** **

**课程实验报告**

课程名： 《人工智能》课程大作业

学 院： 信息工程学院 系 计算机科学与技术系

专 业： 计算机科学与技术

班 级： 计算机科学与技术181班

学 号： 6105118107

姓 名： 刘涔宇

任课教师： 黄伟

授课学期： 2020 年~~~~ 2021 年 2 学期

**旅行商问题的求解**

摘要：TSP问题（旅行商问题）是组合优化问题中最典型的NP完全问题之一，其具有很重要的理论和研究价值。本文讨论了传统算法例如动态规划(DP)、流行的智能算法例如模拟退火算法(SA)、蚁群算法(ACO)、遗传算法(GA)以及神经网络求解例如自组织映射(SOM)、指针网络(PN)。并且对ACO和GA进行混合改进，对标准SOM利用泛化竞争和局部渗透的机制也做出了改进，提出了未来求解TSP问题的发展趋势。

**一、研究背景与意义**

旅行商问题(Traveling Salesman Problem，简称TSP)是一个经典的组合优化问题。TSP问题最早由Dantzig等提出，并迅速成为最优化领域研究的热点，成为许多优化算法的一个测试基准[1]。经典的TSP可以描述为：有n个城市，任何两个城市之间的距离都是确定的，现要求一旅行商从某城市出发必须经过每一个城市且只能在每个城市逗留一次，最后回到原出发城市，问如何事先确定好一条最短的路线使其旅行的费用最少。从[图论](https://baike.baidu.com/item/%E5%9B%BE%E8%AE%BA)的角度来看，该问题实质是在一个带权重的完全无向图中，找到一个权值总和最小的哈密顿回路。由于该问题的可行解是所有顶点的[全排列](https://baike.baidu.com/item/%E5%85%A8%E6%8E%92%E5%88%97)，随着顶点数的增加，会产生组合爆炸，它是一个NP完全问题。

现实生活中有很多问题都可以归结为旅行商问题，其具有较为广泛的工程应用和现实生活背景，比如VLSI芯片设计、物流货物的配送、飞机航线的安排、公路网络的建设、车辆选路、装配线上的螺帽问题和产品的生产安排问题等。因此，如何快速、有效的解决TSP问题具有很高的理论意义和实际意义[2]。

**二、国内外相关研究方法综述**

TSP问题求解的方法目前可以大致分类两大类：精确算法和近似算法。精确算法包括：暴力穷举法、动态规划法等，其缺点很明显，计算时间复杂度呈指数级别，随着路径结点的增加，问题的解空间规模会爆炸增长，且至今为止仍然没有找到多项式时间算法。为了解决大规模的TSP问题，不断的有学者提出用近似算法去无限逼近最优解的方法。

1953年，Metropolibs首先提出模拟退火算法(SA)，但直到1983年Kirkpatrick等才成功地将退火思想引入组合优化领域[3]。模拟退火算法采用类似于物理退火的过程。先在一个高温状态下，然后逐渐退火，在每个温度下慢慢冷却，最终达到物理基态，即找到近似最优解。其缺点在于参数难度控制，使用不当容易陷入局部最优。

1975年，美国密西根大学J.Holla教授首先提出遗传算法[4](Genetic Algorithm)。此算法基于达尔文生物进化论，通过模拟自然界中的生物以优胜劣汰、适者生存的自然进化规则生存和繁衍，并逐步产生出对其生存环境适应高的优良品种，从而获得越来越好的近似解。遗传算法的优点是隐含并行性和全局解空间搜索，但实际运用时容易出现早熟收敛和收敛性能差等缺点。

1996年，意大利学者Dorigo M等首先提出蚁群算法[5](Ant Colony Optimization)。此算法通过模拟蚂蚁寻找食物过程中发现路径的行为，蚂蚁在运动过程中，会在其经过的路径上留下信息素，同时信息素会随着时间的推移而逐渐挥发，蚂蚁会根据信息素的浓度选择运动方向并最终到达食物所在的地方。在较短的路径上信息素的浓度高，更多蚂蚁会选择此路径，从而导致该路径的信息素浓度更高，吸引更多蚂蚁，通过这种正反馈机制，最终找到最优路径。蚁群算法与其它启发式算法相比，在求解性能上具有很强的搜索较好解的能力，并且很容易与多种启发式算法结合，以改善算法性能。缺点是参数太多，参数的设置又对结果有决定性影响。

1988年Angeniol等首次将自组织网络(SOM)应用于TSP问题，并且SOM网络在解决城市数目比较大的问题中虽然解的质量有所下降但是复杂度大幅降低[6]。张军英等人通过泛化竞争和局部渗透两个新的学习机制，改进了标准SOM网络(ORC\_SOM)使得路径尽可能接近最优解[7]。近年来也出现了使用指针网络和强化学习的方式去求解TSP问题，但是其训练时间较长，并且由于实际数据并不是简单的正态分布或均匀分布，实际训练数据获取困难导致训练集成本高且样本少，导致在实际场景中表现不佳，相比于传统算法没有展现出其优势[8]。

本文采用了混合遗传蚁群算法作为求解小中规模的TSP问题，而大规模的TSP问题采取了ORC\_SOM网络进行求解。

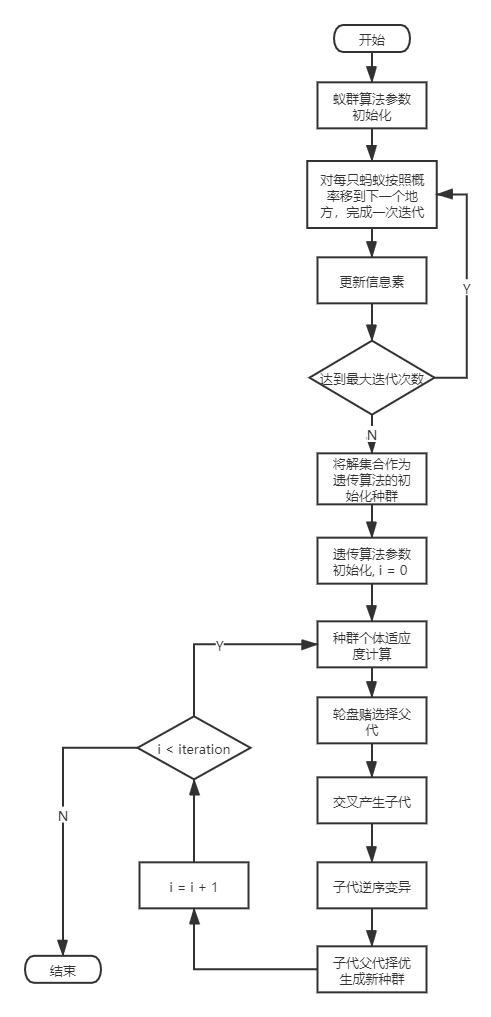
**三、研究方法**

**（一）小中规模TSP问题**

使用混合蚁群遗传算法作为此类规模的解决方法，通过实验发现，使用随机生成编码序列的种群初始化的遗传算法，因为初代种群的适应度很低，即路径和初始就很大，从而导致了一开始繁殖子代的效果并不好，即种群最优适应度也很低，而在使用蚁群算法解决TSP问题的时候发现，通过少数几次全局蚂蚁的循环就已经可以获得一个较为优秀的解集。所以利用蚁群算法作为遗传算法的初始化种群输入，可以减少遗传算法的寻优次数，加快遗传算法求近似最优解的速度。混合蚁群遗传算法工作流程如下：

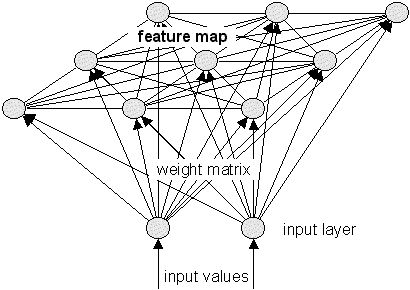
1. 蚁群算法初始化参数。蚁群规模m(一般设定为城市数量的1.5倍)、信息素因子α(根据经验一般取[1,4])、启发函数因子β(根据经验一般取[3,4.5])、信息素挥发因子ρ(根据经验一般取[0.2,0.5])、信息素常量Q(根据经验一般取[10,1000])、最大迭代次数设置为2次。
2. 构建解空间。将各个蚂蚁随机地放置于不同的出发点，对每个蚂蚁k（k=1，2，……，m），计算其下一个访问的城市的概率，设表示蚂蚁从第i个城市到第j个城市的概率，表示第i个城市到第j个目标的信息素，表示第i个城市到第j个城市的距离，则每个城市选择的概率公式为：。再通过轮盘赌法来选择下一个城市，直到所有蚂蚁访问完所有的城市。
3. 更新信息素。计算各个蚂蚁经过的路径长度，记录当前迭代次数中的最优解（最短路径）。同时，对各个路径上的信息素浓度进行更新，更新公式为：，其中(t)表示之前路径中留下来的信息素，表示蚂蚁此次迭代释放在路径上的信息素，且。
4. 保存结果。将当前解空间作为遗传算法的输入。
5. 遗传算法初始化参数。种群规模num\_total（这里和蚁群规模m保持一致）、迭代次数iteration、变异概率mutate\_radio，交叉概率crossover\_ratio。初始种群直接使用④中的解。
6. 计算种群中每个个体的适应度。TSP问题的目标是找到路径和最小的个体，所以适应值函数选择为编码路径和的倒数，即 ，路径和越大，适应值越小，即越应该淘汰。
7. 根据交叉概率，选择此代应该留下来的个体作为父代去繁殖，并且根据轮盘赌法选择父代1和父代2。轮盘赌算法：设群体大小为N，个体的适应度为，则个体的选择概率为：和累积概率为。通过在[0,1]生成的随机数r和比较，判断哪两个父代留下。
8. 交叉操作，随机交叉父代的基因编码产生子代1和子代2，交叉方法为：随机生成两个城市点start和end，然后交换父代start和end之间的序列即得子代1和子代2。
9. 变异操作,随机产生一个概率若小于变异概率则，随机产生两个城市点start和end，对子代两点之间的城市序列逆序排列从而进行变异。
10. 子代父代结合择优将适应度高的个体作为新种群，判断是否满足最大迭代数，若不满足则跳转到⑥继续执行，记录下每次迭代的最好个体,直到达到最大迭代数。
11. 输出最好个体的基因序即路径。

算法流程图如下：

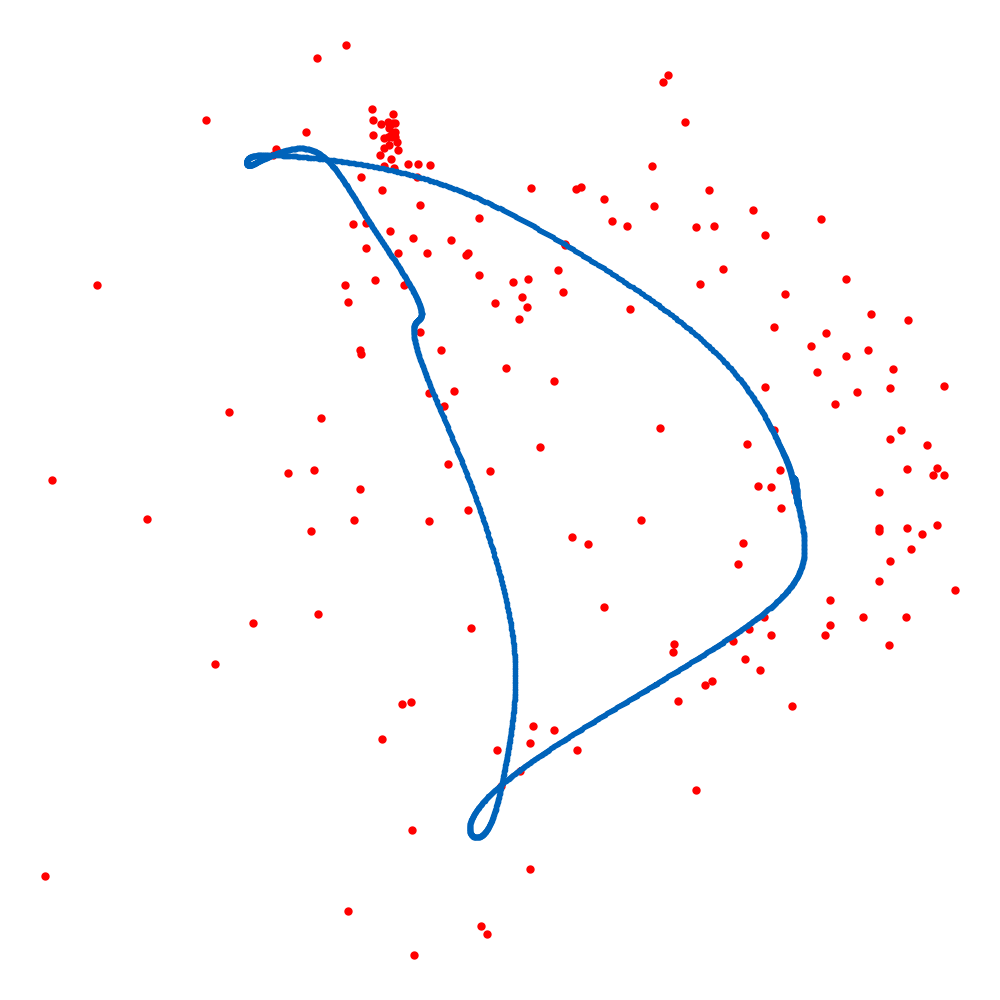


**（二）大规模TSP问题**

使用ORC\_SOM网络作为此类规模的解决方法，首先介绍一下SOM网络，自组织映射（SOM）是一种无监督学习的神经网络，用来解决旅行商问题（TSP）时，二维城市坐标是神经网络的输入，城市空间位置关系是神经网络要学习的模式，一个环形的神经元结构是神经网络的输出。SOM 自组织映射的结构：输入层、全连接（权值矩阵）、输出层（特征映射）。如图：



SOM算法的主要原理就是每次找到离城市最近的神经元，该神经元被称为优胜神经元，然后以该神经元建立一个高斯分布逐渐更新其他神经元的位置，也就是更新输出神经元权值向量，经过多次反复的竞争与更新，神经网络最终学到输入数据的模式（城市的空间关系），并以神经元权值向量的形式保存下来。体现在地图中，就是神经元的位置不断靠近城市，最终与之重合，从而得到了最优路径。如图：



而ORC\_SOM网络引入了泛化竞争和局部渗透这两个新的学习机制，(1)当输入城市与获胜神经元之间的距离较大时(表明尽管该神经元获胜, 但仍不一定将该城市映射到该神经元上), 应使获胜神经元及其邻域神经元向输入城市的位移量减少, 从而允许各个神经元更加公平地参与竞争, 我们称这一机制为泛化竞争机制; (2)当输入城市与获胜神经元之间的距离较小时(表明应该将城市映 射到该神经元上),应使获胜神经元及其邻域神经元向输入城市的位移量增大,从而弱化其他神经元对当前输入城市的竞争而强化获胜神经元及其邻域神经元对当前输入城市的竞争, 我们称这一机制为局部渗透机制[7]。其工作流程如下：

1. 初始化神经元的位置。使用Vieira等人已经证实有效的神经元初始化方式，即将神经元有序的分布在城市外围边框矩阵中[9]。这里神经元的个数采用的是城市个数的3倍。
2. 确定获胜神经元。使用欧几里得距离测量相似性，随机挑选一个城市，找到距离输入城市坐标位置距离最近的神经元，其就是获胜神经元。
3. 更新优胜领域内神经元的权值。引入渗透半径λ作为输入城市与获胜神经元之间距离大小的判断门限。如果它们之间的距离小于λ, 则采用局部渗透机制;反之则采用泛化竞争机制,所以显然λ是应该随着时间由小到大逐步扩张的，。其中取神经元初始矩形框的对角线长度的1/4， 。所以神经元更新规则为：,其中表示当前神经元，表示上一次迭代的神经元的值，表示学习率，表示以该神经元为中心的高斯分布，表示神经元离当前被选中城市的距离。表示高斯分布的方差，该方差初始化为城市个数的十分之一。
4. 在每一次迭代的过程中，和都随着迭代次数增加而不断减小，最终收敛。。分别是对应的衰减率。
5. 判断迭代次数是否到达上限，如果是，则转到步骤②。否则迭代次数到达迭代上限，找到所有城市和其对应的获胜神经元，按获胜神经元的顺序对城市排序就是神经网络找到的最短路径。

**四、实验结果与分析**

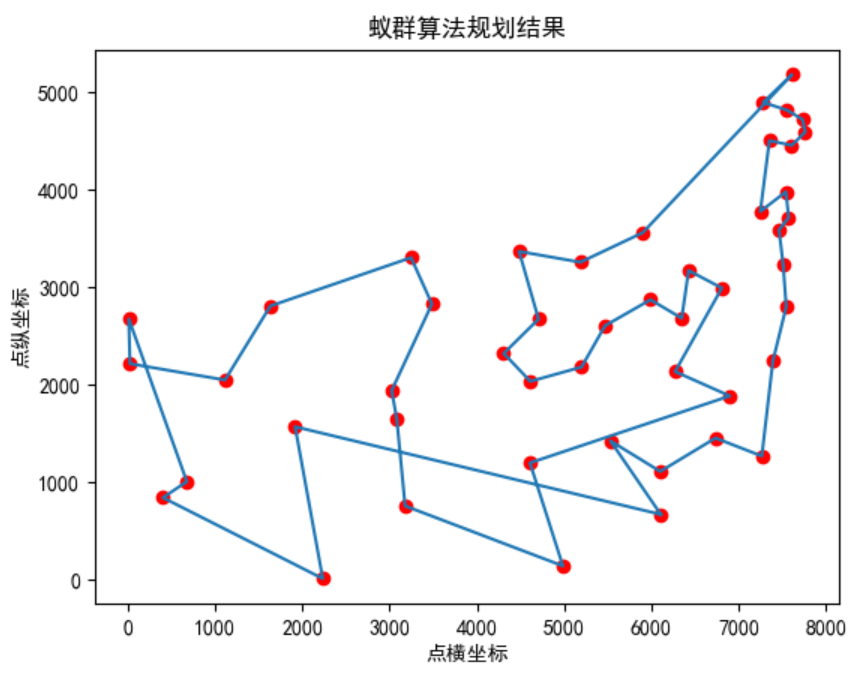
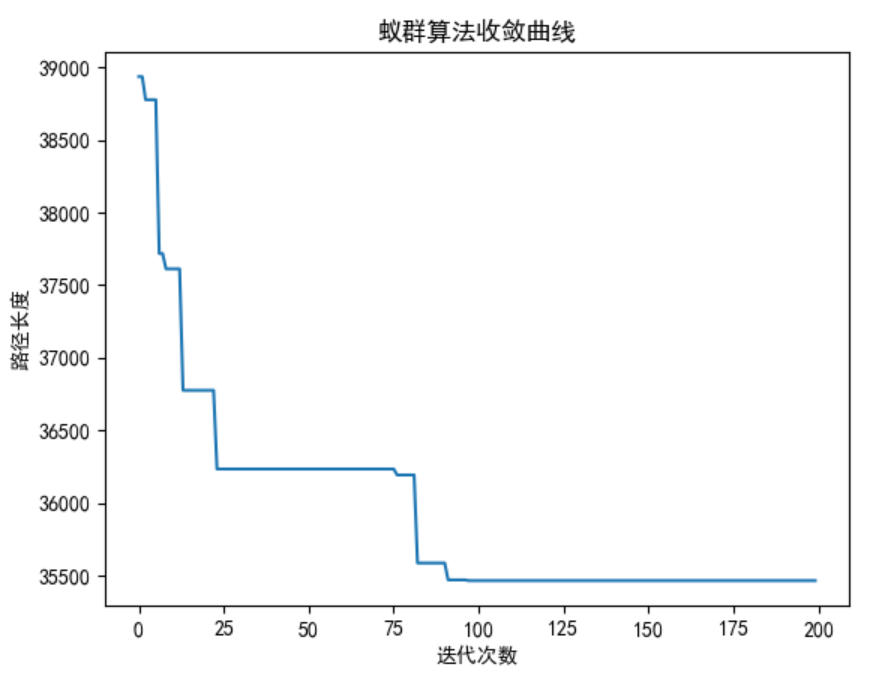
实验中所使用的数据集都是TSPLIB中的。

**（一）小中规模TSP问题**

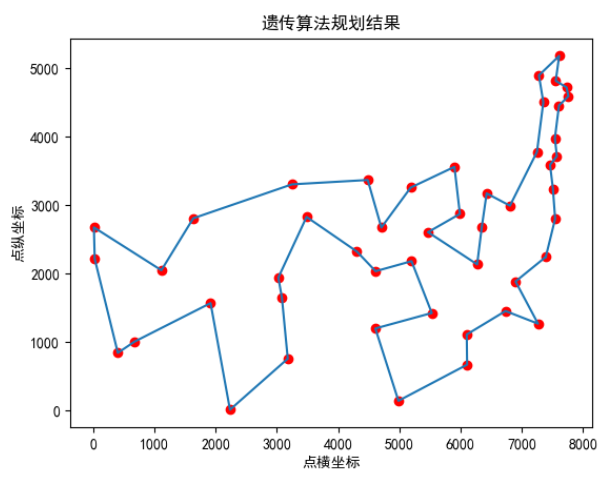
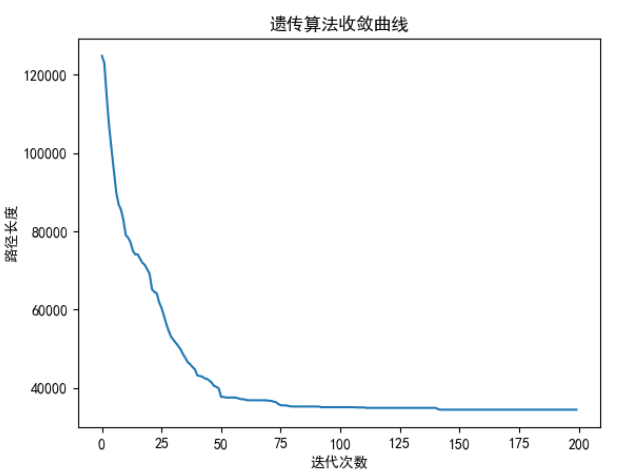
实验1：测试混合遗传蚁群算法在TSP中的att48问题中相对于蚁群算法、遗传算法的性能，att48问题最优解为:33522。结果如下：

|  |
| --- |
| 迭代次数 蚁群算法 遗传算法 混合蚁群遗传算法 |
| 30 36234.986 53106.558 35853.551 |
| 50 36234.986 39888.990 34343.238 |
| 200 35467.076 34468.175 33828.076 |

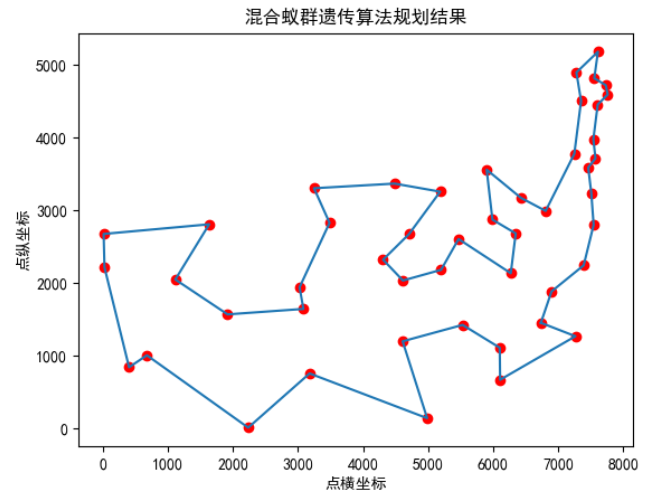
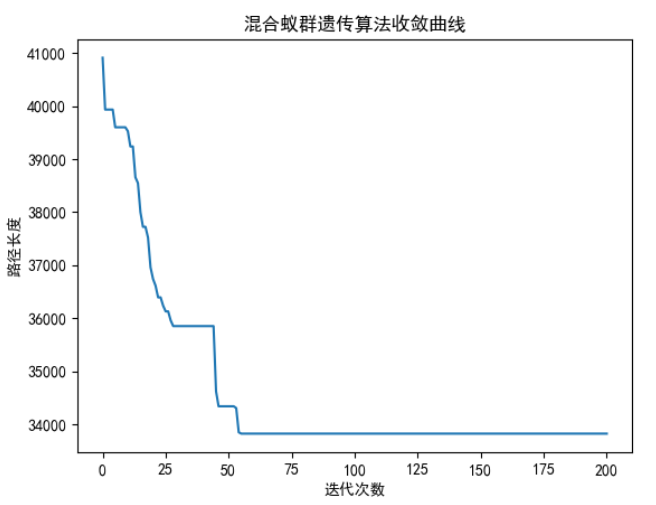
蚁群算法收敛曲线和规划结果如下图：



遗传算法收敛曲线和规划结果如下图：



混合蚁群遗传算法收敛曲线和规划结果如下图：



结果分析：

1、比较上述三种算法的收敛曲线图：可以发现混合算法相比于普通遗传算法的起始路径长度(迭代次数为0时)大大减少了，而蚁群算法的起始路径长度就低得多了，这也是我们一开始为遗传算法使用蚁群算法初始化优化的想法。

2、比较上述三种算法的收敛曲线图：可以发现混合算法的路径长度在迭代次数[50,75]区间内就已经开始趋向于稳定，而遗传算法的路径长度在迭代次数150左右才趋向于稳定，而蚁群算法的路径长度在迭代次数100左右才趋向稳定，所以混合算法确实可以大大减少遗传算法的收敛时间，加快遗传算法的收敛，这和使用蚁群算法作为初始种群输入密不可分，因为初始种群的路径长度小说明种群的适应值高，初始种群相对于普通遗传算法的随机初始化而言就优秀很多，所以导致了收敛的加快。

实验2：测试混合蚁群遗传算法应用于ulysses16、att48、eil76、等不同规模的TSP问题，研究具体问题的规模对混合蚁群遗传算法的影响。

在att48问题上，混合蚁群遗传算法取得了不错的效果，相比于普通遗传算法，它的精度变高，趋向稳定的需要迭代次数也更少。

|  |  |  |
| --- | --- | --- |
| 序号 | ulysses16 | eil76 |
| 1 | 73.999 | 567.987 |
| 2 | 73.987 | 572.094 |
| 3 | 73.999 | 566.779 |
| 4 | 74.593 | 565.802 |
| 5 | 74.870 | 573.266 |
| 6 | 74.001 | 558.367 |
| 7 | 74.593 | 562.654 |
| 8 | 73.987 | 567.530 |
| 9 | 74.108 | 573.714 |
| 10 | 73.987 | 575.328 |
| 平均最优值 | 74.212 | 568.352 |

实验表明：ulysses16的TSP问题中，我使用动态规划求解，其路径长度的最优值就是73.987。而使用混合蚁群遗传算法求解时，多次达到了此最优解，而平均最优值也有74.212，这表明混合算法在解决小规模的TSP问题的时候很优秀，精度也很高。eli70问题，混合蚁群遗传算法最好的最优解是568.352，离官方最优解（538）有一定差距，但差距不大，这表明在规模稍大的问题中，混合蚁群遗传算法仍有用武之地。

实验3测试混合蚁群遗传算法的稳定性。

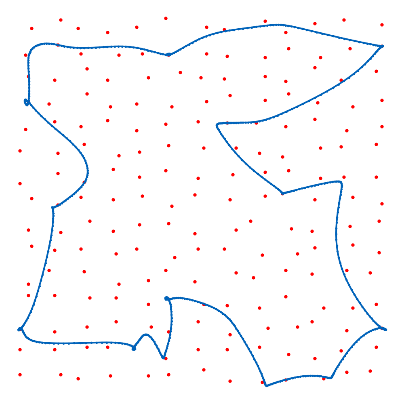
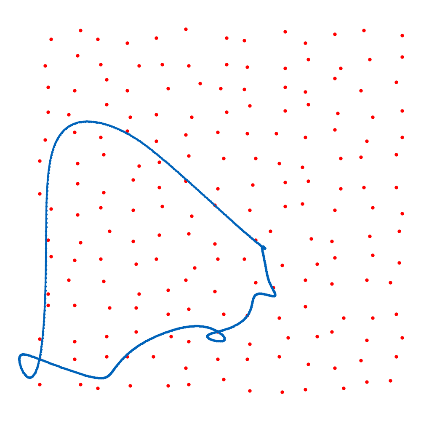
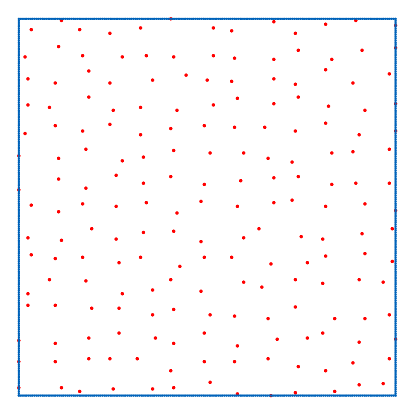
一个算法性能的好坏，除了要求求解最优解的精度，也需要算法具有较高的稳定性，此处对此混合蚁群遗传算法针对eli76.tsp问题数据进行20次求解，并对结果进行分析。见下表。

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 最优路径大小 | 序号 | 最优路径大小 |
| 1 | 567.641 | 11 | 567.238 |
| 2 | 575.144 | 12 | 554.560 |
| 3 | 568.532 | 13 | 576.145 |
| 4 | 571.471 | 14 | 566.673 |
| 5 | 566.763 | 15 | 583.339 |
| 6 | 564.404 | 16 | 572.008 |
| 7 | 570.100 | 17 | 572.754 |
| 8 | 570.862 | 18 | 564.762 |
| 9 | 569.049 | 19 | 562.737 |
| 10 | 566.205 | 20 | 576.740 |

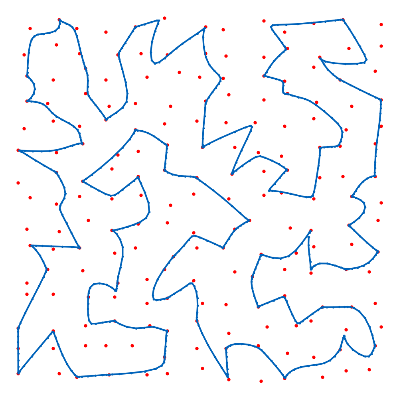
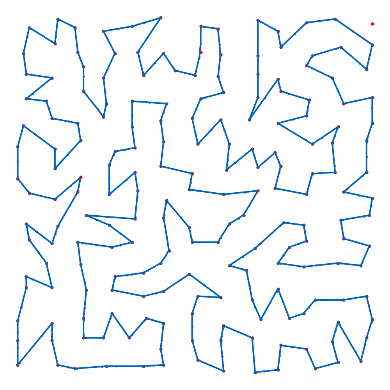
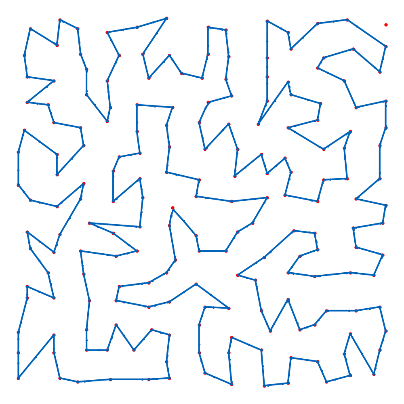
从表中可知，20次计算结果的平均值为569.35635，从表中数据来看，大多数的数据都集中在这个范围内，因此本文认为此混合蚁群遗传算法是稳定的。

**（二）较大规模TSP问题**

实验1:使用ORC\_SOM网络进行解决rat195.tsp问题。

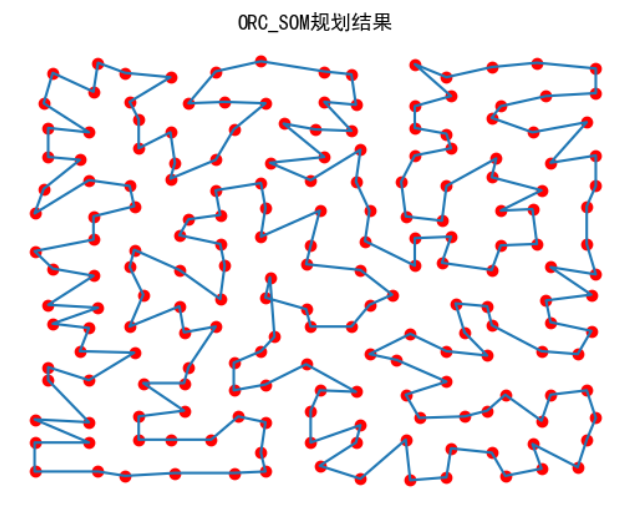
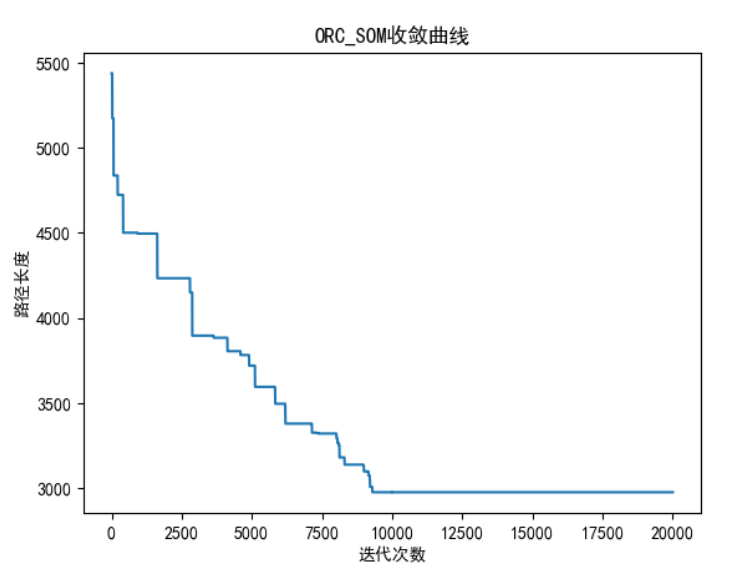


(a)初始化 (b)寻优的中间过程 (c)寻优的中间过程

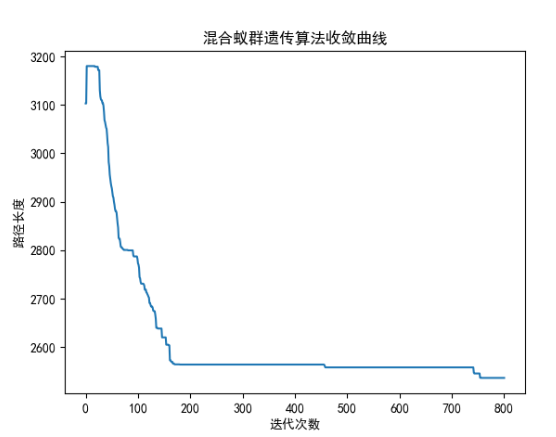
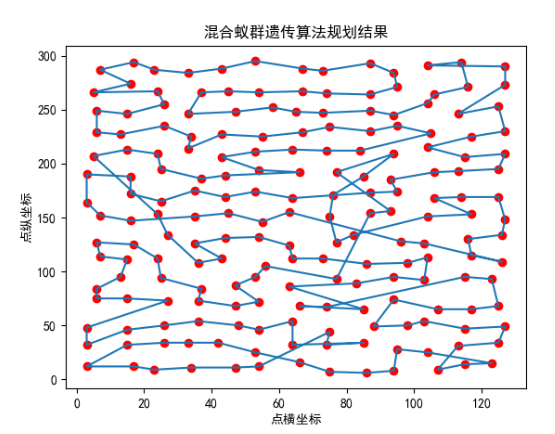
(d)寻优的中间过程 (e)寻优的中间过程 (f)寻优的中间过程

结果如下：



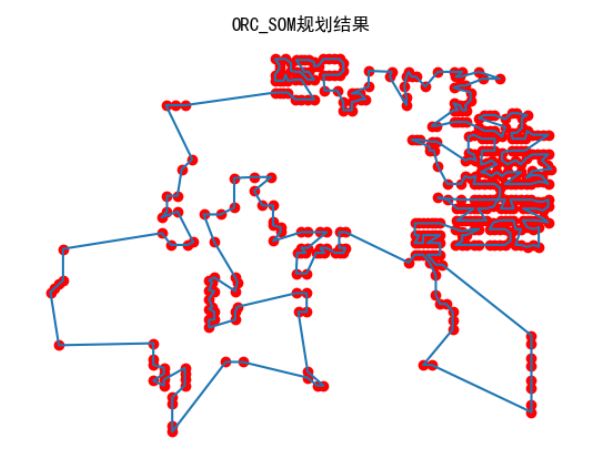
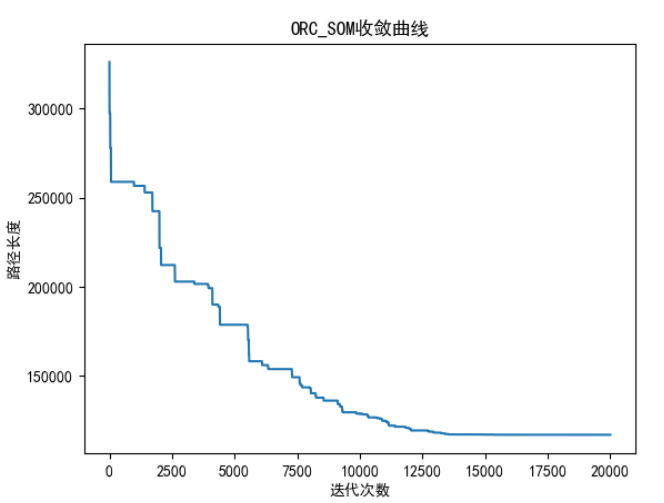
rat195.tsp数据集官方解为：2323。所得路径最优解为: 2677.424，消耗时间145s

而使用混合蚁群遗传算法，所得路径最优解为：2536.471，消耗时间564s

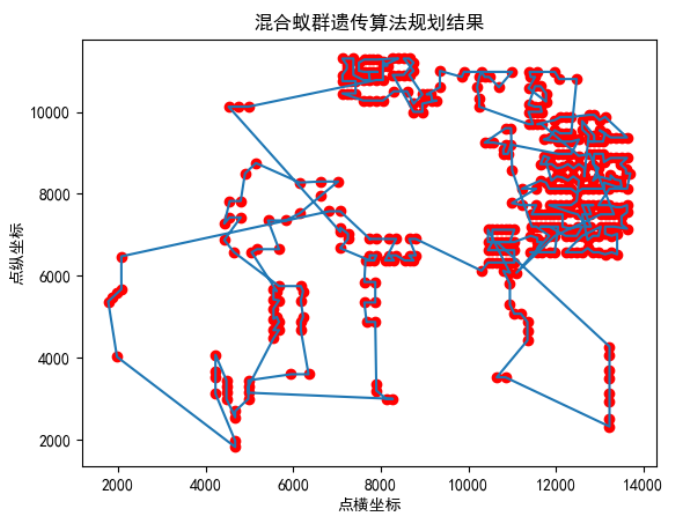
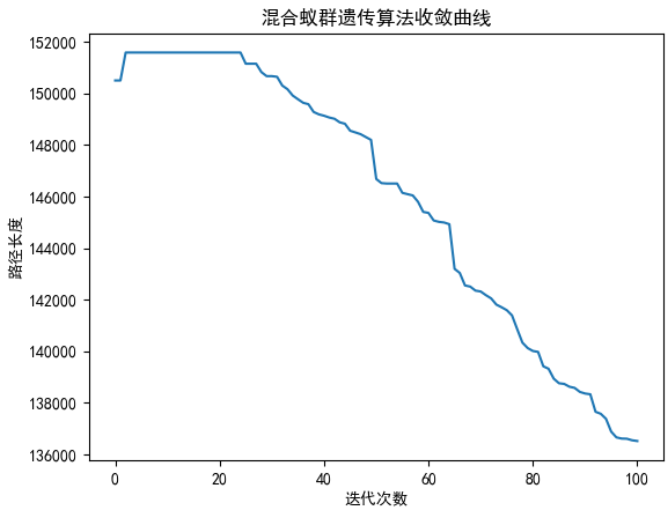
可以发现ORC\_SOM网络在中规模的TSP数据集上的最优解虽然没有混合蚁群遗传算法高，但也差不多，并且消耗的时间很少（可以通过增大神经元的个数和迭代次数来提高精度）。

实验2：使用ORC\_SOM网络进行解决pr439.tsp问题求解实例结果如下：



结果：对于pr439.tsp数据集，官方最优解为107217，而ORC\_SOM算法的最优解为112957.888，消耗用时831s，尽管存在一定的误差，但是我使用混合蚁群遗传算法求解此数据集，消耗了2225s才求解到128617.162的精度。相比之下其消耗时间相对于传统启发式算法降低了很多。

在和ORC\_SOM算法控制相同消耗时间下，混合蚁群遗传算法的收敛曲线和规划路径结果如下图，并且其最优路径求解为：136520.172。



可以发现混合蚁群遗传算法在较大规模TSP问题下的求解效率低于ORC\_SOM，并且想要获得和ORC\_SOM算法相同精度的结果需要消耗更多的时间，这本质上是因为混合算法仍然是遗传算法的变种，并不能改变其在较大规模TSP问题下收敛速度慢的情况。

**五、结论与展望**

结论：本文从不同规模的TSP问题出发，给出了求解不同规模的TSP问题的两种方案，对于小中规模：使用混合蚁群遗传算法，该方法相对于传统的遗传算法，求解精度更高，耗时更少，具有更好的收敛速度。对于较大规模：使用基于泛化竞争和局部渗透改进的自组织网络(ORC\_SOM)，求解精度相对于传统的启发式算法偏弱，但是耗时相对大大减少。

展望：由于时间和水平的限制，本文对于TSP问题的研究有限。TSP问题长期以来一直吸引着众多的研究人员从事对其算法的改进工作，以达到在较低的时间复杂度下能得到更高的求解精度。对于传统启发式算法的改进，可以通过与其它方式的混合从而改进，但这一般都只能提高原算法的收敛速度，对于求解精度的提升很小。而对于ORC\_SOM虽然通过传统SOM学习保证了路径的全局寻优，同时又通过竞争泛化和渗透确保了路径在局部的优化，从而使得算法能够找到更为接近最优的解，但是其求解精度还是不够高，如何提高自组织网络的精度是下一步的研究方向。而近些年来由于神经网络的火热，又提出了基于指针网络和强化学习去求解组合问题，但是其训练集代价太大，并且训练样本的获取是通过传统算法获得的，所以精度不能突破传统算法的上限，训练时间也高，但此种算法有一个很大的优点，就是对于不同规模的城市不需要重复训练，直接修改输入数据就可以获得结果，所以如何减少指针网络的训练代价是下一步的研究方向。

**六、参考文献**

[1]Flood M M. The Traveling-Salesman Problem[J]. Operations Research, 1956, 4(1):61-75.

[2]于莹莹,陈燕,李桃迎.改进的遗传算法求解旅行商问题[J].控制与决策,2014,29(08):1483-1488.

[3]Lin W . Qian C J. A daptive Control of Nonlinearly Parameterized Systems: A Nonsmooth Feedback Framework[J]. IEEE Trans on Automatic Control, 2002,47(5) : 757-774.

[4]Holland J H . Adaptation in Natural and Artificial Systems[M]. The University of Michigan Press. 1975.

[5]Dorigo M. Maniezzo V.Colorni A. The Ant System:Optimization by a Colony of Cooperating Agents[J] . IEEE Trans on Systems. Man. and Cybernetics-Part B. 1996. 26 (1): 29-41.

[6]ANGENIOL B , de La CROIX VAUBOIS G, Le TEXIER J Y. Self-organizing feature maps and the travelling salesman problem [J].Neural Networks, 1988,1(4): 289-293

[7]张军英,周斌.基于泛化竞争和局部渗透机制的自组织网TSP问题求解方法[J].计算机学报,2008(02):220-227.

[8]康康. 基于强化指针网络的TSP问题的求解与优化[D].华中科技大学,2019.

[9]Vieira F C, Neto A D D. An efficient approach to the traveling salesman problem using self-organizing maps. International Journal of Neural Systems, 2003, 13(2): 59-66.

**七、附件**

大作业github地址：<https://github.com/liucenyu>。项目名为：人工智能大作业

混合蚁群遗传算法代码：

|  |
| --- |
| import random import math import numpy as np import matplotlib.pyplot as plt import time  *# 存储存储每次迭代的最佳路径，画出收敛图* iter\_x = [] iter\_y = [] *# 存储每次迭代的最佳路径选择，画出规划结果* iter\_path = []  class GA(object):  def \_\_init\_\_(self, num\_city, num\_total, iteration, data, aco\_init): *#* self.num\_city = num\_city *# 城市数量* self.num\_total = num\_total *# 种群中个体数（等于蚁群数量)* self.scores = [] *# 适应值* self.iteration = iteration *# 迭代轮数* self.location = data *# 城市坐标矩阵* self.crossover\_ratio = 0.2 *# 父代选择比* self.mutate\_ratio = 0.05 *# 变异概率   # fruits中存每一个个体是下标的list* self.dis\_mat = self.calculate\_dis\_mat(num\_city, data) *# 计算城市间的路径长度，即路径矩阵  #self.fruits = self.random\_init(num\_total, num\_city) # 随机初始化开始种群* self.fruits = aco\_init *#直接将蚁群算法得到的解作为遗传算法的初始种群  # print(len(self.fruits))   # 显示初始化后的最佳路径* scores = self.compute\_fitness(self.fruits)  sort\_index = np.argsort(-scores) *# 返回的是种群个体适应值从大到小排序后的索引值的数组* init\_best = self.fruits[sort\_index[0]] *# 初始化种群中适应值最高的个体的适应值* init\_best = self.location[init\_best] *# 适应值最高的路径的沿途城市坐标* def random\_init(self, num\_total, num\_city):  tmp = [x for x in range(num\_city)]  result = []  for i in range(num\_total):  random.shuffle(tmp)  result.append(tmp.copy())  return result   *# 计算不同城市之间的距离* def calculate\_dis\_mat(self, num\_city, location):  dis\_mat = np.zeros((num\_city, num\_city))  for i in range(num\_city):  for j in range(num\_city):  if i == j:  dis\_mat[i][j] = np.inf  continue  a = location[i]  b = location[j]  tmp = np.sqrt(sum([(x[0] - x[1]) \*\* 2 for x in zip(a, b)]))  dis\_mat[i][j] = tmp  return dis\_mat   *# 计算路径长度* def calculate\_pathlen(self, path, dis\_mat):  try:  a = path[0]  b = path[-1]  except:  import pdb  pdb.set\_trace()  result = dis\_mat[a][b]  *# print('路径：' , path)* for i in range(len(path) - 1):  a = path[i]  b = path[i + 1]  result += dis\_mat[a][b]  return result   *# 计算种群个体适应度* def compute\_fitness(self, fruits):  fitness = []  for fruit in fruits:  *# if isinstance(fruit, int):  # import pdb  # pdb.set\_trace()* length = self.calculate\_pathlen(fruit, self.dis\_mat)  fitness.append(1.0 / length)  *# print('种群适应度： ', fitness)* return np.array(fitness)   *# 基因交叉产生下一代* def ga\_cross(self, x, y):  *# print('x' , x , 'y' , y)* len\_ = len(x)  assert len(x) == len(y)  path\_list = [t for t in range(len\_)]  *# print('path\_list' , path\_list)* order = list(random.sample(path\_list, 2))  *# print('order' , order)* order.sort()  start, end = order   *# 找到冲突点并存下他们的下标,x中存储的是y中的下标,y中存储x与它冲突的下标* tmp = x[start:end]  *# print(tmp)* x\_conflict\_index = []  for sub in tmp:  index = y.index(sub)  if not (index >= start and index < end): *# 如果不在start-end之间,则表示如果直接交换会发生冲突* x\_conflict\_index.append(index)   y\_confict\_index = []  tmp = y[start:end]  for sub in tmp:  index = x.index(sub)  if not (index >= start and index < end):  y\_confict\_index.append(index)   assert len(x\_conflict\_index) == len(y\_confict\_index)  *# print(x\_conflict\_index , y\_confict\_index)  # 交叉* tmp = x[start:end].copy()  x[start:end] = y[start:end]  y[start:end] = tmp  *# print('x ' , x , 'y' , y)   # 解决冲突* for index in range(len(x\_conflict\_index)):  i = x\_conflict\_index[index]  j = y\_confict\_index[index]  y[i], x[j] = x[j], y[i]   assert len(set(x)) == len\_ and len(set(y)) == len\_  return list(x), list(y)   *# 选择部分优秀的个体作为父代* def ga\_parent(self, scores, crossover\_ratio):  sort\_index = np.argsort(-scores).copy()  sort\_index = sort\_index[0:int(crossover\_ratio \* len(sort\_index))]  parents = []  parents\_score = []  for index in sort\_index:  parents.append(self.fruits[index])  parents\_score.append(scores[index])  return parents, parents\_score   *# 轮盘赌选择算法选择两个父代。* def ga\_choose(self, genes\_score, genes\_choose):  sum\_score = sum(genes\_score)  score\_ratio = [sub \* 1.0 / sum\_score for sub in genes\_score] *# 划分区域比率* rand1 = np.random.rand()  rand2 = np.random.rand()  for i, sub in enumerate(score\_ratio):  if rand1 >= 0:  rand1 -= sub  if rand1 < 0:  index1 = i  if rand2 >= 0:  rand2 -= sub  if rand2 < 0:  index2 = i  if rand1 < 0 and rand2 < 0:  break  return list(genes\_choose[index1]), list(genes\_choose[index2])   *# 变异* def ga\_mutate(self, gene):  path\_list = [t for t in range(len(gene))]  order = list(random.sample(path\_list, 2))  start, end = min(order), max(order)  tmp = gene[start:end]  *# np.random.shuffle(tmp)* tmp = tmp[::-1]  gene[start:end] = tmp *# 以首尾反转作为变异* return list(gene)   def ga(self):  *# 获得优质父代* scores = self.compute\_fitness(self.fruits)  *# 选择部分优秀个体作为父代候选集合* parents, parents\_score = self.ga\_parent(scores, self.crossover\_ratio)  tmp\_best\_one = parents[0]  tmp\_best\_score = parents\_score[0]  *# 新的种群fruits* fruits = parents.copy()  *# 生成新的种群* while len(fruits) < self.num\_total:  *# 轮盘赌方式对父代进行选择* gene\_x, gene\_y = self.ga\_choose(parents\_score, parents)  *# print('轮盘赌：' , gene\_x , gene\_y)  # 交叉* gene\_x\_new, gene\_y\_new = self.ga\_cross(gene\_x, gene\_y)  *# 变异* if np.random.rand() < self.mutate\_ratio:  gene\_x\_new = self.ga\_mutate(gene\_x\_new)  if np.random.rand() < self.mutate\_ratio:  gene\_y\_new = self.ga\_mutate(gene\_y\_new)  x\_fitness = 1. / self.calculate\_pathlen(gene\_x\_new, self.dis\_mat)  y\_fitness = 1. / self.calculate\_pathlen(gene\_y\_new, self.dis\_mat)  *# 将适应度高的放入种群中* if x\_fitness > y\_fitness and (not gene\_x\_new in fruits):  fruits.append(gene\_x\_new)  elif x\_fitness <= y\_fitness and (not gene\_y\_new in fruits):  fruits.append(gene\_y\_new)   self.fruits = fruits   return tmp\_best\_one, tmp\_best\_score   def run(self):  BEST\_LIST = None  best\_score = -math.inf  self.best\_record = []  for i in range(len(iter\_x), self.iteration + 1):  tmp\_best\_one, tmp\_best\_score = self.ga()  iter\_x.append(i)  iter\_y.append(1. / tmp\_best\_score)  if tmp\_best\_score > best\_score:  best\_score = tmp\_best\_score  BEST\_LIST = tmp\_best\_one  iter\_path.append(BEST\_LIST)  self.best\_record.append(1. / best\_score)  print(i, 1. / best\_score)  print(1. / best\_score)  return BEST\_LIST, self.location[BEST\_LIST], 1. / best\_score   class ACO(object):  def \_\_init\_\_(self, num\_city, data):  self.m = math.ceil(num\_city \* 2) *# 蚂蚁数量，一般设置为目标数的1.5倍* self.alpha = 1 *# 信息素重要程度因子，一般[1 , 4]* self.beta = 5*#4 # 启发函数重要因子，一般[3 , 4.5]* self.rho = 0.1 *#0.2 # 信息素挥发因子* self.Q = 1*#20 # 信息素常量* self.num\_city = num\_city *# 城市规模* self.location = data *# 城市坐标* self.Tau = np.ones([num\_city, num\_city]) *# 信息素矩阵,Tau(i, j)表示路径(i, j)的信息素量,设置为1防止/0错误* self.Table = [[0 for \_ in range(num\_city)] for \_ in range(self.m)] *# 生成的蚁群,路径记录表,第一列是所有蚂蚁的起点城市* self.iter\_max = 2 *#200#迭代最大次数* self.dis\_mat = self.calculate\_dis\_mat(num\_city, self.location) *# 计算城市之间的距离矩阵* self.Eta = 10. / self.dis\_mat *# 启发式函数  #self.Eta = 1. / self.dis\_mat # 启发式函数* self.paths = None *# 蚁群中每个个体的长度   # 轮盘赌选择* def rand\_choose(self, p):  x = np.random.rand()  for i, t in enumerate(p):  x -= t  if x <= 0:  break  return i   *# 生成蚁群* def get\_ants(self, num\_city):  for i in range(self.m):  start = np.random.randint(num\_city - 1)  *#start = np.random.randint(1,num\_city) #设计直接从0出发，最终回到0。* self.Table[i][0] = start *#将各个蚂蚁随机地放置在不同的出发点。* unvisit = list([x for x in range(num\_city) if x != start]) *#把还没走到的点加入  #unvisit = list([x for x in range(1,num\_city) if x != start])* current = start  j = 1  while len(unvisit) != 0: *#蚂蚁访问完所有的城市* P = []  *# 通过信息素计算城市之间的转移概率* for v in unvisit:  P.append(self.Tau[current][v] \*\* self.alpha \* self.Eta[current][v] \*\* self.beta)  P\_sum = sum(P)  P = [x / P\_sum for x in P] *#归一化  # 轮盘赌选择一个一个城市* index = self.rand\_choose(P)  current = unvisit[index]  self.Table[i][j] = current  unvisit.remove(current)  j += 1   *# 计算不同城市之间的距离* def calculate\_dis\_mat(self, num\_city, location):  dis\_mat = np.zeros((num\_city, num\_city))  for i in range(num\_city):  for j in range(num\_city):  if i == j:  dis\_mat[i][j] = np.inf  continue  a = location[i]  b = location[j]  tmp = np.sqrt(sum([(x[0] - x[1]) \*\* 2 for x in zip(a, b)]))  dis\_mat[i][j] = tmp  return dis\_mat   *# 计算一条路径的长度* def calculate\_pathlen(self, path, dis\_mat):  a = path[0]  b = path[-1]  result = dis\_mat[a][b]  for i in range(len(path) - 1):  a = path[i]  b = path[i + 1]  result += dis\_mat[a][b]  return result   *# 计算一个群体的长度* def compute\_paths(self, paths):  result = []  for one in paths:  length = self.calculate\_pathlen(one, self.dis\_mat)  result.append(length)  return result   *# 更新信息素* def update\_Tau(self):  delta\_tau = np.zeros([self.num\_city, self.num\_city])  *#paths = self.compute\_paths(self.Table)* for i in range(self.m):  for j in range(self.num\_city - 1):  a = self.Table[i][j]  b = self.Table[i][j + 1]  delta\_tau[a][b] = delta\_tau[a][b] + self.Q / self.paths[i]  a = self.Table[i][0]  b = self.Table[i][-1]  delta\_tau[a][b] = delta\_tau[a][b] + self.Q / self.paths[i]  self.Tau = (1 - self.rho) \* self.Tau + delta\_tau   def aco(self):  best\_lenth = math.inf  best\_path = None  for cnt in range(self.iter\_max):  *# 生成新的蚁群* self.get\_ants(self.num\_city) *# 输出>>self.Table，蚂蚁访问完所有的城市  #print(self.Table)* self.paths = self.compute\_paths(self.Table)  *# 取该蚁群的最优解* tmp\_lenth = min(self.paths)  tmp\_path = self.Table[self.paths.index(tmp\_lenth)]  *# 可视化每一次迭代的最优路径，从而绘制出路径变化的整个过程  # 更新最优解* if tmp\_lenth < best\_lenth:  best\_lenth = tmp\_lenth  best\_path = tmp\_path  *#记录下每次迭代选到的最优路径规划图* iter\_path.append(best\_path)  *# 更新信息素* self.update\_Tau()   *# 保存结果* iter\_x.append(cnt)  iter\_y.append(best\_lenth)  print(cnt,best\_lenth)  return best\_lenth, best\_path   def run(self):  best\_length, best\_path = self.aco()  print(**'路径：'** , best\_path)  return best\_path, self.location[best\_path], best\_length   *# 读取数据* def read\_tsp(path):  lines = open(path, **'r'**).readlines()  assert **'NODE\_COORD\_SECTION**\n**'** in lines  index = lines.index(**'NODE\_COORD\_SECTION**\n**'**)  data = lines[index + 1:-1]  tmp = []  for line in data:  line = line.strip().split(**' '**)  if line[0] == **'EOF'**:  continue  tmpline = []  for x in line:  if x == **''**:  continue  else:  tmpline.append(float(x))  if tmpline == []:  continue  tmp.append(tmpline)  data = tmp  return data  *#处理信息读入* data = read\_tsp(**'data/rat195.tsp'**) *#结果：[[1.0，64.0，96.0],[],[]...]* data = np.array(data)*#将数据转换为矩阵* data = data[:, 1:]*#去除掉第一列* time\_start=time.time() *#计算算法运行时间* aco = ACO(num\_city=data.shape[0], data=data.copy()) Best\_pathIndex, Best\_path, Best = aco.run() print(**'蚁群算法->遗传初始种群: '** , aco.Table , len(aco.Table)) *#50* ga = GA(num\_city=data.shape[0], num\_total=math.ceil(data.shape[0] \* 2), iteration=800, data=data.copy() , aco\_init = aco.Table) Best\_pathIndex , Best\_path , Best\_len = ga.run() print(**"路径"**, Best\_pathIndex) time\_end=time.time() *#计算算法运行时间* print(**'time cost'**,time\_end-time\_start,**'s'**)  *#绘制收敛曲线图* plt.figure() plt.plot(iter\_x , iter\_y) plt.title(**'混合蚁群遗传算法收敛曲线'**) plt.xlabel(**"迭代次数"**) plt.ylabel(**"路径长度"**) plt.rcParams[**'font.sans-serif'**] = [**'SimHei'**] *# 显示中文标签* plt.rcParams[**'axes.unicode\_minus'**] = False  *#绘制规划结果图* plt.figure() *# 打开交互模式* plt.ion() print(**'长度'**, len(iter\_path)) for i in range(len(iter\_path)):  *# 清除原有图像* plt.cla()  plt.scatter(Best\_path[:, 0], Best\_path[:, 1],color=**'red'**) *# 所有点的0坐标做x，1坐标做y ,在plt中标出来  # 使用annotate函数必备参数绘制注解  # for index, point in zip(Best\_pathIndex, Best\_path):  # plt.annotate(index + 1, xy=(point[0], point[1]))* plt.title(**'混合蚁群遗传算法规划结果'**)  plt.xlabel(**"点横坐标"**)  plt.ylabel(**"点纵坐标"**)  *#print('iter\_path[i]', iter\_path[i])* iter\_path\_coord = aco.location[iter\_path[i]]  iter\_path\_coord = np.vstack([iter\_path\_coord, iter\_path\_coord[0]]) *# 将尾和初始连起来* plt.plot(iter\_path\_coord[:, 0], iter\_path\_coord[:, 1]) *# 将所有点按顺序连起来* plt.savefig(**r'D:/.c学习资料/人工智能/TSP\_collection-master/GA\_ACOgif\{}.png'**.format(i))  *# 暂停* plt.pause(0.1) *# 关闭交互模式* plt.ioff() plt.show() |

ORC\_SOM代码：

|  |
| --- |
| import random import math import numpy as np import matplotlib.pyplot as plt import matplotlib as mpl import time  class SOM(object):  def \_\_init\_\_(self, num\_city, data):  self.num\_city = num\_city *#城市数量* self.location = data.copy() *#城市物理坐标* self.iteraton = 20000  self.learning\_rate = 0.8 *#学习率* self.dis\_mat = self.compute\_dis\_mat(num\_city, self.location)  self.best\_path = []  self.best\_length = math.inf  self.iter\_x = []  self.iter\_y = []   def normalize(self, citys):  *#朴素版直接归一化* x\_min, y\_min = citys.min(axis=0)[:]  x\_max, y\_max = citys.max(axis=0)[:]  citys[:, 0] = (citys[:, 0] - x\_min) / (x\_max - x\_min)  citys[:, 1] = (citys[:, 1] - y\_min) / (y\_max - y\_min)  return citys   def generate\_network(self, size , citys):  *#生成给定大小的神经元网络，返回区间 [0,1] 中二维点的向量。  #return np.random.rand(size, 2)#随机生成符合（0，1）型正态分布的size行2列的向量   # 初始神经元拓扑结构为围绕城市的矩形* x\_min, y\_min = citys.min(axis=0)[:]  x\_max, y\_max = citys.max(axis=0)[:]    *# 初始门限半径,对应论文中的lamda* self.radius\_0 = np.sqrt((x\_max - x\_min) \*\* 2 + (y\_max - y\_min) \*\* 2) / 4  self.radius = 0.00001    network = np.ones((size, 2))  a = int(size / 4)  network[0:a, 0] = np.linspace(x\_min, x\_max, a)  network[0:a, 1] = y\_min  network[a: 2 \* a, 0] = x\_max  network[a: 2 \* a, 1] = np.linspace(y\_min, y\_max, a)  network[2 \* a: 3 \* a, 0] = np.linspace(x\_max, x\_min, a)  network[2 \* a: 3 \* a, 1] = y\_max  network[3 \* a: 4 \* a, 0] = x\_min  network[3 \* a: 4 \* a, 1] = np.linspace(y\_max, y\_min, a)  network = np.row\_stack((network, network[0])) *# 末尾添首项，组成闭环* return network   def get\_neighborhood(self, center, radix, domain):  *#返回优胜邻域，二维排列就是一个圆网  """得到一个中心索引周围给定基数的范围高斯。"""  # 为基数设置上界* if radix < 1:  radix = 1  *# 计算圆网到中心的距离* deltas = np.absolute(center - np.arange(domain))  distances = np.minimum(deltas, domain - deltas)  *# 计算给定中心周围的高斯分布* return np.exp(-(distances \* distances) / (2 \* (radix \* radix)))   def get\_route(self, cities, network):*#返回神经元的路线,按顺序离城市最近的神经元组成路线* f = lambda c: self.select\_closest(network, c)  dis = []  for city in cities:  dis.append(f(city))  index = np.argsort(dis)  *#print('index' , index)* return index *#返回路径(城市索引值)* def select\_closest(self, network, city):  *#返回最接近输入点的神经元的索引。   # 矩阵每个行向量求向量的2范数（欧式距离）* distances = np.linalg.norm(network - city, axis=1)  winner\_idx = distances.argmin()   *# 计算orc系数,用于神经元权值偏移量，对应论文中的Z* distence = distances.min()  *#self.orc = np.exp((-(distence \*\* 2) / (2 \* self.radius \*\* 2)))* self.orc = np.exp((-(distence \*\* 2) / (2 \* self.radius \*\* 2)) + 0.5)   return winner\_idx   *# 计算不同城市之间的距离* def compute\_dis\_mat(self, num\_city, location):  dis\_mat = np.zeros((num\_city, num\_city))  for i in range(num\_city):  for j in range(num\_city):  if i == j:  dis\_mat[i][j] = np.inf  continue  a = location[i]  b = location[j]  tmp = np.sqrt(sum([(x[0] - x[1]) \*\* 2 for x in zip(a, b)]))  dis\_mat[i][j] = tmp  return dis\_mat   *# 计算一条路径的长度* def compute\_pathlen(self, path, dis\_mat):  a = path[0]  b = path[-1]  result = dis\_mat[a][b]  for i in range(len(path) - 1):  a = path[i]  b = path[i + 1]  result += dis\_mat[a][b]  return result   def som(self):  citys = self.normalize(self.location) *#归一化城市坐标* n = citys.shape[0] \* 4 *#输出层神经元的个数：城市个数的八倍* network = self.generate\_network(n , citys) *#随机生成【0，1】分布的神经元的位置* for i in range(self.iteraton):  index = np.random.randint(self.num\_city - 1) *#随机选取一个城市作为网络的输入* city = citys[index] *#获取城市归一化后的坐标* winner\_idx = self.select\_closest(network, city) *#返回最接近输入坐标的神经元的索引。   #以所选到的神经元为中心，均值为0，方差初始化为城市个数的十分之一创建高斯分布* gaussian = self.get\_neighborhood(winner\_idx, n // 10, network.shape[0])  *#所有神经元按照该高斯分布向选中的城市移动* network += self.orc \* gaussian[:, np.newaxis] \* self.learning\_rate \* (city - network)   self.learning\_rate = self.learning\_rate \* 0.99997 *#学习率的衰减值   # 更新渗透区域* self.radius = self.radius\_0 \* (1 / (1 + np.exp(-i / 2)) - 0.5)  if self.radius <= 1:  self.radius = 1   n = n \* 0.9997 *#高斯分布的方差的衰减值* if n < 1: *#方差达到阈值结束* break  if self.learning\_rate < 0.001: *#学习率达到阈值结束* break  route = self.get\_route(citys, network)  route\_l = self.compute\_pathlen(route, self.dis\_mat) *#计算路径长度* if route\_l < self.best\_length: *#记录最优路径* self.best\_length = route\_l  self.best\_path = route  self.iter\_x.append(i) *#记录下收敛过程* self.iter\_y.append(self.best\_length)  print(i, self.iteraton, self.best\_length)   *#画出神经网络的规划图* if not i % 1000:  plot\_network(citys, network, name=**'GSOMgif/{:05d}.png'**.format(i))  plot\_network(citys, network, name=**'GSOMgif/final.png'**)  return self.best\_length, self.best\_path   def run(self):  self.best\_length, self.best\_path = self.som()  return self.best\_path, self.location[self.best\_path], self.best\_length  *#绘制神经网络自组织的规划结果图* def plot\_network(cities, neurons, name):  mpl.rcParams[**'agg.path.chunksize'**] = 10000  plt.rcParams[**'axes.facecolor'**] = **'white'** plt.rcParams[**'savefig.facecolor'**] = **'white'** fig = plt.figure(figsize=(5, 5), facecolor=**'white'**)  axis = fig.add\_axes([0,0,1,1])   axis.set\_aspect(**'equal'**, adjustable=**'datalim'**)  plt.axis(**'off'**)   axis.scatter(cities[:, 0], cities[:, 1], color=**'red'**, s=4)  axis.plot(neurons[:, 0], neurons[:, 1], **'r.'**, ls=**'-'**, color=**'#0063ba'**, markersize=2)   plt.savefig(name , bbox\_inches=**'tight'**, pad\_inches=0, dpi=200)  plt.close()  *# 读取数据* def read\_tsp(path):  lines = open(path, **'r'**).readlines()  assert **'NODE\_COORD\_SECTION**\n**'** in lines  index = lines.index(**'NODE\_COORD\_SECTION**\n**'**)  data = lines[index + 1:-1]  tmp = []  for line in data:  line = line.strip().split(**' '**)  if line[0] == **'EOF'**:  continue  tmpline = []  for x in line:  if x == **''**:  continue  else:  tmpline.append(float(x))  if tmpline == []:  continue  tmp.append(tmpline)  data = tmp  return data   data = read\_tsp(**'data/pr439.tsp'**) data = np.array(data) *#结果：[[1.0，64.0，96.0],[],[]...]* data = data[:, 1:] *#去除掉第一列* time\_start=time.time() *#计算算法运行时间* som = SOM(num\_city=data.shape[0], data=data.copy()) Best\_pathIndex, Best\_path, Best\_length = som.run() print(**"路径"**, Best\_pathIndex) time\_end=time.time() *#计算算法运行时间* print(**'time cost'**,time\_end-time\_start,**'s'**)  *#绘制收敛曲线图* plt.figure() plt.plot(som.iter\_x , som.iter\_y) plt.title(**'ORC\_SOM收敛曲线'**) plt.xlabel(**"迭代次数"**) plt.ylabel(**"路径长度"**) plt.rcParams[**'font.sans-serif'**] = [**'SimHei'**] *# 显示中文标签* plt.rcParams[**'axes.unicode\_minus'**] = False  *#绘制规划结果图* plt.figure() plt.scatter(Best\_path[:, 0], Best\_path[:, 1], color=**'red'**) *#所有点的0坐标做x，1坐标做y ,在plt中标出来* Best\_path = np.vstack([Best\_path, Best\_path[0]]) *#将尾和初始连起来* plt.plot(Best\_path[:, 0], Best\_path[:, 1]) *#将所有点按顺序连起来* plt.title(**'ORC\_SOM规划结果'**) plt.axis(**'off'**) plt.xticks([]) plt.yticks([]) plt.show() |