产生的新路径，接下来讲解的第一个就是变异操作，如算法1所示。

**算法 1** Mutation算法

**输入：**待变异的路径path
**输出：**变异后的路径newpath

```
1:  times = random (1, N - 2)
2:  for i in range(times):
3:     exchange 2 random Nodes in path
4:  newpath = path
5:  append newpath into NG
```

从算法 1 可以看出，每次变异的过程，都是有随机次数的交换操作；而每次发生交换的两个城市，也是随机的。所以变异过后，改变的是城市顺序，每座被交换的城市的上一个城市和下一个城市都被改变了。值得注意的是，在实现的过程中，我还加入了其他代码，保证每个位置的城市只会被改变一次。这条变异产生的路径，也会被加入到候选路径集NG中。

其次是交配操作，顾名思义，就是两条路线互相将自己的一部分顺序交换给对方，故而会产生两条新的路径，如算法2所示。

**算法 2** CrossOver算法

**输入：**待交配的路径path1、path2
**输出：**交配后的路径newpath1、newpath2

```
1:  times = random (1, N - 2)
2:  newpath1 = path1
3:  newpath2 = path2
4:  for i in range(times):
5:     index = random (1, N - 2)
6:     exchange newpath1[index] and newpath2[index]
7:     append newpath1, newpath2 into NG
```

通过算法2，两条原有路径的随机数量、随机位置的城市被交换了，同样会在实际程序中保证每个位置的城市只被交换一次。两条新产生的路径也会被加入到路径候选集 NG 中，至此，路径候选集NG一共收集了3种来源的候选路径，从而需要通过自然选择算法筛选出与上一代保持相同数量的下一代路径。

自然选择算法的目标非常明确：选最短的 N 条路径。所以自然选择算法如算法 3 所示，针对每条路径计算总距离，然后按照距离从小到大排序，选取前N条即是下一代的路径。

**算法 3** Select算法

**输入：**路径候选集NG
**输出：**下一代路径集Paths

```
1:  for path in NG:
2:     calculate its diatance
3:  sort NG by path’ distance
4:  Paths = NG[:N]
5:  return Paths
```

所以，遗传算法总体的框架如算法 4 所示。对于每一代来说，经过遗传、变异和交配产生下一代的候选集，通过自然选择筛选出下一代的路径。如果下一代找出了更优解，就继续遗传算法；如果连续几代找不到更优解，就认为当前求得的路径就是最优解，

并输出这个条路径。

**算法 4** Genetic 算法

**输入：**城市坐标及距离,变异概率 M,交配概率 C,最大停止进化代数 Maxgen,每代路径数 N。

**输出：**最优路径

```
randomly generate Paths
current = shortest distance in Paths
while(stopgen < Maxgen):
  for i in range(N):
    if random < M:
      Mutation (Paths[i])
    for j in range(i, N)
      if random < C:
        CrossOver(Paths[i], Paths[j])
  Paths = Select(NG)
  shortest = shortest path distance in Paths
  if shortest < current
    current = shortest
  else stopgen += 1
return Paths[0]
```

## 3.2 二交换算法原理

二交换算法实际上是以蛮力的方式，寻找比当前路径更短的路径。二交换算法本身并没有什么可探究的，无非是穷举并计算判断。而本节中讨论的，是我关于二交换算法的一些改进。

之前在遗传算法中，所有的交换都是位置上的直接交换，我一开始尝试二交换算法的时候，也是直接交换点与点的位置。但是发现行不通，总是会出现路线上的交叉。我认为不应该交换点的顺序，而应该是交换边的顺序。所以我的实现方法是，如果每次选定两个点作交换，同时要将它们之间的点的顺序换成逆序,如算法 5 所示。

**算法 5** Exchange 算法

**输入：**待处理的数组 Array，交换位置 i、j

**输出：**处理后的数组 Array

```
a, b = i, j
while(a <= b):
    Array[a], Array[b] = Array[b], Array[a]
while Array
```

经过这种改进，我的实验中每次求出的最短路线都不存在交叉了。

而关于二交换的时间复杂度，确实是一个令人头疼的问题，从理论上讲大概接近$O(n^3)$。原因比较简单：二交换算法查找的时候，二层嵌套循环设置了两个指针 i 和 j,每个指针都是从 0 到 N,每查找一次需要执行$n^2$次交换，而交换完计算路径长度的时候，又要做 n 次加法，所以在数组长度达到 100 以上的时候，二交换算法所花费的时间是特别多的。

所以我想了两步的优化策略。第一步，在查找二交换路径的时候，从 i=0，j=1 和 i=N-1,j=N 这两种情况交替开始查找，即一次从前往后找，找到更优解即退出，下一次再从后往前找，这样遍历一次所有解的时间就变成了原来的一半，不过查找过程的时间复杂度依然是$O(n^2)$。

第二步是在计算路线长度的时候，原本是应该执行 n 次加法，而我认为可以用近邻准则，在原路径长度的基础上，减去被替换掉的两条路径的长

度，加入两条新的路径的长度，从而使用 4 次加减法即可完成路径长度计算，计算路径长度的时间复杂度由$O(n)$变为$O(1)$。

也就是说，整个二交换优化部分的时间复杂度由$O(n^3)$被优化到$O(n^2)$，而实际执行时间是原来的$\frac{1}{2n}$，大大加快了算法的效率。二交换部分的算法如算法 6 所示。

**算法 6** 2-Swap 算法

**输入：**路径 Path
**输出：**优化后的 Path

```
PathVal=total distance of the Path
state = 0
Finded = 0
while(Finded == 0):
    CopyPath = Path
    for i in range(0, N-2)
      for j in range(i, N-1)
        if state == 0:
          left =i
           right= j
         else:
          left=N-2-i
          right=N-2-j
         use Algorithm 2 to change CopyPath[left: right]
         NewVal =total distance of the Path
        if NewVal< PathVal:
          Path = CopyPath
          state = not state
          countinue
  Finded = 1
 return Path
```

以上就是二交换算法的全部优化思路，是在做本文实验的过程中，逐步探索所得。

# 4 实验

本文的实验比较多，从基础算法的实现，到算法性能的测试，再到算法参数的调试，最后是尝试贪婪算法优化。这些实验不仅在个人 PC 上运行，还动用了服务器持续跑，整个过程持续了一个多星期。本章所有的实验，城市的横纵坐标都是在 0 到 500 之间随机取的整数，实验结果仅供参考，在不同的取值范围下，结果的距离数值会略有差异，但是依然会求出最优路径。在本章中就会逐一讲解本文的所有实验。

## 4.1 TSP 求解算法演示

本文实验的第一步，是使用遗传算法求得 TSP 问题的近似解并使用二交换算法优化。所以我在开始探索的阶段，编写了 demo_ga_swap.py 脚本，专门用来演示这个过程。

程序运行开始，会自动生成指定数量、随机坐标的城市。每当路线发生变动，程序会通过国 OpenCV 生成一张图，绘制出城市与路线，标明当前正在执行的算法和总距离。两阶段算法对于不同数量的点的处理，如图 4-1 所示：首先使用遗传算法求近似解，并保存近似解的结果图，再由二交换算法优化。

从图 4-1 中可以看出，在点数较少的情况下，非常容易用遗传算法求

出接近最优解的近似解。而随着点数增多，遗传算法求出的近似解，会存在大量的路径交叉。这主要是因为遗传算法在变异和交配的时候，往往更改的是点的顺序，即便是交换两个点，新路线与原路线也有多达四条路线改变，而一般的路径交叉，往往只需要更改两条边，所以遗传算法很难解决这种细微的改动，需要使用二交换算法优化它的近似解。

在算法执行结束后，可以查看output文件夹中的TSP.avi视频文件，这里记录了本次求解的完整过程。

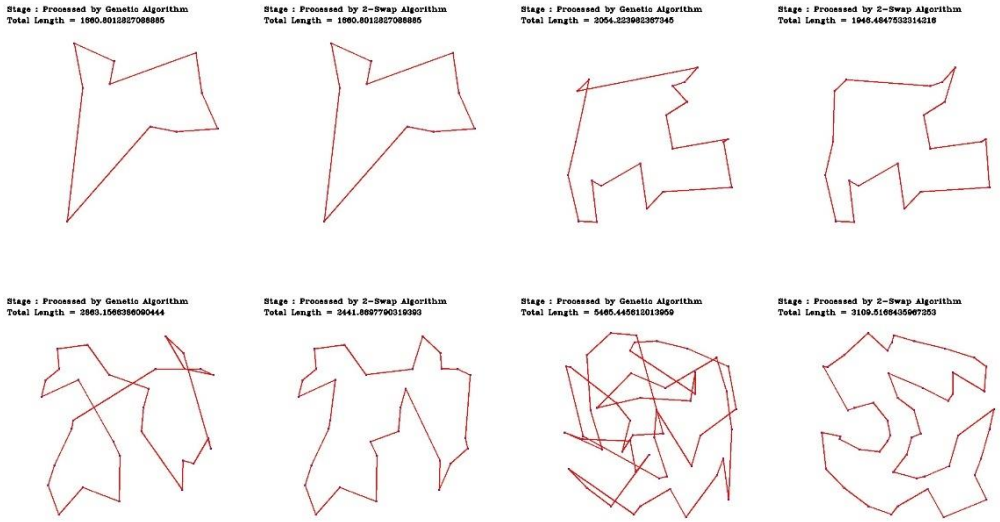



**图 4-1 不同数量城市的遗传算法近似解和二交换优化结果**

## 4.2 算法性能测试

为了测试本文中二阶段 TSP 求解算法的整体性能，我在阿里云服务器上搭建了测试环境。测试环境主要配置参数为Python 3.6、主频为2.5GHz的 Xeon 8163 处理器和 Windows Server 2008 系统。而关于算法的思想，本文第 3.1 节中已经有相关介绍，其中参数配置方面：变异概率设置成0.9，交配概率设置成0.9，每代路线数设置成 40 条，最长停止进化代数为 5 代。因为遗传算法是启发式算法，所以要靠概率求最优解，前三个参数设置的越大，算法的求解结果越好，执行效率越慢，是显而易见的。最后一个停止进化代数设置成5代的原因，会在第4.3中讲解，本文是通过实验得出 5 代比较好的结论。

本文使用demo_test.py脚本，在10到230个点(城市)之间做了测试，每种数量做了50次测试，每次都是随机生成的数据，最终取各个结果的平均值,测试结果如表4-1所示。

首先是观察两阶段算法的求解总距离，数据趋势如图4-2所示，数值结果可以在表 4-1 中查询。图 4-2 中，绿色的线是仅仅经过遗传算法求解后的平均距离，蓝色的线是经过遗传算

法和二交换优化后的平均距离。

**表 4-1 遗传算法与二交换算法在不同数量城市上的测试结果**

| Citie | Average Time(s) | Genetic Distance | Genetic Cost Time | 2-Swap Distance | 2-Swap Cost Time |
|---|---|---|---|---|---|
| 10 | 0.82 | 1401.59 | 0.82 | 1401.59 | 0.00 |
| 20 | 2.94 | 2147.95 | 2.94 | 1955.59 | 0.01 |
| 30 | 5.52 | 3093.09 | 5.48 | 2405.46 | 0.03 |
| 40 | 9.14 | 4446.41 | 9.01 | 2760.34 | 0.13 |
| 50 | 12.34 | 5618.30 | 12.00 | 2978.01 | 0.34 |
| 60 | 16.22 | 6972.90 | 15.52 | 3277.18 | 0.70 |
| 70 | 20.53 | 8371.51 | 18.86 | 3520.19 | 1.67 |
| 80 | 24.28 | 10360.17 | 21.34 | 3680.65 | 2.94 |
| 90 | 31.25 | 11564.18 | 26.60 | 3895.27 | 4.65 |
| 100 | 35.22 | 13472.58 | 27.87 | 4132.30 | 7.35 |
| 110 | 44.50 | 15014.31 | 32.69 | 4356.93 | 11.81 |
| 120 | 51.83 | 16735.21 | 36.12 | 4490.20 | 15.72 |
| 130 | 64.42 | 18954.44 | 39.59 | 4638.21 | 24.83 |
| 140 | 76.58 | 20808.22 | 45.46 | 4879.98 | 31.12 |
| 150 | 93.36 | 22238.07 | 50.69 | 5054.01 | 42.67 |
| 160 | 114.99 | 23703.50 | 57.04 | 5141.70 | 57.95 |
| 170 | 129.38 | 25188.30 | 58.38 | 5302.34 | 71.00 |
| 180 | 155.10 | 27371.27 | 63.05 | 5455.33 | 92.05 |
| 190 | 187.94 | 28756.05 | 70.41 | 5656.92 | 117.53 |
| 200 | 211.77 | 30265.36 | 75.33 | 5738.10 | 136.44 |
| 210 | 244.27 | 32937.27 | 75.31 | 5917.88 | 168.97 |
| 220 | 305.24 | 33972.48 | 84.69 | 6000.29 | 220.55 |
| 230 | 336.23 | 35786.87 | 90.13 | 6095.31 | 246.09 |

从理论上来说，使用遗传算法求出的近似解，距离是在随点数线性增长的；二阶段求出的最优解，不是线性增长的原因，是点的分布空间有限，在十分松散的情况和十分密集的情况下，增加相同的点数，距离增长肯定是不一样的。图 4-2 可以反映出这两阶段算法的求解距离趋势，在点数比较多

的时候，二交换的优化效果是非常明显的，这可能是因为二交换可以顺利处理路径交叉的情况。。

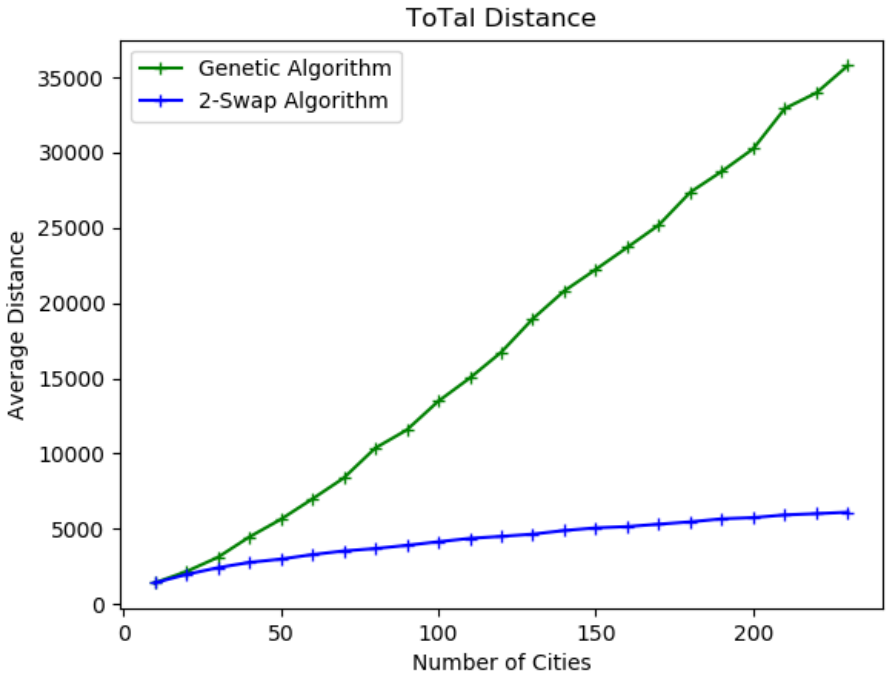



图 4-2 二阶段算法求解后的总距离平均值

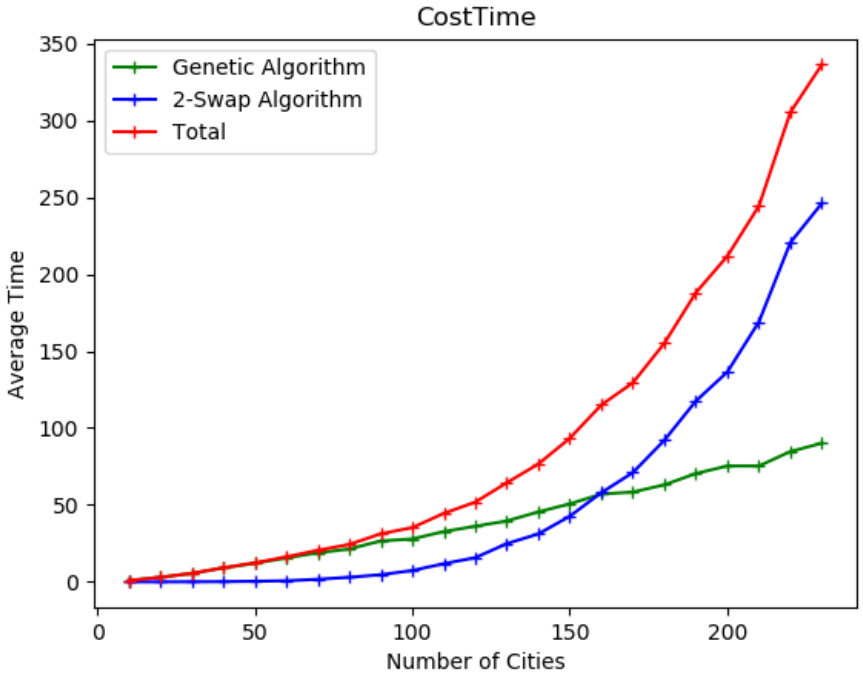



图 4-3 二阶段算法花费的时间的平均值

其次是从图 4-3 中观察两阶段算法花费的时间，其中蓝色的线是遗传

算法花费的平均时间，绿色的线是二交换算法花费的平均时间，红色的线则是两个阶段算法所花费的平均总时间。因为停止遗传代数不多，所以遗传算法的花费时间几乎是随着点数线性增长的。但本文的第 3.2 节分析过，二交换算法的时间复杂度是$O(n^2)$，所以在点数比较多的情况下，二交换算法的花费时间是在很快增长的，可以预测它接下来的趋势。

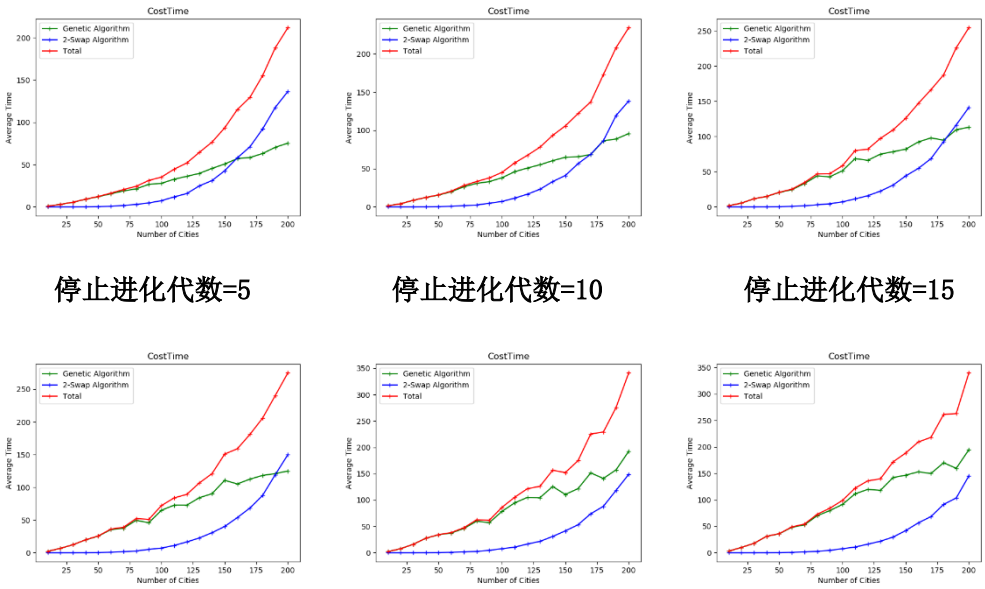
总体来看，当点数比较少的时候，主要是遗传算法消耗时间；而点数比较多的情况下，则是二交换算法比较消耗时间。怎样提高算法的整体速度，是一个值得深究的问题。在本文第 3.2 节，实际上已经对二交换算法有了一些优化，本文的 4.3 节和 4.4 节，即是对遗传算法的一些探索。

## 4.3 最优停止遗传代数实验

在进行这个实验的时候，依然是使用了 3.2 中的 demo_test.py 脚本。因为整个工程对代码进行了封装，所以进行本节的实验，实际上是把工程复制成多份，然后将 lib 文件夹中 parameter.py 文件里的停止进化代数做更改，同时运行这些工程里的 demo_test.py 脚本。

因为我的服务器是一个 8 线程的服务器，在做实验的同时还运行了 2 个其他的服务。所以为了保证正常运转，我同时运行了 6 个测试线程，分别测试停止进化代数为 5、10、15、20、25、30 的情况下，算法所花费的总时间和求解路线总距离。

本次测试，测试了 6 种情况下，算法在 10 到 200 个点上的运行情况。算法花费的总时间取图 4-4 所示。




停止进化代数=5　　停止进化代数=10　　停止进化代数=15

停止进化代数=20　　停止进化代数=25　　停止进化代数=30

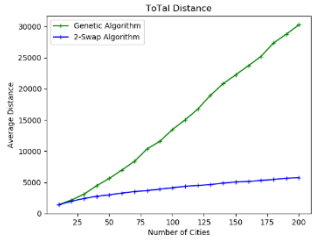
图 4-4 算法在停止进化代数不同时的花费时间

表 4-2 算法在停止进化代数不同时的总花费时间（秒）

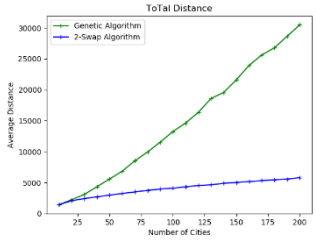
| Citie | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|
| 10.00 | 0.82 | 1.39 | 1.67 | 2.12 | 2.11 | 2.98 |
| 20.00 | 2.94 | 4.03 | 5.29 | 6.68 | 7.57 | 9.97 |
| 30.00 | 5.52 | 8.63 | 11.55 | 12.26 | 16.01 | 17.69 |
| 40.00 | 9.14 | 12.40 | 14.96 | 19.80 | 27.57 | 31.11 |
| 50.00 | 12.34 | 15.74 | 20.77 | 25.64 | 34.36 | 36.09 |
| 60.00 | 16.22 | 20.56 | 25.16 | 36.13 | 38.26 | 48.82 |
| 70.00 | 20.53 | 27.91 | 34.71 | 38.97 | 47.56 | 54.21 |
| 80.00 | 24.28 | 33.15 | 46.91 | 52.27 | 62.35 | 72.63 |
| 90.00 | 31.25 | 37.82 | 47.23 | 50.96 | 61.58 | 84.06 |
| 100.00 | 35.22 | 45.07 | 58.49 | 72.08 | 86.10 | 98.80 |
| 110.00 | 44.50 | 57.37 | 79.93 | 83.80 | 105.17 | 122.02 |
| 120.00 | 51.83 | 67.28 | 81.98 | 89.16 | 121.43 | 135.76 |
| 130.00 | 64.42 | 78.03 | 97.26 | 106.71 | 126.09 | 139.58 |
| 140.00 | 76.58 | 93.44 | 109.32 | 120.91 | 156.54 | 171.60 |
| 150.00 | 93.36 | 105.82 | 125.92 | 150.81 | 152.14 | 188.32 |
| 160.00 | 114.99 | 121.88 | 146.94 | 158.89 | 174.70 | 209.27 |
| 170.00 | 129.38 | 137.14 | 166.45 | 180.92 | 225.37 | 217.93 |
| 180.00 | 155.10 | 172.80 | 187.59 | 205.80 | 229.06 | 261.23 |
| 190.00 | 187.94 | 208.05 | 226.12 | 240.23 | 275.66 | 262.91 |
| 200.00 | 211.77 | 234.12 | 254.61 | 274.98 | 341.34 | 339.81 |

因为遗传算法具有随机性，所以在遗传代数增多的情况下，随机性也变得非常大，从图 4-4 中的六条绿线可以验证这个结论；而二交换算法就比较稳定。但是从表 4-2 中的数值结果总体来看，随着停止遗传代数增多，遗传算法所花费的时间实际上是一直在增长的。从原理上来讲，如果遗传算法所花费的时间增多，停止遗传的代数增多，应该是更有机会得出更优的路径。而且通过图 4-4 来观察，随着停止的代数增加，算法所花费时间的随机性也比较大，运行过程具有很强的随机性。

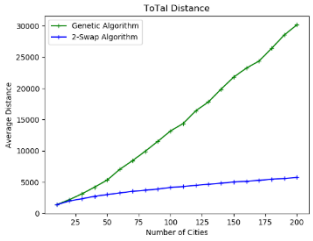
但是事实上的情况，还需要观察两阶段计算后得到的路径距离，是否真的可以通过增加遗传代数，使遗传算法得到更优的解。接下来可以从图 4-5 中观察两阶段计算后，距离的变化情况。



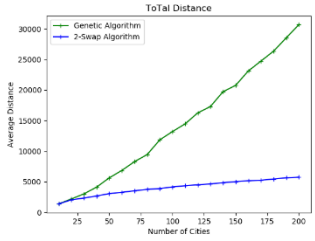

停止进化代数=5



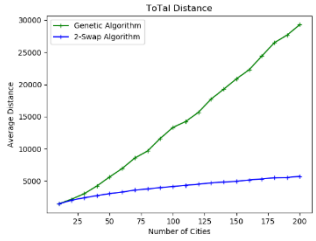

停止进化代数=10



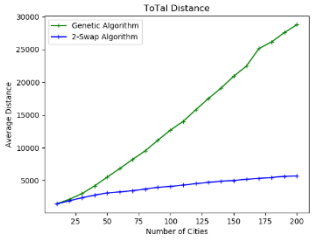

停止进化代数=15




停止进化代数=20




停止进化代数=25




停止进化代数=30

图 4-5 算法在停止进化代数不同时的求解距离

表 4-3 算法在停止进化代数不同时的求解距离

| Citie | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|
| 10.00 | 1401.59 | 1421.87 | 1378.89 | 1386.29 | 1440.64 | 1424.84 |
| 20.00 | 1955.59 | 2060.85 | 1975.24 | 2042.78 | 2009.87 | 1886.66 |
| 30.00 | 2405.46 | 2421.60 | 2339.37 | 2338.13 | 2395.58 | 2369.69 |
| 40.00 | 2760.34 | 2700.77 | 2727.45 | 2682.29 | 2722.95 | 2758.10 |
| 50.00 | 2978.01 | 2977.47 | 3002.52 | 3062.28 | 3034.08 | 3094.25 |
| 60.00 | 3277.18 | 3240.54 | 3266.43 | 3282.72 | 3278.56 | 3251.93 |
| 70.00 | 3520.19 | 3493.20 | 3524.36 | 3528.26 | 3589.21 | 3436.76 |
| 80.00 | 3680.65 | 3747.70 | 3700.74 | 3760.05 | 3758.48 | 3692.20 |
| 90.00 | 3895.27 | 3955.07 | 3890.51 | 3889.19 | 3968.83 | 3949.48 |
| 100.00 | 4132.30 | 4096.04 | 4145.35 | 4163.97 | 4144.28 | 4092.68 |
| 110.00 | 4356.93 | 4320.34 | 4287.70 | 4350.21 | 4328.75 | 4293.89 |
| 120.00 | 4490.20 | 4530.21 | 4487.39 | 4510.82 | 4502.74 | 4511.25 |
| 130.00 | 4638.21 | 4645.09 | 4654.54 | 4647.47 | 4708.18 | 4711.11 |
| 140.00 | 4879.98 | 4879.33 | 4827.13 | 4852.72 | 4832.44 | 4879.99 |
| 150.00 | 5054.01 | 5027.01 | 5022.06 | 5019.90 | 4947.64 | 4996.13 |
| 160.00 | 5141.70 | 5156.58 | 5137.01 | 5170.02 | 5156.57 | 5179.54 |
| 170.00 | 5302.34 | 5323.16 | 5283.96 | 5254.50 | 5325.57 | 5320.58 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 180.00 | 5455.33 | 5454.78 | 5463.68 | 5451.44 | 5500.85 | 5451.59 |
| 190.00 | 5656.92 | 5565.92 | 5577.14 | 5651.32 | 5537.52 | 5630.56 |
| 200.00 | 5738.10 | 5812.33 | 5740.42 | 5744.80 | 5730.32 | 5684.58 |

所以根据图 4-5 和表 4-3 中的数据，尽管随着停止遗传代数增大，程序在遗传算法中多运行了几代，但是经过遗传算法求得的近似解，并没有明显的距离缩短。所以本节的实验证明，遗传算法求近似解的过程中，不需要盲目增加遗传的代数；在过多的代数后停止遗传，遗传算法在相同范围内可以得到的近似解是与原来差不多的，并没有实质性的提升。

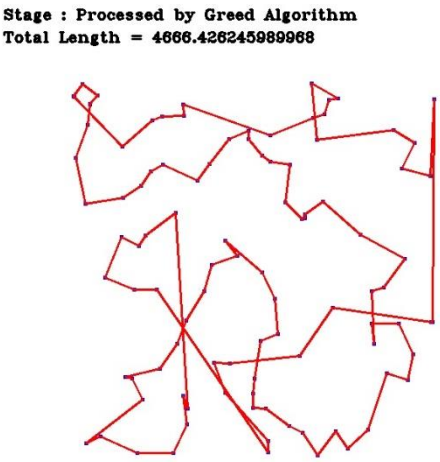



**图 4-6 使用贪心算法找到的最短路线**

## 4.4 贪婪算法初始化探索

在本文的第 3.1 节提到，遗传算法的初始化，实际上是随机生成的第一代路径，继而进行遗传算法的工作。但是我在后期优化中，想到随机生成的路径，不如随机选择起点，采用贪婪算法生成几十条第一代的路径，这样从遗传算法的角度，第一代会有很多优秀的“基因”，更容易求的更高的解。于是诞生了 demo_greed_ga_swap.py 脚本。

在这个脚本中，我使用贪心算法为遗传算法生成了第一代路径，并试图通过遗传算法找到更优解，再通过二交换优化。如图 4-6，我使用贪心算法生成了 40 条路线，并找到其中最短的一条。

继而使用遗传算法对贪心算法生成的路径优化，发现遗传算法几乎不会发挥作用，从运行的十几次程序来看，仅有一次优化过一步，其余都是在浪费计算资源。而运行完遗传算法，继续使用二交换优化的时候，发现依然可以得到最优的结果，如图 4-7 所示。

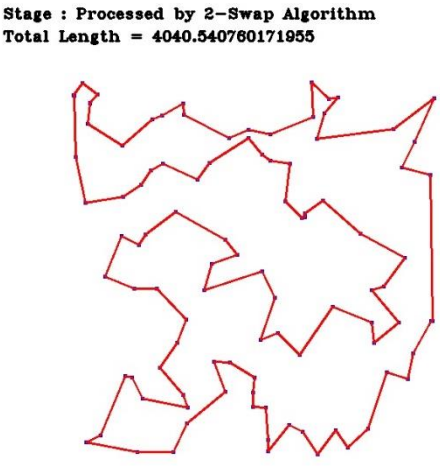



**图 4-7 使用二交换算法优化的图 4-6**

这引起了我的思考，我又尝试了一下遗传算法和贪婪算法，发现在 20 个点以内，两者表现差不多。但是点多的情况下，贪婪算法直接得出的任意一条路径，都比遗传算法求解的要好很多，速度也比较快。所以用一种比较好的算法来为一种比较差的算法做优化，是可行但没意义的。我验证了一下这个结论，编写 demo_greed_swap.py 脚本做实验，原本在表 4-2 中需要花 200 多秒才能解决的 200 个点，使用贪婪算法加二交换算法，只需要 9 到 12 秒即可解决。

所以，其实贪婪算法加上二交换算法，是很容易找到 TSP 问题最优解的。如图 4-8，我在个人电脑上使用这两种算法，花了 4000 多秒，画出了 1000 个点的 TSP 路径。

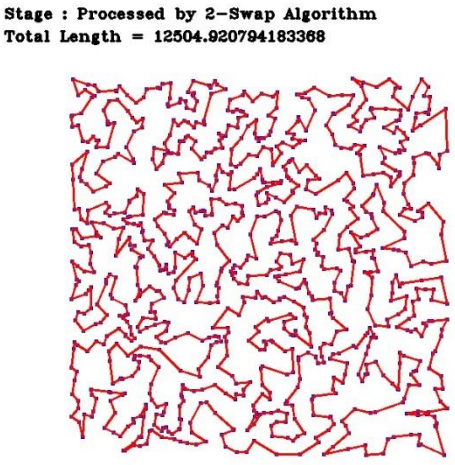



图 4-8 1000 个点的 TSP 路径

# 5 总结

关于遗传算法和二交换优化的探索，做到这里告一段落。本文的实验代码参考过网上的代码，但是因为有诸多缺陷，本文的代码都是重写、优化再封装进库中的；而本文的一字一句，则完全是自行完成的。算法的研究如星辰大海，本文中所做的这点工作显得微不足道，不过也加入了我自己的一些思考。至此，本文顺利完成 TSP 求解算法，通过图形化界面展示过程核测试性能等工作；而第四章的后两节，则是浓缩了一周多的思考与探究。希望在未来可以接触到更前沿的遗传算法理论，有更多的探索呈现在世人眼前。