编程题

刘治学 2021091202022

问题：

将课程所学旅行商问题（TSP）应用于配送问题，并将生成的路径可视化。算法不限，可用启发式，也可以用精确解法。

要求：在成都市任取10个需求点。

解决：

**1.算法原理**

1.1 算法背景

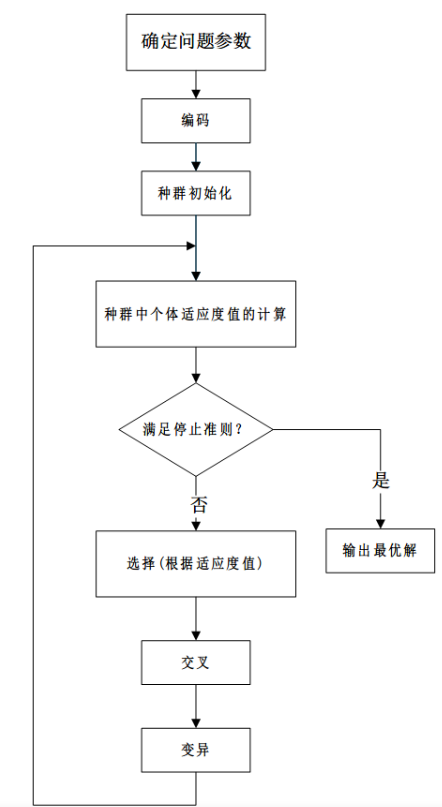
遗传算法是Holland教授于20世纪60年代提出，它主要借用了生物进化中“物竞天择，适者生存”的自然机理，通过选择、遗传和变异等机制，模拟自然进化过程来求解复杂问题的一种搜索算法。从某种程度上说遗传算法是对生物进化过程进行的数学仿真。与自然界类似，遗传算法从代表问题可能潜在解集的一个种群开始，将择优与随机信息结合起来并逐步迭代，在每一代中通过一定的规则，使用上代中适应性最好的个体，来形成新的种群，如此重复，直到满足指定条件为止。

1.2 算法和自然界的对应关系



1.3 遗传算法的主要步骤

遗传算法的控制参数包括群体规模N、算法停止准则以及交配概率pc和变异概率pm



**2.遗传算法的实现**

2.1 编码问题

对一个具体的应用问题如何编码是应用遗传算法的首要问题，也是遗传算法应用的难点。事实上，还不存在一种通用的编码方法，特殊的问题往往需要采用特殊的编码方法。目前几种常用的编码技术主要包括二进制编码、浮点数编码、字符编码等。

2.2 适应函数

遗传算法中的适应度函数也称为评价函数，是用来判断群体中的个体的优劣程度的指标，是根据所求问题的目标函数来进行评估的。适应度函数是算法演化过程的驱动力，是进行自然选择的唯一依据。在具体应用中，适应度函数的设计要结合求解问题本身的要求而定。适应度函数设计直接影响到遗传算法的性能。

非线性加速适应函数。该方法利用已有的信息构造适应函数。

线性加速适应函数。该方法是将当前得到的最优指标函数值放大为指标函数值平均值的M倍。

另一种定义适应函数的方法是利用染色体指标函数值从小到大的排列序号作为适应函数值，利用该值采用“赌轮盘”的方法得到每个染色体被选中的概率。

2.3 初始种群的产生

遗传算法中初始种群中的个体可以是随机产生的，但是最好采用如下策略来设定：

根据问题的固有知识，设法把握最优解所占空间在整个问题空间中的分布范围，然后，在此分布范围内设定初始群体。

先随机生成一定数目的个体，然后从中挑出最好的个体加入到初始群体中。这种过程不断迭代，直到初始群体中个体数达到了预先确定的规模。

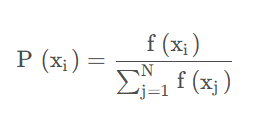
然后进行选择、交叉和变异等运算，解决问题。

2.4 选择策略

选择操作也称为复制，是指从当前种群中选出个体以生成交配池的过程。所选出的这些个体具有良好的特征，以便产生优良的后代。较大的选择压力使最优个体有较高的复制数目，从而使算法收敛速度快，但也较容易出现过早收敛的现象。相对而言，较小的选择压力一般能使群体保持足够的多样性，从而增大算法收敛到全局最优的概率，但算法的收敛速度一般较慢。

2.4.1 轮盘赌选择

轮盘赌选择（Roulette Wheel Selection）策略在遗传算法中使用的最多。在轮盘赌选择方法中先按个体的选择概率产生一个轮盘，轮盘每个区的角度与个体的选择概率成正比，然后产生一个随机数，它落入转盘的哪个区域就选择相应的个体交叉。很显然，选择概率大的个体被选中的可能性较大，获得交叉的机会也就越大。



在实际计算时，可以按照个体顺序求出每个个体的累积概率，然后产生一个随机数，它落入累积概率的哪个区域就选择相应的个体交叉。

2.4.2 锦标赛选择

锦标赛选择（Tournament Selection）是先在群体中随机选择 k 个个体（放回或不放回）进行比较，适应值最好的个体被选择作为生成下一代的父体。反复执行该过程，直到下一代个体数量达到预定的群体规模。参数k称为竞赛规模，一般取k = 2。

锦标赛选择常常比轮盘赌选择方法能得到更加多样化的群体。这种方法使得适应值好的个体具有较大的生存机会，与此同时，由于它只使用适应值的相对值作为选择的标准，而与适应值的数值大小不成直接比例，从而能避免超级个体的影响，一定程度上避免了过早收敛和停滞现象的发生。

2.5 交配（交叉）策略

当两个生物机体配对或者复制时，它们的染色体相互混合，产生一个由双方基因组成的全新的染色体组。这一过程称为重组（recombination）或者交叉（crossover）。交叉得到的后代可能继承了上代的优良基因，其后代会比它们的父母更加优秀，但也可能继承了上代的不良基因，其后代则会比它们的父母差，难以生存，甚至不能再复制自己。越能适应环境的后代越能继续复制自己并将其基因传给后代。因此，每一代总是比其父母一代生存和复制得更好。交叉的具体步骤如下：

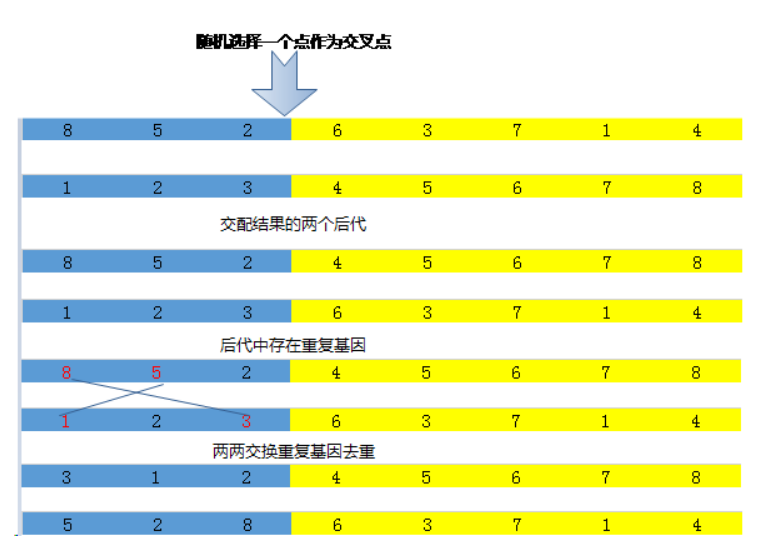
① 从交配池中随机取出要交配的一对个体；

② 根据位串长度L，对要交配的一对个体，随机选取[1, L-1]中一个或多个的整数k作为交叉点；

③ 根据交叉概率pc（0 < pc < 1）实施交叉操作，配对个体在交叉点处，相互交换各自的部分内容，从而形成一对新的个体。

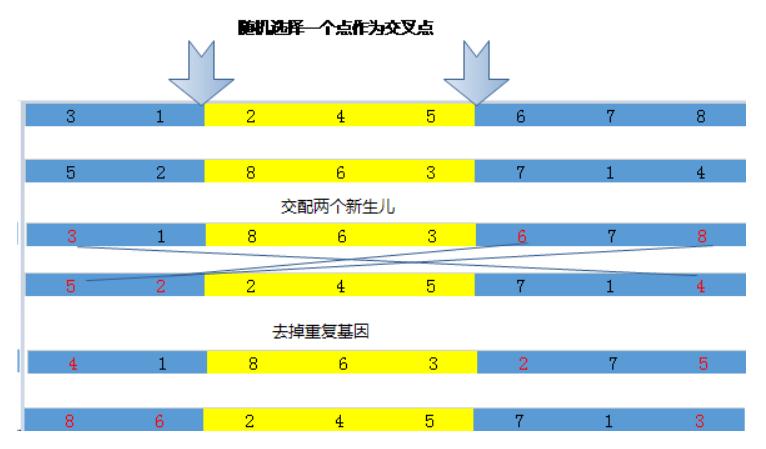
（1）一点交叉

一点交叉（Single-point Crossover）是最基础的一种交叉方式。从交配池中随机选择两个个体，然后随机选择一个交叉点，该点前后的两个个体的部分结构进行互换，并生成两个新的个体。Holland教授曾指出，一点交叉算子不利于长距模式的保留和重组，而且位串末尾的重要基因总是被交换，因此在实际应用中采用的并不多。



（2）两点交叉

两点交叉（Two-point Crossover）的操作与一点交叉类似，只是设置了两个交叉点，将两个交叉点之间的串互相交换。



（3）多点交叉

类似于两点交叉，多点交叉（Multipoint Crossover）是指对于选定的两个个体，随机选择多个交叉点，然后进行基因交换。

2.6 变异策略

对于位串而言，变异可以使串中的某些位的数字发生变化。变异与选择和交叉结合在一起，保证了遗传算法的有效性，使遗传算法具有局部的搜集搜索能力；同时使得遗传算法保持种群的多样性，以防止非成熟收敛。在变异操作中，变异概率不能取的太大，如果变异概率大于0.5，则遗传算法就退化为随机算法，而遗传算法的一些重要特性和搜索能力也不复存在。

变异的具体操作为：对 Pt 中任一个体，随机产生一个实数，0 ≤ ρ ≤1，如果 ρ 大于已定义的变异概率的阈值 ρm，就对该个体进行变异。

基于位置的变异——2136457，变异后为2413657

基于次序的变异——2136457，变异后为2436157

打乱变异——2136457，一变异结果为2463157

**3.算法具体实现**

3.1 代码部分

我用了两个python程序，一个程序负责计算最短路径并且进行迭代，一个程序负责根据前面程序进行绘图。具体的代码都在文件夹当中打包好了。

3.1.1 main.py

1. import numpy as np
2. import pandas as pd
3. from draw import \*
4. class TSP(object):
5. citys = np.array([])  *# 城市数组*
6. citys\_name = np.array([])
7. pop\_size = 50  *# 种群大小*
8. c\_rate = 0.7  *# 交叉率*
9. m\_rate = 0.05  *# 突变率*
10. pop = np.array([])  *# 种群数组*
11. fitness = np.array([])  *# 适应度数组*
12. city\_size = -1  *# 标记城市数目*
13. ga\_num = 200  *# 最大迭代次数*
14. best\_dist = -1  *# 记录目前最优距离*
15. best\_gen = []  *# 记录目前最优旅行方案*
16. dw = Draw()  *# 绘图类*
17. def \_\_init\_\_(self, c\_rate, m\_rate, pop\_size, ga\_num):
18. self.fitness = np.zeros(self.pop\_size)
19. self.c\_rate = c\_rate
20. self.m\_rate = m\_rate
21. self.pop\_size = pop\_size
22. self.ga\_num = ga\_num
23. def init(self):
24. tsp = self
25. tsp.load\_Citys() *#加载城市数据*
26. tsp.pop = tsp.creat\_pop(tsp.pop\_size)  *# 创建种群*
27. tsp.fitness = tsp.get\_fitness(tsp.pop)  *# 计算初始种群适应度*
28. tsp.dw.bound\_x = [np.min(tsp.citys[:, 0]), np.max(tsp.citys[:, 0])]  *# 计算绘图时的X界*
29. tsp.dw.bound\_y = [np.min(tsp.citys[:, 1]), np.max(tsp.citys[:, 1])]  *# 计算绘图时的Y界*
30. tsp.dw.set\_xybound(tsp.dw.bound\_x, tsp.dw.bound\_y)  *# 设置边界*
31. def creat\_pop(self, size):
32. pop = []
33. for i in range(size):
34. gene = np.arange(self.citys.shape[0])  *# 问题的解，基因，种群中的个体：[0，...，city\_size]*
35. np.random.shuffle(gene)  *# 打乱数组[0，...，city\_size]*
36. pop.append(gene)  *# 加入种群*
37. return np.array(pop)
38. def get\_fitness(self, pop):
39. d = np.array([])  *# 适应度记录数组*
40. for i in range(pop.shape[0]):
41. gen = pop[i]  *# 取其中一条基因（编码解，个体）*
42. dis = self.gen\_distance(gen)  *# 计算此基因优劣（距离长短）*
43. dis = self.best\_dist / dis  *# 当前最优距离除以当前pop[i]（个体）距离；越近适应度越高，最优适应度为1*
44. d = np.append(d, dis)  *# 保存适应度pop[i]*
45. return d
46. def get\_local\_fitness(self, gen, i):
47. '''
48. 计算地i个城市的邻域
49. 交换基因数组中任意两个值组成的解集：称为邻域。计算领域内所有可能的适应度
50. :param gen:城市路径
51. :param i:第i城市
52. :return:第i城市的局部适应度
53. '''
54. di = 0
55. fi = 0
56. if i == 0:
57. di = self.ct\_distance(self.citys[gen[0]], self.citys[gen[-1]])
58. else:
59. di = self.ct\_distance(self.citys[gen[i]], self.citys[gen[i - 1]])
60. od = []
61. for j in range(self.city\_size):
62. if i != j:
63. od.append(self.ct\_distance(self.citys[gen[i]], self.citys[gen[i - 1]]))
64. mind = np.min(od)
65. fi = di - mind
66. return fi
67. def EO(self, gen):
68. *# 极值优化，传统遗传算法性能不好，这里混合EO*
69. *# 其会在整个基因的领域内，寻找一个最佳变换以更新基因*
70. local\_fitness = []
71. for g in range(self.city\_size):
72. f = self.get\_local\_fitness(gen, g)
73. local\_fitness.append(f)
74. max\_city\_i = np.argmax(local\_fitness)
75. maxgen = np.copy(gen)
76. if 1 < max\_city\_i < self.city\_size - 1:
77. for j in range(max\_city\_i):
78. maxgen = np.copy(gen)
79. jj = max\_city\_i
80. while jj < self.city\_size:
81. gen1 = self.exechange\_gen(maxgen, j, jj)
82. d = self.gen\_distance(maxgen)
83. d1 = self.gen\_distance(gen1)
84. if d > d1:
85. maxgen = gen1[:]
86. jj += 1
87. gen = maxgen
88. return gen
89. def select\_pop(self, pop):
90. *# 选择种群，优胜劣汰，策略1：低于平均的要替换改变*
91. best\_f\_index = np.argmax(self.fitness)
92. av = np.median(self.fitness, axis=0)
93. for i in range(self.pop\_size):
94. if i != best\_f\_index and self.fitness[i] < av:
95. pi = self.cross(pop[best\_f\_index], pop[i])
96. pi = self.mutate(pi)
97. *# d1 = self.distance(pi)*
98. *# d2 = self.distance(pop[i])*
99. *# if d1 < d2:*
100. pop[i, :] = pi[:]
101. return pop
102. def select\_pop2(self, pop):
103. *# 选择种群，优胜劣汰，策略2：轮盘赌，适应度低的替换的概率大*
104. probility = self.fitness / self.fitness.sum()
105. idx = np.random.choice(np.arange(self.pop\_size), size=self.pop\_size, replace=True, p=probility)
106. n\_pop = pop[idx, :]
107. return n\_pop
108. def cross(self, parent1, parent2):
109. """交叉p1,p2的部分基因片段"""
110. if np.random.rand() > self.c\_rate:
111. return parent1
112. index1 = np.random.randint(0, self.city\_size - 1)
113. index2 = np.random.randint(index1, self.city\_size - 1)
114. tempGene = parent2[index1:index2]  *# 交叉的基因片段*
115. newGene = []
116. p1len = 0
117. for g in parent1:
118. if p1len == index1:
119. newGene.extend(tempGene)  *# 插入基因片段*
120. if g not in tempGene:
121. newGene.append(g)
122. p1len += 1
123. newGene = np.array(newGene)
124. if newGene.shape[0] != self.city\_size:
125. print('c error')
126. return self.creat\_pop(1)
127. *# return parent1*
128. return newGene
129. def mutate(self, gene):
130. """突变"""
131. if np.random.rand() > self.m\_rate:
132. return gene
133. index1 = np.random.randint(0, self.city\_size - 1)
134. index2 = np.random.randint(index1, self.city\_size - 1)
135. newGene = self.reverse\_gen(gene, index1, index2)
136. if newGene.shape[0] != self.city\_size:
137. print('m error')
138. return self.creat\_pop(1)
139. return newGene
140. def reverse\_gen(self, gen, i, j):
141. *# 函数：翻转基因中i到j之间的基因片段*
142. if i >= j:
143. return gen
144. if j > self.city\_size - 1:
145. return gen
146. parent1 = np.copy(gen)
147. tempGene = parent1[i:j]
148. newGene = []
149. p1len = 0
150. for g in parent1:
151. if p1len == i:
152. newGene.extend(tempGene[::-1])  *# 插入基因片段*
153. if g not in tempGene:
154. newGene.append(g)
155. p1len += 1
156. return np.array(newGene)
157. def exechange\_gen(self, gen, i, j):
158. *# 函数：交换基因中i,j值*
159. c = gen[j]
160. gen[j] = gen[i]
161. gen[i] = c
162. return gen
163. def evolution(self):
164. *# 主程序：迭代进化种群*
165. tsp = self
166. for i in range(self.ga\_num):
167. best\_f\_index = np.argmax(tsp.fitness)
168. worst\_f\_index = np.argmin(tsp.fitness)
169. local\_best\_gen = tsp.pop[best\_f\_index]
170. local\_best\_dist = tsp.gen\_distance(local\_best\_gen)
171. if i == 0:
172. tsp.best\_gen = local\_best\_gen
173. tsp.best\_dist = tsp.gen\_distance(local\_best\_gen)
174. if local\_best\_dist < tsp.best\_dist:
175. tsp.best\_dist = local\_best\_dist  *# 记录最优值*
176. tsp.best\_gen = local\_best\_gen  *# 记录最个体基因*
177. *# 绘图*
178. tsp.dw.ax.cla()
179. tsp.re\_draw()
180. tsp.dw.plt.pause(0.001)
181. else:
182. tsp.pop[worst\_f\_index] = self.best\_gen
183. print('迭代次数:%d 次 , 最短路径长度 :%s' % (i+1, self.best\_dist))
184. tsp.pop = tsp.select\_pop(tsp.pop)  *# 选择淘汰种群*
185. tsp.fitness = tsp.get\_fitness(tsp.pop)  *# 计算种群适应度*
186. for j in range(self.pop\_size):
187. r = np.random.randint(0, self.pop\_size - 1)
188. if j != r:
189. tsp.pop[j] = tsp.cross(tsp.pop[j], tsp.pop[r])  *# 交叉种群中第j,r个体的基因*
190. tsp.pop[j] = tsp.mutate(tsp.pop[j])  *# 突变种群中第j个体的基因*
191. *# self.best\_gen = self.EO(self.best\_gen) #极值优化，防止收敛局部最优*
192. tsp.best\_dist = tsp.gen\_distance(self.best\_gen)  *# 记录最优值*
193. def load\_Citys(self, file='chengdu.csv', delm=';'):
194. *# 中国34城市经纬度*
195. data = pd.read\_csv(file, delimiter=delm, header=None).values
196. self.citys = data[:, 1:]
197. self.citys\_name = data[:, 0]
198. self.city\_size = data.shape[0]
199. def gen\_distance(self, gen):
200. *# 计算基因所代表的总旅行距离*
201. distance = 0.0
202. for i in range(-1, len(self.citys) - 1):
203. index1, index2 = gen[i], gen[i + 1]
204. city1, city2 = self.citys[index1], self.citys[index2]
205. distance += np.sqrt((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2)
206. return distance
207. def ct\_distance(self, city1, city2):
208. *# 计算2城市之间的欧氏距离*
209. d = np.sqrt((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2)
210. return d
211. def draw\_citys\_way(self, gen):
212. '''
213. 根据一条基因gen绘制一条旅行路线
214. :param gen:
215. :return:
216. '''
217. tsp = self
218. dw = self.dw
219. m = gen.shape[0]
220. tsp.dw.set\_xybound(tsp.dw.bound\_x, tsp.dw.bound\_y)
221. for i in range(m):
222. if i < m - 1:
223. best\_i = tsp.best\_gen[i]
224. next\_best\_i = tsp.best\_gen[i + 1]
225. best\_icity = tsp.citys[best\_i]
226. next\_best\_icity = tsp.citys[next\_best\_i]
227. dw.draw\_line(best\_icity, next\_best\_icity)
228. start = tsp.citys[tsp.best\_gen[0]]
229. end = tsp.citys[tsp.best\_gen[-1]]
230. dw.draw\_line(end, start)
231. def draw\_citys\_name(self, gen, size=5):
232. '''
233. 根据一条基因gen绘制对应城市名称
234. :param gen:
235. :param size: text size
236. :return:
237. '''
238. tsp = self
239. m = gen.shape[0]
240. tsp.dw.set\_xybound(tsp.dw.bound\_x, tsp.dw.bound\_y)
241. for i in range(m):
242. c = gen[i]
243. best\_icity = tsp.citys[c]
244. tsp.dw.draw\_text(best\_icity[0], best\_icity[1], tsp.citys\_name[c], 10)
245. def re\_draw(self):
246. *# 重绘图；每次迭代后绘制一次，动态展示。*
247. tsp = self
248. tsp.dw.draw\_points(tsp.citys[:, 0], tsp.citys[:, 1])
249. tsp.draw\_citys\_name(tsp.pop[0], 8)
250. tsp.draw\_citys\_way(self.best\_gen)
251. def main():
252. tsp = TSP(0.5, 0.1, 100, 500)
253. tsp.init()
254. tsp.evolution()
255. tsp.re\_draw()
256. tsp.dw.plt.show()
257. *#输出最终结果*
258. print(f"迭代{tsp.ga\_num}次之后，最终的最短路径如图，最短路径的长度是{tsp.best\_dist}")
259. if \_\_name\_\_ == '\_\_main\_\_':
260. main()

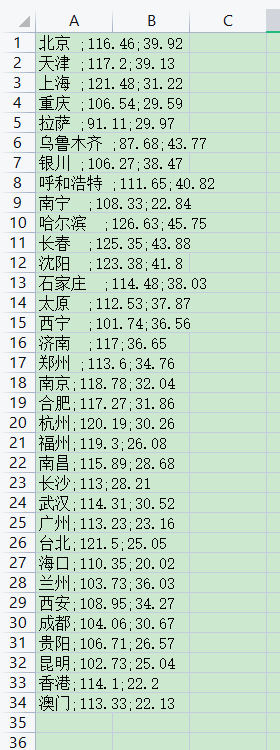
3.1.2 draw.py

1. import matplotlib.pyplot as plt
2. from matplotlib.lines import Line2D
3. import matplotlib.animation as animation
4. class Draw(object):
5. bound\_x = []
6. bound\_y = []
7. def \_\_init\_\_(self):
8. self.fig, self.ax = plt.subplots()
9. self.plt = plt
10. self.set\_font()
11. def draw\_line(self, p\_from, p\_to):
12. line1 = [(p\_from[0], p\_from[1]), (p\_to[0], p\_to[1])]
13. (line1\_xs, line1\_ys) = zip(\*line1)
14. self.ax.add\_line(Line2D(line1\_xs, line1\_ys, linewidth=1, color='blue'))
15. def draw\_points(self, pointx, pointy):
16. self.ax.plot(pointx, pointy, 'ro')
17. def set\_xybound(self, x\_bd, y\_bd):
18. self.ax.axis([x\_bd[0], x\_bd[1], y\_bd[0], y\_bd[1]])
19. def draw\_text(self, x, y, text, size=8):
20. self.ax.text(x, y, text, fontsize=size)
21. def set\_font(self, ft\_style='SimHei'):
22. plt.rcParams['font.sans-serif'] = [ft\_style]  *# 用来正常显示中文标签*

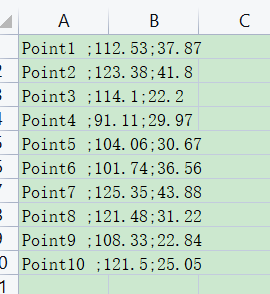
3.2 测试数据

我在文件当中准备了两组数据，分别是chengdu.csv和china.csv，分别存储了随机的十个点数据和中国所有省会城市的坐标数据。

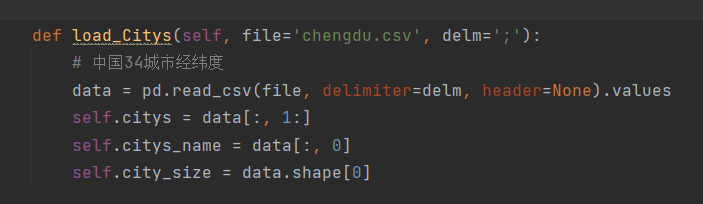
china.csv



chengdu.csv



如何选择这两个数据，只需要在main.py当中修改如下就可以了：



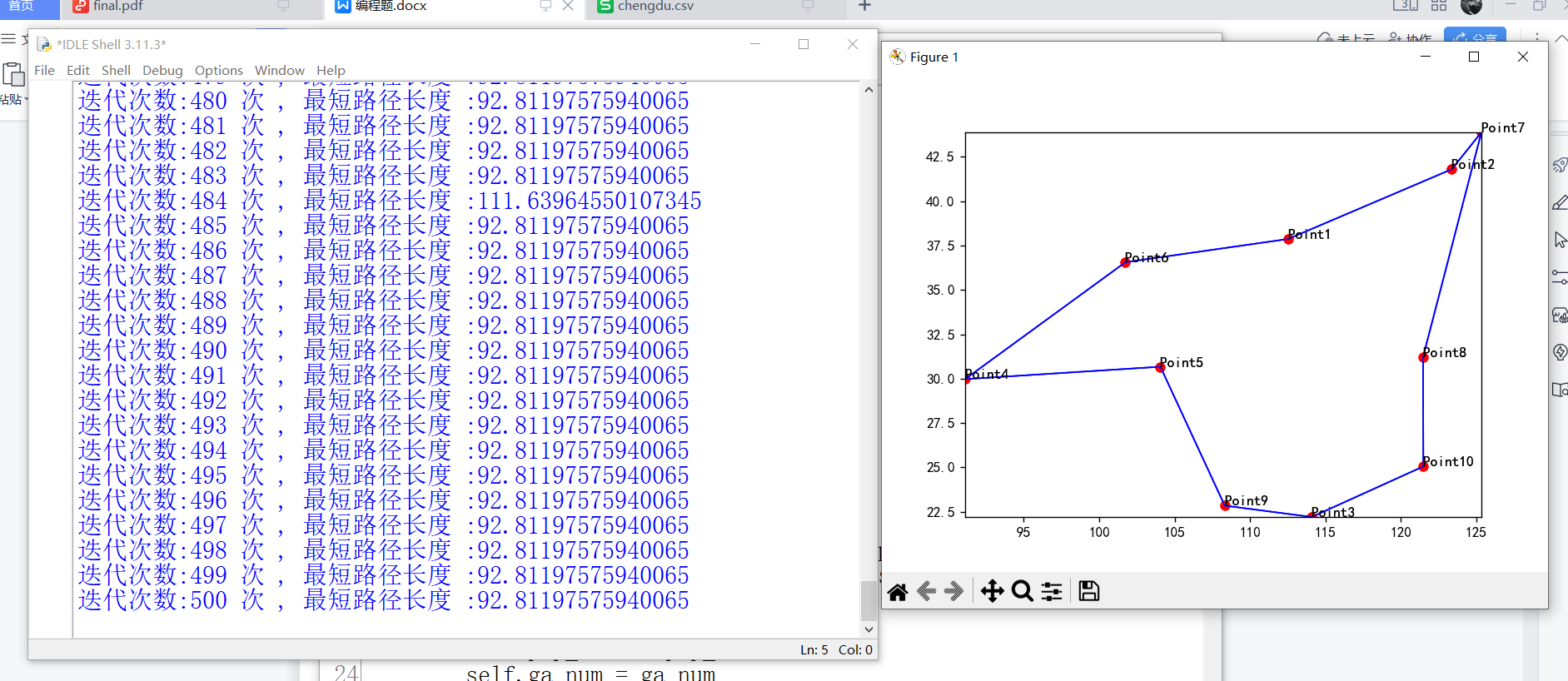
将file=’’中的内容填成我们想要的文件就可以了。

3.3 程序测试

注意程序的测试需要引入外部库numpy和pandas。

3.3.1 chengdu.csv

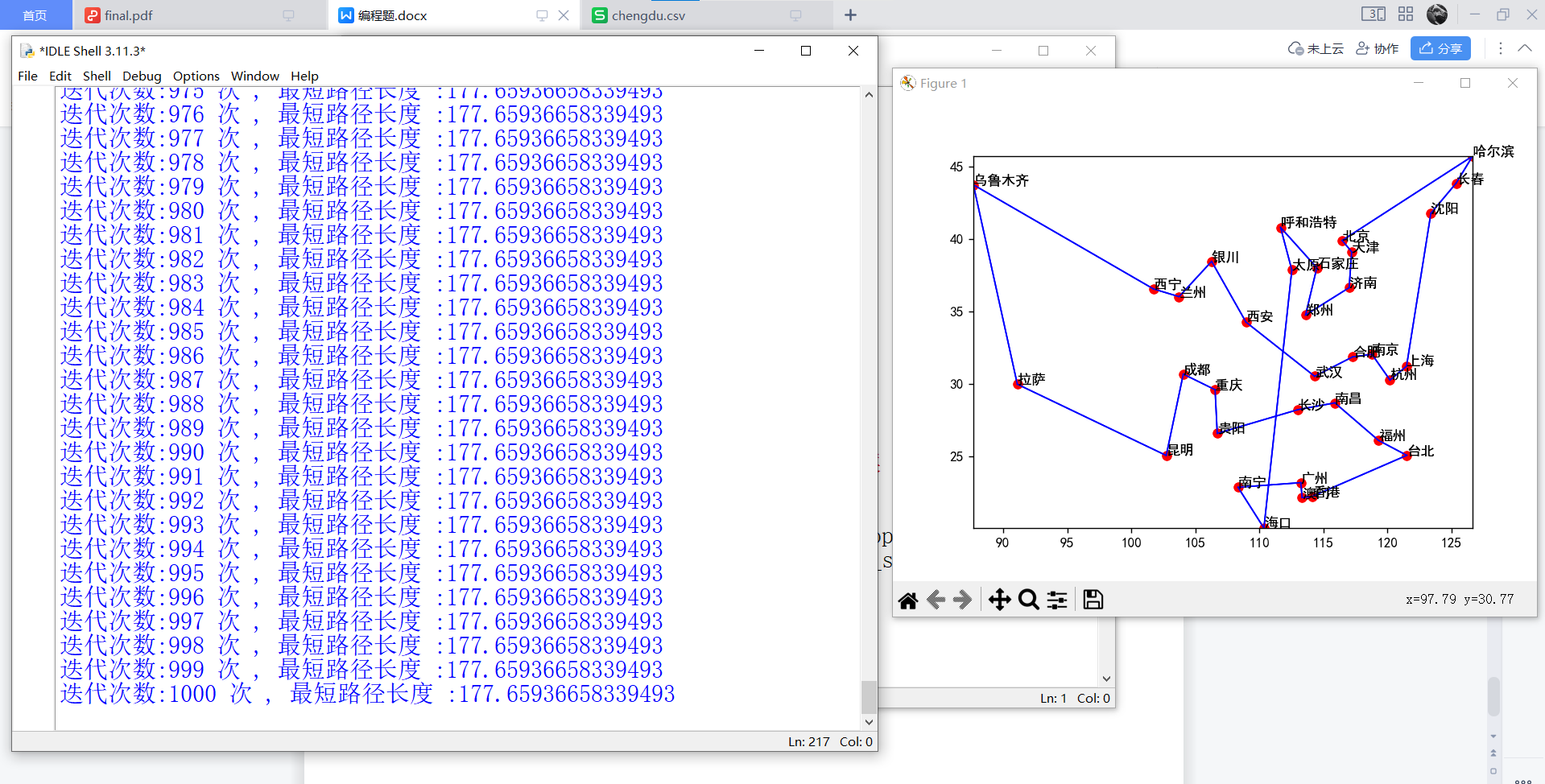
进行对chengdu.csv的测试。



可以看出，迭代了500次之后最短路径稳定下来，如右图所示。

3.3.2 china.csv

进行china.csv的测试。



可以看出，迭代了1000次之后最短路径稳定下来，如右图所示。

为了方便检查，还录制了两段测试视频，分别命名为chengdu.mp4和china.mp4。

注意，多次测试的结果可能不同，因为这种算法得出的只能是近似解，不能在规定的次数之内就能保证得到最优解，所以两次得到不一样的路径是很正常的，但是他们的路径值偏差应该不大。