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 100x100 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

② 散点图如下：

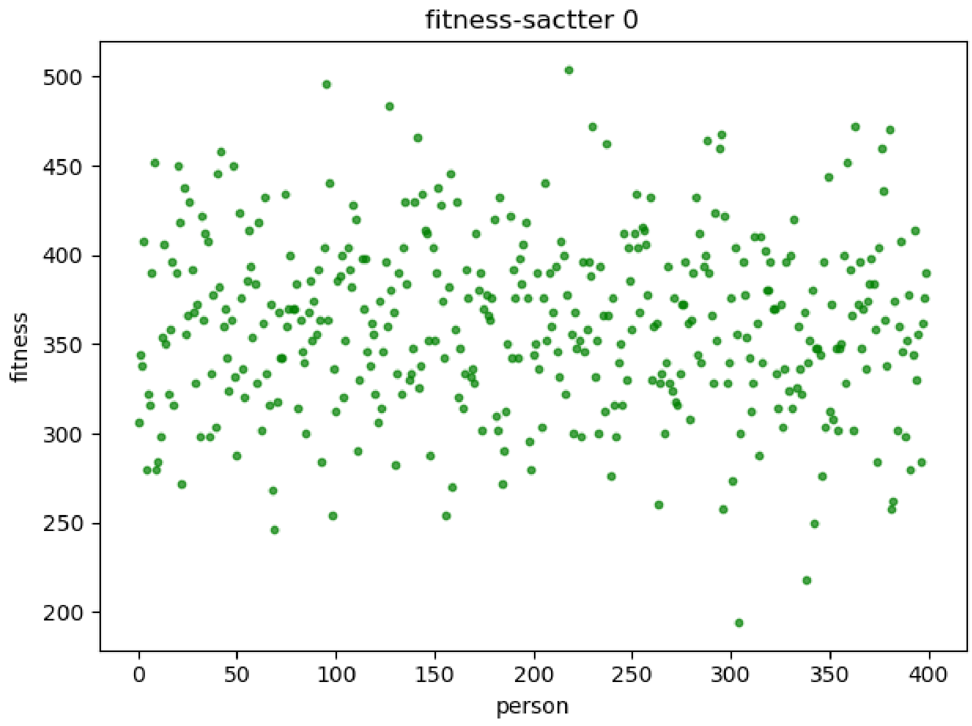

图 1 原始种群适应值散点图

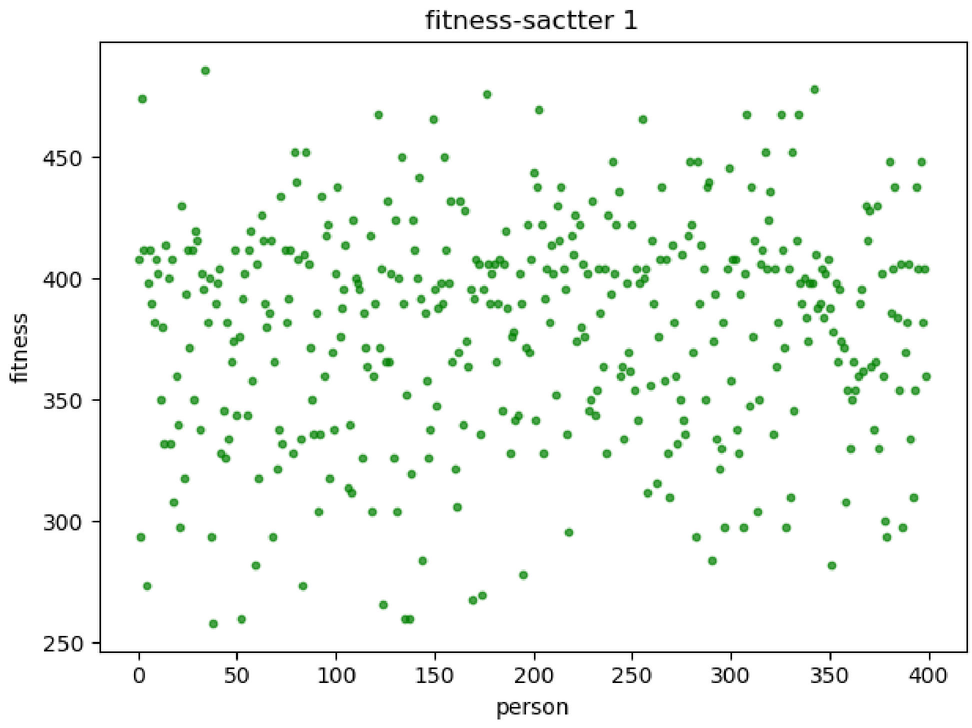

图 2 第 1 次迭代适应值散点图

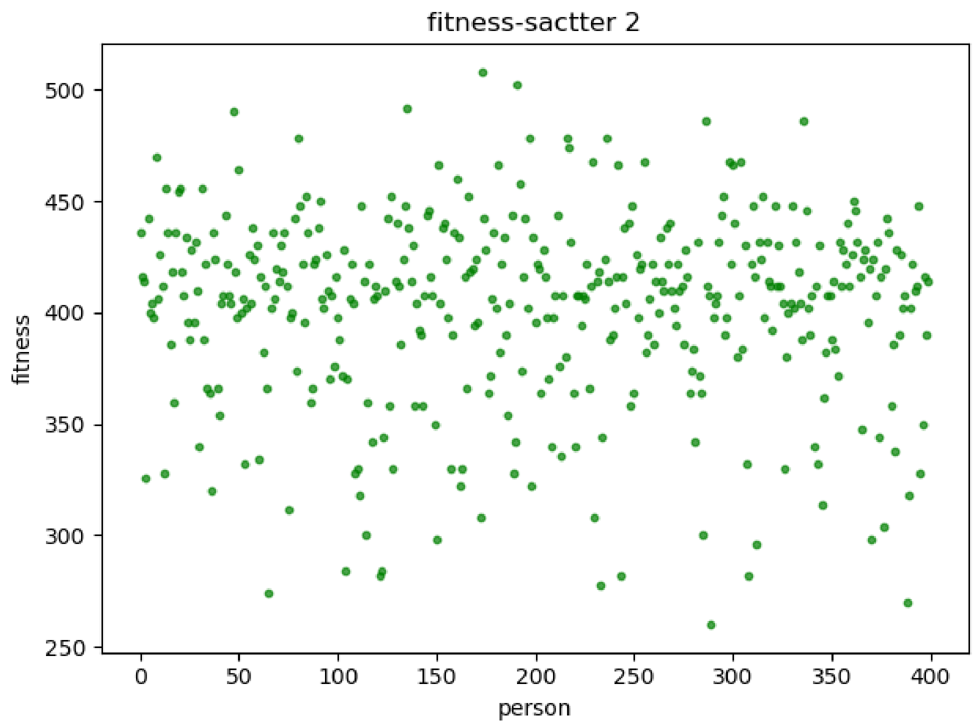

图 3 第 2 次迭代适应值散点图

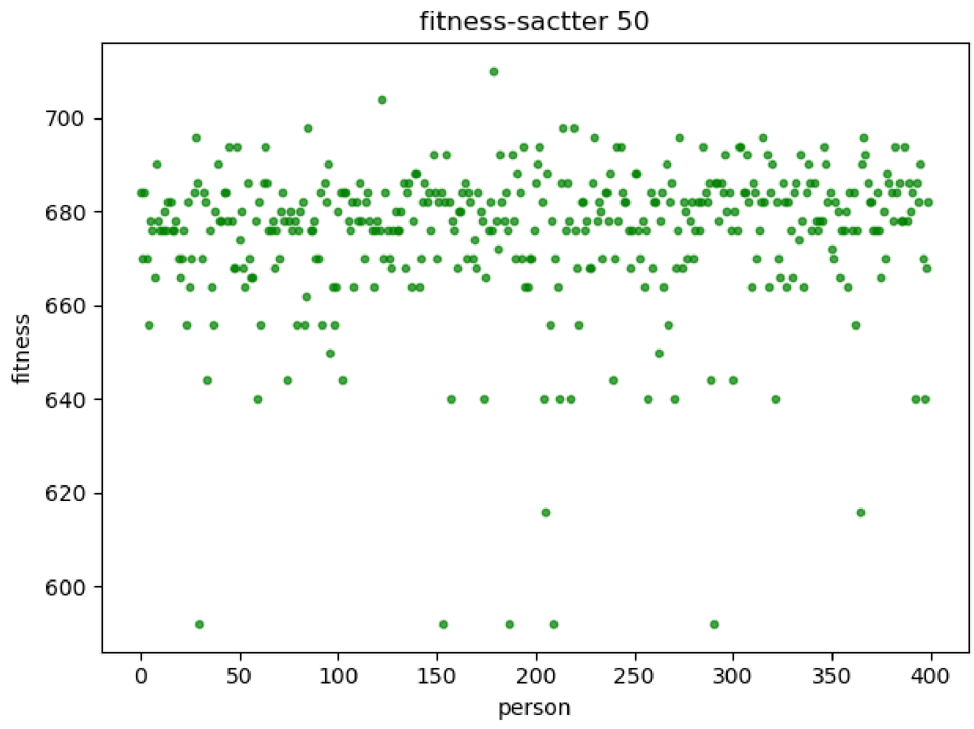

图 4 第 50 次迭代适应值散点图

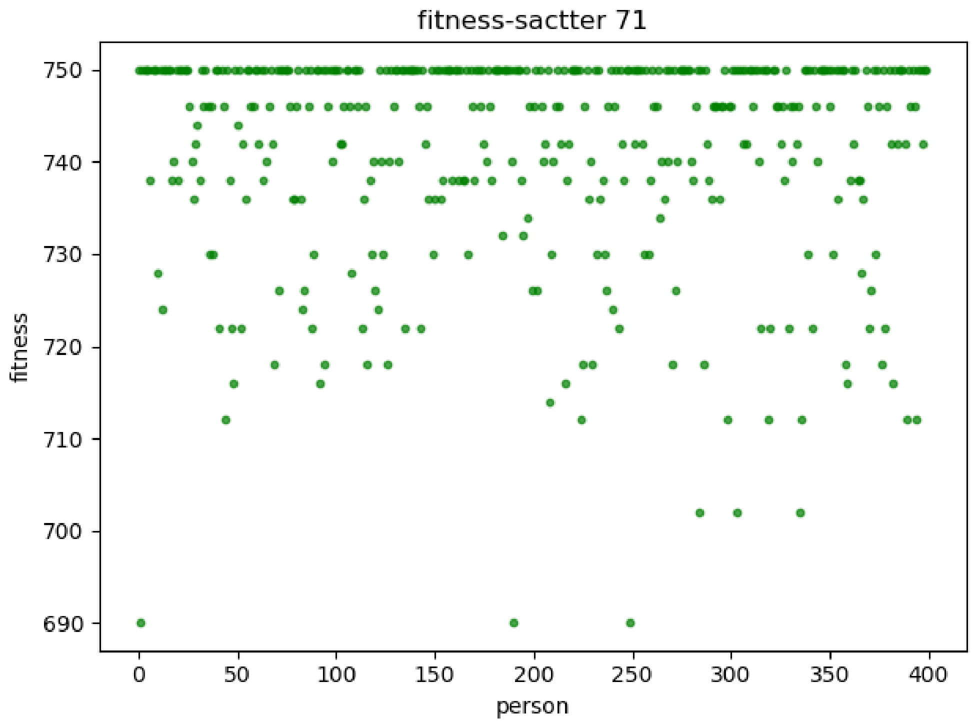

图 5 第 71 次迭代适应值散点图

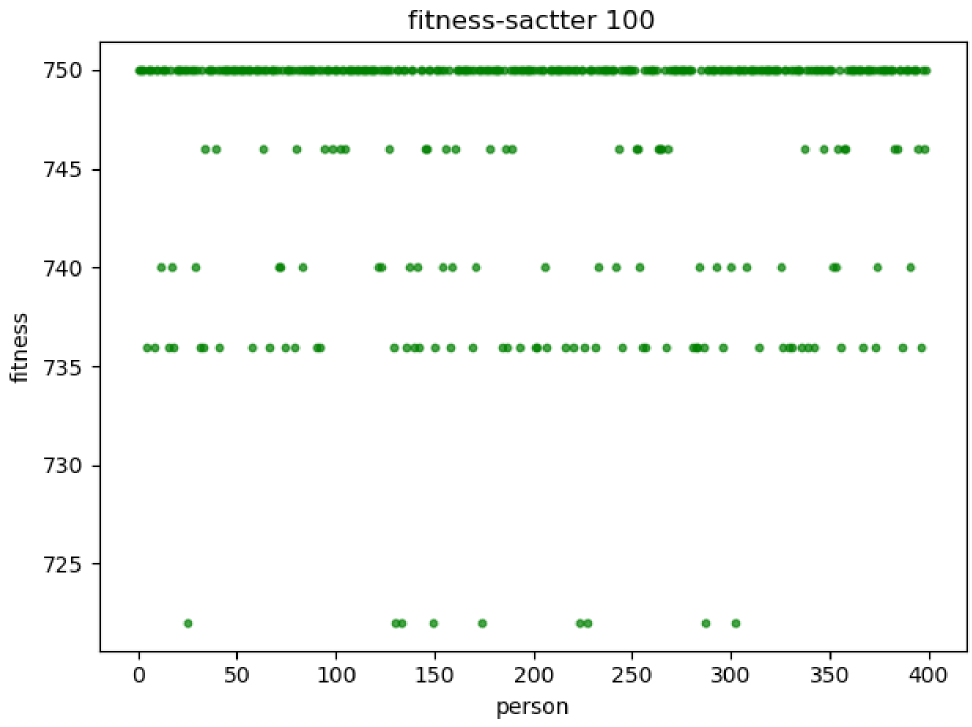

图 6 第 100 次迭代适应值散点图

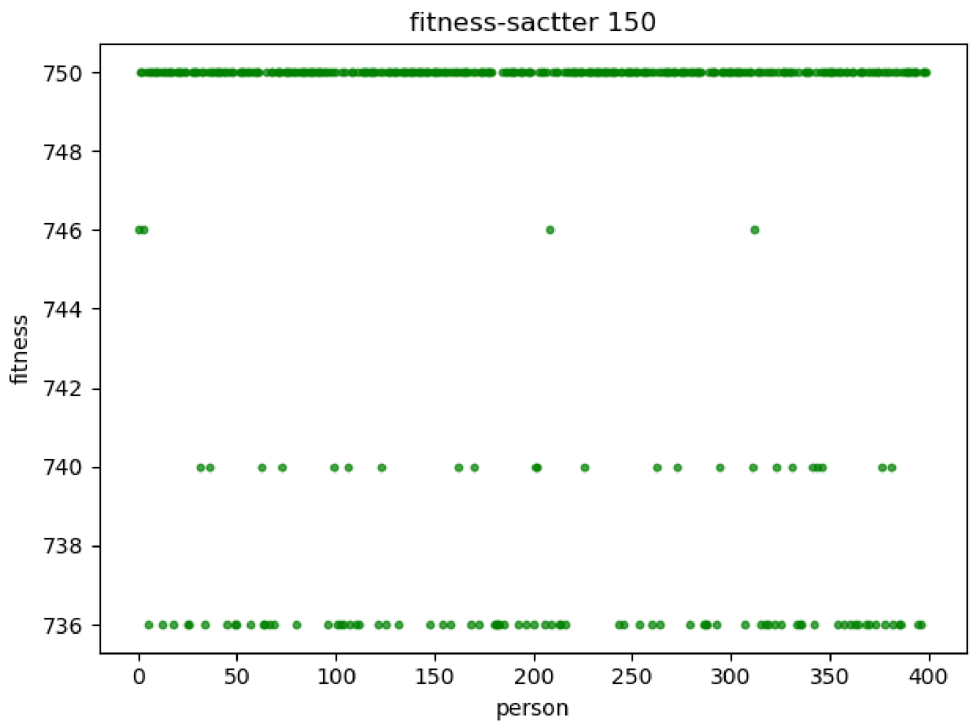

图 7 第 150 次迭代适应值散点图

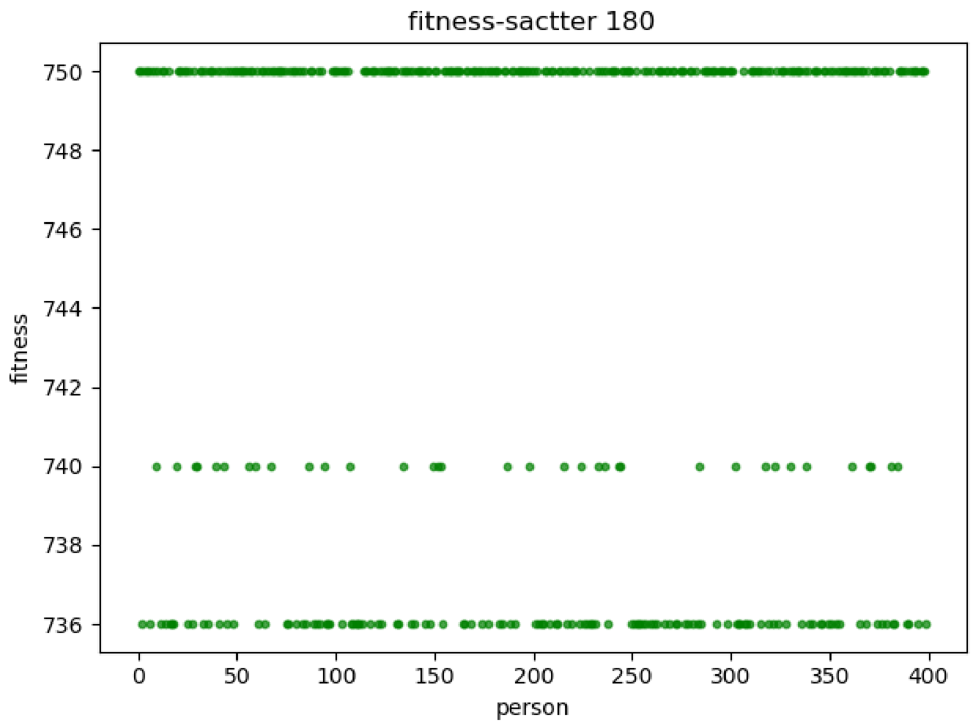

图 8 第 180 次迭代适应值散点图

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。