|  |
| --- |
| 遗传算法运行程序(python) |
| # -\*- encoding: utf-8 -\*-  import random  import math  import pos\_data  import sys  if sys.version\_info.major < 3:  import Tkinter  else:  import tkinter as Tkinter    from GA import GA  pos\_list = pos\_data.pos\_list  class TSP\_WIN(object):  def \_\_init\_\_(self, aRoot, aLifeCount = 100, aWidth = 560, aHeight = 330):  self.root = aRoot  self.lifeCount = aLifeCount  self.width = aWidth  self.height = aHeight  self.canvas = Tkinter.Canvas(  self.root,  width = self.width,  height = self.height,  )  self.canvas.pack(expand = Tkinter.YES, fill = Tkinter.BOTH)  self.bindEvents()  self.initCitys()  self.new()  self.title("TSP")  def initCitys(self):  self.citys = pos\_data.pos\_list  #坐标变换  minX, minY = self.citys[0][0], self.citys[0][1]  maxX, maxY = minX, minY  for city in self.citys[1:]:  if minX > city[0]:  minX = city[0]  if minY > city[1]:  minY = city[1]  if maxX < city[0]:  maxX = city[0]  if maxY < city[1]:  maxY = city[1]  w = maxX - minX  h = maxY - minY  xoffset = 30  yoffset = 30  ww = self.width - 2 \* xoffset  hh = self.height - 2 \* yoffset  xx = ww / float(w)  yy = hh / float(h)  r = 5  self.nodes = []  self.nodes2 = []  for city in self.citys:  x = (city[0] - minX ) \* xx + xoffset  y = hh - (city[1] - minY) \* yy + yoffset  self.nodes.append((x, y))  node = self.canvas.create\_oval(x - r, y -r, x + r, y + r,  fill = "#ff0000",  outline = "#000000",  tags = "node",)  self.nodes2.append(node)    def distance(self, order):  distance = 0.0  for i in range(-1, len(self.citys) - 1):  index1, index2 = order[i], order[i + 1]  city1, city2 = self.citys[index1], self.citys[index2]  distance += math.sqrt((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2)  return distance  def matchFun(self):  return lambda life: 1.0 / self.distance(life.gene)  def title(self, text):  self.root.title(text)  def line(self, order):  self.canvas.delete("line")  for i in range(-1, len(order) -1):  p1 = self.nodes[order[i]]  p2 = self.nodes[order[i + 1]]  self.canvas.create\_line(p1, p2, fill = "#000000", tags = "line")    def bindEvents(self):  self.root.bind("n", self.new)  self.root.bind("g", self.start)  self.root.bind("s", self.stop)  def get\_distance(self,p1,p2):  return math.sqrt(math.pow(p1[0]-p2[0],2)+math.pow(p1[1]-p2[1],2))  def get\_list(self):  tdm()  D = []  for j in range(280):  dis = []  for i in range(280):  if i==j:  dis.append(9e9)  else:  dis.append(self.get\_distance(pos\_list[j],pos\_list[i]))    D.append(dis)  return D  def new(self, evt = None):  self.isRunning = False  order = range(len(self.citys))  self.line(order)  self.ga = GA(aCrossRate = 0.8,  aMutationRage = 0.02,  aLifeCount = self.lifeCount,  aGeneLenght = len(self.citys),  aD = self.get\_list(),  aMatchFun = self.matchFun())  def start(self, evt = None):  self.isRunning = True  while self.isRunning:  self.ga.next()  distance = self.distance(self.ga.best.gene)  self.line(self.ga.best.gene)  self.title("TSP-gen: %d" % self.ga.generation)  print("%d : %f"%(self.ga.generation,distance))  self.canvas.update()  def stop(self, evt = None):  self.isRunning = False  def mainloop(self):  self.root.mainloop()  print(self.ga.best.gene)  def main():  #tsp = TSP()  #tsp.run(10000)  tsp = TSP\_WIN(Tkinter.Tk())  tsp.mainloop()  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| 遗传算法主程序 |
| # -\*- coding: utf-8 -\*-  import random  import math  import pos\_data  from Life import Life  pos\_list=pos\_data.pos\_list  class GA(object):  """遗传算法类"""  def \_\_init\_\_(self, aCrossRate, aMutationRage, aLifeCount, aGeneLenght, aD, aMatchFun = lambda life : 1):  self.croessRate = aCrossRate  self.mutationRate = aMutationRage  self.lifeCount = aLifeCount  self.geneLenght = aGeneLenght  self.matchFun = aMatchFun # 适配函数  self.lives = [] # 种群  self.best = None # 保存这一代中最好的个体  self.generation = 1  self.D = aD  self.crossCount = 0  self.mutationCount = 0  self.bounds = 0.0 # 适配值之和，用于选择是计算概率  dis\_min = []  for i in range(self.geneLenght):  \_temp = self.D[i][0]  for j in range(self.geneLenght):  if \_temp > self.D[i][j]:  \_temp = self.D[i][j]  temp = j  dis\_min.append(temp)  self.Min = dis\_min  self.initPopulation()  def initPopulation(self):  """初始化种群"""  path\_solve = pos\_data.path\_solve  path\_greddy = pos\_data.path\_greddy  self.lives = []  for i in range(self.lifeCount):  gene = [x for x in range(0,self.geneLenght)]  # gene = []  # rate =random.randint(1,100)  # if rate > 55:  # gene.extend(path\_greddy)  # else:  # gene.extend(path\_solve)  # rate =random.randint(1,100)  # if rate > 50:  # random.shuffle(gene)  random.shuffle(gene)  # gene.insert(0,0)  # gene.append(0)  life = Life(gene)  self.lives.append(life)  def judge(self):  """评估，计算每一个个体的适配值"""  self.bounds = 0.0  self.best = self.lives[0]  for life in self.lives:  life.score = self.matchFun(life)  self.bounds += life.score  if self.best.score < life.score:  self.best = life  def get\_distance(self,c1,c2):  p1=pos\_list[c1]  p2=pos\_list[c2]  return math.sqrt(math.pow(p1[0]-p2[0],2)+math.pow(p1[1]-p2[1],2))    def miner\_one(self,index,lis):  num = int(index)  if num==0:  num = len(lis) - 1  else:  num -= 1  return num  def plus\_one(self,index,lis):  num = int(index)  if num==(len(lis) - 1):  num = 0  else:  num += 1  return num  def cross(self, parent1, parent2):  index = random.randint(0, self.geneLenght - 1)  rp1,rp2=list(parent1.gene),list(parent2.gene)  lp1,lp2=list(parent1.gene),list(parent2.gene)  index2 = int(index)  node = rp1[index]  newGene1,newGene2 = [node],[node]  rp1.remove(node)  rp2.remove(node)  lp1.remove(node)  lp2.remove(node)  while(len(rp1)>=1):  temp\_index = self.miner\_one(index,rp1)  d1 = self.D[rp1[temp\_index]][node]  d2 = self.D[rp2[temp\_index]][node]  if d1<d2:  node = rp1[temp\_index]  else:  node = rp2[temp\_index]  index = temp\_index  newGene1.append(node)  rp1.remove(node)  rp2.remove(node)  while(len(lp1)>=1):  temp\_index = self.plus\_one(index,lp1)  d1 = self.D[lp1[temp\_index]][node]  d2 = self.D[lp2[temp\_index]][node]  if d1<d2:  node = lp1[temp\_index]  else:  node = lp2[temp\_index]  index2 = temp\_index  newGene2.append(node)  lp1.remove(node)  lp2.remove(node)  self.crossCount += 1  return newGene1,newGene2  def cross\_bk(self, parent1, parent2):  """交叉"""  index1 = random.randint(0, self.geneLenght - 1)  index2 = random.randint(index1, self.geneLenght - 1)  tempGene = parent2.gene[index1:index2] # 交叉的基因片段  newGene = []  p1len = 0  for g in parent1.gene:  if p1len == index1:  newGene.extend(tempGene) # 插入基因片段  p1len += 1  if g not in tempGene:  newGene.append(g)  p1len += 1  self.crossCount += 1  return newGene  def mutation\_bk(self, gene):  index = random.randint(1, self.geneLenght - 1)  min\_to = self.Min[gene[index]]  index2 = gene.index(min\_to)  if index < index2:  part1 = gene[0:index+1]  part2 = gene[index+1:index2+1][::-1]  part2.reverse()  part3 = gene[index2+1:]  else:  part1 = gene[0:index2]  part2 = gene[index2:index][::-1]  part3 = gene[index:]  newGene = []  newGene.extend(part1)  newGene.extend(part2)  newGene.extend(part3)  self.mutationCount += 1  return newGene  def mutation(self, gene):  """突变"""  index1 = random.randint(1, self.geneLenght - 2)  index2 = random.randint(1, self.geneLenght - 2)  newGene = gene[:] # 产生一个新的基因序列，以免变异的时候影响父种群  newGene[index1], newGene[index2] = newGene[index2], newGene[index1]  self.mutationCount += 1  return newGene  def getOne(self):  """选择一个个体"""  r = random.uniform(0, self.bounds)  for life in self.lives:  r -= life.score  if r <= 0:  return life  raise Exception("选择错误", self.bounds)  def newChild(self):  """产生新后的"""  parent1 = self.getOne()  parent2 = self.getOne()  rate = random.random()  # 按概率交叉  if rate < self.croessRate:  # 交叉  gene1,gene2 = self.cross(parent1, parent2)  else:  gene1,gene2 = parent1.gene,parent2.gene  # 按概率突变  rate = random.random()  if rate < self.mutationRate:  gene1,gene2 = self.mutation(gene1),self.mutation\_bk(gene2)  return Life(gene1),Life(gene2)  def next(self):  """产生下一代"""  self.judge()  newLives = []  newLives.append(self.best) #把最好的个体加入下一代  while len(newLives) < self.lifeCount-1:  child1,child2=self.newChild()  newLives.append(child1)  newLives.append(child2)  if len(newLives) < self.lifeCount:  newLives.append(child1)  self.lives = newLives  self.generation += 1  class Life(object):  """个体类"""  def \_\_init\_\_(self, aGene = None):  self.gene = aGene  self.score = -1 |
| 贪婪算法 |
| from \_\_future\_\_ import print\_function, division  from itertools import islice  from array import array as pyarray  ################################################################################  # A simple algorithm for solving the Travelling Salesman Problem  # Finds a suboptimal solution  ################################################################################  if "xrange" not in globals():  #py3  xrange = range  else:  #py2  pass    def optimize\_solution( distances, connections ):  """Tries to optimize solution, found by the greedy algorithm"""  N = len(connections)  path = restore\_path( connections )  def ds(i,j): #distance between ith and jth points of path  return distances[path[i]][path[j]]  d\_total = 0.0  optimizations = 0  for a in xrange(N-1):  b = a+1  for c in xrange( b+2, N-1):  d = c+1  delta\_d = ds(a,b)+ds(c,d) -( ds(a,c)+ds(b,d))  if delta\_d > 0:  d\_total += delta\_d  optimizations += 1  connections[path[a]].remove(path[b])  connections[path[a]].append(path[c])  connections[path[b]].remove(path[a])  connections[path[b]].append(path[d])  connections[path[c]].remove(path[d])  connections[path[c]].append(path[a])  connections[path[d]].remove(path[c])  connections[path[d]].append(path[b])  path[:] = restore\_path( connections )    return optimizations, d\_total    def restore\_path( connections ):  """Takes array of connections and returns a path.  Connections is array of lists with 1 or 2 elements.  These elements are indices of teh vertices, connected to this vertex  Guarantees that first index < last index  """  #there are 2 nodes with valency 1 - start and end. Get them.  start, end = [idx  for idx, conn in enumerate(connections)  if len(conn)==1 ]  path = [start]  prev\_point = None  cur\_point = start  while True:  next\_points = [pnt for pnt in connections[cur\_point]  if pnt != prev\_point ]  if not next\_points: break  next\_point = next\_points[0]  path.append(next\_point)  prev\_point, cur\_point = cur\_point, next\_point  return path  def pairs\_by\_dist(N, distances):  #Sort coordinate pairs by distance  indices = [None] \* (N\*(N-1)//2)  idx = 0  for i in xrange(N):  for j in xrange(i+1,N):  indices[idx] = (i,j)  idx += 1    indices.sort(key = lambda ij: distances[ij[0]][ij[1]])  return indices    def solve\_tsp( distances, optim\_steps=3, pairs\_by\_dist=pairs\_by\_dist ):  """Given a distance matrix, finds a solution for the TSP problem.  Returns list of vertex indices.  Guarantees that the first index is lower than the last"""  N = len(distances)  if N == 0: return []  if N == 1: return [0]  for row in distances:  if len(row) != N: raise ValueError( "Matrix is not square")  #State of the TSP solver algorithm.  node\_valency = pyarray('i', [2])\*N #Initially, each node has 2 sticky ends    #for each node, stores 1 or 2 connected nodes  connections = [[] for i in xrange(N)]  def join\_segments(sorted\_pairs):  #segments of nodes. Initially, each segment contains only 1 node  segments = [ [i] for i in xrange(N) ]    def filtered\_pairs():  #Generate sequence of  for ij in sorted\_pairs:  i,j = ij  if not node\_valency[i] or\  not node\_valency[j] or\  (segments[i] is segments[j]):  continue  yield ij  for i,j in islice( filtered\_pairs(), N-1 ):  node\_valency[i] -= 1  node\_valency[j] -= 1  connections[i].append(j)  connections[j].append(i)  #Merge segment J into segment I.  seg\_i = segments[i]  seg\_j = segments[j]  if len(seg\_j) > len(seg\_i):  seg\_i, seg\_j = seg\_j, seg\_i  i, j = j, i  for node\_idx in seg\_j:  segments[node\_idx] = seg\_i  seg\_i.extend(seg\_j)  join\_segments(pairs\_by\_dist(N, distances))  for passn in range(optim\_steps):  nopt, dtotal = optimize\_solution( distances, connections )  if nopt == 0:  break  path = restore\_path( connections )  return path |
| 贪婪算法运行程序 |
| from greedy import solve\_tsp  import math  import numpy as np  import matplotlib.pyplot as plt  import pos\_data  pos\_list = pos\_data.pos\_list  D = []  T = []  X = []  Y = []  def get\_list():  tdm()  for j in range(280):  dis = []  for i in range(280):  if i==j:  dis.append(9e9)  else:  dis.append((get\_distance(pos\_list[j],pos\_list[i])-T[i]-T[j])\*0.001)  # dis.append(get\_distance(pos\_list[j],pos\_list[i]))  D.append(dis)  def get\_distance(p1,p2):  return math.sqrt(math.pow(p1[0]-p2[0],2)+math.pow(p1[1]-p2[1],2))  def tdm():  x\_bar = reduce(lambda x,y:x+y, X)/280  y\_bar = reduce(lambda x,y:x+y, Y)/280  dis = []  for i in range(280):  dis.append(get\_distance([x\_bar,y\_bar],pos\_list[i]))  dis\_bar = reduce(lambda x,y:x+y, dis)/280  for i in range(280):  T.append(get\_distance([x\_bar,y\_bar],pos\_list[i])-dis\_bar)  def init():  for i in range(280):  X.append(pos\_list[i][0])  Y.append(pos\_list[i][1])  def get\_fit(path):  fit = 0 #functools.reduce(get\_distance,target)  for i in range(279):  dis = get\_distance(pos\_list[path[i]],pos\_list[path[i+1]])  # print('dis btw %d %d is %f'%(target[i],target[i+1],dis))  fit = fit + dis  return fit  init()  get\_list()  # path = solve\_tsp(D,12)  path= [ 40, 27, 14, 3, 2, 1, 0, 9, 12, 11, 10, 25, 24, 23, 32, 41, 56, 49, 57, 48, 64, 81, 82, 98, 112, 127, 146, 160, 159, 178, 177, 187, 198, 197, 188, 164, 165, 158, 145, 144, 128, 133, 134, 115, 114, 113, 99, 96, 83, 65, 71, 72, 67, 84, 97, 100, 116, 129, 147, 161, 175, 189, 196, 207, 214, 231, 215, 208, 209, 216, 233, 232, 247, 248, 259, 279, 278, 258, 246, 257, 277, 261, 252, 251, 260, 276, 256, 245, 234, 217, 210, 199, 190, 176, 162, 148, 131, 117, 101, 85, 75, 90, 106, 89, 102, 137, 132, 121, 138, 150, 167, 180, 192, 201, 218, 219, 220, 221, 228, 229, 230, 237, 236, 235, 255, 275, 274, 273, 269, 272, 270, 271, 268, 254, 267, 266, 265, 264, 253, 263, 262, 250, 249, 223, 211, 212, 213, 244, 206, 181, 182, 169, 168, 151, 139, 126, 125, 140, 152, 163, 183, 205, 224, 243, 242, 241, 240, 239, 238, 225, 226, 227, 222, 200, 204, 195, 191, 179, 186, 166, 174, 156, 173, 185, 194, 203, 202, 193, 184, 172, 154, 143, 142, 155, 171, 170, 153, 141, 124, 109, 93, 94, 110, 111, 95, 80, 79, 78, 63, 47, 38, 22, 36, 21, 20, 37, 45, 62, 46, 61, 77, 76, 92, 91, 108, 123, 122, 107, 118, 103, 86, 70, 55, 60, 73, 87, 104, 119, 136, 157, 149, 135, 130, 120, 105, 88, 74, 59, 43, 44, 34, 19, 18, 35, 31, 8, 7, 6, 5, 4, 16, 17, 33, 42, 58, 53, 68, 69, 54, 52, 66, 51, 50, 39, 26, 13, 15, 28, 30, 29]  # path.append(path[0])  print(get\_fit(path))  # print(path)  plt.figure()  plt.plot(X,Y,'bo')  plt.plot(X[40],Y[40],'ro')  plt.plot(X[29],Y[29],'go')  a = []  for i in range(279):  xp = X[path[i]]  yp = Y[path[i]]  xl = X[path[i+1]] - xp  yl = Y[path[i+1]] - yp  temp = [xp, yp, xl, yl]  a.append(temp)  # plt.setp(lines, color='r', linewidth=2.0)  soa = np.array(a)  px,py,u,v = zip(\*soa)  ax = plt.gca()  ax.quiver(px,py,u,v,color='b',angles='xy',scale\_units='xy',scale=1)  plt.draw()  # plt.plot(px,py)  plt.show() |
|  |
|  |