通过改进的蚁群算法解决TSP问题

摘要：针对传统的蚁群算法求解蚁群算法容易陷入局部最优的问题，提出一直基于传统蚁群算法的改进模型，来提升算法的全局搜索能力。改进模型相较于基线模型有三个方面的改进，改进信息素浓度的初始化，改进信息更新策略，增加融断机制。最后使用TSPLIB上的数据集，通过平均相对误差来对比基线模型和改进模型，最终的实验结果的表明改进模型在全局搜素能力上相较于基线模型有明显的提高

关键词：TSP问题; 蚁群算法; 全局搜索能力

Abstract:Aiming at the problem that the traditional ant colony algorithm improved ant colony algorithm is easy to achieve local optimization, an improved model that has been based on the traditional ant colony algorithm is proposed to improve the sorting and search ability of the algorithm. Compared with the improvement of the model in three aspects, the improved model is finally used on the data set on TSPLIB to compare the comparison model and the improved model through the average relative error. The final experimental results show that the improved model is compared with The aforementioned model has been significantly improved

keyword:TSP problem; ant colony algorithm; global search ability

1. TSP 问题和蚁群算法
   1. TSP问题的描述

TSP*（Traveling Salesman Problem）*问题，即履行商问题，又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。经典TSP问题中结点之间的路径长度为欧式距离。本文将用改进的蚁群算法解决TSP问题，减少求解陷入局部最优的可能性，增加全局搜索能力，使得得到TSP问题的更优解。

* 1. 蚁群算法的描述

蚁群算法的思路来基于自然界蚂蚁的觅食行为，蚂蚁在寻找食物时，会在路径上留下蚂蚁独有的路径标识——信息素，蚂蚁从蚁穴到食物所进过的路进越短，在该路径上留下的信息素浓度就越高，蚂蚁会根据留下的信息浓度来选择爬行路径，会更倾向于选择信息浓度高的路径，随着时间的退役，越来越多的蚂蚁会集中于某一条路径，而该路径因为蚂蚁的集中爬行其信息素浓度也会越来越高，从而有进一步加强了对蚂蚁的吸引性，形成正反馈机制。

最终蚁群集中爬行的路径即为最优路径

1. 蚁群算法在TSP算法中的应用
   1. 基线模型

对于TSP问题，设置蚁群中蚂蚁数量为m，结点数为n, 结点i和结点j之间的距离为, t时刻结点i和结点j之间的信息素浓度为。

在初始时刻，所有边的信息素浓度设置为一个固定值Q，蚂蚁随机从某一个结点出发，按照一定的概率选择下一个结点，最终生成一条哈密顿路径。不妨设为t时刻处于结点i的蚂蚁选择结点j作为下一个目标的概率，其概率会受到两方面的影响，一是当前路径是上的信息素浓度，二是当前路径上的长度，其具体公式为：

(1)

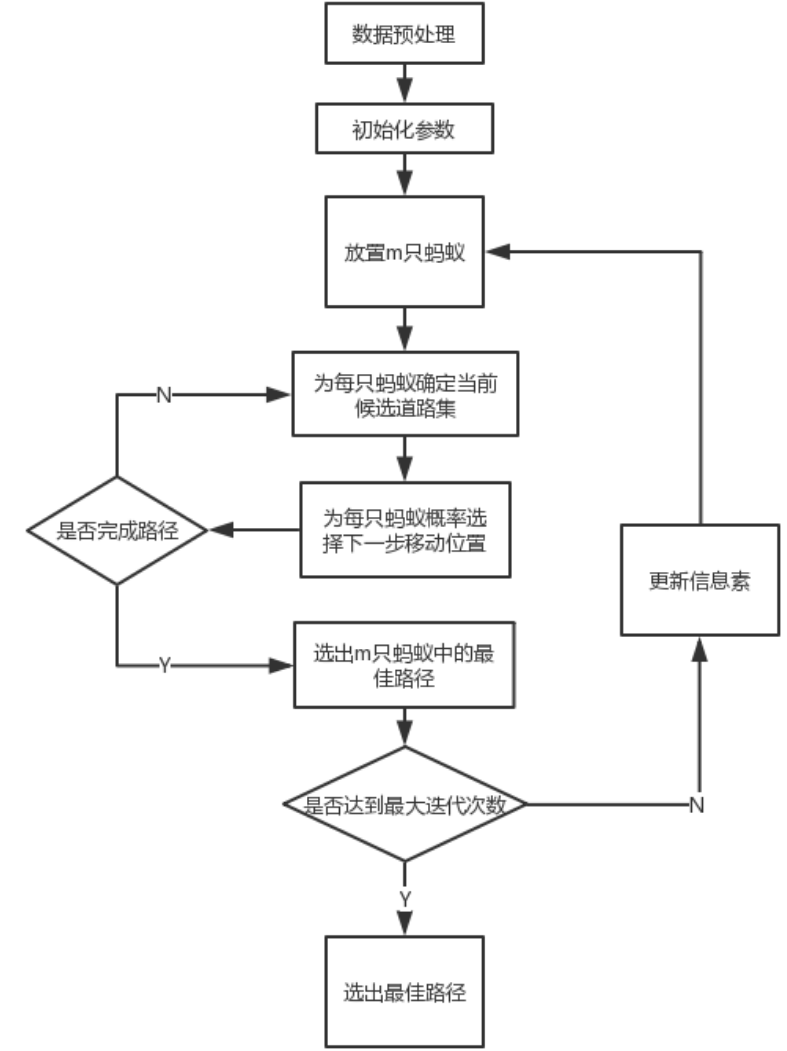
其中，为蚂蚁k下一步可以到达的结点集合，随着时间的推移，中的元素会越来越少，最终变为空集。a和b分别为信息素浓度和路径倒数的重要程度因子。

随着时间的推移，路径上的信息素浓度会不断挥发，而蚂蚁也会不断增加路径上的信息素浓度，路径上的信息素浓度会产生差异，使得整个蚁群集中于某一条路径中。下面定义信息素浓度的状态转移方程：

(2)

其中h为信息素浓度挥发率，为t时刻第k只蚂蚁在结点i和结点j之间路径增加的信息素浓度，为t时刻第k只蚂蚁经过哈密顿回路的总长度。

基线模型的程序流程图如下：



* 1. 改进模型
     1. 改进信息素浓度的初始化

在基线模型中，每条边赋予的信息素浓度初值是一定的，我认为这并不是一个好的方式，原因如下：

1. 相同的初始信息素浓度不具有指导性，使得算法的收敛速度下降
2. 相同的初始信息素浓度可能让蚂蚁一开始向着错误的路径前进，使得信息素浓度在错误的路径上积累，容易陷入局部最优

为了改进以上相同初始信息素浓度的缺点，我们可以根据边的长度来给其的信息素浓度赋予不同的初值，即边越长，则赋予的初始信息素浓度越低，具体公式如下：

(3)

其中，Q为常数，其中为蚂蚁数量m除以贪心选择得到的一个解，numEdge为结点网络的边数。该信息素浓度初始化方法使得每条边上的初始信息素浓度为Q，但是具体每条边上的初始信息素浓度会收到其长度的影响。

* + 1. 改进信息素的更新策略

在基线模型中，信息素浓度的更新策略是不变的，我们知道，在算法运行一段时间之后，蚁群会集中于某条路径，蚁群对解空间探索范围就会越来越小，容易陷入局部最有。为了解决上面的问题，提升算法的全局搜索能力，我们需要一种可变的信息素更新策略。当迭代次数较小时，需要保证蚂蚁在路径上留下的信息素足够多，同时信息素浓度不能过大，确保算法可以更快的收敛; 而当算法收敛到一定程度时，搜索范围越来越局限时，就需要适当增加信息素浓度挥发速率和减少蚂蚁增加的信息素浓度，来避免信息素浓度过于集中，使得蚂蚁能有更大的概率去探索潜在的最优路径，跳出原来狭窄的搜索范围，提高算法的全局搜索能力。下面给出具体的更新公式：

为了更好的理解算法，先给出稳定时刻点的定义：如果存在某个时刻，在一定的时间范围内，当前最优路径一直为发生变化，则称时刻为稳定时刻点

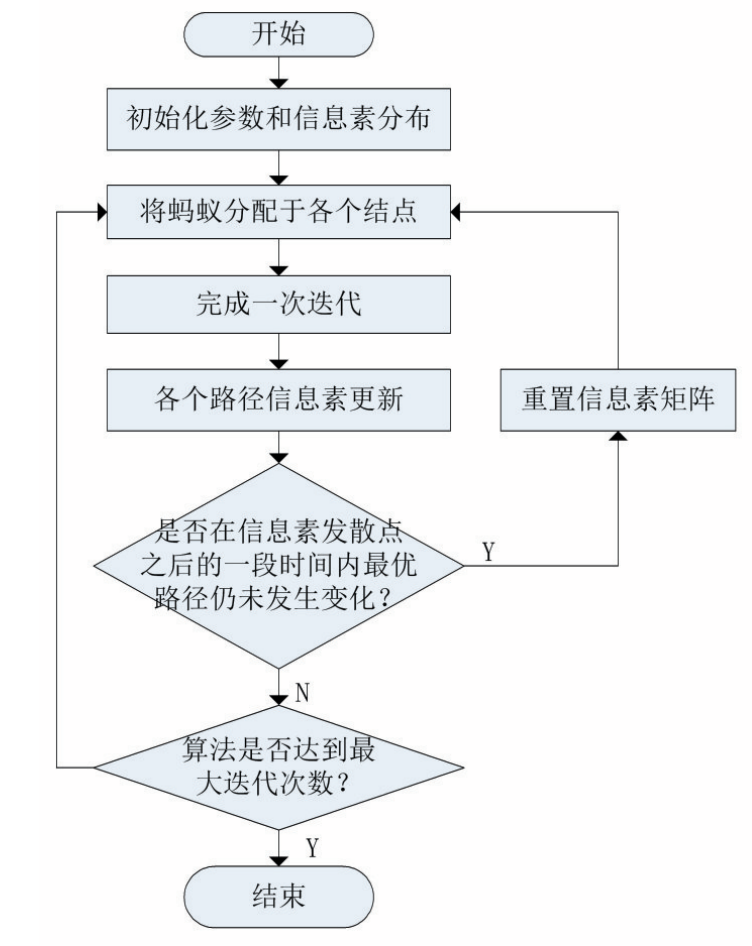
(4)

从上面的公式我们可以看到，当时，信息素更新策略和基线模型一致; 而当时，信息素更新值会除以，减少了信息素浓度，时间越大，减少的幅度就越大。

* + 1. 信息素矩阵的重置

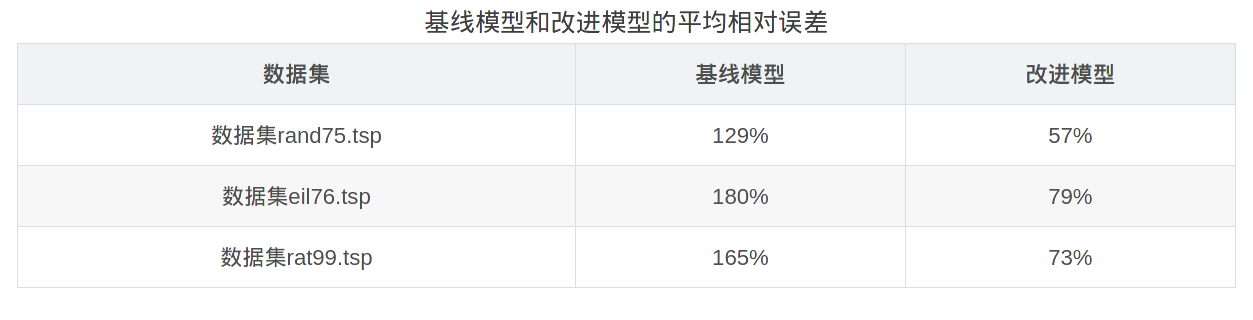
为了进一步增加算法的全局搜索能力，这里我们引入融断机制。当算法进入稳定时刻后的一段时间内，当前的全局最优解仍然没有发生变化，则将信息素浓度矩阵重新初始化。

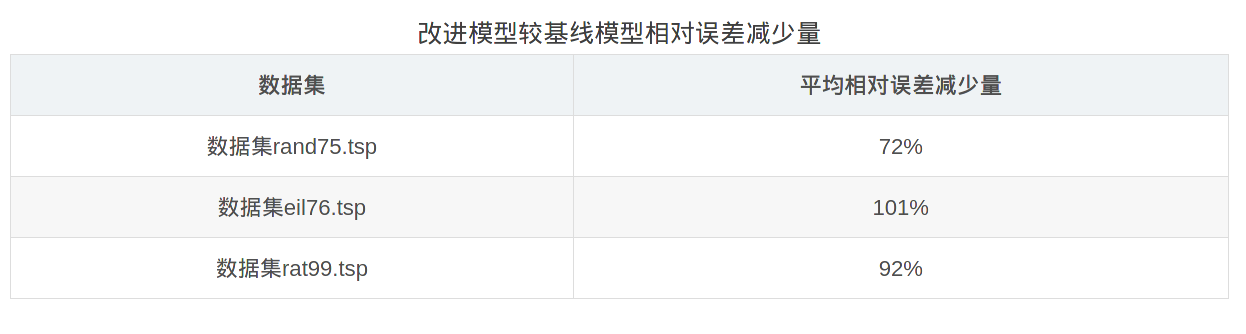
* + 1. 改进模型的程序流程图



1. 实验评估

本次实验选用了TSPLIB提供的三个数据集，rand75.tsp、eil76.tsp、rat99.tsp，初始化参数为，重复实验次数选择10, 通过平均相对误差来比较基线模型和改进模型的优劣。实验结果如下：





由上面的实验结果可以看到，改进模型相较于基线模型有明显的提升

1. 结论

综上所述，改进模型相较于基线模型有一定的性能提升，全局搜索能力更强，但是依然存在着一些不足：

1. 算法的结果相较于真实结果依然存在着较大的误差
2. 算法的时间开销较大

参考文献：

徐书杨 王海红.改进蚁群算法在TSP问题中的应用.现代计算机.2020,(25),22-26

代码我放在GitHub上，上面的东西更完整(有数据集)

GitHub仓库链接<https://github.com/Shuojia-Huang/TSP.git>

下面我只贴出关键的代码

基线模型：

*import* numpy *as* np

*import* random *as* ra

def TSP(*net*):

'''初始化'''

n = net.shape[0]

h = 0.1 *#挥发率*

m = 3 *#蚂蚁数量*

t0 = m / TanXin(n, net) *#初始信息素浓度*

a = 1

b = 2

*#存储结构*

R = np.ones([m, n+1], *dtype*=int) *#蚂蚁路径矩阵*

T= np.ones([n, n]) \* t0 *#信息浓度矩阵*

C = np.ones(m) *#路径长度序列*

global\_best = float('inf') *#全局最优值*

global\_bestR = None *#全局最优路径*

'''循环'''

*for* turn *in* range(50):

*for* k *in* range(m): *# 蚂蚁k*

*for* p *in* range(n + 1):

*if* p == 0:

R[k][0] = ra.randint(0, n-1)

*elif* p == n:

R[k][n] = R[k][0]

*else*:

i = R[k][p - 1] *# 前一个结点*

js = neighbor(n, net, i)

js = list(set(js) - set(R[k][:p])) *# 下一个可能节点*

tjs = [T[i, j]\*\*a \* (1 / net[i, j])\*\*b *for* j *in* js]

sum\_tjs = sum(tjs)

pjs = [tj / sum\_tjs *for* tj *in* tjs] *# 下一个可能节点的概率*

*# 轮盘算法求下一个节点j*

r = ra.random()

*for* z *in* range(len(pjs)):

*if* r <= pjs[z]:

j = js[z]

*else*:

r -= pjs[z]

R[k][p] = j *# 把j加入k蚂蚁的路径*

C[k] = sum([net[R[k][i], R[k][i+1]] *for* i *in* range(n)])

*# 更新t*

*for* i *in* range(n):

*for* j *in* range(n):

T[i, j] = (1-h)\* T[i, j]

*for* k *in* range(m):

*if* j == R[k][list(R[k]).index(i)+1]:

T[i, j] += 1 / C[k]

*#更新全局最优值*

local\_best = min(C) *#局部最优值*

bestk = list(C).index(local\_best) *#局部最优值对应的蚂蚁编号*

*if* local\_best < global\_best:

global\_best = local\_best

global\_bestR = R[bestk].copy()

*# '''输出'''*

*# print('最短路径为：', global\_best)*

*# print('最短路径长度：', global\_bestR)*

*return* global\_best

def TanXin(*n*, *net*):

R = np.ones(n+1, *dtype*=int)

*while* True:

*for* p *in* range(n+1):

*if* p == 0:

R[0] = ra.randint(0, n-1)

*elif* p == n:

R[n] = R[0]

*else*:

i = R[p-1]

js = neighbor(n, net, i)

js = list(set(js)-set(R[:p]))

ljs = [net[i, j] *for* j *in* js]

j = js[ljs.index(min(ljs))]

R[p] = j

C = sum([net[R[i], R[i+1]] *for* i *in* range(n)])

*if* C != float('inf'):

*break*

*return* C

def neighbor(*n*, *net*, *i*):

*return* [j *for* j *in* range(n) *if* j != i *if* net[i][j] != float('inf')]

*if* \_\_name\_\_ == '\_\_main\_\_':

net = np.array([[float('inf'), 3, 1, 2],

[3, float('inf'), 5, 4],

[1, 5, float('inf'), 2],

[2, 4, 2, float('inf')]])

TSP(net)

改进模型

*import* numpy *as* np

*import* random *as* ra

def TSP(*net*):

'''初始化'''

n = net.shape[0]

h = 0.1 *#挥发率*

m = 3 *#蚂蚁数量*

t0 = m / TanXin(n, net) *#初始信息素浓度*

a = 1

b = 2

*#存储结构*

R = np.ones([m, n+1], *dtype*=int) *#蚂蚁路径矩阵*

T = np.ones([n, n]) *#信息浓度矩阵*

*#初始化信息浓度矩阵*

sum\_edge = 0

num\_edge = 0

*for* i *in* range(n):

*for* j *in* range(n):

*if* i == j or net[i, j] == float('inf'):

*continue*

sum\_edge += 1.0 / net[i, j]

num\_edge += 1;

*for* i *in* range(n):

*for* j *in* range(n):

*if* i == j or net[i, j] == float('inf'):

*continue*

T[i, j] = num\_edge \* t0 \* 1 / net[i, j] / sum\_edge

C = np.ones(m) *#路径长度序列*

global\_best = float('inf') *#全局最优值*

global\_bestR = None *#全局最优路径*

*#改进部分*

x = 5

y = 5

u = 5

turn0times = 0 *#计时器1（记录到turn0前未变化的次数）*

turn1times = 0 *#计时器2 (记录到turn0后为变化的次数)*

turn1num = 0 *#计数器2 (记录重来的次数)*

turn0 = None *#稳定时刻点*

isturn0 = False *#是否到达稳定时刻点*

isturn1 = True *#是否到达重来时刻点*

'''循环'''

turn = 0

*while* turn <= 50:

*if* isturn0:

turn1times += 1

*for* k *in* range(m): *# 蚂蚁k*

*for* p *in* range(n + 1):

*if* p == 0:

R[k][0] = ra.randint(0, n-1)

*elif* p == n:

R[k][n] = R[k][0]

*else*:

i = R[k][p - 1] *# 前一个结点*

js = neighbor(n, net, i)

js = list(set(js) - set(R[k][:p])) *# 下一个可能节点*

tjs = [T[i, j]\*\*a \* (1 / net[i, j])\*\*b *for* j *in* js]

sum\_tjs = sum(tjs)

pjs = [tj / sum\_tjs *for* tj *in* tjs] *# 下一个可能节点的概率*

*# 轮盘算法求下一个节点j*

r = ra.random()

*for* z *in* range(len(pjs)):

*if* r <= pjs[z]:

j = js[z]

*else*:

r -= pjs[z]

R[k][p] = j *# 把j加入k蚂蚁的路径*

C[k] = sum([net[R[k][i], R[k][i+1]] *for* i *in* range(n)])

*#更新T*

*for* i *in* range(n):

*for* j *in* range(n):

*if* isturn0:

T[i, j] = (1 - h) \* T[i, j] / (turn-turn0)

*else*:

T[i, j] = (1-h) \* T[i, j]

*for* k *in* range(m):

*if* j == R[k][list(R[k]).index(i)+1]:

*if* isturn0:

deltaT = 1 / C[k] / (turn-turn0)

*else*:

deltaT = 1 / C[k]

T[i, j] += deltaT

*#更新全局最优值*

local\_best = min(C) *#局部最优值*

bestk = list(C).index(local\_best) *#局部最优值对应的蚂蚁编号*

*if* local\_best < global\_best:

global\_best = local\_best

global\_bestR = R[bestk].copy()

turn0times = 0

*if* isturn0:

isturn1 = False

*else*:

turn0times += 1

*if* turn0times == x:

turn0 = turn

isturn0 = True

turn += 1

*if* turn1times == y and isturn1 and turn1num <= u:

turn1num += 1

*for* i *in* range(n):

*for* j *in* range(n):

*if* i == j or net[i, j] == float('inf'):

*continue*

T[i, j] = num\_edge \* t0 \* 1 / net[i, j] / sum\_edge

turn0times = 0

turn1times = 0

isturn0 = False

isturn1 = True

turn = 0

*# '''输出'''*

*# print('最短路径为：', global\_best)*

*# print('最短路径长度：', global\_bestR)*

*return* global\_best

def TanXin(*n*, *net*):

R = np.ones(n+1, *dtype*=int)

*while* True:

*for* p *in* range(n+1):

*if* p == 0:

R[0] = ra.randint(0, n-1)

*elif* p == n:

R[n] = R[0]

*else*:

i = R[p-1]

js = neighbor(n, net, i)

js = list(set(js)-set(R[:p]))

ljs = [net[i, j] *for* j *in* js]

j = js[ljs.index(min(ljs))]

R[p] = j

C = sum([net[R[i], R[i+1]] *for* i *in* range(n)])

*if* C != float('inf'):

*break*

*return* C

def neighbor(*n*, *net*, *i*):

*return* [j *for* j *in* range(n) *if* j != i *if* net[i][j] != float('inf')]

*if* \_\_name\_\_ == '\_\_main\_\_':

net = np.array([[float('inf'), 3, 1, 2],

[3, float('inf'), 5, 4],

[1, 5, float('inf'), 2],

[2, 4, 2, float('inf')]])

TSP(net)