《最优化技术》实验报告

一、实验目的

理解遗传算法的基本思想，并应用于求解实际问题。

二、实验项目内容

请利用遗传算法解决以下的旅行商问题：

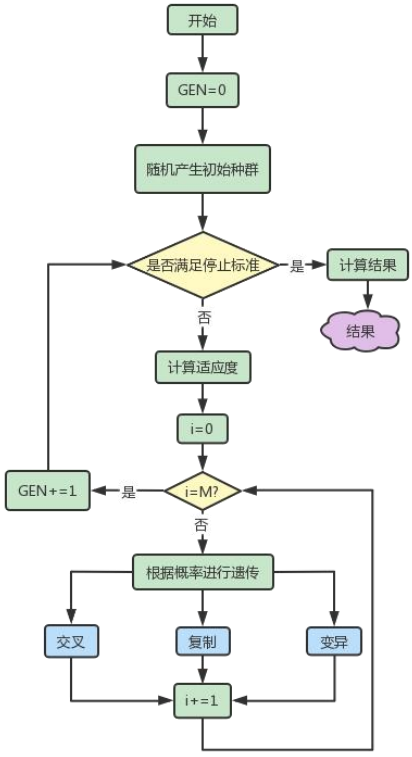
一个旅行者需要到国内的 10 个城市旅行，各城市的坐标见 cities.csv 文档。请设计一个合理的线路使旅行者所行的路程之和最小。注意：每个城市只能访问一次，且最后要回到原来出来的城市。

要求：输出适应度函数的进化曲线和最终选择的线路图。

注意：所有程序请用 python 语言实现。只提交本电子文档，注意本文件末尾的文件命名要求；源程序一节请用代码备注的方式说明你的算法和思路；实验结果一节需要提供测试结果截图并给出结果分析。

三、实验过程或算法（源程序）

1. 遗传算法流程图：



报告创建时间：2022.5.9

2. 实验代码：

```
'''
    用基因表示个体特征,即城市编号;染色体表示一组解,即城市路线编号;一定数量的染色体组成种群,并
通过适应度函数进行评估其存活率。
    算法随机生成初始种群,通过复制、交叉、逆转、变异并进行自然选择,直至符合优解条件满足
'''
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import math
from random import random,randint,shuffle
import csv
matplotlib.rcParams['font.family']='SimHei'     #设置中文字体

def read_data():         #载入数据
    name=[]             #城市名
    location=[]         #城市坐标
    f=open('OptimizationTechnology\实验 3\city.csv')
    reader=csv.reader(f)
    rows=[row for row in reader]
    for i in range(1,len(rows)):
        city=rows[i][0]
        latitude=rows[i][1]
        longitude=rows[i][2]
        name.append(city)
        location.append([float(latitude),float(longitude)])
    location=np.array(location)
  return name,location

class TSP:                          #定义旅行商类
    def __init__(self,org):
        self.origin=org             #起始点
        self.epochTime=20           #进化次数
        self.populationCount=1000   #种群数
        self.retainRate=0.3         #保持率
        self.mutationRate=0.1       #变异率
        self.reverseCount=10        #逆转次数
        self.cityNames,self.location=read_data()
        self.cityCount=len(self.cityNames)
        self.matrix=self.get_distance_matrix() #建立邻接矩阵
        self.cityIndex=[i for i in range(self.cityCount)]#城市索引
        self.cityIndex.remove(org)

    def get_distance_matrix(self):          #建立图的邻接矩阵
        num=self.cityCount
```

```
        loc=self.location
        matr=np.zeros([num,num])
        for i in range(num):
            for j in range(i,num):
                if i==j:
                    matr[i][j]=0
                else:
                    x2=(loc[i][0]-loc[j][0])**2
                    y2=(loc[i][1]-loc[j][1])**2
                    matr[j][i]=matr[i][j]=math.sqrt(x2+y2)
        return matr

    def get_total_distance(self,path):      #求路径 path 长度
        matr=self.matrix
        org=self.origin
        dis=0                               #记录路径长度
        dis+=matr[org][path[0]]
        for i in range(len(path)):
            if i==len(path)-1:
                dis+=matr[org][path[i]]
            else:
                dis+=matr[path[i]][path[i+1]]
        return dis

    def Reverse(self,path):                 #逆转操作,改善局部搜索能力
        rcnt=self.reverseCount
        dis=self.get_total_distance(path)
        for i in range(rcnt):               #逆转次数
            u=randint(0,len(path)-1)
            v=randint(0,len(path)-1)
            if u!=v:
                new_path=path.copy()        #拷贝 path 对其副本进行操作
                t=new_path[u]               #swap(path,u,v)
                new_path[u]=new_path[v]
                new_path[v]=t
                new_distance=self.get_total_distance(new_path)
                if new_distance<dis:
                    dis=new_distance
                    path=new_path.copy()
            else:
                continue
        return path

    def selection(self,population):         #对新种群的自然选择
        retain_rate=self.retainRate         #保持率
        #对总距离从小到大进行排序
        graded=[[self.get_total_distance(path),path] for path in
population]#[距离,路径]
        graded=[path[1] for path in sorted(graded)] #路径
```

```
        retain_length=int(len(graded)*retain_rate)
        new_pop=graded[:retain_length]      #选出适应性强的
len(graded)*retain_rate 条染色体
        random_select_rate=0.2              #弱者存活率
        for chromosome in graded[retain_length:]:
            if random() < random_select_rate:
                new_pop.append(chromosome)
        return new_pop

    def crossover(self,parents):            #交叉
        # 生成子代的个数,保证种群数目稳定
        target_count=self.populationCount-len(parents)
        # 孩子列表
        cs=[]
        while len(cs)<target_count:
            male_index=randint(0,len(parents)-1)
            female_index=randint(0,len(parents)-1)
            if male_index!=female_index:    #保证不和自己交配
                male=parents[male_index]
                female=parents[female_index]
                left=randint(0,len(male)-2) #随机生成两点交叉点位
                right=randint(left+1,len(male)-1)

                gene1=male[left:right]      #待交叉片段
                gene2=female[left:right]
                miss1=[]
                miss2=[]
                for item in gene1:
                    if item not in gene2:
                        miss1.append(item)          #交换后 male 缺失的
部分(female 多余的部分)
                for item in gene2:
                    if item not in gene1:
                        miss2.append(item)          #交换后 female 缺失
的部分(male 多余的部分)
                child1_tmp=male[:left]+male[right:] #不包含交换部分的基
因
                child2_tmp=female[:left]+female[right:]
                for i in range(len(miss1)):
                    for j in range(len(child1_tmp)):
                        if child1_tmp[j]==miss2[i]:
                            child1_tmp[j]=miss1[i]
                for i in range(len(miss2)):
                    for j in range(len(child2_tmp)):
                        if child2_tmp[j]==miss1[i]:
                            child2_tmp[j]=miss2[i]
                child1=child1_tmp[:left]
                child1_r=child1_tmp[left:]
```

```
                child1.extend(gene2)
                child1.extend(child1_r)
                child2=child2_tmp[:left]
                child2_r=child2_tmp[left:]
                child2.extend(gene1)
                child2.extend(child2_r)
                cs.append(child1)
                cs.append(child2)
        return cs

    def mutation(self,cs):                  #变异
        mutation_rate=self.mutationRate
        new_cs=[]                           #生成变异后结果
        for i in range(len(cs)):
            child=cs[i]
            if random()<mutation_rate:
                u=randint(1,len(child)-4)
                v=randint(u+1,len(child)-3)
                w=randint(v+1,len(child)-2)
                child=child[0:u]+child[v:w]+child[u:v]+child[w:]
            new_cs.append(child)
        return new_cs

    def get_best_result(self,population):   #得到最佳结果
        graded=[[self.get_total_distance(path), path] for path in
population]
        graded=sorted(graded)
        return graded[0][0],graded[0][1]

    def evolution(self):                    #进化
        org=self.origin
        population=[]
        for i in range(self.populationCount):
            x=self.cityIndex.copy()         #随机生成个体
            shuffle(x)
            x=self.Reverse(x)               #优化个体
            population.append(x)
        #遗传优化
        ans=[]
        i=0
        distance,result_path=self.get_best_result(population)
        while i<self.epochTime:
            #选择繁殖个体群
            parents=self.selection(population)
            #交叉繁殖
            cs=self.crossover(parents)
            #变异操作
            cs=self.mutation(cs)
            #更新种群
```

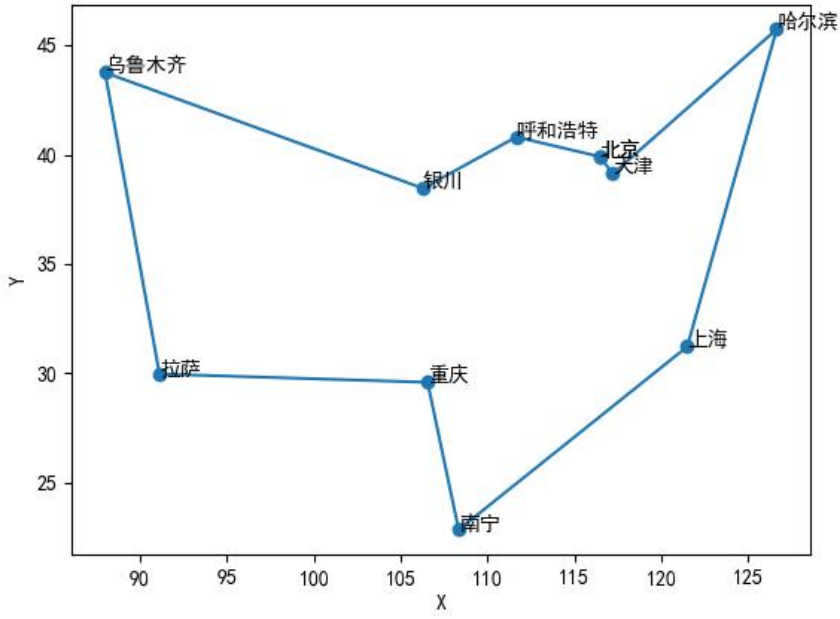
```
            population=parents+cs
            distance,result_path=self.get_best_result(population)
            ans.append(distance)
            i+=1
        result_path=[org]+result_path+[org]
        #输出
        print(distance)
        print(result_path)
        for i in range(len(result_path)):
            if i!=0:
                print("—>",end='')
            print(self.cityNames[result_path[i]],end='')
        #绘图
        xs=list()
        ys=list()
        nam=self.cityNames
        loc=self.location
        for j in result_path:
            # 画注解
            plt.annotate(nam[j],[loc[j, 0],loc[j, 1]])
            xs.append(loc[j,0])
            ys.append(loc[j,1])
        #绘图
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.plot(xs,ys,'-o')
        plt.show()
        plt.xlabel('训练次数')
        plt.ylabel('距离')
        plt.title('Evolution of Fitness function')
        plt.plot(list(range(len(ans))),ans)
        plt.show()

if __name__ == '__main__':
    t1=TSP(8)
    t1.evolution()
```

四、实验结果及分析和（或）源程序调试过程

1. 实验结果：

对于**不同初始点**（即出发的城市），根据遗传算法所得**路径均相同，最短路径长度大小为** 109.988246258313**，路径为南宁—>上海—>哈尔滨—>天津—>北京—>呼和浩特—>银川—>乌鲁木齐—>拉萨—>重庆—>南宁（初始点随意，围绕上述路径形成闭环即可）**。但**适应度函数的进化各不相同**，下给出 5 组适应度函数进化过程的示例。

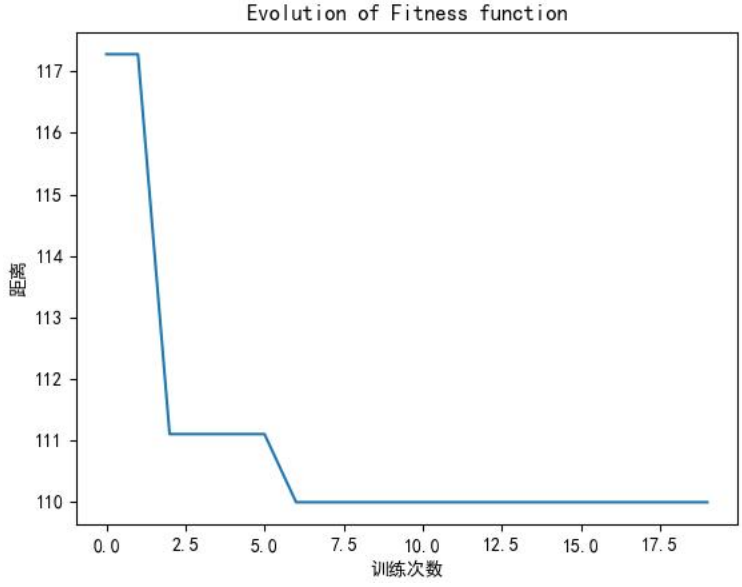
路径如下图：



不同初始点的适应度函数的进化过程如下所示：

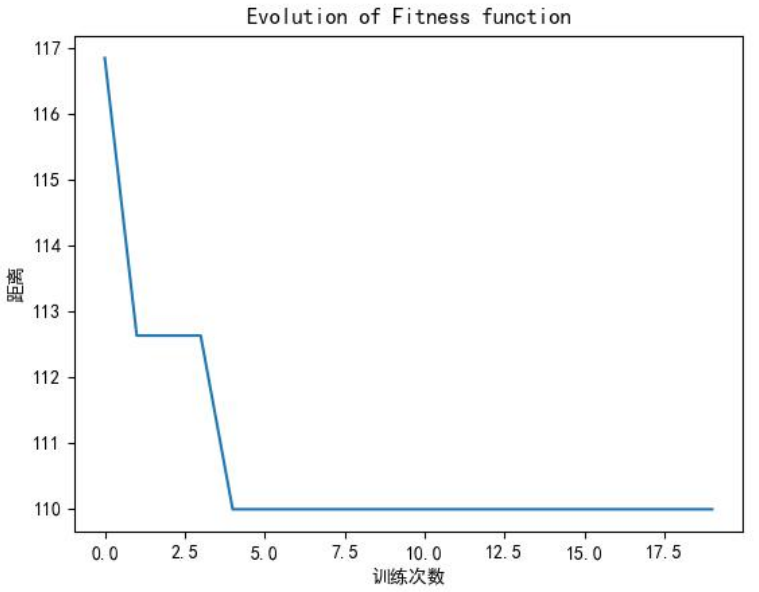
（1）从**北京**开始

```
109.988246258313
[0, 1, 9, 2, 8, 3, 4, 5, 6, 7, 0]
北京-->天津-->哈尔滨-->上海-->南宁-->重庆-->拉萨-->乌鲁木齐-->银川-->呼和浩特-->北京
```



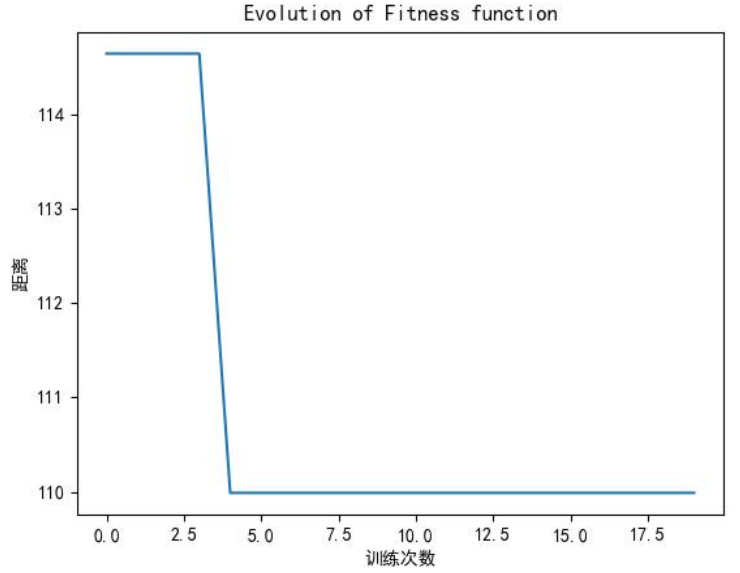
（2）从**天津**开始

```
109.98824625831301
[1, 0, 7, 6, 5, 4, 3, 8, 2, 9, 1]
天津-->北京-->呼和浩特-->银川-->乌鲁木齐-->拉萨-->重庆-->南宁-->上海-->哈尔滨-->天津
```



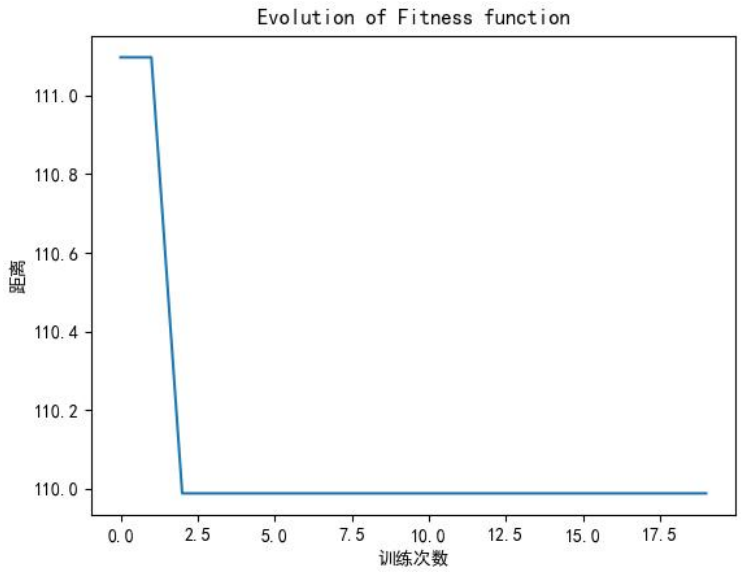
（3）从**上海**开始

```
109.98824625831301
[2, 8, 3, 4, 5, 6, 7, 0, 1, 9, 2]
上海-->南宁-->重庆-->拉萨-->乌鲁木齐-->银川-->呼和浩特-->北京-->天津-->哈尔滨-->上海
```



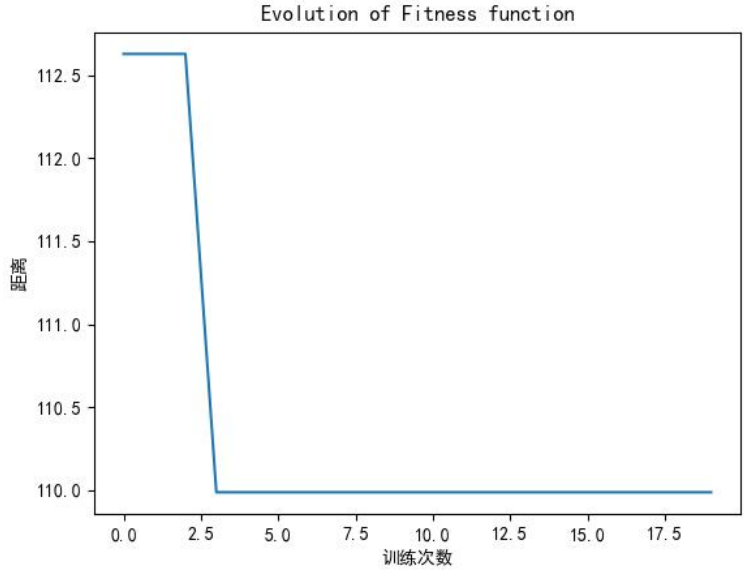
（4）从**重庆**开始

```
109.988246258313
[3, 4, 5, 6, 7, 0, 1, 9, 2, 8, 3]
重庆-->拉萨-->乌鲁木齐-->银川-->呼和浩特-->北京-->天津-->哈尔滨-->上海-->南宁-->重庆
```



（5）从**银川**开始

```
109.98824625831301
[6, 5, 4, 3, 8, 2, 9, 1, 0, 7, 6]
银川-->乌鲁木齐-->拉萨-->重庆-->南宁-->上海-->哈尔滨-->天津-->北京-->呼和浩特-->银川
```



2. 调试过程：

（1）**遗传**时分为**复制**、**交叉**、**变异**和**逆转**，其中复制可以直接将父代压入种群，逆转和变异随机选取一个染色体的点位进行交换即可，但**交叉**难以实现，因为**两组染色体交叉后容易出现重复的 DNA**（城市编号）。采取**记录交换中重复基因**的方式，在交叉完成后将未交叉部分中重复元素取出，换成缺失部分。具体操作如下，主要利用 list 切片实现：

```
gene1=male[left:right]      #待交叉片段
gene2=female[left:right]
miss1=[]
miss2=[]
for item in gene1:
    if item not in gene2:
        miss1.append(item)          #交换后male缺失的部分(female多余的部分)
for item in gene2:
    if item not in gene1:
        miss2.append(item)          #交换后female缺失的部分(male多余的部分)
child1_tmp=male[:left]+male[right:] #不包含交换部分的基因
child2_tmp=female[:left]+female[right:]
for i in range(len(miss1)):
    for j in range(len(child1_tmp)):
        if child1_tmp[j]==miss2[i]:
            child1_tmp[j]=miss1[i]
for i in range(len(miss2)):
    for j in range(len(child2_tmp)):
        if child2_tmp[j]==miss1[i]:
            child2_tmp[j]=miss2[i]
child1=child1_tmp[:left]
child1_r=child1_tmp[left:]
child1.extend(gene2)
child1.extend(child1_r)
child2=child2_tmp[:left]
child2_r=child2_tmp[left:]
child2.extend(gene1)
child2.extend(child2_r)
```

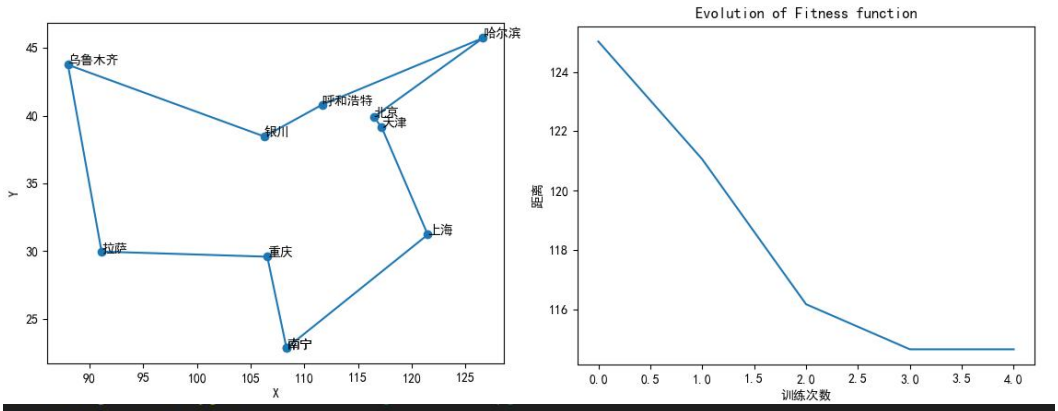
（2）python 程序读取 csv 文件时注意表头标题，为**无效数据**，不应读入数据列表，将遍历改为从 1 开始：

```
for i in range(1,len(rows)):
    city=rows[i][0]
    latitude=rows[i][1]
    longitude=rows[i][2]
    name.append(city)
    location.append([float(latitude),float(longitude)])
location=np.array(location)
```

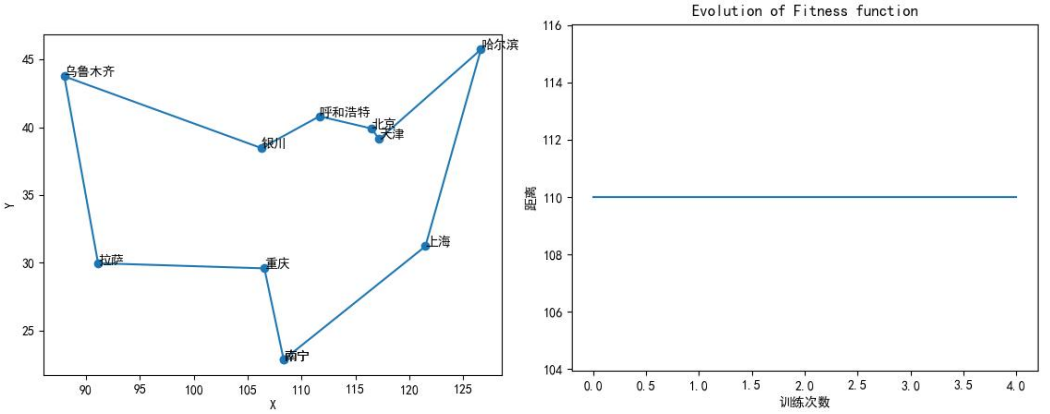
（3）为了保证**种群数量稳定**，需在自然选择时设定种群数量，否则有可能陷入死循环

```
# 生成子代的个数,保证种群数目稳定
target_count=self.populationCount-len(parents)
```

（4）由于初始种群是**随机**生成的，遗传过程也夹杂着大量的随机数，因此在进化次数较少的情况下会出现**不稳定**情形，比如无法生成最优解的情况：




```
114.64401518698126
[8, 2, 1, 0, 9, 7, 6, 5, 4, 3, 8]
南宁-->上海-->天津-->北京-->哈尔滨-->呼和浩特-->银川-->乌鲁木齐-->拉萨-->重庆-->南宁
```




```
109.98824625831301
[8, 2, 9, 1, 0, 7, 6, 5, 4, 3, 8]
南宁-->上海-->哈尔滨-->天津-->北京-->呼和浩特-->银川-->乌鲁木齐-->拉萨-->重庆-->南宁
```

上面两组图为同一程序进化次数为 5 时某两次运行生成的解，前一组显然未达到最优，后一组为最优，但由于巧合，使得初始种群一次遗传便达到最优解。易见算法中**随机数带来的不稳定性**。

（5）遗传算法的**参数根据经验所得**，进化次数、种群数量、逆转次数、保持率、变异率、弱者存活率均为人为预知参数，若参数设定不当，很容易难以得到最优解。本程序中设定的参数为：

参数	参数值
进化次数	20
种群数量	1000
逆转次数	10
保持率	0.3
变异率	0.1
弱者存活率	0.2

接下来固定其他变量分别探究各个**参数变量对结果的影响**：

5.1）进化次数

进化次数	1	2	3	4	5	6	7	8	9	10

最优解	119.98730 112222073	116.84488 230462189	114.64401 518698126	113.73576 754363006	112.62679 39532163	112.62679 395321632	111.09721 984872675	111.09721 984872675	109.98824 625831301	109.98824 625831301

显然**进化次数越多越容易得到最优解**，由实验得**进化次数达到 10 次**以后，在其他参数不变的情况下将得到**稳定的最优解**。

5.2）种群数量

种群数量	100	200	300	400	500	600	700	800	900	1000
最优解	124.06415 081964239	122.33151 069363818	121.75403 962405629	120.01572 036233794	116.24838 720358666	115.49389 867878264	114.64401 518698125	112.62679 39532163	111.09721 984872675	109.98824 625831301

显然**种群数量越大**，可以选择最优解的范围越大，**越可能得到最优解**。当种群数量达到 1000 以后，最优解趋于稳定。

5.3）逆转次数

逆转次数	1	2	3	4	5	6	7	8	9	10
最优解	115.49389 867878264	113.73576 754363006	113.73576 754363006	111.09721 984872675	109.98824 625831301	109.98824 625831301	109.98824 625831301	109.98824 625831301	109.98824 625831301	109.98824 625831301

由尝试可知，**逆转次数越多**，得到的解越符合适应度函数，**解越优**。当逆转次数达到 5 以后，最优解稳定于 109.988。

5.4）保持率

保持率	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
最优解	122.33151 069363818	116.24838 720358666	109.98824 625831301	112.62679 39532163	114.64401 518698125	116.24838 720358666	119.98730 112222073	121.75403 962405629	125.30671 028460529	123.14757 090065358

保持率为一个**居中最优参数**，若**保持率太低**，会使得最优解**难以延续**；**保持率过高**，会使得**初始种群难以进化**。当保持率在 0.3 附近处，最优解接近 109.988。

5.5）变异率

变异率	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
最优解	109.98824 625831301	109.98824 625831301	111.09721 984872675	111.09721 984872675	114.64401 518698125	114.64401 518698125	116.24838 720358666	116.24838 720358666	117.28256 288188454	120.01572 036233794

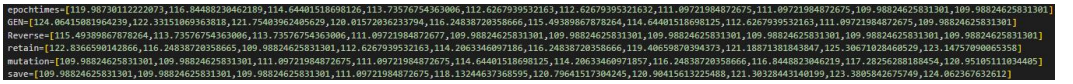
保持率也是一个**居中最优参数**，若**变异率太低**，会使得最优解**难以进化**；**变异率过高**，会使得**初始种群难以保持**。由常识可知，变异发生概率较低，因此设为 0.1。

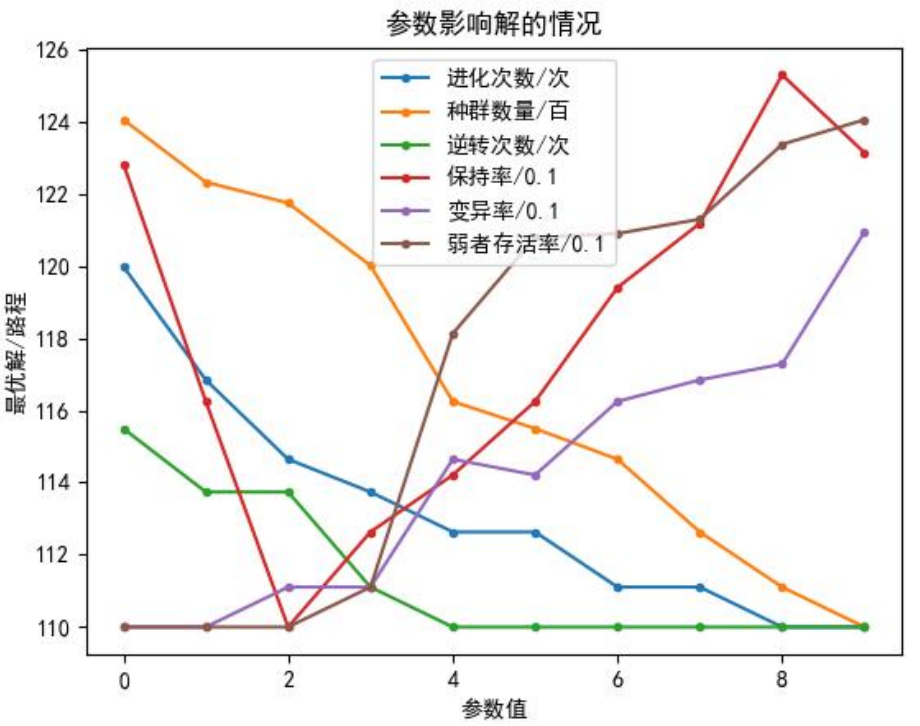
5.6）弱者存活率

弱者存活率	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
最优解	109.98824 625831301	109.98824 625831301	109.98824 625831301	111.09721 984872675	118.13244 637368595	120.01572 036233794	120.01572 036233794	121.75403 962405629	123.38058 42675749	124.06415 081964239

由常识可知，自然会淘汰弱者，但又不会完全淘汰，总会留下少量侥幸的弱者存活，因此弱者存活率设为 0.2。显然**弱者存活率越低，解越优**。

综合上述 6 个参数分析，得到表格和图像如下：





与上述分析一致：**进化次数越多越容易得到最优解；种群数量越大，越可能得到最优解；逆转次数越多，越容易得到最优解；最优解随保持率增加先增加后减少；最优解随变异率增加先增加后减少；弱者存活率越低解越优。**

3. 总结：

遗传算法首先实现从性状到基因的映射，即**编码**工作，然后从代表问题可能潜在解集的一个种群开始进行进化**求解**。接着初代种群产生后，按照**优胜劣汰**的原则，根据个体适应度大小挑选；然后进行**复制、交叉、变异、逆转**，产生出代表**新的群体**，再对其进行挑选以及一系列遗传操作，如此往复，逐代演化产生出越来越好的近似解。

遗传算法与经典算法存在明显区别：遗传算法**以决策变量的编码作为运算对象**，直接**以目标函数值作为搜索信息**，引入和应用遗传操作。传统的**优化算法**往往**直接利用决策变量的实际值本身进行优化计算**，并且往往不只需要目标函数值，还需要目标函数的导数等其它信息。这样对许多目标函数无法求导或很难求导的函数，遗传算法就比较方便。

遗传算法同时进行解空间的多点搜索。**传统的优化算法**往往从解空间的一个初始点开始搜索，这样容易陷入**局部极值点**。**遗传算法**进行**群体搜索，，避免陷入局部最优**。而且在搜索的过程中引入遗传运算，使群体又可以不断进化。这也是遗传算法所特有的一种**隐含并行性**。

遗传算法使用**概率搜索**技术，属于一种**自适应概率搜索技术**，其选择、交叉、变异等运算都是以一种概率的方式来进行的，从而增加了其搜索过程的灵活性。

不过遗传算法依赖可靠经验值参数，否则将无法得到最优解，因此对不同模型应用遗传算法需要提前做好预测，从而给定合理的参数值。