无线充电器节点部署的优化算法

本算法先是构造了一块100\*100的区域，然后随机摆放了100个传感器节点，再摆放40个无线充电装置对传感器节点进行充电，通过改变无线充电装置的位置，使得整个网络里所有传感器节点获得的接受效率最大。本算法分别使用了布谷鸟算法，粒子群算法，蝙蝠算法对这个问题进行寻优，每个算法循环了100次，最终结果显示蝙蝠算法求出的整个网络里所有传感器节点获得的接受效率最大，并且打印出了此时无线充电装置的位置，另存为result.txt里。

import random

import numpy as np

from time import time

import math

import copy

with open('result.txt', 'a') as f:

for iters in range(0, 100):

t1 = time()

X = []

for i in range(0, 100):

ax = random.randint(0, 99)

bx = random.randint(0, 99)

X.append([ax, bx])

print('X=', X)

class Cuckoo\_Search():

def \_\_init\_\_(self, nest\_number, nest\_size, pa, Lowerbands, Upperbands, Num\_iter):

# 适应度函数

# self.func = func

self.n = nest\_number

self.nd = nest\_size

self.pa = pa

self.Lb = Lowerbands

self.Ub = Upperbands

self.N\_IterTotal = Num\_iter

self.sigma = self.sigma\_cal()

# 用于适应度函数

def sigma\_cal(self):

# 定义伽马函数

def gamma(x):

\_a = (1.00000000000000000000,

0.57721566490153286061,

-0.65587807152025388108,

-0.04200263503409523553,

0.16653861138229148950,

-0.04219773455554433675,

-0.00962197152787697356,

0.00721894324666309954,

-0.00116516759185906511,

-0.00021524167411495097,

0.00012805028238811619,

-0.00002013485478078824,

-0.00000125049348214267,

0.00000113302723198170,

-0.00000020563384169776,

0.00000000611609510448,

0.00000000500200764447,

-0.00000000118127457049,

0.00000000010434267117,

0.00000000000778226344,

-0.00000000000369680562,

0.00000000000051003703,

-0.00000000000002058326,

-0.00000000000000534812,

0.00000000000000122678,

-0.00000000000000011813,

0.00000000000000000119,

0.00000000000000000141,

-0.00000000000000000023,

0.00000000000000000002)

y = float(x) - 1.0

sm = \_a[-1]

for an in \_a[-2::-1]:

sm = sm \* y + an

return 1.0 / sm

beta = 1.5

sigma = (gamma(1+beta)\*math.sin(math.pi\*beta/2)/(gamma((1+beta)/2)\*beta\*(2\*\*((beta-1)/2))))\*\*(1/beta)

return sigma

def cs\_run(self):

# 随机初始解决方案

# 随机初始化一组解并将它们存储到矩阵巢中

nest = [[random.uniform(self.Lb[idx], self.Ub[idx]) for idx in range(self.nd)] for row in range(self.n)]

fitness = [100000 for idx in range(self.n)]

fmin, bestnest, nest, fitness = self.get\_best\_nest(nest, nest, fitness)

N\_iter = 0

# 开始迭代

for iter in range(self.N\_IterTotal):

# 生成新的解决方案(但保持当前最好的解决方案)

new\_nest = self.get\_cuckoos(nest, bestnest, self.Lb, self.Ub)

fnew, best, nest, fitness = self.get\_best\_nest(nest, new\_nest, fitness)

# 更新计数器

N\_iter = N\_iter+self.n

# 发现和随机化

new\_nest = self.empty\_nests(nest, self.Lb, self.Ub, self.pa)

# 打印new\_nest

# 求这组解的值

fnew, best, nest, fitness = self.get\_best\_nest(nest, new\_nest, fitness)

# 再次更新计数器

N\_iter = N\_iter+self.n

# 找到目前为止最好的目标

if fnew < fmin:

fmin = fnew

bestnest = best

# 结束迭代

# Post-optimization处理

# 展示所有鸟巢

# print 'Total number of iterations='

# print N\_iter

# print fmin

# print bestnest

return fmin, bestnest

def get\_cuckoos(self, nest, best, Lb, Ub):

# 莱维飞行

# Matlab code: n=size(nest,1);

n = len(nest)

# 莱维指数与系数

# 详细内容见公式(2.21)，第16页(第2章)

# 杨，自然启发元启发式算法，第二版，Luniver出版社，(2010).

# 打印sigma

for j in range(n):

# matlab code s=nest(j,:);

s = copy.deepcopy(nest[j])

# 这是实现Levy flights的一种简单方法

# 对于标准随机游动，使用step=1;

# 使用Mantegna算法进行Levy flights

# in matlab u = randn(size(s))\*sigma

# in matlab v = randn(size(s))

# randn: Matlab中的标准正态分布函数

# normal alvariate(0,1)用于标准正态分布

u = [self.sigma \* random.normalvariate(0, 1) for idx in range(len(s))]

v = [random.normalvariate(0, 1) for idx in range(len(s))]

# step=u./abs(v).^(1/beta)

# 矩阵u除以矩阵v的绝对值

# 矩阵中元素的幂(1/)

# 方程2.21，第16页(第2章)

# 杨，自然启发元启发式算法，第二版，Luniver出版社，(2010).

step = [0 for idx in range(len(s))]

# Matlab code: step=u./abs(v).^(1/beta)

beta = 1.5

for i in range(len(s)):

step[i] = u[i]/((abs(v[i]))\*\*(1.0/beta))

# 在下一个方程中，差因子(s-best)表示

# 当解决方案是最佳解决方案时，它保持不变.

# Matlab : stepsize = 0.01\*step.\*(s-best)

stepsize = [0 for idx in range(len(s))]

stepsize\_factor = 0.01

for i in range(len(s)):

stepsize[i] = stepsize\_factor \* step[i] \* (s[i] - best[i])

# 这里的因子0.01来自于L/100应该是典型的

# L为典型长度尺度的步行/飞行步长;

# 否则，Levy flights可能会变得过于激进/高效，

# 是什么让新的解决方案(甚至)跳出了设计领域

# (从而浪费评估)

# 现在是实际的随机漫步或飞行

# Matlab code: s=s + stepsize.\*randn(size(s))

coef\_a = [random.normalvariate(0, 1) for idx in range(len(s))]

for i in range(len(s)):

s[i] = s[i] + stepsize[i] \* coef\_a[i]

# 应用简单的边界/限制

# print '--------------------------------'

# print s

nest[j] = self.simplebounds(s, Lb, Ub)

# print nest[j]

return nest

# 找到当前最好的巢

def get\_best\_nest(self, nest, newnest, fitness):

# 评估所有新解决方案

for j in range(len(nest)):

fnew = self.fobj(newnest[j])

if fnew <= fitness[j]:

fitness[j] = fnew

for i in range(len(nest[j])):

# print newnest[j][i]

# print nest[j][i]

nest[j][i] = newnest[j][i]

# print newnest[j][i]

# print nest[j][i]

# 找到当前最好的

fmin = min(fitness)

K = fitness.index(fmin)

best = nest[K]

return fmin, best, nest, fitness

# 通过构建新的解决方案/鸟巢来替换一些鸟巢

def empty\_nests(self, nest, Lb, Ub, pa):

# 一小部分较差的巢穴被发现的概率为pa

n = len(nest)

nd = len(nest[0])

# 发现与否——一个状态向量

# Matlab code: K=rand(size(nest))>pa

K\_temp = [[random.random() for idx in range(nd)] for row in range(n)]

K = [[0 for idx in range(nd)] for row in range(n)]

for i in range(n):

for j in range(nd):

if K\_temp[i][j] > pa:

K[i][j] = 1

else:

K[i][j] = 0

# 在现实世界中，如果布谷鸟的蛋和寄主的非常相似，那么

# 这只布谷鸟的蛋不太可能被发现，因此适合度应该

# 与解的不同有关。因此，这是一个好主意

# 用随机步长有偏差的方法进行随机行走.

# 偏置/选择性随机游动的新解

# Matlab: stepsize=rand\*(nest(randperm(n),:)-nest(randperm(n),:));

# randperm(n): 返回一个元素值为从1到n的随机整数的矩阵

stepsize = [[0 for idx in range(nd)] for row in range(n)]

nest\_idx1\_ran = [i for i in range(n)]

random.shuffle(nest\_idx1\_ran)

nest\_idx2\_ran = [i for i in range(n)]

random.shuffle(nest\_idx2\_ran)

size\_cofe = random.random()

for i in range(n):

idx1 = nest\_idx1\_ran[i]

idx2 = nest\_idx2\_ran[i]

for j in range(nd):

stepsize[i][j] = size\_cofe\*(nest[idx1][j]-nest[idx2][j])

# Matlab: new\_nest=nest+stepsize.\*K

new\_nest = [[0 for idx in range(nd)] for row in range(n)]

for i in range(n):

for j in range(nd):

new\_nest[i][j] = nest[i][j] + stepsize[i][j] \* K[i][j]

for i in range(n):

# print new\_nest[j]

new\_nest[i] = self.simplebounds(new\_nest[i], Lb, Ub)

# print new\_nest[j]

# new\_nest[j]=simplebounds(new\_nest[j],self.Lb,self.self.Ub)

return new\_nest

# 简单约束的应用

def simplebounds(self, s, Lb, Ub):

# 应用下界

# matlab code:

# ns\_tmp = copy.deepcopy(s)

# I=ns\_tmp<self.Lb

# ns\_tmp(I)=self.Lb(I)

for i in range(len(s)):

if s[i] < Lb[i]:

s[i] = Lb[i]

else:

pass

# 应用上界

# J=ns\_tmp>self.self.Ub

# ns\_tmp(J)=self.self.Ub(J)

for i in range(len(s)):

if s[i] > Ub[i]:

s[i] = Ub[i]

else:

pass

# 更新这个新动作

# s=ns\_tmp

return s

# 您可以用自己的函数替换以下内容

# 一个d维目标函数

def fobj(self, u):

y = []

for j in range(0, 40):

ay = u[0]

by = u[1]

c = random.randint(0, 1)

if c == 0:

continue

y.append([ay, by])

P\_n = 0

for k in range(0, len(X)):

p\_n = 0

for o in range(0, len(y)):

d = np.sqrt(np.square(X[k][0] - y[o][0]) + np.square(X[k][1] - y[o][1]))

p = 100 / np.square(d + 40)

p\_n += p

# p\_n += float(format(p, '.4f'))

P = p\_n

if P >= 0.3:

P = 0.3

P = float(format(P, '.4f'))

P\_n += P

P\_n = float(format(P\_n, '.4f'))

z = 100 - P\_n

return z

cs = Cuckoo\_Search(100, 2, 0.25, [0, 0], [99, 99], 10)

fmin, bestnest = cs.cs\_run()

y = []

for j in range(0, 40):

ay = bestnest[0]

by = bestnest[1]

# ay = float(format(ay, '.4f'))

# by = float(format(by, '.4f'))

c = random.randint(0, 1)

if c == 0:

continue

# print(c)

y.append([int(ay), int(by)])

# print('y=', y)

print('len=', len(y))

P\_n = 0

for k in range(0, len(X)):

p\_n = 0

for o in range(0, len(y)):

d = np.sqrt(np.square(X[k][0]-y[o][0]) + np.square(X[k][1]-y[o][1]))

p = 100 / np.square(d + 40)

p\_n += p

# p\_n += float(format(p, '.4f'))

P = p\_n

if P >= 0.3:

P = 0.3

P\_n += P

P = float(format(P, '.4f'))

# print('P=', P)

P\_n = float(format(P\_n, '.4f'))

print('P\_n\_fitness', P\_n)

class PSO(object):

def \_\_init\_\_(self, population\_size, max\_steps):

self.w = 0.6 # 惯性权重

self.c1 = self.c2 = 2

self.population\_size = population\_size # 粒子群数量

self.dim = 2 # 搜索空间的维度

self.max\_steps = max\_steps # 迭代次数

self.x\_bound = [0, 99] # 解空间范围

self.x = np.random.randint(self.x\_bound[0], self.x\_bound[1],

(self.population\_size, self.dim)) # 初始化粒子群位置

self.v = np.random.rand(self.population\_size, self.dim) # 初始化粒子群速度

fitness = self.calculate\_fitness(self.x)

self.p = self.x # 个体的最佳位置

self.pg = self.x[np.argmin(fitness)] # 全局最佳位置

self.individual\_best\_fitness = fitness # 个体的最优适应度

self.global\_best\_fitness = np.max(fitness) # 全局最佳适应度

def calculate\_fitness(self, x):

P\_n = 0

for k in range(0, len(X)):

p\_n = 0

for o in range(0, len(x)):

Y = x[o]

d = np.sqrt(np.square(X[k][0]-Y[0]) + np.square(X[k][1]-Y[1]))

p = 100 / ((d + 40) \*\* 2)

p\_n += p

P = p\_n

if P >= 0.3:

P = 0.3

P\_n += P

list = []

list.append(100 - P\_n)

# print(list)

for p in range(0, len(x)-1):

list.append(100)

array = np.array(list)

# print(array)

return array

# return z

def evolve(self):

for step in range(self.max\_steps):

r1 = np.random.rand(self.population\_size, self.dim)

r2 = np.random.rand(self.population\_size, self.dim)

# 更新速度和权重

self.v = self.w \* self.v + self.c1 \* r1 \* (self.p - self.x) + self.c2 \* r2 \* (self.pg - self.x)

self.x = self.v + self.x

fitness = self.calculate\_fitness(self.x)

# 需要更新的个体

update\_id = np.greater(self.individual\_best\_fitness, fitness)

self.p[update\_id] = self.x[update\_id]

self.individual\_best\_fitness[update\_id] = fitness[update\_id]

# 新一代出现了更小的fitness，所以更新全局最优fitness和位置

if np.min(fitness) < self.global\_best\_fitness:

self.pg = self.x[np.argmin(fitness)]

self.global\_best\_fitness = np.min(fitness)

# print('best fitness: %.5f, mean fitness: %.5f' % (100-self.global\_best\_fitness, np.mean(fitness)))

pso = PSO(len(y), 100)

pso.evolve()

print('global\_best\_fitness=%.4f' % (100 - pso.global\_best\_fitness))

# print('pg=[%.4f %.4f]' % (pso.pg[0], pso.pg[1]))

# print('fitness=', )

# print('pg=', 100 - pso.pg)

class BA(object):

def \_\_init\_\_(self, d, N\_p, N\_gen, Qmin, Qmax, lower\_bound, upper\_bound, func):

self.d = d # 搜索维度

self.N\_p = N\_p # 个体数

self.N\_gen = N\_gen # 迭代次数

self.A = 1 + np.random.random(self.N\_p) # 响度

self.r = np.random.random(self.N\_p) # 脉冲发射率

self.Qmin = Qmin # 最小频率

self.Qmax = Qmax # 最大频率

self.lower\_bound = lower\_bound # 搜索区间下限

self.upper\_bound = upper\_bound # 搜索区间上限

self.func = func # 目标函数

self.alpha = 0.85

self.gamma = 0.9

self.r0 = self.r

self.Lb = self.lower\_bound \* np.ones(self.d)

self.Ub = self.upper\_bound \* np.ones(self.d)

self.Q = np.zeros(self.N\_p) # 频率

self.v = np.zeros((self.N\_p, self.d)) # 速度

self.sol = np.zeros((self.N\_p, self.d)) # 种群

self.fitness = np.zeros(self.N\_p) # 个体适应度

self.best = np.zeros(self.d) # 最好的位置

self.fmin = 0.0 # 最小fitness

# 初始化蝙蝠种群

def init\_bat(self):

for i1 in range(self.N\_p):

self.sol[i1, :] = self.Lb + (self.Ub - self.Lb) \* np.random.uniform(0, 1, self.d)

self.fitness[i1] = self.func(self.sol[i1, :])

self.fmin = np.min(self.fitness)

fmin\_arg = np.argmin(self.fitness)

self.best = self.sol[fmin\_arg, :]

# 越界检查

def simplebounds(self, s, lb, ub):

for j1 in range(self.d):

if s[j1] < lb[j1]:

s[j1] = lb[j1]

if s[j1] > ub[j1]:

s[j1] = ub[j1]

return s

# 迭代部分

def start\_iter(self):

S = np.zeros((self.N\_p, self.d))

self.init\_bat()

for step in range(self.N\_gen):

for i2 in range(self.N\_p):

self.Q[i2] = self.Qmin + (self.Qmin - self.Qmax) \* np.random.uniform(0, 1)

self.v[i2, :] = self.v[i2, :] + (self.sol[i2, :] - self.best) \* self.Q[i2]

S[i2, :] = self.sol[i2, :] + self.v[i2, :]

S[i2, :] = self.simplebounds(S[i2, :], self.Lb, self.Ub) # 越界检查

if np.random.random() > self.r[i2]:

S[i2, :] = self.best + 0.001 \* np.random.randn(self.d) # 此处没有实现乘以响度平均值

S[i2, :] = self.simplebounds(S[i2, :], self.Lb, self.Ub) # 越界检查

Fnew = self.func(S[i2, :])

if (Fnew <= self.fitness[i2]) and (np.random.random() < self.A[i2]):

self.sol[i2, :] = S[i2, :]

self.fitness[i2] = Fnew

self.A[i2] = self.alpha \* self.A[i2] # 响度更新

self.r[i2] = self.r0[i2] \* (1 - np.exp(-1 \* self.gamma \* step)) # 脉冲发射率更新

if Fnew <= self.fmin:

self.best = S[i2, :]

self.fmin = Fnew

# print(step, ':', '\n', 'BEST=', self.best, '\n', 'min of fitness=', self.fmin)

return self.best, self.fmin

def func(x):

# print('x=', x)

x = list(x)

P\_n = 0

for k in range(0, len(X)):

p\_n = 0

for i in range(0, len(x), 2):

d = np.sqrt(np.square(X[k][0] - int(x[i])) + np.square(X[k][1] - int(x[i+1])))

p = 100 / ((d + 40) \*\* 2)

p\_n += p

P = p\_n

# print('P=', P)

if P>= 0.3:

P = 0.3

P\_n += P

# print('p\_n', P\_n)

z = 100 - P\_n

# print('z=', z)

return z

ba = BA(len(y)\*2, 100, 100, 0, 2, 0, 99, func)

best, fmin = ba.start\_iter()

for k in range(0, len(X)):

p\_n = 0

for i in range(0, len(best), 2):

d = np.sqrt(np.square(X[k][0] - int(best[i])) + np.square(X[k][1] - int(best[i + 1])))

p = 100 / ((d + 40) \*\* 2)

p\_n += p

P = p\_n

if P >= 0.3:

P = 0.3

print('P=%.4f' % P)

print('min of fitness=%.4f' % (100-fmin))

print('best\_location', best)

t2 = time()

f.write(str(X)+'\n')

f.write(str(y)+'\n')

f.write(str(len(y))+'\n')

f.write(str(P\_n)+'\n')

f.write(str(100-pso.global\_best\_fitness)+'\n')

f.write(str(100-fmin)+'\n')

print('time=%f s' % (t2-t1))