智能算法与应用 实验一

实验题目：模拟退火算法与遗传算法（单点与多点智能优化）解决TSP问题

> 成员1：唐瑞怡 18340159，负责模拟退火算法实现
>
> 成员2：张琛颐 18340205，负责遗传算法实现

> 日期：2021/5/19

摘要

本实验利用局部搜索策略、模拟退火策略和遗传策略，从单点与多点智能优化两个角度解决旅行商问题（TSP问题）；首先从局部搜索策略开始，之后改进加入温度、随机性等因素的模拟退火策略来弥补局部搜索的不足。最后还使用了遗传算法，通过调试种群大小，使用了精英算法和轮盘赌算法结合的选择方法，以优化TSP问题尝试找到最优解。

在单点智能优化的探索中，对相同规模的TSP问题，比较了使用局部搜索策略（$Local\ Search$）和添加模拟退火策略（$Simulated\ Annealing\ Algorithm, SAA$）之后的运行结果，找出了与TSP最优解10%误差之内的路径。通过实验发现，得到的模拟退火的最优解普遍比纯局部搜索策略得到的最优解好，并得出结论：局部搜索策略容易陷入局部最优解，而模拟退火算法因为其以一定概率接受差解的优点跳出局部最优，从而可逐渐收敛到全局最优。

在遗传算法（$Simple\ Genetic\ Algorithm, SGA$）的探索中，在两个测试样例中使用遗传算法，成功找到了与最优解10%误差之内的路径。并且在相同的TSP数据上，与模拟退火算法进行了比较，得出了遗传算法最优解普遍比模拟退火算法效果优的结论，这是因为多点搜索从多个解出发进行迭代，可以产生更多候选解，更容易找到更优解，不容易陷入局部最优点。

一、导言

1.1 TSP问题

TSP问题提出了这样的情景：假设一个商人需要到各个城市进行销售，每两个城市都是连通的（也就是说城市间的路程都是已知的），如何找到一条从起点开始，访问每一个城市一次，最后回到起点的最短路径？这个问题是一个NP问题，需要进行组合优化，在这里，我们使用了模拟退火算法和遗传算法来对这个问题进行优化，找到尽可能好的路径。

在本实验中，路径的起点是可变的，即不固定路径的起点（实际上，由于路径是回路，可设置路径中任一城市作为起点）。使用的数据集是著名的$TSPLIB$，并在里面选取了其中问题规模为105或131个城市的数据集，每个数据集都包含两个文件：各个城市对应的坐标和最优路径。

1.2 模拟退火算法（SAA)

SAA 是解决陷入局部最优解问题的有效方法，本质为一种引入了**随机因素**的**贪心算法**。SAA以一定的概率接受一个比当前解要差的解，因此有可能会跳出这个局部的最优解，达到全局的最优解。

SAA 来源于晶体冷却的过程，如果固体不处于最低能量状态（即“差解”），给固体加热再冷却（即“退火”），随着温度缓慢下降，固体中的原子按照一定形状排列，形成高密度、低能量的有规则晶体（即“全局最优解”）。而如果温度下降过快，可能导致原子缺少足够的时间排列成晶体的结构，结果产生了具有较高能量的非晶体（即“局部最优解”）。因此，可根据退火的过程，给其在增加一点能量，然

后再冷却，若增加能量后跳出了局部最优解，则本次退火是成功的。

SAA的基本要素有：

- 初始温度与终结条件的设置；
- 局部搜索方法的选择；
- 接受概率的设置（即接受新解作为当前解的概率）；
- 冷却控制的实现，即降温方式的选择（退火系数）；
- 内层平衡的设置（设置内循环次数等）。

1.3 遗传算法

遗传算法模拟了达尔文的生物进化论，通过程序推演来一步步推演和模拟生物种群进化的过程。在进化论理论中，最著名的论述有：过度繁衍，生存斗争，遗传变异和适者生存，根据这个理论，可以运用到TSP问题上，以希望随着一代代种群不断演化，找到问题更优的解。

在TSP问题中，将问题的解集设成初始的种群，种群中不同的个体代表不同的解，当种群开始演化的过程中，先是大量繁殖（过度繁衍），繁殖的过程中出现遗传变异，然后表现更有的个体获得继续生存的机会，形成新的种群。遗传算法希望可以通过种群迭代过程中，将优秀的基因保留下来，通过交叉和变异操作又提供了随机性，可以产生新的基因，让算法更加灵活。

遗传算法的基本要素有：

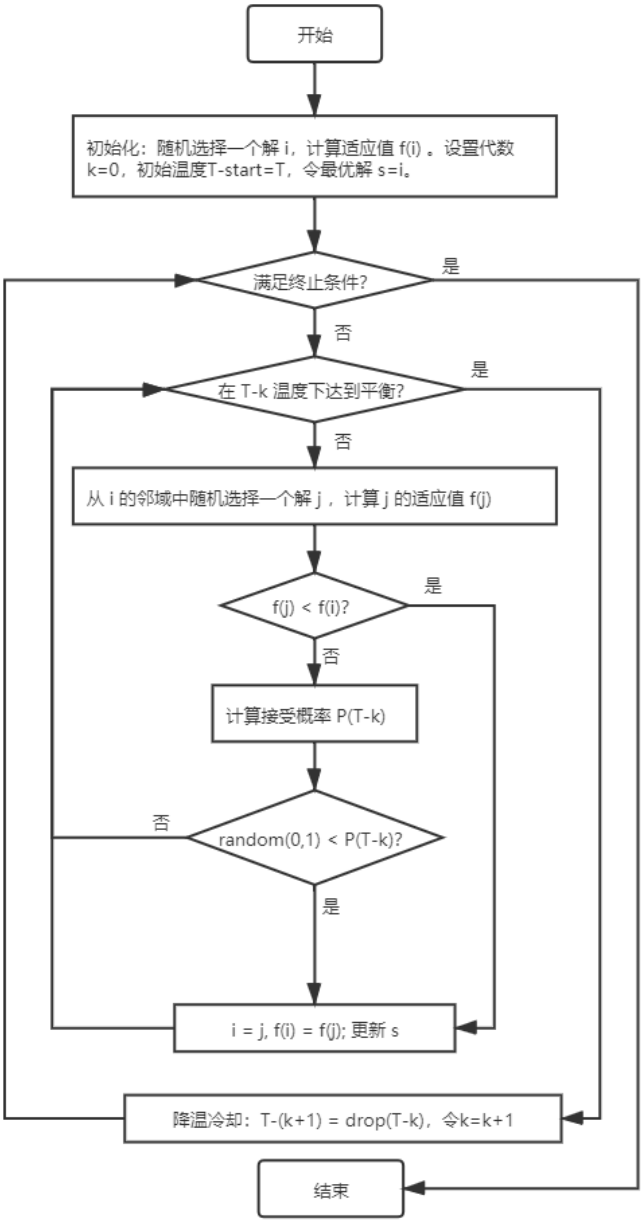
- 种群中个体基因的表示
- 交叉和变异的方式
- 选择下一代种群的方式
- 计算每一个个体适应值的方式

二、实验过程

2.1 模拟退火算法

2.1.1 算法流程

SAA算法的流程图如下：



其中，在本次实验中，对各**基本要素**进行如下设置：

1. 初始温度：设置为充分大的数（如50000）。
2. 初始解状态：对n个城市组成的序列 $[0, 1, 2, \ldots, n-1]$ 进行随机排序，作为算法迭代的起点。
3. 局部搜索方法：即获取i 的邻域，引入了swap, 2-opt等方法。
4. 适应值的计算：使用欧氏距离，计算回路中相邻两点之间的距离之和；
5. 接受概率：采用Metropolis准则

> Metropolis准则：以一定的概率接受恶化解，从而使算法具有逃脱局部极值和避免过早收敛的全局优化能力.

$$P(T_k) = \begin{cases} 1 & f(j) < f(i) \\ e^{-\frac{f(j)-f(i)}{T_k}} & f(j) \geq f(i) \end{cases}$$

6. 降温方式（退火系数）：有多种降温方法，如
   - 设置降温系数：取 $\alpha < 1$，一般取[0.8, 0.99]，以$T_{n+1} = \alpha T_n$的形式下降；
   - 经典模拟退火降温方式：$T_k = \frac{T_0}{log(1+k)}$；

- 快速模拟退火降温方式：$T_k = \frac{T_0}{1+k}$；

在本实验中，我们使用了经典模拟退火降温方式。

7. 内层平衡的设置：通过设置内循环次数实现。
8. 算法终止条件：通过设置终止温度来实现。

因此，算法的伪代码如下：

```
Procedure SAA
    随机生成一个初始解X_0，并计算其适应值f(X_0)
    X_best:=X_0, k:=0, T_k=T, T_end:=1e-8
    while T_k>T_end then
        for i:=1 to L then
            利用局部搜索策略，在当前解X_k的基础上生成新解X_new，并计算其适应值f(X_new)
            if f(X_new)<f(X_k) then
                X_k := X_new
                if f(X_k)<f(X_best) then
                    X_best := X_k
                end if
                continue
            end if
            计算接受概率P(T_k)=e^{(df)/T_k}
            if (0,1)中的随机数小于P(T_k) then
                X_k := X_new
            end if
        end for
        T_{k+1} := T_k/log(1+k)
        k := k+1
    end while
    print X_best
end procedure
```

2.1.2 主要流程及功能说明

初始化

随机初始化一个城市访问序列。即使用$[x_0, x_1, \ldots, x_{n-1}]$来表示城市访问序列，其中$x_0, x_1, \ldots, x_{n-1}$为$[0, n-1]$范围内两两互不相等的$n$个数，表示$n$个城市。

```
def initialize_path(num)
```

评价函数

在本实验中，评价函数是计算回路长度的函数，我们使用了欧氏距离作为评价指标。即从头到尾遍历城市数组，根据相邻两个城市的坐标来进行计算它们之间的直线距离，最后累加起来即可（注意头尾相接）。

```
# calculate the distance between two points: start and end
def calculate_distance(city, start, end)
# calculate the total distances of the whole route.
def calculate_route(route, city)
```

获取两个随机点

在局部搜索策略中，需要获取路径中的两个不相同的随机点$p1, p2$:

```
def get_2_points(num)
```

局部搜索函数

在本实验中，我们设置了5种局部搜索策略来找邻域中的解：

1. swap操作：交换路径中两个随机点，要求两个点不相同；
2. reverse操作（2-opt）：将两个随机点x_i, x_j之间序列反转；
3. swap_2_arrays操作：交换两个随机点x_i, x_j外的数组A和B：

$$\begin{aligned} &交换前：A\ [x_i, \ldots, x_j]\ B \\ &交换后：B\ [x_i, \ldots, x_j]\ A \end{aligned}$$

4. extract_to_head操作：将两个随机点x_i, x_j之间的序列提取到路径的头部：

$$\begin{aligned} &提取前：A\ [x_i, \ldots, x_j]\ B \\ &提取后：[x_i, \ldots, x_j]\ A\ B \end{aligned}$$

5. extract_to_tail操作：将两个随机点x_i, x_j之间的序列提取到路径的尾部：

$$\begin{aligned} &提取前：A\ [x_i, \ldots, x_j]\ B \\ &提取后：A\ B\ [x_i, \ldots, x_j] \end{aligned}$$

并提供一个选择函数local_search，以选择使用对应的策略生成新路径：

```
# get the new route(choice- method chosen)
def local_search(route, choice)
```

五种策略函数如下：

```
# 1: To swap 2 random points in the route.
def swap_2_points(route)
#:2: reverse the route between two random points. Aka "2opt".
def reverse(route)
#:3: swap 2 arrays outside 2 random points.
def swap_2_arrays(route)
#:4: extract the array between two random points to the head of the route.
def extract_to_head(route)
#:5: extract the array between two random points to the tail of the route.
def extract_to_tail(route)
```

纯局部搜索算法

当只使用局部搜索策略时，需要设置循环次数来进行遍历。且局部搜索只接受好解，故当陷入了局部最优解时，无法跳出来。

```
def Local_Search_only(choice):
     # initialization is omitted
    while count <= LOOP_TIME:
        new_route = local_search(route, choice)
        d1 = calculate_route(route, city)
        d2 = calculate_route(new_route, city)
        diff = d2 - d1
        if diff<0:   # good solution
```

```
            route = new_route
            dist.append(d2)
        else:       # bad solution
            dist.append(d1)
        count += 1
        dist_index.append(count)
```

添加模拟退火操作

在局部搜索策略基础上，添加温度的控制，实现以一定概率接受差解，从而避免陷入局部最优解。

设定参数如下：

```
# SA 参数
T_start = 50000     # initial tempreture
T_end = 1e-8         # terminal tempreture
N = 2000             # inner loop time(iteration time for each tempreture)
```

将局部搜索设置在每一个温度的内循环中，并且判断新解与旧解对应的适应函数差值。利用这个差值及当前温度来计算接受概率概率P，判断是否接受这个差解（好解全部接受）。执行完一次温度的N次内循环，进行降温操作。

```
def SA(choice):
    # initialization is omitted
    while T > T_end:
        for count in range(N):
            new_route = local_search(route, choice)
            d1 = calculate_route(route, city)
            d2 = calculate_route(new_route, city)
            diff = d2 - d1
            r = random.random()
            if diff>0 and math.exp(-diff / T) <= r:   # bad solution and not
accepted
                dist.append(d1)
            else:          # good solution or bad one being accepted
                route = new_route
                dist.append(d2)
            dist_index.append(T_index * N + count)
        T = 1 / math.log10(1+1+T_index) * T     # drop tempreture control
        T_index += 1
```

2.2 遗传算法

2.2.1 算法流程

算法伪代码如下：

```
procedure: GA for TSP
input: n citys with 2d coordinates
output: best tour route

t := 0
随机初始化初始种群 P(t)
计算适应值函数 fit(P)
```

```
while
    对 P(t) 进行交叉，产生子代 C1(t)
    对 P(t) 进行变异，产生变异个体 C2(t)
    计算 C1(t) 和 C2(t) 中所有个体的适应值 fit(C)
    从 P(t) 和 C(t)中，选择部分个体形成下一代种群 P(t+1)
    t := t+1
until 满足终止条件
```

2.2.2 主要流程及功能说明

种群初始化

种群初始化主要是根据设置的初始种群数量n_0，随机初始化n_0个不同基因。这里说的基因，是指TSP问题的一个解，也就是城市访问序列$[x_0, x_1, \ldots, x_{n-1}]$，与一般的种群算法不同，这里需要保证序列中任意两个x_i,x_j都不相同。初始化城市访问序列的方法和退火算法时相同，这里就不赘叙了。

计算适应值

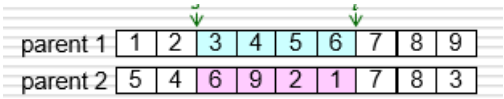
适应度表示一个个体在这个种群中的生存能力，适应性更强的个体具有更大生存的可能，在TSP问题中，我们希望可以让访问城市的总路程更短，因此总路程更短的个体更容易生成下来，这里定义适应值为：

$$fit(X) = \frac{1}{route(X)}$$

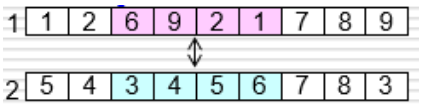
其中$route(X)$表示这个城市序列的总路程

交叉

这里使用了Partial-Mapped Crossover(PMX)的变异方法，方法是这样的：先在两个父母（A,B）上随机选到2个点(p_1,p_2):



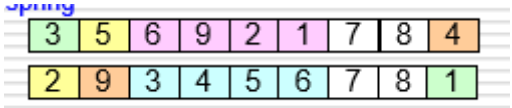
先将A和B上p_1和p_2之间的序列交换来产生下一代C_1, C_2:



由于简单的交换操作可能会导致C_1, C_2的序列上出现相同的城市序列号，不满足TSP问题，因此，还应该将交换时的对应关系存储起来，以修改最终的C_1和C_2序列:

(对应关系):

(修改之后的子代):



伪代码如下:

```
procedure: PMX
```

```
input: Parent1,Parent2
output: child1,child2
//选出两个随机点
p1 := random_select()
p2 := random_select()
//通过交换生成子代
l := len(parent1)
child1 := join(parent1[0:p1], parent2[p1:p2], parent1[p2:l])
child2 := join(parent2[0:p1], parent1[p1:p2], parent2[p2:l])
根据p1,p2 生成一个在这段序列中，parent1和parent2的对应关系r1,r2
//根据对应关系，修改子代
for i:=0 to p1:
    if child1[i]可以在r中找到对应关系：
        child1[i]=r1(child2[i])
    if child2[i]可以在r中找到对应关系：
        child2[i]=r2(child2[i])
for i:=p2 to l:
    if child1[i]可以在r中找到对应关系：
        child1[i]=r1(child2[i])
    if child2[i]可以在r中找到对应关系：
        child2[i]=r2(child2[i])
```

变异

变异是指在一个个体中，其中的序列发送突变，为的是给种群中增加更多的基因种类，添加随机性，这里变异的方法与模拟退火中局部搜索的策略相同，分别是：

1. swap操作：交换路径中两个随机点，要求两个点不相同；
2. reverse操作（2-opt）：将两个随机点x_i, x_j之间序列反转；
3. swap_2_arrays操作：交换两个随机点x_i, x_j外的数组A和B：
4. extract_to_head操作：将两个随机点x_i, x_j之间的序列提取到路径的头
5. extract_to_tail操作：将两个随机点x_i, x_j之间的序列提取到路径的尾部：

选择

选择过程模拟进化论中优胜劣汰的策略，在经过交叉和变异之后，得到大量的子代，这使得种群数量大大膨胀，而在自然界中，由于资源有限，只有部分个体能适应下来，这个称为最大种群数量。而适应值更高的个体具有更大的几率可以生存下来。这里的选择，使用了精英选择策略和轮盘赌算法的结合，来实现选择下一代。在实现上，可以设定一个比例，也就是有m%子代是通过精英选择策略产生的，(1-m%)的子代是由轮盘赌产生的。

- 精英选择策略

  精英选择策略是为了避免当前种群中最优的个体在下一代中丢失，而失去了优秀的解，导致算法难以收敛。同时这里保留种群中最好的部分个体，是为了避免轮盘赌中，无法将一些好个体保留下来，导致算法难以收敛。

- 轮盘赌算法

  轮盘赌算法是根据各个个体的适应值，来计算每一个个体的被选中的概率，适应值大的个体有更大的概率。计算时，先计算每一个个体的累计概率，然后通过随机均匀生成一个随机数，来选择一个个体。

  具体实现如下：

```
def wheel_select(group,node_list):
    #通过轮盘算法，得到选择之后的种群，返回group
    if len(group)<=group_size: #总数比最大种群数小，不需要选择
        return group
```

```
    #计算每一个个体的适应值
    fitness_list=[]
    for item in group:
        fitness_list.append(fitness(item,node_list))
    fitness_sum=sum(fitness_list) #总的适应值
    #计算每一个个体被选择的累计概率
    sum_p=0
    cumulative_p=[]#每一个个体被选择累计概率
    for item in fitness_list:
        p=item/fitness_sum
        cumulative_p.append(p+sum_p)
        sum_p+=p
    rand_num = np.random.uniform(low=0.0,high=1.0,size=group_size) #生成
group_size个随机小数
    choice_index=np.searchsorted(cumulative_p,rand_num) #得到选出的个体对应的下
标
    new_group=group[choice_index]
    return new_group
```

三、结果分析

3.1 实验环境

- 操作系统：Windows 10
- 编程语言：Python 3.8
- 本地IDE： pycharm / Jupyter notebook

3.2 局部搜索算法的结果

3.2.1 测试样例1：

城市数为105，参考最优解为14379。

实验结果

- 可调参数为Local Search Choice（其中，1-`swap`，2-`reverse`，3-`swap_2_arrrays`，4-`extract_to_head`，5-`extract_to_tail`）

序号	Choice	解（路径长度）	误差	运行时间(s)
1	1	31216.17	117.10%	8.07
2	2	15354.60	6.7%	7.75
3	3	21828.31	51.81%	8.42
4	4	40683.42	182.93%	8.72
5	5	25328.21	76.15%	8.49
6	2	15750.47	9.54%	7.86
7	2	16010.55	11.34%	8.02
8	2	15740.65	7.59%	8.05
9	2	15489.61	7.72%	8.55
10	2	15968.83	11.06%	8.13

根据以上结果可以看出，Local Search Choice=2对应的`reverse`操作是效果最好的，其他4个操作的误差都过大。因此，**以下的分析都基于choice=2进行**：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	14379	15354.60	16010.55	15719.12	236.04	55713.76
误差		6.7%	11.34%	9.32%		

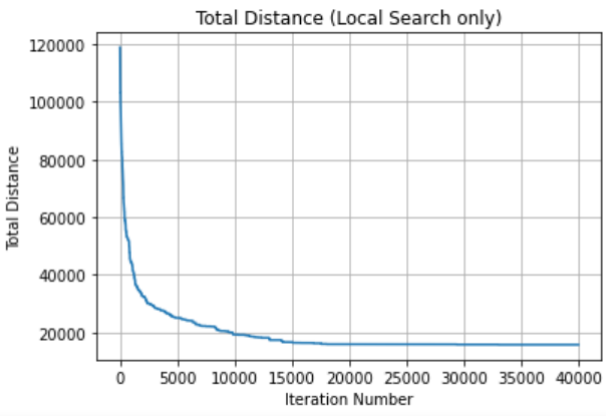
算法性能

- **平均速度**：8.06s
- **其中一次的运行结果**如下：

> 由于画出了中间结果的图，运行时间大大增加。

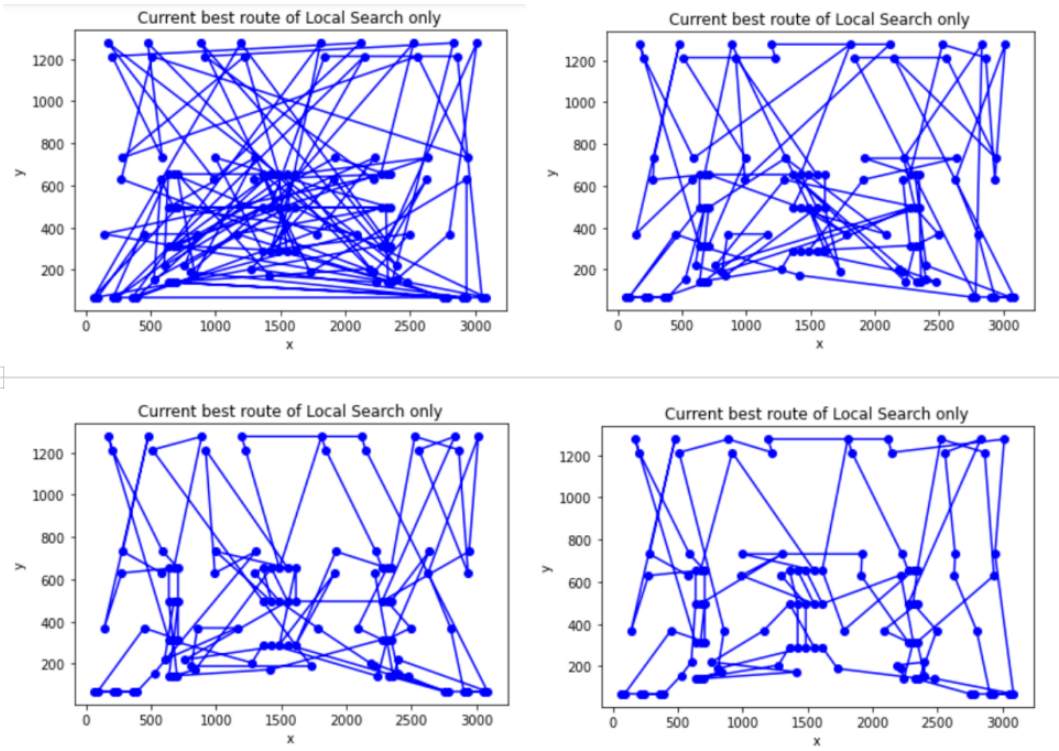
```
Duration for only Local Search(s):  20.863093852996826
Length of the best path is : 15752.626491018618
The best solution for 40000 loops is :
 [68, 69, 62, 61, 56, 104, 58, 55, 54, 49, 50, 53, 57, 52, 51, 45, 42, 40, 41, 36, 35, 103, 43, 46, 44, 47, 48, 39, 31, 30, 28, 21, 20, 10
2, 14, 10, 9, 6, 5, 1, 0, 2, 7, 8, 11, 19, 22, 27, 29, 32, 26, 23, 18, 25, 24, 17, 15, 16, 4, 3, 12, 13, 33, 34, 37, 38, 59, 60, 64, 65, 8
6, 87, 93, 99, 94, 89, 88, 98, 97, 92, 101, 100, 96, 95, 91, 90, 84, 82, 81, 77, 70, 67, 66, 63, 72, 71, 76, 78, 83, 85, 79, 75, 80, 74, 7
3]
```

  - 由图可看出，此时最优路径长度为15752.62
  - 随着迭代次数的增加，路径收敛情况如下：（迭代次数为40000）

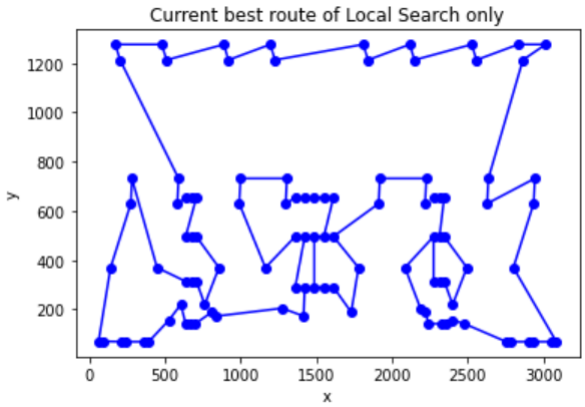


分析：从图中可以看出，总路径长度在循环迭代次数为20000左右开始收敛。

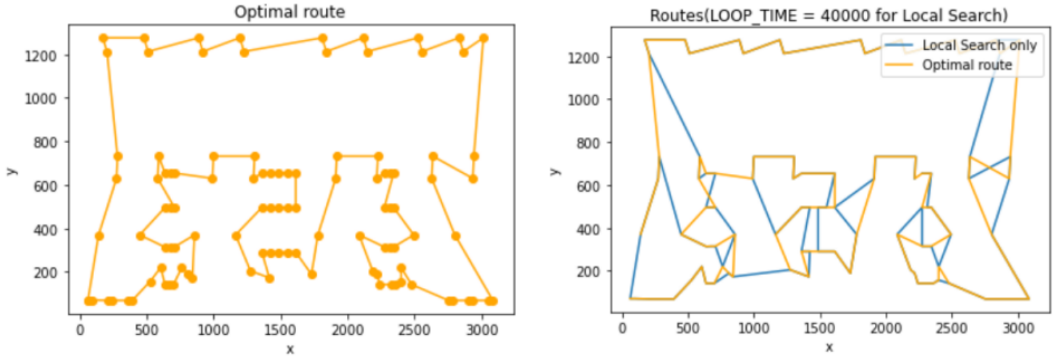
- 以下呈现前2000次迭代中，每500次的**路径变化情况**：



在迭代完成后，得到的**最终路径**如下：



而此问题的**最优路径**和**两者的比较**如下：



分析：从路径变化可以看出，随着迭代次数的增加，路径的交叉程度逐渐降低，解的精度逐渐变大。且经过40000轮的迭代后，算法收敛到了局部最优点，路径显然得到了简化，最终达到了9.54%的误差，相互交叉的路径数量大大减少，和最优路径也比较接近。

3.2.2 测试样例2

城市数为130，参考最优解为6110。

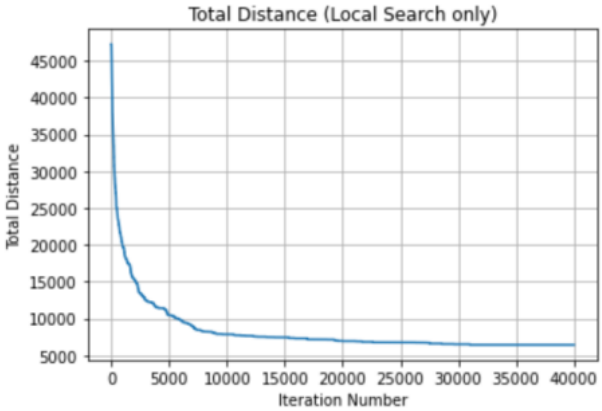
实验结果

- 根据测试样例1的经验，发现**Local Search Choice=2**的实验效果最优，因此以下分析只基于choice=2的结果。
- 具体分析步骤与测试样例1大致相同，故以下只给出最终结果：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	6110	6426.39	7137.22	6757.12	241.41	58280.59
误差		5.17%	16.81%	10.59%		

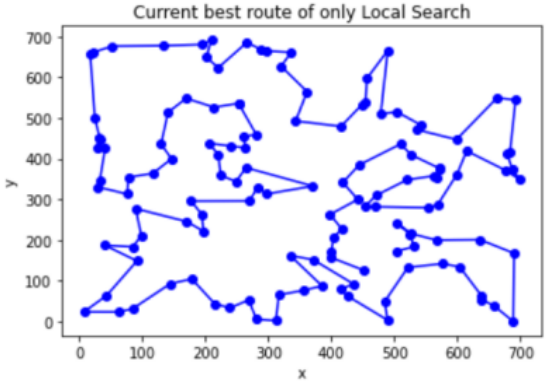
算法性能

- **平均速度**：11.51s
- **最优解的运行结果**如下：
  - 最优解的路径长度为6426.39
  - 随着迭代次数的增加，路径收敛情况如下：（迭代次数为40000）



分析：从图中可以看出，总路径长度在循环迭代次数为30000左右开始收敛。

  - 在迭代完成后，得到的**最终路径**如下：



3.3 模拟退火算法的结果

3.3.1 测试样例1：

城市数为105，参考最优解为14379。

实验参数

```
T_start = 50000     # initial tempreture
T_end = 1e-8         # terminal tempreture
N = 2000             # inner loop time(iteration time for each tempreture)
choice = 2          # default local choice = 2
```

实验结果

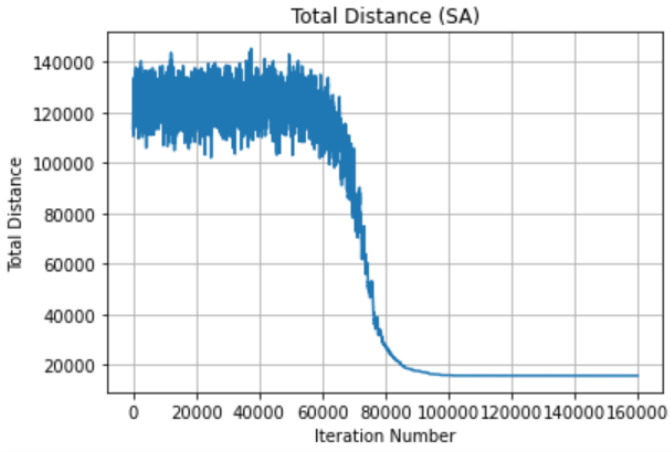
序号	解（路径长度）	误差	运行时间(s)
1	15231.76	5.93%	36.12
2	15707.69	9.24%	32.61
3	15667.37	8.96%	33.97
4	15927.17	10.76%	33.54
5	15596.92	8.47%	33.73
6	14899.35	3.61%	34.35

根据以上的6次运行情况（Local Search Choice=2），可知：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	14379	14899.35	15927.17	15505.04	340.48	115932.67619
误差		3.61%	10.76%	7.83%		

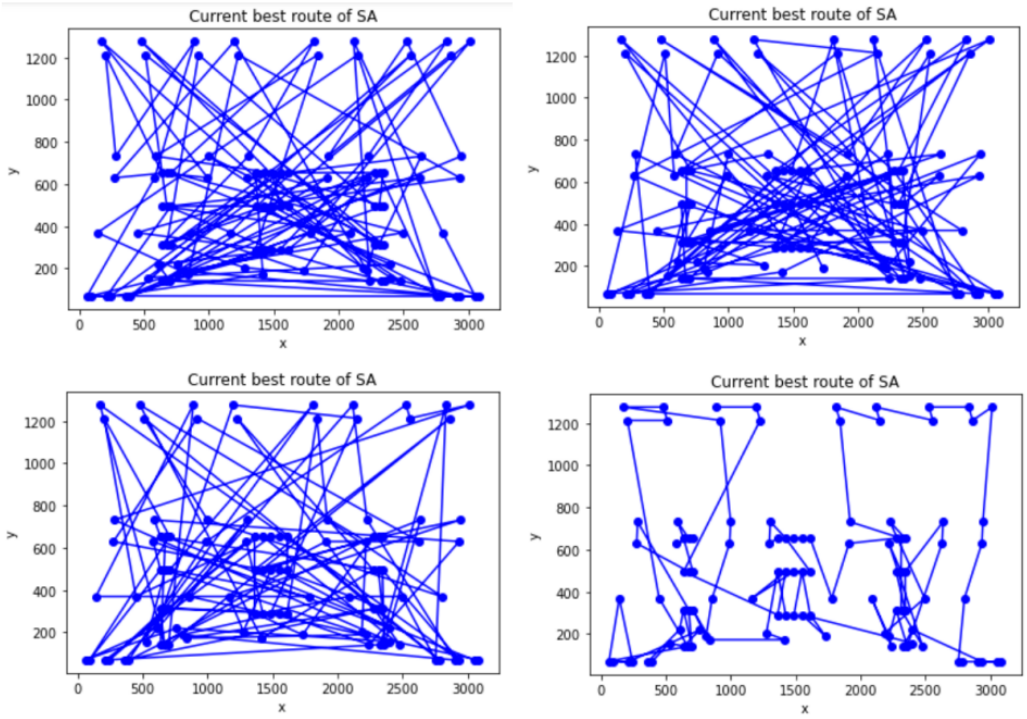
算法性能

- **平均速度**：34.05s
- **其中一次的运行结果**如下：
  - 由图可看出，此时最优路径长度为15760.82
  - 随着迭代次数的增加，路径收敛情况如下：（迭代次数为160000）

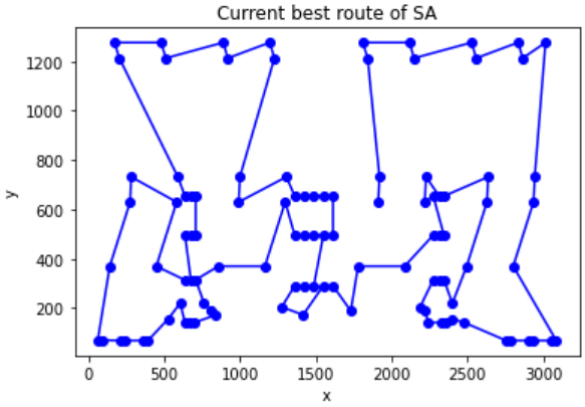


分析：从图中可以看出，在初始降温阶段（迭代次数在0~ 60000左右），总路径长度的变化波动很大，在图中表现为震荡非常频繁，在60000~ 80000次迭代中急速下降，而在迭代次数100000左右接近收敛。因此，可看出在添加了模拟退火操作，以一定概率接受差解之后，总路径长度震荡频繁，收敛速度减慢。

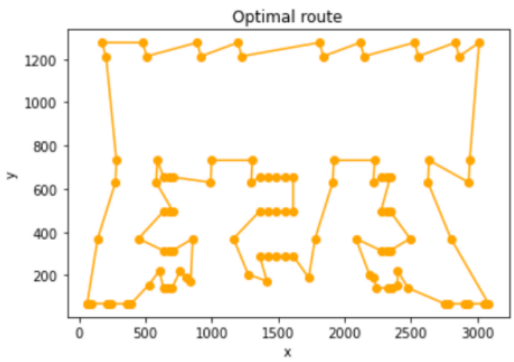
- 以下呈现前80000次迭代中，每20000次的**路径变化情况**：

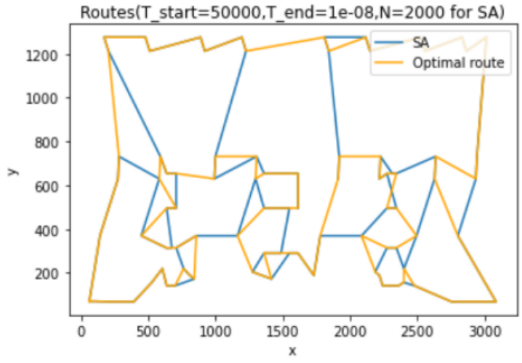


在迭代完成后，得到的**最终路径**如下：



而此问题的**最优路径**和**两者的比较**如下：





分析：从路径变化可以看出，随着迭代次数的增加，路径的交叉程度先震荡后大幅降低并平稳，解的精度也随之逐渐变大。且经过90000轮的迭代后，算法收敛到了局部最优点，路径显然得到了简化，最终达到了9.60%的误差，相互交叉的路径数量大大减少，和最优路径也比较接近。

3.3.2 测试样例2

城市数为130，参考最优解为6110。

实验参数

> 与测试样例1相同，此处省略。

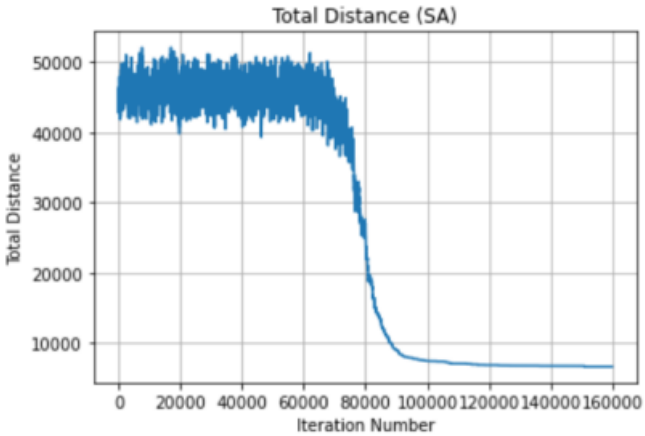
实验结果

- 具体分析步骤与测试样例1大致相同，故以下只给出最终结果：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	6110	6511.07	6841.38	6,638.34	101.55	10314.14
误差		6.56%	11.97%	8.64%		

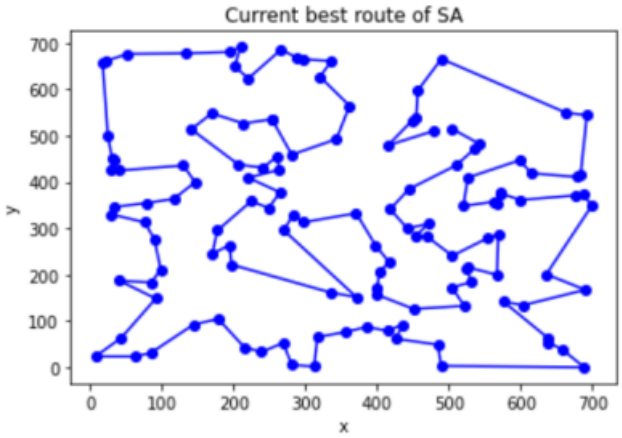
算法性能

- **平均速度**：45.03s
- **其中一次解对应的运行结果**如下：
  - 解的路径长度为6597.19
  - 随着迭代次数的增加，路径收敛情况如下：（迭代次数为160000）



分析：从图中可以看出，在初始降温阶段（迭代次数在0~ 70000左右），总路径长度的变化波动很大，在图中表现为震荡非常频繁，在70000~ 90000次迭代中急速下降，而在迭代次数120000左右接近收敛。因此，可看出在添加了模拟退火操作，以一定概率接受差解之后，总路径长度震荡频繁，收敛速度减慢。

- 在迭代完成后，得到的**最终路径**如下：



3.4 遗传算法的结果

3.4.1 测试样例1：

城市数为105，参考最优解为14379。

实验参数

```
group_size = 200 #种群大小
initial_group_size=30 # 初始种群数量
intersect_p = 0.8 #发生交叉的几率
variation_p=0.5 #变异几率
elite_rate=0.8 #精英选择比例
choice=2 #变异方式：reverse
```

实验结果

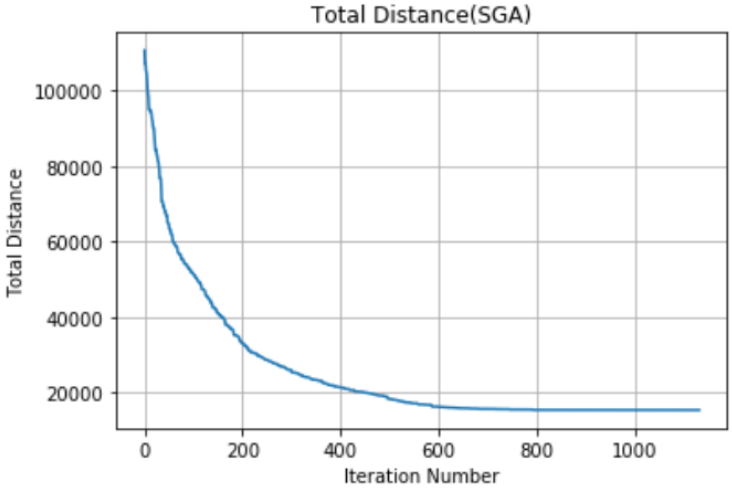
序号	解（路径长度）	误差	运行时间(s)
1	15438.82	7.37%	211.52
2	15209.08	5.77%	232.62
3	15416.12	7.21%	188.52
4	15615.94	8.60%	255.22
5	14839.62	3.20%	260.87
6	15567.58	8.27%	208.87

根据以上的6次运行情况，可知：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	14379	14839.62	15615.94	15347.86	261.61	68440.78
误差		3.20%	8.60%	6.74%		

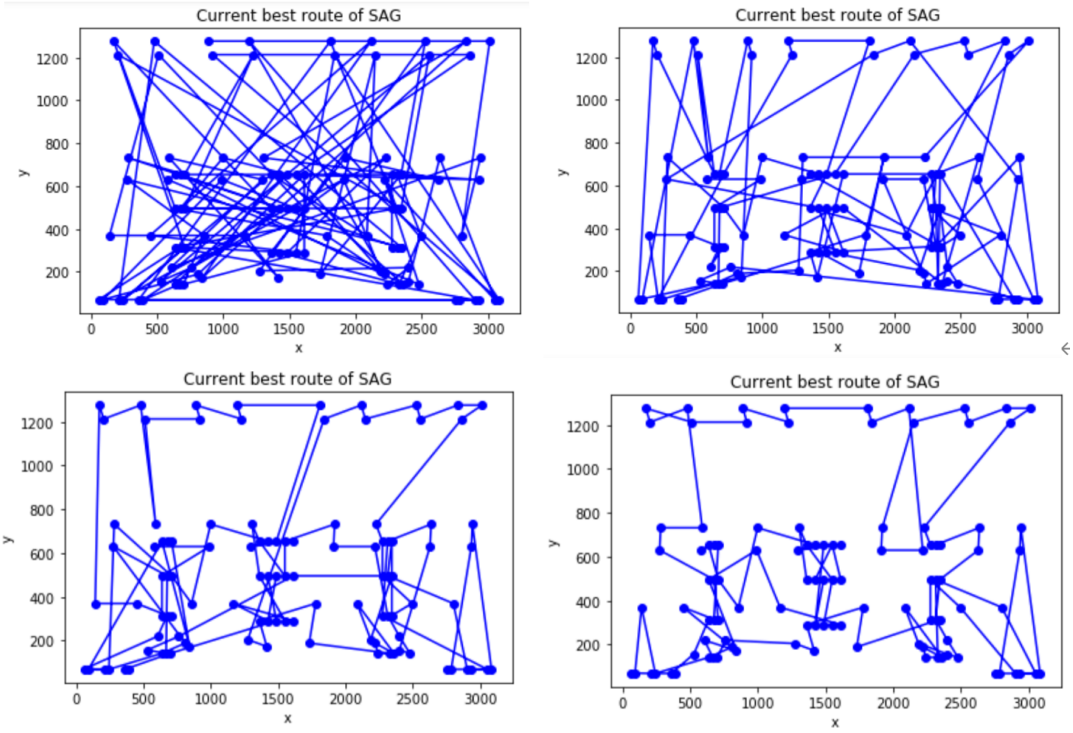
算法性能

- **平均时间**：
- **其中一次运行结果如下**：
  - 最优路径长度：15438.82
  - 随着迭代次数增加，（最佳）路径变化如下：

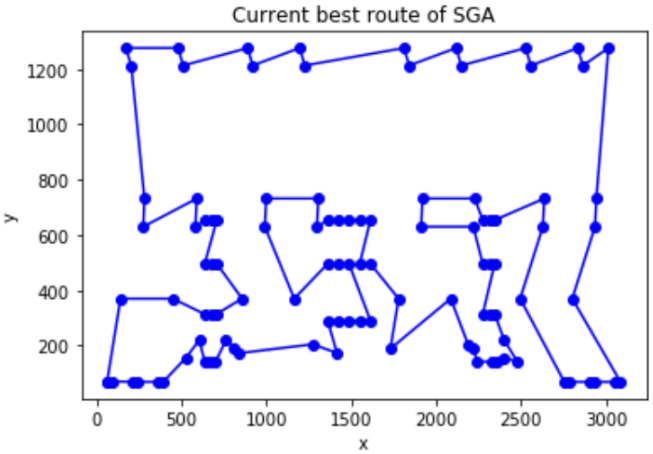


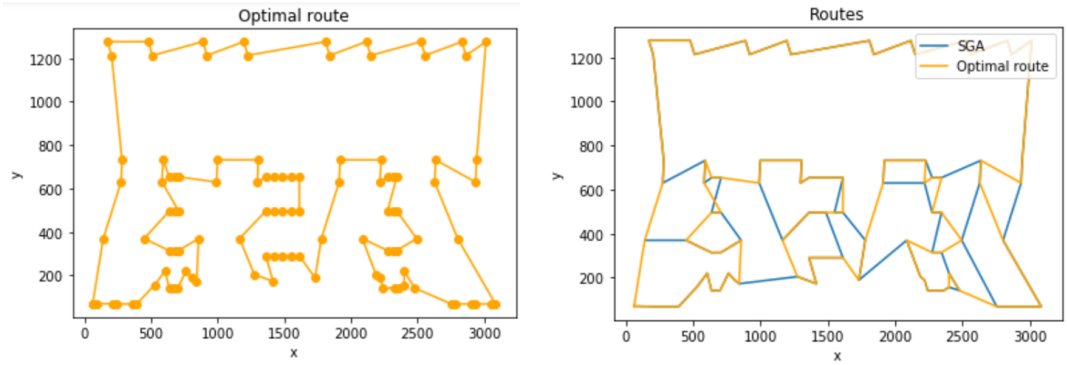
分析：从上面可以看到，随着迭代次数的增加，一开始总距离下降迅速，等到了迭代600次以后，距离下降非常慢，直到迭代1000次左右总距离才不变。这个说明在开始阶段，由于使用了精英选择策略，使得每次种群迭代中，好的基因可以被留下来，使得种群演化不断朝着距离越来越短的道路前进。而随着迭代次数的增加，由于每次迭代都会留下最优的个体，要想通过交叉和变异来产生更优的个体，变得越来越难，因此最优距离减少得越来越慢。

  - 以下呈现前400次迭代中，每100次的**路径变化情况**：



在迭代完成后，得到的最终路径如下：





分析：根据上面的图可以看到，路径一开始交叉非常严重，随着迭代次数的增加，路径的交叉情况越来越少，到了最后的结果时，图上基本没有交叉了，这非常符合常理，并且和最优路径的差别比较小。同时，通过观察算法生成的路径与已知最优路径可以看到，在节点不密集的区域（比如图的上面部分），算法生成的路径和最优路径是一样的，而在节点比较密集的区域（图下面部分），算法生成的路径和最优路径相查比较多，但这些区域由于点与点之间距离比较小，所以最后的路径长度相差不大。

3.4.2 测试样例2

城市数为130，参考最优解为6110。

实验参数

实验参数和测试样例1相同

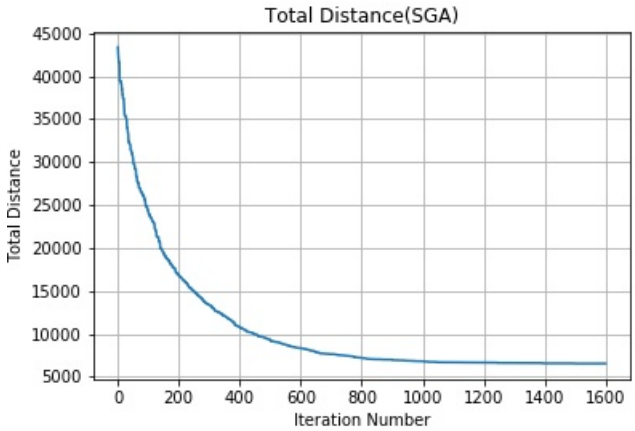
实验结果

根据以上的6次运行情况，可知：

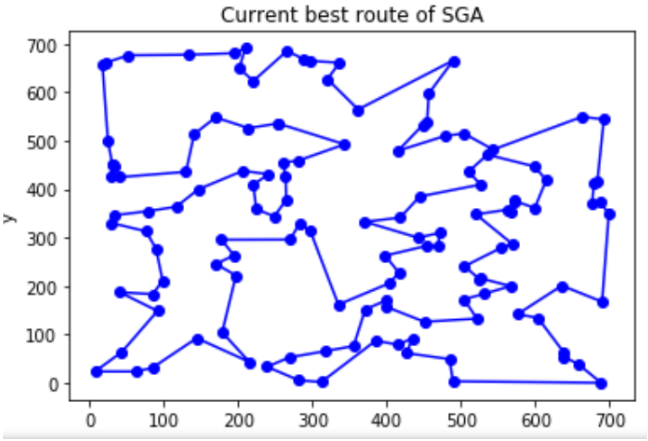
	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	6110	6449.94	6672.09	6563.99	70.97	5037.21
误差		5.56%	9.20%	7.43%		

算法性能

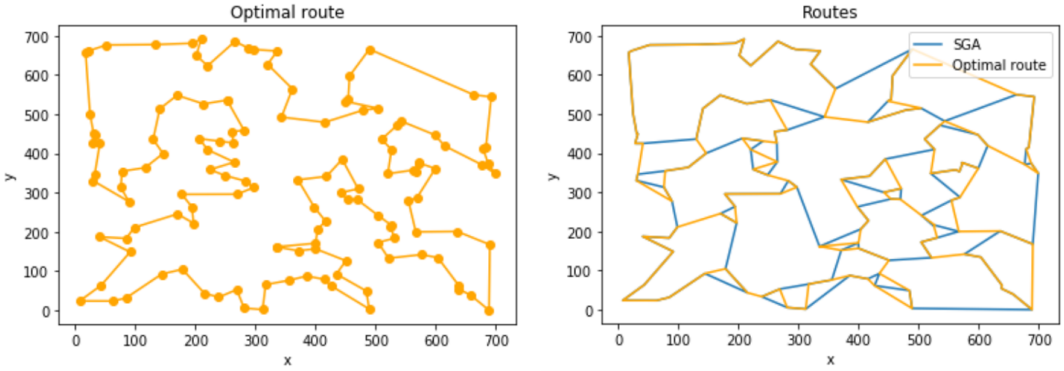
- **平均时间**：
- **其中一次运行结果如下**：



- 最优路径长度：6672.09
- 随着迭代次数增加，（最佳）路径变化如下：



- 而此问题的最优路径和两者的比较如下：



分析：从上面的结果可以看到，算法最终产生的路径和最近路径是非常接近的，重复部分非常多。而且最终产生的路径基本没有交叉，是比较合理的。

对比种群大小对实验结果的影响

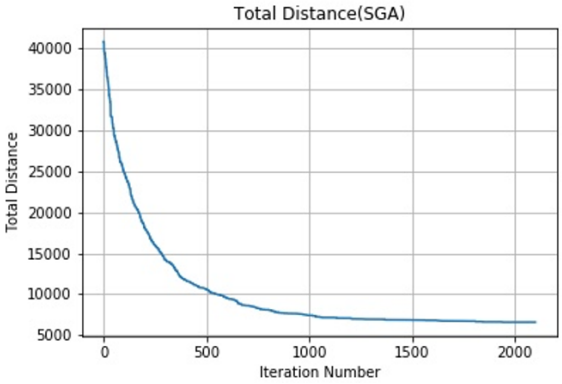
下面数据是实验参数不变情况下，种群大小为100时的数据：

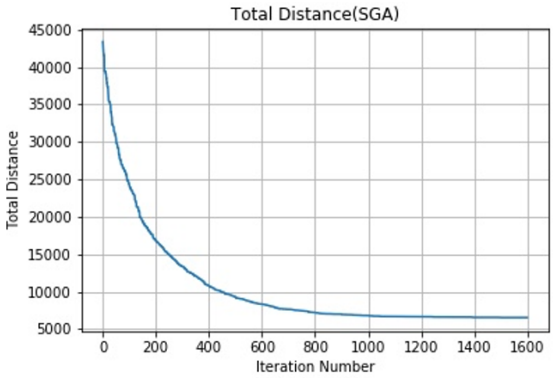
	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	6110	6567.45	6678.11	6,635.09	38.50924	1482.96122
误差		7.48%	9.30%	8.59%		

相比于种群数量为200时的数据：

	参考最优解	最好解	最差解	平均值	标准差	方差
路径长度	6110	6449.94	6672.09	6563.99	70.97	5037.21
误差		5.56%	9.20%	7.43%		

对于最优总路程的变化来说（下面左图是种群数量为100的，下面右图是种群数量为200的）





可以看到种群数量为100时，解明显比200时差，但标准差和方差比较小，数据波动没有这么明显。从最优总路程来看，种群数量为200时需要的迭代次数更少，权衡来看，使用种群数量为200明显更优。

3.5 三种算法的比较

3.5.1 局部搜索和模拟退火算法的对比

由3.2和3.3的分析可知，局部搜索算法和模拟退火算法的误差比较如下：

测试样例	算法	最好解误差	最差解误差	平均值误差	平均时间
1	Local Search	6.7%	11.34%	9.32%	8.06s
1	SA	3.61%	10.76%	7.83%	34.05s
2	Local Search	5.17%	16.81%	10.59%	11.51s
2	SA	6.56%	11.97%	8.64%	45.03s

从两个测试样例的各个指标都可看出，模拟退火算法(SA)的实验效果是优于局部搜索算法(Local Search)的。原因在于，模拟退火存在一定的随机性（引入了“以一定概率接受差解”的特性），这点通过分析总距离随迭代次数变化的曲线也可以看出来：局部搜索策略是一条较光滑下降的曲线，而模拟退火是会先剧烈抖动再突然下降。也正由于这个特性，模拟退火算法具有跳出局部最优，逼近全局最优的可能，达到更高的精度。且因为加入了温度因素，后期温度下降后，抖动的概率会逐渐降低（Metropolis准则），最后慢慢收敛到最优情况。从运行用时的角度分析，局部搜索策略也有其优势：由于迭代设置更简单，训练时间较少，收敛速度提升。而模拟退火算法由于随机性的引入，需要更多的迭代次数才能达到比较好的效果。

因此，在需要一些不太精确的近似解时，可利用局部搜索算法进行探索；而在对时间要求不高、且需要高精度的解的情况下，可利用模拟退火算法进行探索。

3.5.2 模拟退火算法和遗传算法对比

由3.3和3.4的分析可知，模拟退火算法和种群算法的误差比较如下：

值得注意的是，模拟退火和种群算法都是用`reverse`这个局部搜索策略

测试样例	算法	最好解误差	最差解误差	平均值误差	平均时间
1	SGA	3.20%	8.60%	6.74%	226.27
1	SA	3.61%	10.76%	7.83%	34.05s
2	SGA	5.56%	9.20%	7.43%	428.71s
2	SA	6.56%	11.97%	8.64%	45.03s

从算法误差上看，种群算法（SGA）明显优于模拟退火算法（SA）；而在运行时间上看，种群算法呢运行时间是模拟退火算法的好几倍。在误差上，由于种群算法利用了多点搜索的策略，并且在模拟退火的局部搜索基础上增加了交叉的操作，使得每一次迭代都可以产生更多的新解，遇到更好的解概率也就更大，最终算法的误差也就更少。同时，由于种群算法中，每次迭代都会保留一部分解，而不像模拟退火算法那样当前解可能被差解覆盖，这样种群算法其实是可以保留住大部分优秀的解的，算法也就效果更好。从运行时间上看，模拟退火算法由于每次迭代都要计算种群中各个个体交叉变异之后的结果，需要更大的计算量，在时间上花费得更多。

因此，在需要一些不太精确的近似解时，可利用模拟退火算法进行探索；而在对时间要求不高、且需要高精度的解的情况下，可利用种群算法进行探索。

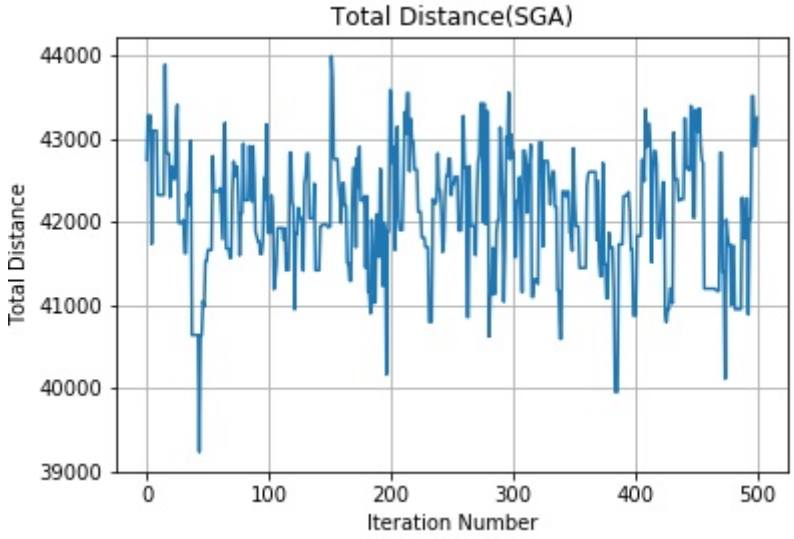
四、结论

如何设计高效的遗传算法

在这次实验中，通过对实验参数进行调整，发现种群精英算法，种群数量大小，交叉和变异方法对算法效果影响很大。

1. 精英算法：

   使用精英算法是希望可以子代中可以保留父代中优秀基因，让它不会在交叉和变异中丢失，使得算法难以收敛。在这次实验中，使用精英算法的另一个原因是，当种群大小为200时，产生的下一代加上父代的总数可能高达600，这个情况下，如果只使用轮盘赌算法可能无法选出大部分优秀的个体。当数量很大时，每一个个体的适应值占总适应值都是很小的，适应值大的个体被选中的概率与适应值小的差别不大，如果这个时候只使用轮盘赌算法的话，选出来的个体和随机选择的效果差不多，无法让种群朝着适应值不断增加的方向前进。下面是只使用轮盘赌算法，最优总路径长度与迭代次数的对比：



从上面的图可以看到，如果只用轮盘赌算法，在500次迭代里面，最优路径的变化震荡明显，而且没有下降的趋势，一直在一个差解附近摆动，算法无法收敛。

2. 种群数量：

   实验中，对比了种群大小为100和200的情况，明显可以看到，种群大小为200时效果更好，这主要是因为种群数量比较大时，相当于从更多点开始搜索，每次通过交叉和变异产生的类型也会更多，更容易找到更优的路径。

3. 交叉和变异方法：

   交叉和变异方法是种群算法中的精髓，通过交叉和变异可以产生新的解，因此好的交叉和变异可以直接影响到算法的效果。在这次实验中，使用了Partial-Mapped Crossover方法进行交叉，并且发现使用reverse这个变异方法比较优秀。

单点搜索和多点搜索的比较

在这次实验中，实现了单点搜索和多点搜索算法，分别是：纯局部搜索和模拟退火算法、遗传算法。在单点搜索中，主要初始解只有一个，每次通过局部搜索策略来寻找更优的解，并以一定概率接受差解。而在多点搜索中，初始解就有多个，每个解可以通过交叉和变异（也就是单点搜索中的局部搜索算法）来找更优的解。这个方法相比于单点搜索，不仅是由多个初始解作为起点来搜索，而且由于交叉的存在，多个解之间相互关联，使得搜索更加效果更好。

五、主要参考文献

[1] 模拟退火算法和遗传算法 https://www.cnblogs.com/houqingying/p/12743159.html

[2] 遗传算法 https://blog.csdn.net/greedystar/article/details/80343841

[3] 遗传算法 https://www.jianshu.com/p/b344e96c9770