求解问题概述

1 TSP问题简介

旅行商问题是一种组合优化问题.**组合优化问题**(Combinatorial Optimization Problem，COP)是一类在**离散状态下求极值**的问题。把某种离散对象按某个确定的约束条件进行安排，当已知合乎这种**约束条件**的特定安排存在时寻求这种特定安排在**某个优化准则**下的**极大解或极小解**。

TSP 的经典提法是：有一个销售员要去若干个城市销售货品，从某个固定城市出发(假设每个城市之间的距离固定)，经过剩下的每个城市至少一次，然后回到起始城市，问题是选择哪条线路，才能使总行程最短。即已有**无向图 G**，假设从**任意一个顶点出发**，剩下每一个顶点恰好**遍历一次**，最后**回到原点**.

TSP 描述为寻找一条巡回路径, 并满足目标函数:

$$f(V)=\min \sum_{i=1}^{n-1} d\left(v_i, v_{i+1}\right)+d\left(v_n, v_1\right) \tag{4}$$

式中 v_i 为城市号 $v_i \in N, 1 \leq v_i \leq n, d\left(v_i, v_j\right)$ 表示城市 i 和城市 j 之间的权值 (可代表：距离、时间等开销）, 若为对称 TSP, 则有 $d\left(v_i, v_j\right)=d\left(v_j, v_i\right)$ 。

2 问题验证数据集

TSPLIB是一个包含了TSP及其相关问题的问题库,本文从中选取10个数据集进行测试

值得注意的是TSPLIB部分数据集最优解距离的计算：取伪欧式 (pseudo Euclidean) 距离，计算方法如下（向上取整）：

$$dist=\sqrt{\frac{\left(X_1-X_2\right)^2+\left(Y_1-Y_2\right)^2}{10}} \tag{5}$$

数据集	最优解
att48	10628
eil51	426
eil76	538
eil101	629
pr76	108159
pr107	44303
pr124	59030
pr136	96772
pr144	58537
pr152	73682

优化方法概述

1 蚁群算法

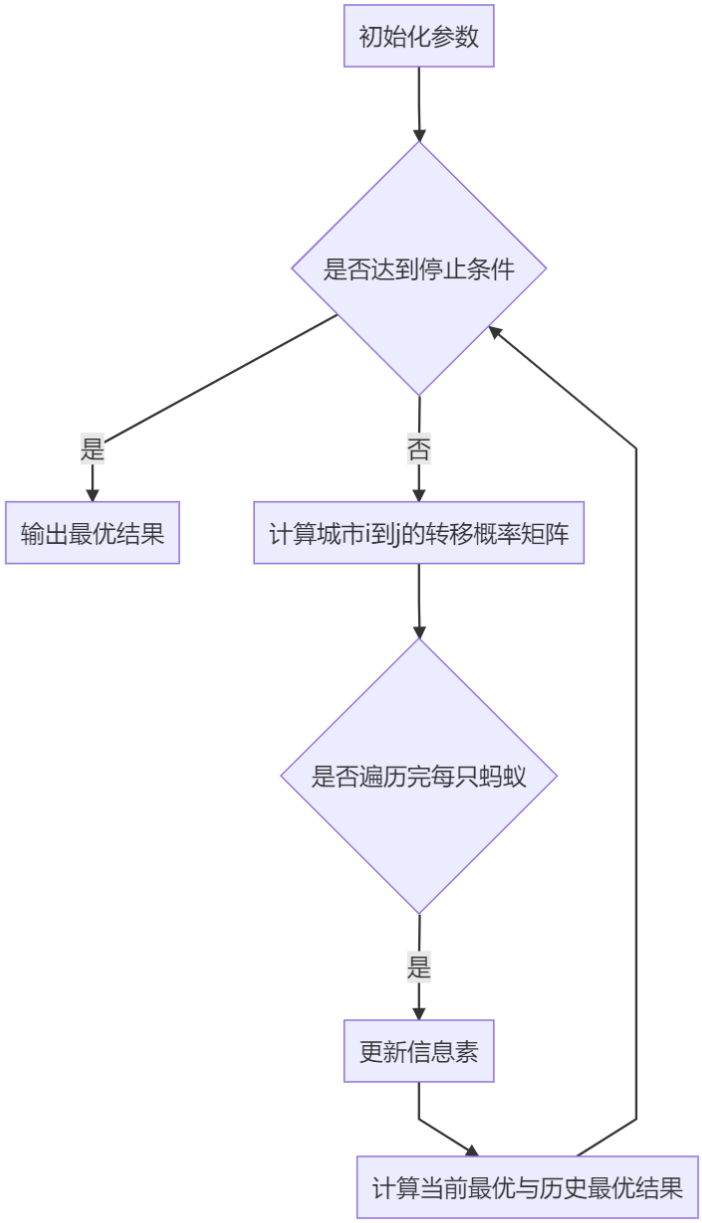
1.1 蚁群算法概述

蚁群算法（Ant Clony Optimization， ACO）是一种群智能算法，它是由一群无智能或有轻微智能的个体（Agent）通过相互协作而表现出智能行为，从而为求解复杂问题提供了一个新的可能性。蚁群算法最早是由意大利学者Colorni A., Dorigo M. 等于1991年提出。

蚁群算法是一种仿生学算法，是由自然界中蚂蚁觅食的行为而启发的。在自然界中，蚂蚁觅食过程中，蚁群总能够按照寻找到一条从蚁巢和食物源的最优路径。

1.2 蚁群算法流程

蚁群算法流程图如下



- 获取城市距离矩阵
- 初始化蚁群算法
- 开始迭代,计算转移概率矩阵

$$p^k(i,j)=\begin{cases}\frac{[\tau(i,j)]^\alpha[\eta(i,j)]^\beta}{\sum_{u\in J_k(i)}[\tau(i,u)]^\alpha[\eta(i,u)]^\beta}, & j\in J_k(i)\\ 0, & \text{otherwise}\end{cases} \tag{6}$$

由公式知,长度越短、信息素浓度越大的路径被蚂蚁选择的概率越大。 α 和 β 是两个预先设置的参数,用来控制信息素浓度与启发式信息的权重。

- 遍历每只蚂蚁 ,对每只蚂蚁设置一个起点,遍历每一个城市,利用转移概率轮盘赌选择城市,直到遍历完.最终得到每只蚂蚁的路径矩阵
- 计算当代最优情况
- 更新信息素 ,遍历每个蚂蚁 ,利用那只蚂蚁总路径长度的倒数去涂抹信息素

初始化: $\tau_0 = m/L^{nn}$

更新公式:

$$\begin{aligned} \tau(i,j) &= (1-\rho)\cdot\tau(i,j) + \sum_{k=1}^{m}\Delta\tau^k(i,j) \\ \Delta\tau^k(i,j) &= \begin{cases} (L_k)^{-1}, & (i,j)\in\mathbf{R}^k \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{7}$$

其中, m 是蚂蚁的个数; L^{nn} 是由贪心算法构造的路径的长度; $\rho\in(0,1]$ 是信息素的蒸发率, 通常设置 为 $\rho = 0.5$; $\Delta\tau^k(i,j)$ 是第 k 只蚂蚴在它经过的边上释放的信息素量; L_k 表示路径的长度, 它是 $\mathbf{R}^k$ 中所有边 的长度和.

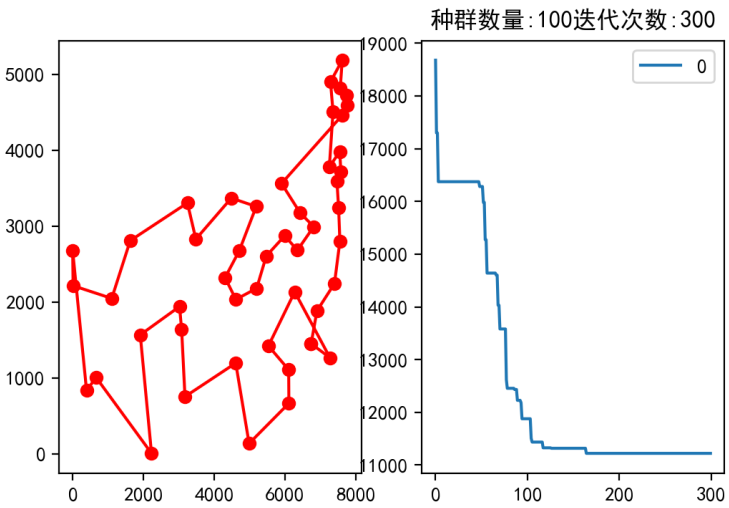
实验结果

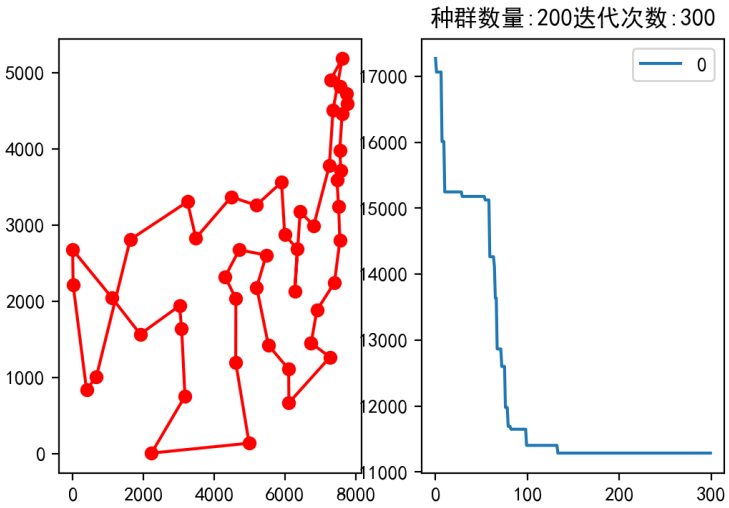
1 参数对比

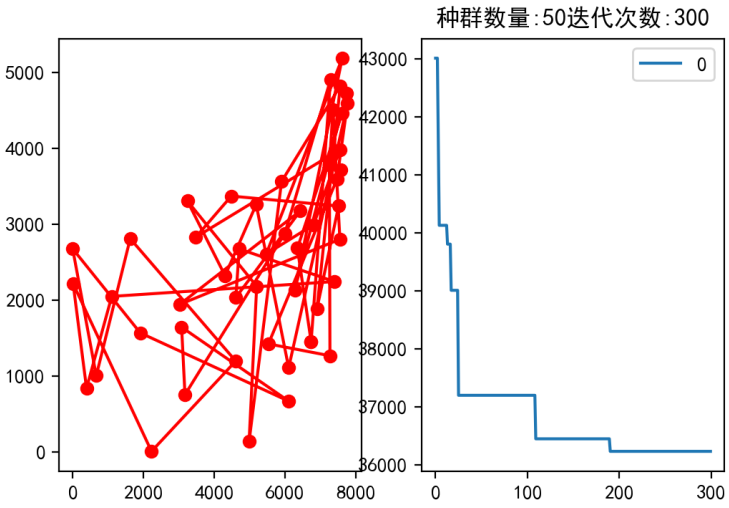
在att48数据集对不同参数进行对比，可得出以下结论:

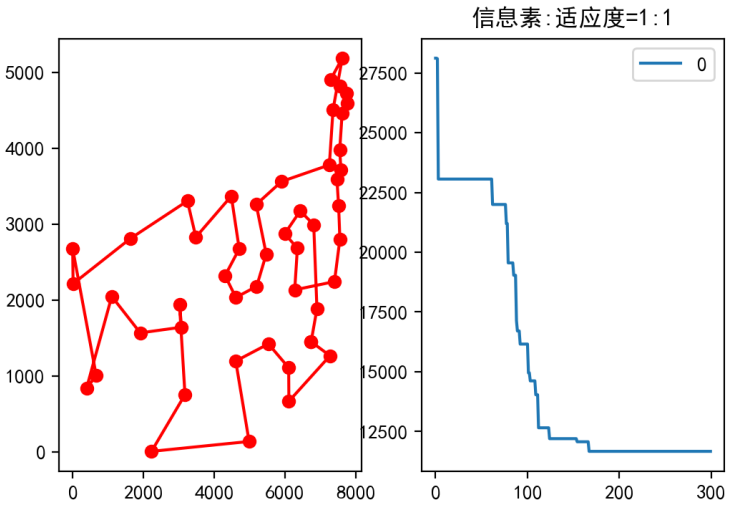
- 蚂蚁数量在一定程度上增加有助于求得更好解，但数量过多时则未必更好
- 仅考虑信息素，而不考虑启发因子，求解质量严重下降。
- 信息素不挥发，易陷入局部最优，求解质量严重下降。

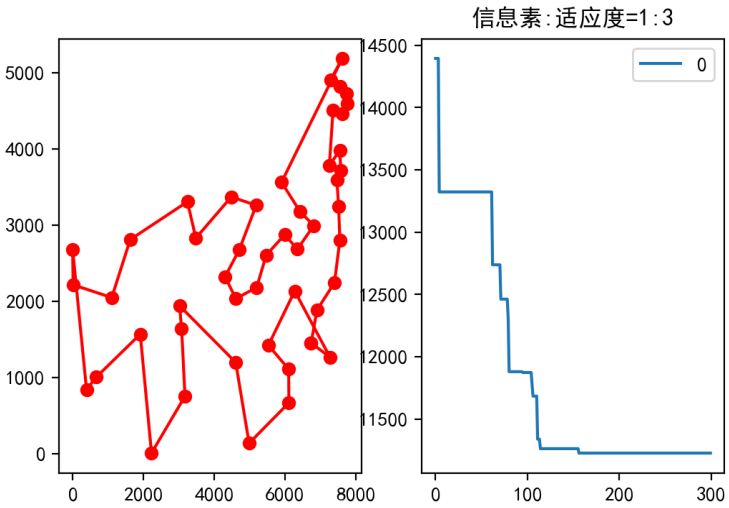
数据集	最优解	信息素重要程度	适应度的重要程度	信息素挥发速度	size_pop	max_iter	我的解	用时	与最优解的差距
att48	10628	1	2	0.1	50	300	11488.76	46.40	860.76
att48	10628	1	2	0.1	100	300	11222.61	130.99	594.61
att48	10628	1	2	0.1	200	300	11285.76	294.98	657.76
att48	10628	1	0	0.1	50	300	36230.20	69.49	25602.20
att48	10628	1	1	0.1	50	300	11670.34	54.98	1042.34
att48	10628	1	3	0.1	50	300	11225.45	81.65	597.45
att48	10628	1	2	0	50	300	15442.38	62.00	4814.38

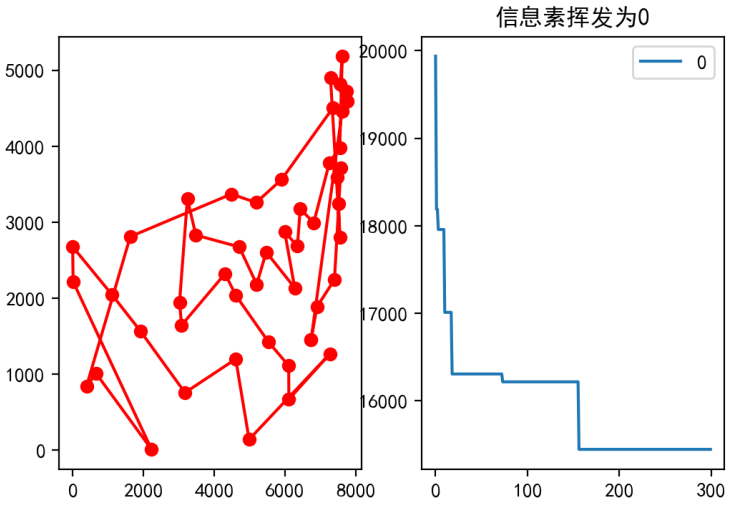

种群数量:100迭代次数:300
0
0
2000
4000
6000
8000
1000
3000
5000
11000
12000
13000
14000
15000
16000
17000
18000
19000
100
200
300


种群数量:200迭代次数:300
0
0
2000
4000
6000
8000
1000
3000
5000
11000
12000
13000
14000
15000
16000
17000
100
200
300


种群数量:50迭代次数:300
0
0
2000
4000
6000
8000
1000
3000
5000
36000
37000
38000
39000
40000
41000
42000
43000
100
200
300







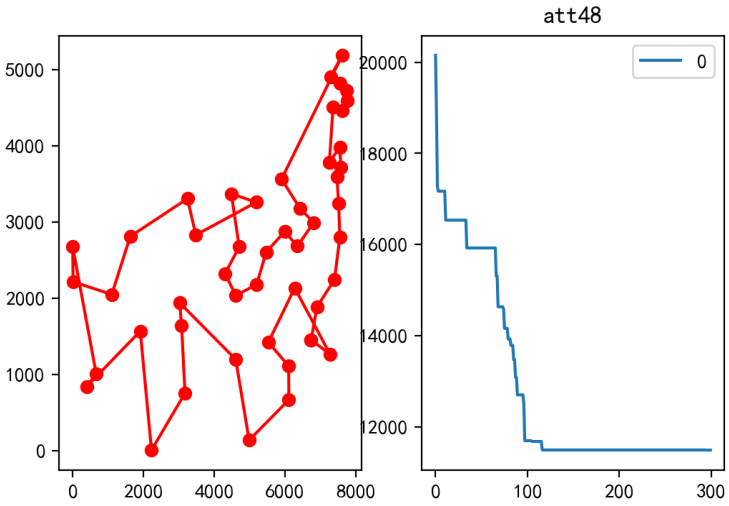
2 实验结果与分析

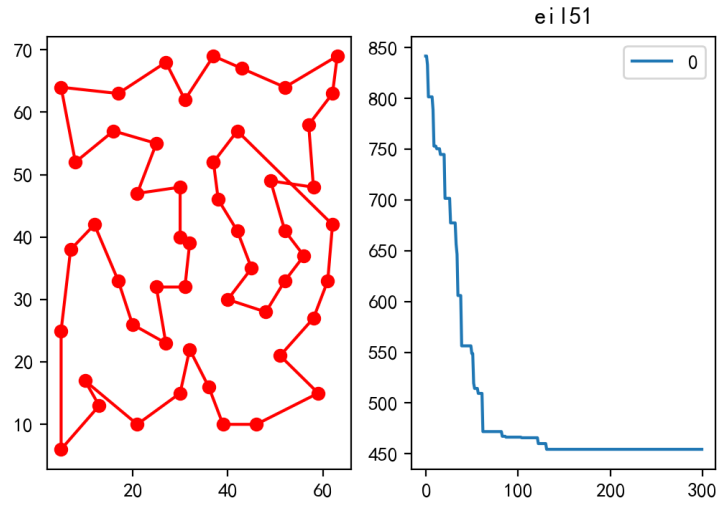
参数为size_pop=50，max_iter=300，信息素重要程度alpha=1，适应度的重要程度beta=2，信息素挥发速度rho=0.1。

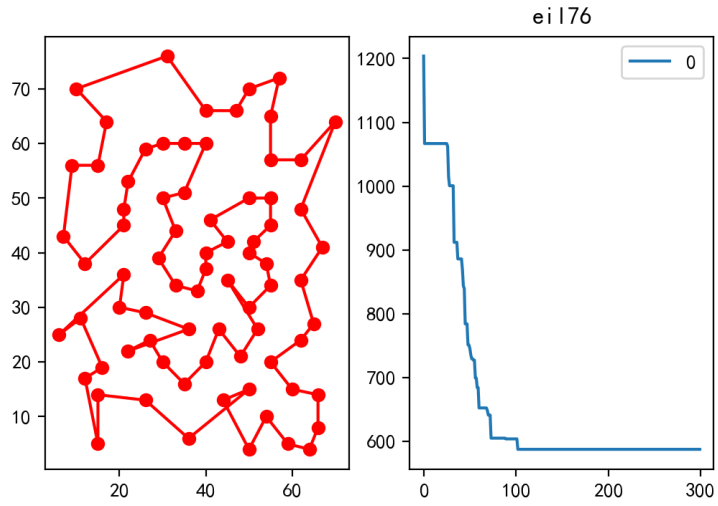
总体来说，算法常在100-200代收敛，所求得的解接近最优解，但仍有一定的差距，或可通过提升蚂蚁数量等方法解决。

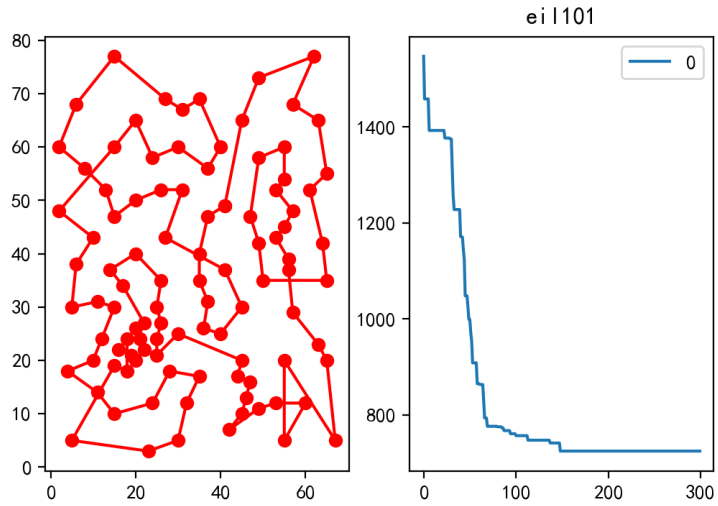
此外随着城市数的增加，算法用时也有显著的增大，数据集计算耗时均在半分钟以上，时间效率稍低。

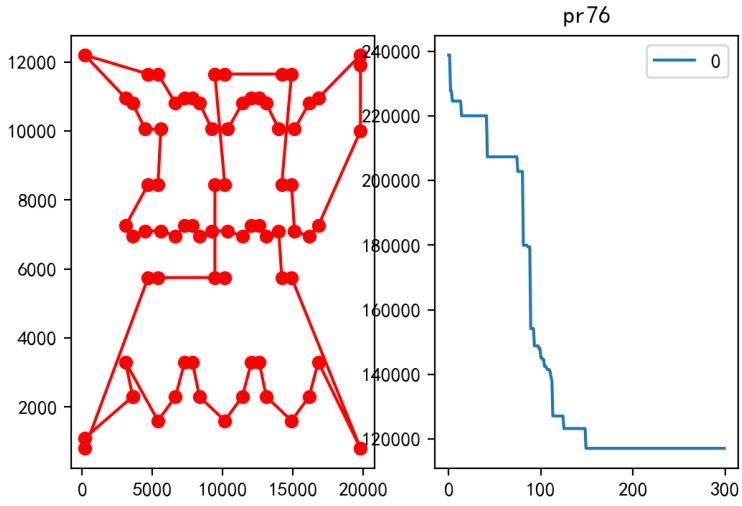
数据集	最优解	size_pop	max_iter	我的解	用时	与最优解的差距
att48	10628	50	300	11488.76	46.40	860.76
eil51	426	50	300	454.36	60.81	28.36
eil76	538	50	300	587.39	109.11	49.39
eil101	629	50	300	724.81	172.34	95.81
pr76	108159	50	300	117118.43	137.87	8959.43
pr107	44303	50	300	48016.21	221.03	3713.21
pr124	59030	50	300	64527.02	142.37	5497.02
pr136	96772	50	300	116592.71	178.13	19820.71
pr144	58537	50	300	62125.74	201.32	3588.74
pr152	73682	50	300	83006.50	152.38	9324.50

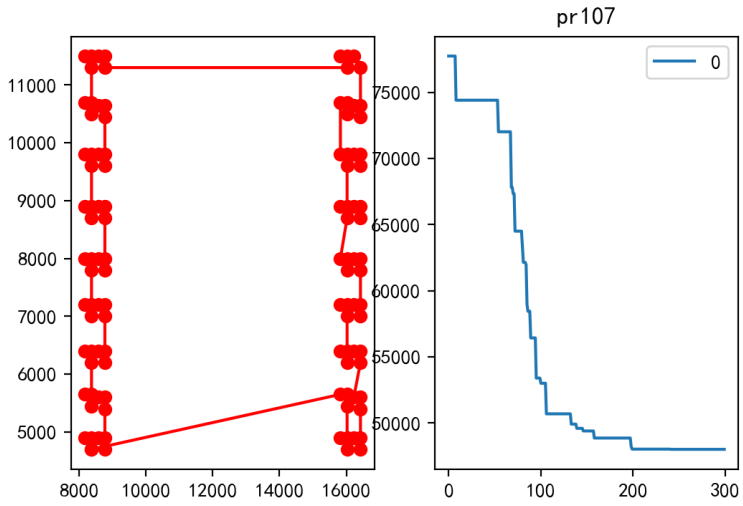


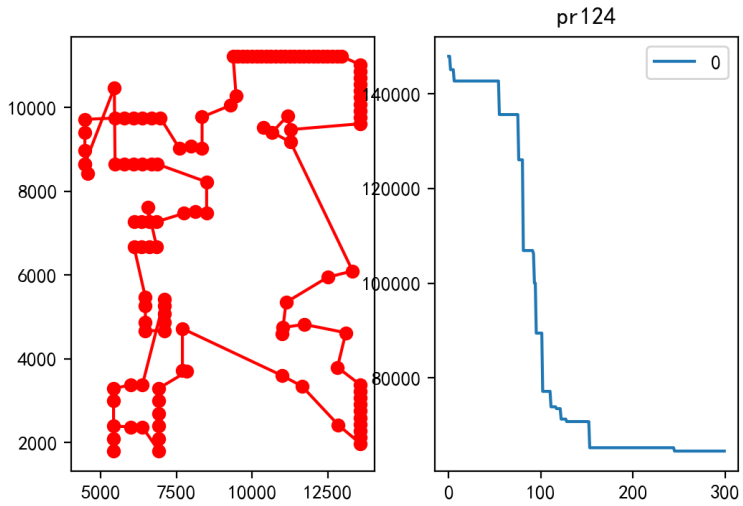


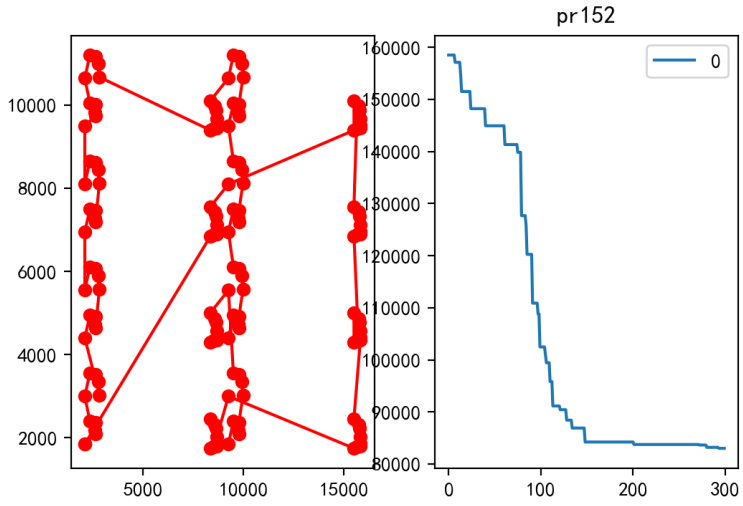

eil76
0
70
60
50
40
30
20
10
20
40
60
1200
1100
1000
900
800
700
600
0
100
200
300


eil101
0
80
70
60
50
40
30
20
10
0
0
20
40
60
1400
1200
1000
800
0
100
200
300


pr76
0
12000
10000
8000
6000
4000
2000
0
5000
10000
15000
20000
240000
220000
200000
180000
160000
140000
120000
0
100
200
300







编程体会

本次实验我利用蚁群算法尝试求解TSP问题,对距离权重,信息素权重,蚂蚁数量等蚁群算法的参数进行分析并实验对比蚁群算法的性能,对于蚁群算法的参数选择有了更多认识.之后对TSPLIB的十个数据集进行实验比对，证明蚁群算法很适合求解TSP问题，算法结果接近最优解，但显而易见的缺点是计算速度有待提高。

在实验过程中，因为不知道部分数据集距离的度量方式不同（其采用伪欧式距离而非欧氏距离），造成了不少不必要的麻烦，这对我也颇有警示作用，即在行动前应当多了解如所使用数据集的一些重要定义等。

总体而言，通过本次实验的学习，我对蚁群算法与TSP问题都有了更深的认识，特别是对蚁群算法思想领悟更加深刻，同时对如何利用智能优化算法求解实际场景中的优化问题有了更多的了解。

核心源代码

1 蚁群算法

```
class ACA_TSP:
    def __init__(self, func, n_dim,
                 size_pop=10, max_iter=20,
                 distance_matrix=None,
                 alpha=1, beta=2, rho=0.1,
                 ):
        self.func = func
        self.n_dim = n_dim  # 城市数量
        self.size_pop = size_pop  # 蚂蚁数量
        self.max_iter = max_iter  # 迭代次数
        self.alpha = alpha  # 信息素重要程度
        self.beta = beta  # 适应度的重要程度
        self.rho = rho  # 信息素挥发速度

        self.prob_matrix_distance = 1 / (distance_matrix + 1e-10 *
np.eye(n_dim, n_dim))  # 路径距离矩阵,避免除零错误，加上一个很小的数

        self.Tau = np.ones((n_dim, n_dim))  # 信息素矩阵，每次迭代都会更新,默认都为
1
        self.Table = np.zeros((size_pop, n_dim)).astype(np.int)  # 某一代每个蚂
蚁的爬行路径,50只*25个城市
        self.y = None  # 某一代每个蚂蚁的爬行总距离
        self.generation_best_X, self.generation_best_Y = [], []  # 记录各代的最
佳情况
        self.x_best_history, self.y_best_history = self.generation_best_X,
self.generation_best_Y  # 历史原因，为了保持统一
        self.best_x, self.best_y = None, None# 最佳情况

    def run(self, max_iter=None):
        for i in range(self.max_iter):  # 对每次迭代
            prob_matrix = (self.Tau ** self.alpha) *
(self.prob_matrix_distance) ** self.beta  # 转移概率，无须归一化。数学公式:
Pij=tau^alpha*Pij^beta
            for j in range(self.size_pop):  # 对每个蚂蚁
                self.Table[j, 0] = 0  # start point，其实可以随机，但没什么区
别,Table[j,0]表示第j只蚂蚁的起点
                for k in range(self.n_dim - 1):  # 蚂蚁按照顺序到达的每个节点
                    taboo_set = set(self.Table[j, :k + 1])  # 已经经过的点和当前
点，不能再次经过
```

```
                    allow_list = list(set(range(self.n_dim)) - taboo_set)  # 在这些点中做选择
                    prob = prob_matrix[self.Table[j, k], allow_list]# 第j只蚂蚁，第k个节点，通往其他可以到达的点的概率
                    prob = prob / prob.sum()  # 概率归一化
                    next_point = np.random.choice(allow_list, size=1, p=prob)[0]# 选择下一个点,np.random.choice是按照概率选择的，p是概率分布,size=1表示只选择一个
                    self.Table[j, k + 1] = next_point

            # 计算距离
            y = np.array([self.func(i) for i in self.Table])# 通过路径矩阵,计算每只蚂蚁的总距离,func是计算距离的函数

            # 顺便记录历史最好情况
            index_best = y.argmin()# 记录最好情况,argmin返回最小值的索引,即最好的路径的蚂蚁的索引
            x_best, y_best = self.Table[index_best, :].copy(), y[index_best].copy()# 找到那只蚂蚁,取出最好的路径
            self.generation_best_X.append(x_best)#记录各代的最佳情况
            self.generation_best_Y.append(y_best)
############更新信息素
            # 计算需要新涂抹的信息素
            delta_tau = np.zeros((self.n_dim, self.n_dim))#初始化新涂抹的信息素 25*25
            for j in range(self.size_pop):  # 每个蚂蚁
                for k in range(self.n_dim - 1):  # 每个节点
                    n1, n2 = self.Table[j, k], self.Table[j, k + 1]  # 蚂蚁从n1节点爬到n2节点
                    delta_tau[n1, n2] += 1 / y[j]  # 涂抹的信息素
                n1, n2 = self.Table[j, self.n_dim - 1], self.Table[j, 0]  # 蚂蚁从最后一个节点爬回到第一个节点
                delta_tau[n1, n2] += 1 / y[j]  # 涂抹信息素

            # 信息素飘散+信息素涂抹,rho是挥发率,挥发之前的信息素
            self.Tau = (1 - self.rho) * self.Tau + delta_tau#公式为：Tau(t+1)=(1-rho)*Tau(t)+delta_tau

        best_generation = np.array(self.generation_best_Y).argmin()#找出最好的一代,argmin()返回最小值的索引
        self.best_x = self.generation_best_X[best_generation]#最佳路径
        self.best_y = self.generation_best_Y[best_generation]#最佳距离
        return self.best_x, self.best_y

    fit = run

def cal_total_distance(routine):# 计算总路程,routine是城市列表
    num_points, = routine.shape# 路径中城市数
```

```
    return sum([distance_matrix[routine[i % num_points], routine[(i + 1) % num_points]] for i in range(num_points)])# 遍历坐标点,计算总路程

```

2 可视化

```

df = pd.read_csv(r'./data/pr152.tsp',sep="",skiprows=6,header=None,encoding='utf8')#读取城市坐标
city = np.array(df[0][0:len(df)-1])  # 最后一行为EOF, 不读入
city_name = city.tolist()
city_x = np.array(df[1][0:len(df)-1])
city_y = np.array(df[2][0:len(df)-1])
city_location = list(zip(city_x, city_y))
city_location=np.array(city_location)
num_points = len(df)-1 #点的数量
t1=time.time()
distance_matrix = spatial.distance.cdist(city_location, city_location, metric='euclidean')# 初始化路线,spatial.distance.cdist()计算两个数组之间的距离,metric='euclidean'表示欧式距离
aca = ACA_TSP(func=cal_total_distance, n_dim=num_points,
              size_pop=30, max_iter=300,
              distance_matrix=distance_matrix)#设置参数,并初始化
best_x, best_y = aca.run()#运行,得到最佳路径和最佳距离
t2=time.time()

# %%
aca.y_best_history=np.array(aca.y_best_history)
fig, ax = plt.subplots(1, 2)#创建一个包含两个子图的窗口
best_points_ = np.concatenate([best_x, [best_x[0]]])#首尾相连成回路,26个点
best_points_coordinate = city_location[best_points_, :]#把点的标号转为实际坐标,方便绘图
ax[0].plot(best_points_coordinate[:, 0], best_points_coordinate[:, 1], 'o-r')#画出最佳路径
pd.DataFrame(aca.y_best_history).cummin().plot(ax=ax[1])#.cummin()函数用于计算数组中各个值的累积和,即最短距离的变化情况
plt.title('pr152')
plt.show()
# print(best_x)
print(best_y)
print(t2-t1)
```