

计算机学院

计算机科学与技术

2020级

2020416240

彭清元

人工智能实训

曲阜师范大学实验报告

**学院 计算机学院 年级、专业 20级计算机科学与技术二班 学号 2020416240姓名 彭清元 课程名称 人工智能 上课时间 第 1 节 — 第 11 节**

**实验日期 2023 年 5 月 27 日 星期 六 教师签名 雷玉霞 成绩**

实验三 遗传算法解决TSP问题

# 【实训目的】

1. 学习遗传算法的基本思想
2. 使用遗传算法解决实际问题，如TSP问题
3. 将理论知识转成实际代码进行展示，运行

# 【实验设备】

1. 计算机一台
2. Idea软件

# 【实验原理】

1. TSP问题

TSP问题（Travelling Salesman Problem）即旅行商问题，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。

1. 遗传算法

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。以其简单通用、鲁棒性强、适合并行处理及应用范围广，大体流程主要分为：初始化种群、计算适应度函数、选择、交叉、变异然后不断重复直到找到理想的解

# 【实验数据】

数据集：cities.txt。

数据集介绍：127个城市的坐标数据。第一列为城市序号，第二列为城市的x坐标，第三列为城市的y坐标。

# 【实验分析】

将我们能够成功访问的所有的城市作为一个种群，将一个路径看成一个个体，在遗传学中的染色体是每一个城市，在这个基础上我们使用遗传算法进行求解，第一步：把问题的解，表示成 “染色体”，在算法中就是以二进制编码的串，给出一群 “染色体”，也就是假设的可行解；第二步：把这些可行解置于问题的 “环境” 中，按适者生存的原则，选取较适应环境的“染色体”进行复制，并通过交叉、变异过程产生更适应环境的新一代 “染色体”群；第三步：经过这样的一代地进化，最后就会收敛到最适应环境的一个 “染色体”上，它就是问题的最优解。将这个应用到旅行商这个具体的问题上，从cities.txt获取127个城市之间的信息，利用遗传算法，对这些城市之间的信息进行“遗传”“变异”“交叉”，最后得到一个最优解

开始

适应度函数

生成最初路线

变异

适应度计算

遗传

满足停止标准

**否**

选择

**是**

输出最优解

# **【实验过程】**

1. 数据准备

从cities.txt中获取127个城市的相关信息，创建一个127\*127的矩阵进行存储每两个城市之间的距离，这个距离使用欧式距离进行表示，初始化这个迭代次数以及这个发生变异的概率

|  |
| --- |
| cities = load\_data('cities.txt') # 导入数据  n\_routes = 100 # 路线 epoch = 100000 # 迭代次数 prob = 0.01 # 变异的概率 |

对两个城市之间的距离进行计算，并存入到矩阵当中去，使用dist\_matrix[i,j] 来表示城市i和城市j之间的距离，

|  |
| --- |
| def get\_cities\_distance(cities):  dist\_matrix = np.zeros((127, 127)) # 根据数据的长度来界定---127\*127  n\_cities = len(cities)  for i in range(n\_cities - 1):  for j in range(i + 1, n\_cities):  dist = get\_two\_cities\_dist(cities[i], cities[j]) # 勾股定理求距离  dist\_matrix[i, j] = dist # 城市i到城市j的距离  dist\_matrix[j, i] = dist  return dist\_matrix |

1. 适应性函数

先随机生成100路径，使用np.random.choice从 n\_cities当中选取size个范围是n\_cities的数据返回的也就是一个一维数组，这个数组作为一个数据样本进行选择处理，每两个城市之间的距离使用欧式距离计算得到之后，将所有的距离进行相加，将这个倒数作为适应度函数。

|  |
| --- |
| def init\_route(n\_route, n\_cities):  routes = np.zeros((n\_route, n\_cities)).astype(int)  for i in range(n\_route):  routes[i] = np.random.choice(range(n\_cities), size=n\_cities, replace=False)  return routes  def get\_route\_fitness\_value(route, dist\_matrix):  dist\_sum = 0  for i in range(len(route) - 1):  dist\_sum += dist\_matrix[route[i], route[i + 1]] # $$$$$对于一条路线的计算，此时的route为routes[i]  dist\_sum += dist\_matrix[route[len(route) - 1], route[0]] # 由于循环的局限性此处累加单拿出来  return 1 / dist\_sum # 越短越好，自然选择用倒数 |

1. 选择操作

选择操作也称为复制（reproduction），是指从当前种群中选出个体以生成交配池（mating pool）的过程。这里使用的是轮盘赌法，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。

|  |
| --- |
| def selection(routes, fitness\_values):  selected\_routes = np.zeros(routes.shape).astype(int)  probability = fitness\_values / np.sum(fitness\_values)  n\_routes = routes.shape[0]  for i in range(n\_routes):  choice = np.random.choice(range(n\_routes), p=probability) # 概率越大，取值次数越多  selected\_routes[i] = routes[choice]  return selected\_routes |

1. 交叉操作

在这里选择的交叉的方式是使用分割的方法，因此存在许多不同的方案处理，比如存在两条路径A-B-C-D，C-D-A-B，我们可以将1，3号进行交叉，但是不能存在重复的元素，由于这些路径经过不断地选择和交叉之后最终会形成最优的方案。在每次进行分割的时候选择两条路径，将这两条路径进行分割后进行合并操作，具体实现是

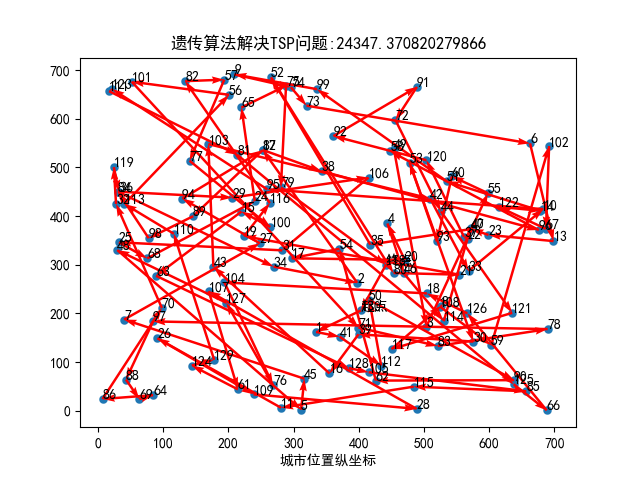
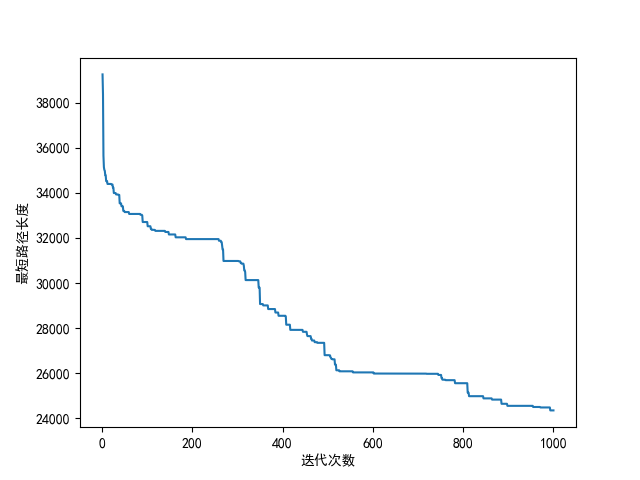
|  |
| --- |
| def crossover(routes, n\_cities):  for i in range(0, len(routes), 2):  r1\_new, r2\_new = np.zeros(n\_cities), np.zeros(n\_cities)   seg\_point = np.random.randint(0, n\_cities)  cross\_len = n\_cities - seg\_point   r1, r2 = routes[i], routes[i + 1]  r1\_cross, r2\_cross = r2[seg\_point:], r1[seg\_point:]   r1\_non\_cross = r1[np.in1d(r1, r1\_cross) == False]  r2\_non\_cross = r2[np.in1d(r2, r2\_cross) == False]   r1\_new[:cross\_len], r2\_new[:cross\_len] = r1\_cross, r2\_cross  r1\_new[cross\_len:], r2\_new[cross\_len:] = r1\_non\_cross, r2\_non\_cross   routes[i], routes[i + 1] = r1\_new, r2\_new  return routes |

1. 变异操作

在样本中选取的变异的概率是0.01，我们只需要在交叉之后，再随机选择几个位置进行改变值就可以了。当然变异的概率是很小的，并且是随机的，这一点要注意。并且由于变异是随机的，所以不排除生成比原来还更加糟糕的个体。具体的变异实现是

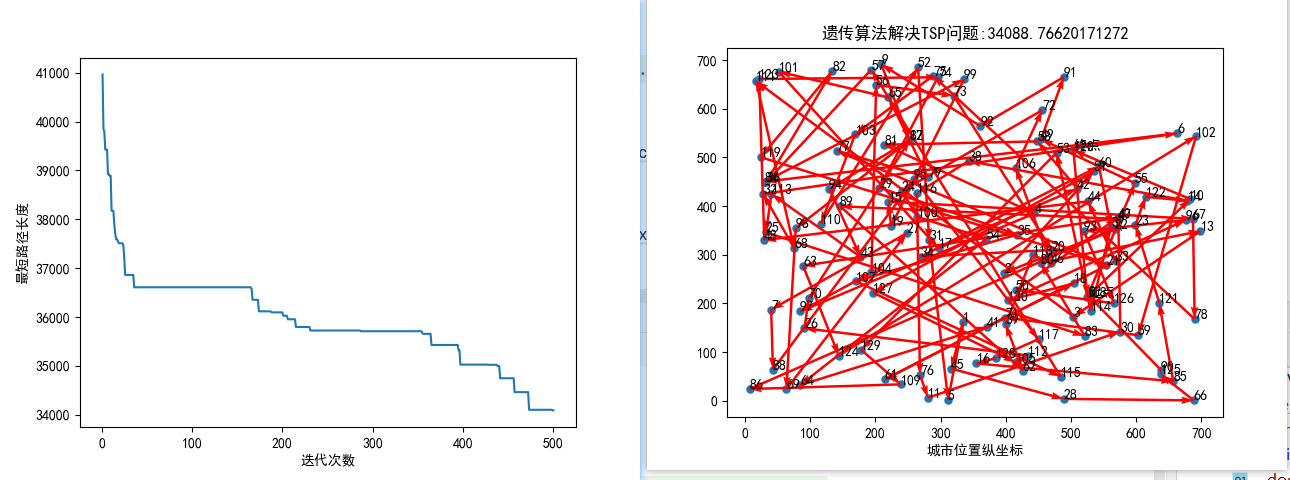
|  |
| --- |
| def mutation(routes, n\_cities):  prob = 0.01  p\_rand = np.random.rand(len(routes)) # 此处的长度的计算返回列表的行数  # 此处的随机数是生成的01之间的小数，长度为1000  for i in range(len(routes)):  if p\_rand[i] < prob:  mut\_position = np.random.choice(range(n\_cities), size=2, replace=False)  # 随机抽取不重复的两个数交换  l, r = mut\_position[0], mut\_position[1]  routes[i, l], routes[i, r] = routes[i, r], routes[i, l]  return routes |

# **【实验结果及分析】**



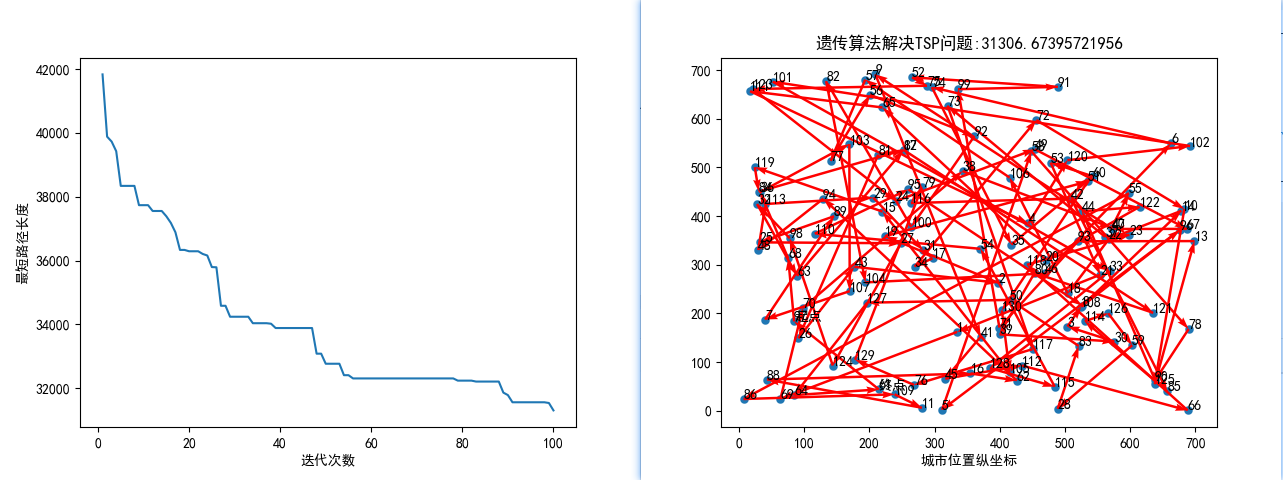
|  |
| --- |
| 最佳路径为：  130-->1-->41-->112-->50-->90-->62-->54-->17-->20-->4-->46-->114-->40-->22-->37-->120-->92-->91-->72-->44-->121-->117-->8-->108-->59-->102-->67-->122-->14-->51-->99-->9-->74-->73-->6-->13-->23-->47-->35-->52-->80-->21-->19-->116-->82-->57-->77-->100-->15-->63-->128-->105-->85-->115-->11-->26-->109-->28-->107-->61-->124-->129-->110-->113-->56-->101-->71-->83-->78-->97-->88-->69-->64-->86-->70-->32-->119-->36-->34-->2-->12-->38-->96-->58-->49-->3-->55-->60-->93-->53-->30-->39-->16-->43-->103-->42-->33-->10-->95-->123-->111-->81-->125-->66-->126-->18-->104-->76-->127-->48-->5-->45-->7-->27-->94-->87-->89-->98-->84-->29-->106-->31-->25-->68-->24-->65-->75-->79-->118-->130  当前最佳距离为: 24347.370820279866 |

在控制变量的情况下，将交叉概率由原来的0.95变成0.9，变异概率由0.5变成0.1，迭代次数由原来的1000次变成500次，进行计算，得出最小路径为



|  |
| --- |
| 最佳路径为：  108-->55-->49-->91-->92-->9-->87-->98-->69-->27-->7-->88-->95-->117-->128-->16-->85-->121-->23-->122-->14-->38-->116-->76-->11-->30-->47-->89-->62-->39-->118-->46-->106-->72-->24-->100-->54-->64-->41-->18-->130-->96-->93-->126-->50-->40-->63-->124-->102-->59-->71-->61-->33-->44-->48-->123-->74-->29-->35-->107-->115-->57-->36-->82-->119-->21-->77-->79-->65-->101-->17-->42-->58-->81-->4-->97-->104-->56-->73-->20-->34-->60-->37-->114-->53-->129-->109-->86-->22-->2-->51-->127-->105-->112-->26-->70-->12-->94-->43-->113-->25-->83-->80-->13-->3-->111-->68-->32-->6-->84-->99-->19-->15-->67-->78-->75-->103-->110-->52-->31-->5-->1-->45-->28-->66-->125-->90-->8-->10-->120-->108  当前最佳距离为: 34088.76620171272 |

在控制变量的情况下，将交叉概率保持原来的0.95，变异概率保持原来的0.5，迭代次数由原来的1000次变成100次，进行计算，得出最小路径为



|  |
| --- |
| 最佳路径为：  97-->68-->36-->103-->107-->7-->89-->25-->27-->124-->98-->32-->63-->87-->81-->26-->43-->2-->19-->51-->130-->105-->15-->119-->84-->12-->121-->80-->39-->30-->3-->108-->96-->5-->58-->78-->90-->44-->122-->20-->65-->111-->4-->120-->102-->101-->33-->128-->115-->54-->110-->95-->8-->14-->22-->18-->106-->57-->77-->56-->92-->100-->82-->104-->46-->6-->74-->35-->60-->53-->21-->1-->69-->70-->38-->40-->55-->86-->109-->11-->88-->16-->62-->94-->48-->113-->29-->31-->17-->34-->72-->10-->125-->13-->93-->71-->99-->91-->52-->75-->123-->24-->49-->67-->47-->9-->41-->45-->112-->66-->118-->23-->37-->73-->117-->76-->129-->79-->116-->42-->85-->114-->126-->59-->28-->83-->50-->127-->64-->61-->97  当前最佳距离为: 31306.67395721956 |

通过多组数据表示，交叉概率和变异概率变化对最终结果影响较大

# **【实验心得】**

本次实验学习了这遗传算法的基本原理，并且将这个遗传算法结合了TSP问题，使用代码将理论和实践进行结合，解决了关于TSP最短距离问题。

在使用遗传算法解决这个TSP问题，最重要的也就是遗传算法的选择、交叉、变异等操作。在选择过程中，计算出每一条路线与适应性相关的概率；在进行交叉的时候，顺序选出两条路线，数据切分执行后再合并操作；在变异过程中，使用随机函数，随机生成小数与设定突变概率进行比较，将交换位置的数值进行交换

通过这次实训进一步掌握了关于使用遗传算法来解决实际问题，将理论和实践进一步结合，提高了编码能力，加深了对遗传算法的理解。

# **【附录】**

**代码**

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
  
plt.rcParams['font.sans-serif'] = ["SimHei"]  
  
# 载入数据  
def load\_data():  
 df = pd.read\_csv('cities.txt', sep=" ", skiprows=6, header=None)  
 city = np.array(df[0][0:len(df) - 1]) # 最后一行为EOF，不读入  
 city\_name = city.tolist()  
 city\_x = np.array(df[1][0:len(df) - 1])  
 city\_y = np.array(df[2][0:len(df) - 1])  
 city\_location = list(zip(city\_x, city\_y))  
 return city\_name, city\_location  
  
# 计算两个城市的欧式距离  
def dist\_cal(x, y):  
 return ((x[0] - y[0]) \*\* 2 + (x[1] - y[1]) \*\* 2) \*\* 0.5  
  
# 求距离矩阵  
def matrix\_dis(city\_name, city\_location):  
 city\_num = len(city\_name)  
 res = np.zeros((city\_num, city\_num))  
 for i in range(city\_num):  
 for j in range(i + 1, city\_num):  
 res[i, j] = dist\_cal(city\_location[i], city\_location[j]) # 求两点欧式距离  
 res[j, i] = res[i, j] # 距离矩阵：对角线为0，对称  
 return res  
  
# 初始化种群  
def rand\_pop(city\_num, pop\_num, pop, distance, matrix\_distance):  
 rand\_ch = np.array(range(city\_num))  
 for i in range(pop\_num):  
 np.random.shuffle(rand\_ch)  
 pop[i, :] = rand\_ch  
 distance[i] = comp\_dis(city\_num, matrix\_distance, rand\_ch) # 这里的适应度其实是距离  
  
# 计算每个个体的总距离  
def comp\_dis(city\_num, matrix\_distance, one\_path):  
 res = 0  
 for i in range(city\_num - 1):  
 res += matrix\_distance[one\_path[i], one\_path[i + 1]]  
 res += matrix\_distance[one\_path[-1], one\_path[0]] # 最后一个城市和第一个城市的距离，需单独处理  
 return res  
  
# 打印出当前路径  
def print\_path(city\_num, one\_path):  
 res = str(one\_path[0] + 1) + '-->'  
 for i in range(1, city\_num):  
 res += str(one\_path[i] + 1) + '-->'  
 res += str(one\_path[0] + 1)  
 print("最佳路径为：")  
 print(res)  
  
# 轮盘赌的方式选择子代  
def select\_sub(pop\_num, pop, distance):  
 fit = 1. / distance # 适应度函数  
 p = fit / sum(fit)  
 q = p.cumsum() # 累积概率  
 select\_id = []  
 for i in range(pop\_num):  
 r = np.random.rand() # 产生一个[0,1)的随机数  
 for j in range(pop\_num):  
 if r < q[0]:  
 select\_id.append(0)  
 break  
 elif q[j] < r <= q[j + 1]:  
 select\_id.append(j + 1)  
 break  
 next\_gen = pop[select\_id, :]  
 return next\_gen  
  
# 交叉操作-每个个体对的某一位置进行交叉  
def cross\_sub(city\_num, pop\_num, next\_gen, cross\_prob, evbest\_path):  
 for i in range(0, pop\_num):  
 best\_gen = evbest\_path.copy()  
 if cross\_prob >= np.random.rand():  
 next\_gen[i, :], best\_gen = intercross(city\_num, next\_gen[i, :], best\_gen)  
  
# 具体的交叉方式：部分映射交叉  
def intercross(city\_num, ind\_a, ind\_b):  
 r1 = np.random.randint(city\_num)  
 r2 = np.random.randint(city\_num)  
 while r2 == r1:  
 r2 = np.random.randint(city\_num)  
 left, right = min(r1, r2), max(r1, r2)  
 ind\_a1 = ind\_a.copy()  
 ind\_b1 = ind\_b.copy()  
 for i in range(left, right + 1):  
 ind\_a2 = ind\_a.copy()  
 ind\_b2 = ind\_b.copy()  
 ind\_a[i] = ind\_b1[i]  
 ind\_b[i] = ind\_a1[i]  
 # 每个个体包含的城市序号是唯一的，因此交叉时若两个不相同，就会产生冲突  
 x = np.argwhere(ind\_a == ind\_a[i])  
 y = np.argwhere(ind\_b == ind\_b[i])  
 # 产生冲突，将不是交叉区间的数据换成换出去的原数值，保证城市序号唯一  
 if len(x) == 2:  
 ind\_a[x[x != i]] = ind\_a2[i]  
 if len(y) == 2:  
 ind\_b[y[y != i]] = ind\_b2[i]  
 return ind\_a, ind\_b  
  
# 变异方式：翻转变异  
def mutation\_sub(city\_num, pop\_num, next\_gen, mut\_prob):  
 for i in range(pop\_num):  
 if mut\_prob >= np.random.rand():  
 r1 = np.random.randint(city\_num)  
 r2 = np.random.randint(city\_num)  
 while r2 == r1:  
 r2 = np.random.randint(city\_num)  
 if r1 > r2:  
 temp = r1  
 r1 = r2  
 r2 = temp  
 next\_gen[i, r1:r2] = next\_gen[i, r1:r2][::-1]  
  
# 绘制路径图  
def draw\_path(city\_num, city\_location, pop, distance):  
 fig, ax = plt.subplots()  
 x, y = zip(\*city\_location)  
 ax.scatter(x, y, linewidths=0.1)  
 for i, txt in enumerate(range(1, len(city\_location) + 1)):  
 ax.annotate(txt, (x[i], y[i]))  
 res0 = pop  
 x0 = [x[i] for i in res0]  
 y0 = [y[i] for i in res0]  
 ax.annotate("起点", (x0[0], y0[0]))  
 ax.annotate("终点", (x0[-1], y0[-1]))  
 # 绘制箭图  
 for i in range(city\_num - 1):  
 plt.quiver(x0[i], y0[i], x0[i + 1] - x0[i], y0[i + 1] - y0[i], color='r', width=0.005, angles='xy', scale=1,  
 scale\_units='xy')  
 plt.quiver(x0[-1], y0[-1], x0[0] - x0[-1], y0[0] - y0[-1], color='r', width=0.005, angles='xy', scale=1,  
 scale\_units='xy')  
 plt.title("遗传算法解决TSP问题:" + str(distance))  
 plt.xlabel("城市位置横坐标")  
 plt.xlabel("城市位置纵坐标")  
 plt.savefig("map.png")  
 plt.show()  
  
# 绘制最优解随迭代次数的关系  
def draw\_iter(iteration, best\_distance\_list):  
 iteration = np.linspace(1, iteration, iteration)  
 plt.plot(iteration, best\_distance\_list)  
 plt.xlabel("迭代次数")  
 plt.ylabel("最短路径长度")  
 plt.savefig("figure.png")  
 plt.show()  
  
def main():  
 city\_name, city\_location = load\_data()  
 matrix\_distance = matrix\_dis(city\_name, city\_location)  
 city\_num = len(city\_name) # 城市数量  
 pop\_num = 300 # 群体个数  
 cross\_prob = 0.95# 交叉概率  
 mut\_prob = 0.5 # 变异概率  
 iteration = 1000 # 迭代次数  
  
 # 初始化初代种群和距离，个体为整数，距离为浮点数  
 pop = np.array([0] \* pop\_num \* city\_num).reshape(pop\_num, city\_num)  
 #print(pop)  
 distance = np.zeros(pop\_num)  
 #print(distance)  
 # 初始化种群  
 rand\_pop(city\_num, pop\_num, pop, distance, matrix\_distance)  
 #draw\_path(city\_num, city\_location, pop[0], distance) # 绘制初代图像  
  
 evbest\_path = pop[0]  
 evbest\_distance = float("inf")  
 best\_path\_list = []  
 best\_distance\_list = []  
 # 循环迭代遗传过程  
 for i in range(iteration):  
 # 选择  
 next\_gen = select\_sub(pop\_num, pop, distance)  
 # 交叉  
 cross\_sub(city\_num, pop\_num, next\_gen, cross\_prob, evbest\_path)  
 # 变异  
 mutation\_sub(city\_num, pop\_num, next\_gen, mut\_prob)  
  
 # 更新每个个体距离值(1/适应度)  
 for j in range(pop\_num):  
 distance[j] = comp\_dis(city\_num, matrix\_distance, next\_gen[j, :])  
 index = distance.argmin() # 记录最小总路程  
  
 # 为了防止曲线波动，每次记录最优值，如迭代后出现退化，则将当前最好的个体回退,取历史最佳结果  
 if distance[index] <= evbest\_distance:  
 evbest\_distance = distance[index]  
 evbest\_path = next\_gen[index, :]  
 else:  
 distance[index] = evbest\_distance  
 next\_gen[index, :] = evbest\_path  
 # 存储每一步的最优路径(个体)及距离  
 best\_path\_list.append(evbest\_path)  
 best\_distance\_list.append(evbest\_distance)  
  
 # 绘制迭代次数和最优解的关系曲线  
 draw\_iter(iteration, best\_distance\_list)  
  
 best\_path = evbest\_path  
 best\_distance = evbest\_distance  
 # 迭代完成，打印出最佳路径  
 print\_path(city\_num, best\_path)  
 print("当前最佳距离为:", best\_distance)  
 # 绘制路径图  
 draw\_path(city\_num, city\_location, best\_path, best\_distance)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

