作业 5：GA 实验

The deceptive functions are a family of functions in which there exists low-order building blocks that do not combine to form the higher-order building blocks. Here, a deceptive problem that consists of 25 copies of the order-4 fully deceptive function DF2 is constructed for this paper. DF2 can be described as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| f(0000)=28 | f(0001)=26 | f(0010)=24 | f(0011)=18 |
| f(0100)=22 | f(0101)=6 | f(0110)=14 | f(0111)=0 |
| f(1000)=20 | f(1001)=12 | f(1010)=10 | f(1011)=2 |
| f(1100)=8 | f(1101)=4 | f(1110)=6 | f(1111)=30 |

This problem has a maximal function value of 750.

代码如下：

**import** numpy **as** np  
**import** math  
**from** matplotlib **import** pyplot **as** plt  
  
DNA\_size = 100 # DNA长度  
population\_size = 400 # 种群大小  
cross\_rate = 0.85 # 交叉率  
mutation\_rate = 0.0005 # 变异率  
iter\_N = 180 # 迭代270次  
population\_value = [] # 适应值数组，用来装适应值  
child\_new = []  
  
population = np.random.randint(2, size=(population\_size, DNA\_size)) # 初始化种群DNA,小于2表示数值为0或1。大小为400x100的矩阵  
np.set\_printoptions(threshold=np.inf)  
  
  
**def** translation(population): # 翻译DNA  
 **for** i **in** range(population\_size):  
 sum = 0 # 归0  
 **for** j **in** range(25):  
 x\_bits = list(population[i][4 \* j:4 \* (j + 1)])  
 **if** (x\_bits == [0, 0, 0, 0]):  
 sum = sum + 28  
 **elif** (x\_bits == [0, 0, 0, 1]):  
 sum = sum + 26  
 **elif** (x\_bits == [0, 0, 1, 0]):  
 sum = sum + 24  
 **elif** (x\_bits == [0, 0, 1, 1]):  
 sum = sum + 18  
 **elif** (x\_bits == [0, 1, 0, 0]):  
 sum = sum + 22  
 **elif** (x\_bits == [0, 1, 0, 1]):  
 sum = sum + 6  
 **elif** (x\_bits == [0, 1, 1, 0]):  
 sum = sum + 14  
 **elif** (x\_bits == [0, 1, 1, 1]):  
 sum = sum + 0  
 **elif** (x\_bits == [1, 0, 0, 0]):  
 sum = sum + 20  
 **elif** (x\_bits == [1, 0, 0, 1]):  
 sum = sum + 12  
 **elif** (x\_bits == [1, 0, 1, 0]):  
 sum = sum + 10  
 **elif** (x\_bits == [1, 0, 1, 1]):  
 sum = sum + 2  
 **elif** (x\_bits == [1, 1, 0, 0]):  
 sum = sum + 8  
 **elif** (x\_bits == [1, 1, 0, 1]):  
 sum = sum + 4  
 **elif** (x\_bits == [1, 1, 1, 0]):  
 sum = sum + 6  
 **elif** (x\_bits == [1, 1, 1, 1]):  
 sum = sum + 30  
  
 population\_value.append(sum)  
  
 **return** population\_value  
  
  
**def** select(population, fitness): # 自然选择，选择适应值比较大的进行交叉  
  
 index = np.random.choice(np.arange(population\_size), size=population\_size, replace=**True**,  
 p=fitness / fitness.sum()) # 轮盘赌的方式选择  
 **return** population[index]  
  
  
**def** crossover(parent, person): # 交叉  
 **if** np.random.rand() < cross\_rate:  
 i = np.random.randint(0, population\_size, size=1) # 随机选择另一个个体进行交叉  
 cross\_points1 = int(np.random.randint(0, DNA\_size, size=1)) # 随机选择交叉点、双点交叉  
 cross\_points2 = int(np.random.randint(0, DNA\_size, size=1)) # 随机选择交叉点、双点交叉  
 **if** (cross\_points1 < cross\_points2):  
 parent[cross\_points2 - cross\_points1:DNA\_size - cross\_points1] = person[i, cross\_points2:] # 交叉  
 **else**:  
 parent[cross\_points1 - cross\_points2:DNA\_size - cross\_points2] = person[i, cross\_points1:] # 交叉  
  
 **return** parent  
  
  
**def** mutate(child): # 变异  
 **for** point **in** range(DNA\_size):  
 **if** np.random.rand() < mutation\_rate:  
 child[point] = 1 **if** child[point] == 0 **else** 0  
 **return** child  
  
  
# 计算交叉变异后子代的值  
**def** child\_value(population): # 翻译DNA  
 sum = 0 # 归0  
 **for** j **in** range(25):  
 y\_bits = list(population[4 \* j:4 \* (j + 1)])  
 **if** (y\_bits == [0, 0, 0, 0]):  
 sum = sum + 28  
 **elif** (y\_bits == [0, 0, 0, 1]):  
 sum = sum + 26  
 **elif** (y\_bits == [0, 0, 1, 0]):  
 sum = sum + 24  
 **elif** (y\_bits == [0, 0, 1, 1]):  
 sum = sum + 18  
 **elif** (y\_bits == [0, 1, 0, 0]):  
 sum = sum + 22  
 **elif** (y\_bits == [0, 1, 0, 1]):  
 sum = sum + 6  
 **elif** (y\_bits == [0, 1, 1, 0]):  
 sum = sum + 14  
 **elif** (y\_bits == [0, 1, 1, 1]):  
 sum = sum + 0  
 **elif** (y\_bits == [1, 0, 0, 0]):  
 sum = sum + 20  
 **elif** (y\_bits == [1, 0, 0, 1]):  
 sum = sum + 12  
 **elif** (y\_bits == [1, 0, 1, 0]):  
 sum = sum + 10  
 **elif** (y\_bits == [1, 0, 1, 1]):  
 sum = sum + 2  
 **elif** (y\_bits == [1, 1, 0, 0]):  
 sum = sum + 8  
 **elif** (y\_bits == [1, 1, 0, 1]):  
 sum = sum + 4  
 **elif** (y\_bits == [1, 1, 1, 0]):  
 sum = sum + 6  
 **elif** (y\_bits == [1, 1, 1, 1]):  
 sum = sum + 30  
  
 **return** sum  
  
  
**def** scatt(j):  
 # 散点图，数据可视化  
 plt.figure() # 画布  
 # s表示点点的大小，c是color嘛，marker就是点点的形状o,x,\*><^,都可以啦  
 # alpha是点点的亮度，label是标签啦  
 plt.scatter(np.arange(population\_size), values\_new, s=10, c=**'green'**, marker=**'o'**, alpha=0.7)  
 plt.title(**"fitness-sactter "** + str(j + 1))  
 plt.xlabel(**"person"**)  
 plt.ylabel(**"fitness"**)  
 # plt.legend(loc='upper right') # 右上角标签  
  
 plt.show() # 显示图像  
  
  
# 主函数  
**for** j **in** range(iter\_N):  
 F\_values = translation(population) # 计算适应值对应的值  
  
 population = select(population, np.array(F\_values)) # 选择  
 pop\_copy = population.copy()  
  
 **for** parent **in** population: # 对种群中的每个个体都进行交叉变异  
 child = crossover(parent, population) # 交叉  
 child = mutate(child) # 变异  
  
 ch = child\_value(child)  
 **if** (ch > np.median(F\_values)): # 子代适应值超过中位数则接受  
 pop\_copy[np.argmin(F\_values), :] = child # 将原种群中的适应值最小的一个替换掉  
 F\_values[np.argmin(F\_values)] = ch # 更新适应值  
  
 population = pop\_copy.copy() # 将跟新后的种群重新给population  
  
 population\_value.clear() # 清空适应值，非常重要，浅复制  
 values\_new = translation(pop\_copy) # 计算新的值  
 x = np.argmax(values\_new) # 找出最大值所在的位置argmax返回数值最大数的下标  
 print(**"Most fitted DNA: "**, population[x]) # 输出最大值对应的基因  
 print(**"适应值: "**, values\_new[x]) # 输出最大适应值  
  
 scatt(j) # 画散点图，数据可视化  
 population\_value.clear() # 清空适应值，进行下一次迭代

参数设定如下：

DNA\_size = 100 # DNA长度  
population\_size = 400 # 种群大小  
cross\_rate = 0.85 # 交叉率  
mutation\_rate = 0.0005 # 变异率  
iter\_N = 180 # 迭代180次

运行结果如下：

1. 最后4次迭代的基因和适应值：

基因：

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

适应值: 750

基因：

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

适应值: 750

基因：

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

适应值: 750

基因：

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

适应值: 750

1. 散点图如下：

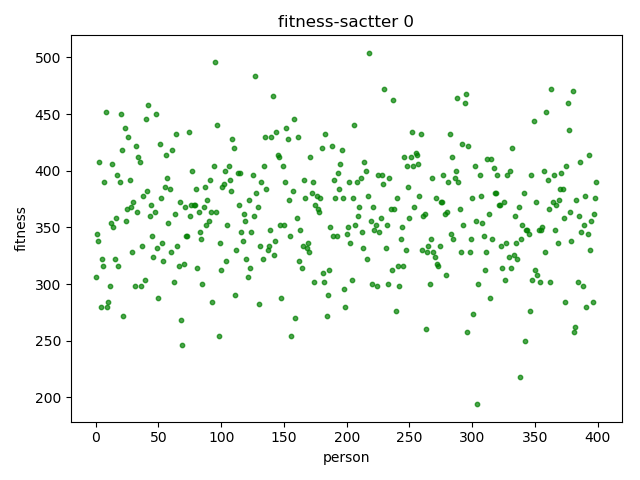


图1 原始种群适应值散点图

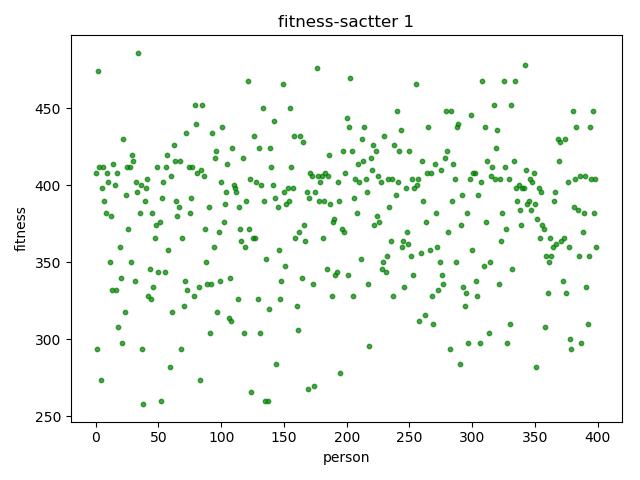


图2 第1次迭代适应值散点图

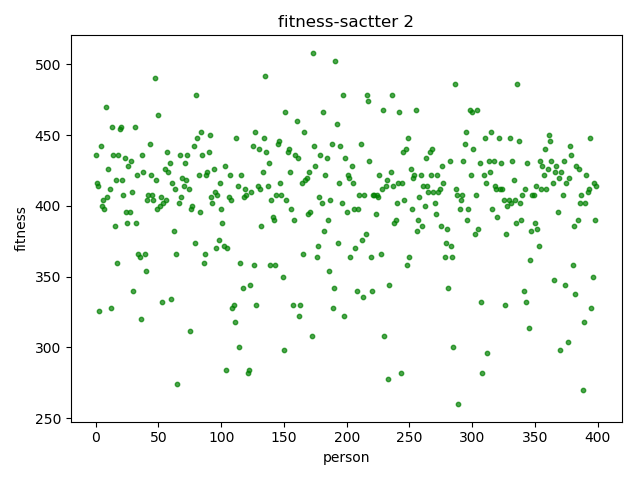


图3 第2次迭代适应值散点图

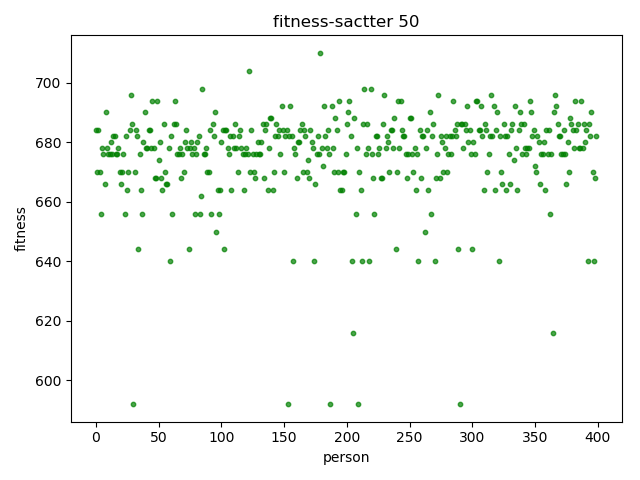


图4 第50次迭代适应值散点图

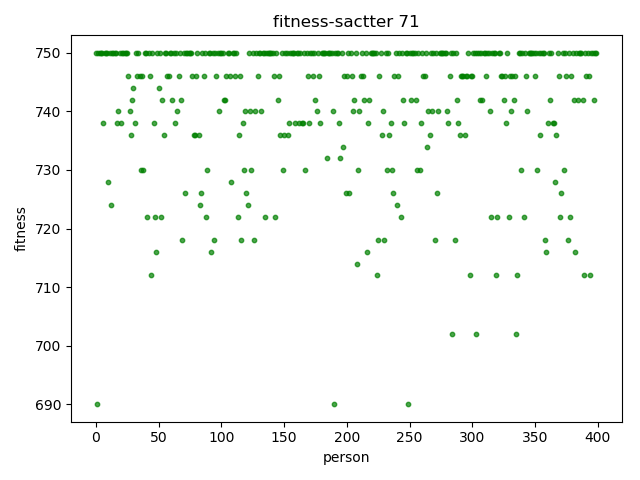


图5 第71次迭代适应值散点图

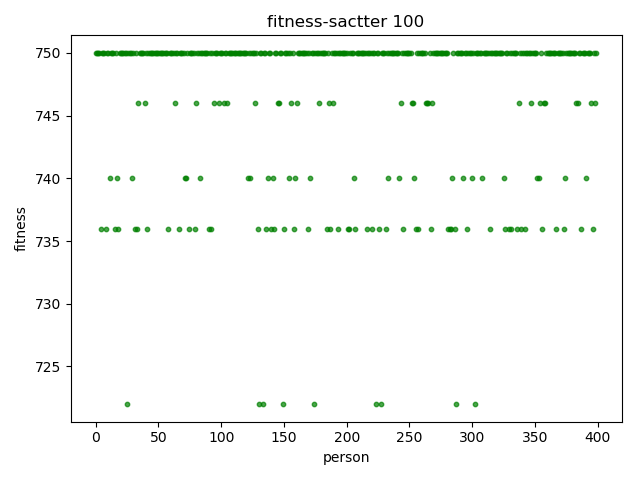


图6 第100次迭代适应值散点图

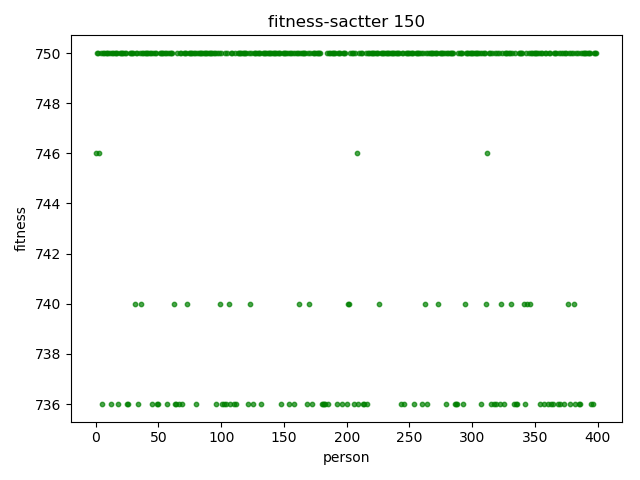


图7 第150次迭代适应值散点图

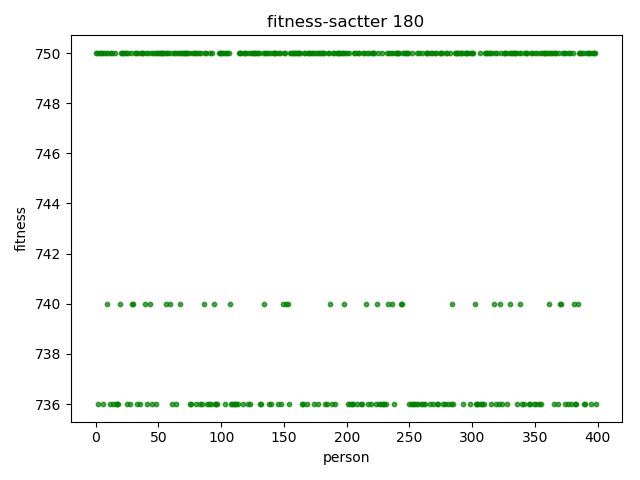


图8 第180次迭代适应值散点图

结果分析：

1. 当迭代71次之后运行结果稳定在750
2. 变异率取为0.0005，本题为欺骗性问题，当变异率过高时及其容易陷入局部最优
3. 交叉方法最初使用传统的双点交叉，结果陷入局部最优。因为当适应值达到较高值但未达到最大值750时，种群中很多个体基因已经完全相同，这时再在对应位置进行双点交叉毫无意义。
4. 改进的交叉方法：采用不同位置双点交叉，即随机选择父亲的某一段染色体，再随机选择母亲的某段染色体，但这两段染色体的位置不同进行交叉，成功跳出局部最优解。
5. 择优部分：当产生的子代的适应值大于原来种群适应值的中位数时，该子代替换原来种群中适应值最小的个体。之所以没有大于原来种群适应值的最小值就替换的原因是这样可以极大地增大找到最优解的速度。同时也保留了物种的多样性。由散点图可以看出，最后每个个体的适应值都变得比较大，但是还是分为三个层级，保留了多样性。