关于生产过程中决策问题的研究

摘要

本文主要研究生产过程中的决策问题。通过建立抽样检测模型及不同标准下的决策模型，帮助企业在最大化收益指标的目标下完成抽样检测及生产决策工作。

针对问题一，我们建立了一个**动态迭代优化模型**，用于处理次品率数据的不确定性。首先，分析了二项分布的正态近似，并推导了样本量的计算公式。接着，通过动态调整样本量和检验策略，根据实际的质控要求，构建了优化模型。模型基于电子产品行业标准，通过**蒙特卡洛模拟和状态转移规则**来优化检测过程，提高了生产决策的准确性和鲁棒性 (结果见下文)。

针对问题二，首先设置二元决策变量，建立在给定的生产流程中分别建立基于整数规划的支出与收入模型。考虑到企业除盈利额外还需兼顾生产效率等非收入因素，对决策依据指标多元化，并建立可加权的多目标规划模型。求解过程中，通过模拟退火算法将整体优化问题变为多个局部最优化问题，并根据结果选择最佳决策方案。最终在不同情况下所得决策方案（如 [0,0,1,0]）具有准确性及稳定性。（符号含义见下文）

针对问题三，首先对多工序多零件进行决策分段，以问题二中的生产流程作为基本单元，重新设置一阶段内的决策变量。考虑到不合格半成品数量的动态继承关系，建立基于迭代的阶段加工品数量模型，并在此基础上求出阶段费用的递推关系。由于问题三生产工序及中间产品数量较多，需考虑生产效率作为决策指标，故在此基础上结合费用递推构建**基于迭代优化的多目标动态规划模型**。针对模型特点，使用**遗传算法**求解多目标的优化，最终得到在 m 道工序，n 个零件的生产过程中的决策方案。特别的，在 m=2，n=8 时，获得决策方案 [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1]（符号含义见下文）

针对问题四，建立**鲁棒优化模型**，当次品率因为抽样分布带来不确定性时，通过拉丁超立方采样抽取大量均匀分布样本，并根据次品率的正态分布计算样本对应概率，得到目标函数的期望。模型求解过程本质上是对遗传算法中个体适应度的计算引入了不确定性。沿用问题三的遗传算法进行演化，即可得到新的问题二和问题三结果。问题二的最终结果保持不变，仍为 [0,0,1,0]，问题三结果变为 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]。

关键字：**整数规划　动态优化　多目标决策　遗传算法**

一、 问题重述

1.1 背景介绍

电子企业在产品生产过程中涉及多道工序及对应零件或半成品。由于生产中有一定概率产生不合格品，企业需要对不同工序中的相关决策进行选择。同时，由于零件及半成品的总体具有一定次品率，因此需制定相关抽样检测方案保证企业在面对不同工序及相关费用时做出正确的决策。[1]

1.2 赛题概述

- 针对问题一，在电子产品零配件进货检测环节，供应方声称一批零配件的次品率不会超过某个标称值。企业方使用抽样检测方法决定是否接收从供应商购买的这批零配件。问题的核心在于设计检验方案能够在保证检测准确性的同时，减少检测次数。具体包括两种情形：(1) 在 95% 的信度下认定零配件次品率超过标称值，则拒收这批零配件；(2) 在 90% 的信度下认定零配件次品率不超过标称值，则接收这批零配件。问题涉及到假设检验、抽样理论。
- 针对问题二，考虑一个 1 道工序、4 个零配件的给定生产过程，在已知零件及工序相关信息的条件下针对六种不同情况提出具体决策方案，即是否进行“零件检测”、“成品检测”以及“不合格成品拆解”，并给出决策的依据及相应的指标结果。
- 针对问题三，考虑一个 m 道工序，n 个零配件的生产过程。在生产工序及相关费用未知的情况下提出具体决策方案，即判断在各个工序是否进行“零件检验”、“成品检验”以及“不合格成品拆解”。并对实际的所给生产流程进行分析，给出最佳决策方案。
- 针对问题四，题目要求零件、半成品以及成品的次品率需通过抽样检测方法得到，并根据抽样检测方法重新对问题二，问题三的情况进行决策。

二、 问题分析

1. 问题一：企业需要通过抽样检测来评估在两种情形下供应商声称的零配件次品率是否超过标称值 10%。有效的抽样方案将确保决策的准确性和成本的合理性。
2. 问题二：对于给定的生产过程，决策数量及决策流程为已知量。通过构建决策变量按照生产流程从前到后的顺序依次对各个阶段产生的费用进行建模。同时，考虑到次品可能由两个合格零件或一个合格一个不合格零件组成，需要对成品中的不合格品数量分类讨论并单独建模。最后根据每一步向后传递的物品数量及决策带来的费

用支出或收益，加和得到最终模型。对于有限参数的整数规划问题，可基于决策变量的迭代首先寻找局部最优值，再将多个局部最优值作最优化处理得到模型的求解结果。

3. 问题三：相比于问题二，问题三在生产过程上做出一般性推广。由于生产工序及零件数量未知，需要依据从前向后的递推关系建立最终模型。同时，注意到由上一阶段生成的不合格品数量会对下一阶段半成品不合格数产生影响，因此考虑引入对各个阶段合格品及不合格品的动态规划，并随工序向后发展做出递推，计算出各个阶段的相关费用。由于问题三中生产流程更加复杂，因此将决策标准多元化（需同时考虑最终利润及生产效率。即满足单个成品收益最大化的同时，尽量减少检测、拆解次数以保证生产效率)。通过建立基于迭代优化的多目标动态规划模型。
4. 问题四：问题四假设问题二和问题三中的零配件、半成品和成品的次品率是通过抽样检测方法 (如问题一中的抽样检测方法) 得到的。在这种假设下，重新在问题二和问题三的背景下给出决策方案，处理检测数据的不确定性，并将这种不确定性融入到生产过程的决策中，以最小化总成本。

三、 基本假设

1. 抽样检测假设：通过抽样检测得到的次品率能够准确反映整批产品的质量水平，允许使用有限样本的检测结果对整体质量进行推断。
2. 检测准确性假设：假设检测过程能够 100% 准确识别出不合格品，从而简化质量控制，但在实际场景中需要考虑检测误差。
3. 无外部干扰假设：假设整个生产过程中不存在其他外部干扰因素，入市场需求波动、生产设备故障
4. 市场需求稳定性假设：假设市场对产品需求不因产品质量波动而产生显著波动，从而简化收益计算，但在实际场景中需要考虑复杂市场环境及波动。

四、 符号说明

符号	意义	计量单位
p_0	标称次品率	
N	样本大小或检测次数	个
ϵ	允许的误差范围	
α	显著性水平	
$Z_{\alpha/2}$	在 $1-\alpha$ 置信水平下的 z 分数	
x_1	单一生产工序下零件一购买价格	元
x_2	单一生产工序下零件二购买价格	元
a_1	零件一是否需要检测的决策变量	
a_2	零件二是否需要检测的决策变量	
c	单一生产工序下是否对成品检测的决策变量	
d	单一生产工序下是否对不合格品拆解的决策变量	
p_i	单一工序下各个生产阶段的次品率	
α_i	单一工序下各个生产阶段的费用	元

五、 模型建立与求解

5.1 问题一的模型建立

5.1.1 自适应阈值抽样检验模型先验

设次品数量 $X \sim B(n,p)$，当样本量 n 很大的时候，二项分布可近似为正态分布，根据中心极限定理:

$$X \sim N(Np, Np(1-p)) \tag{1}$$

故次品率:

$$\hat{p} \sim N(p, \frac{p(1-p)}{N}) \tag{2}$$

在 $\alpha = 0.05$ 的信度下认定零配件次品率超过标称值拒收这批零配件，即:

$$P\{\frac{\hat{p}-p_0}{\sqrt{\frac{p_0(1-p_0)}{N}}} > Z_{1-\alpha}\} = \alpha \tag{3}$$

设 $\hat{p}-p_0=\epsilon$，ϵ 表示测量值 $\hat{p}$ 与标称值 p_0 之间允许的误差，则情形 1 中 N 的表达式为:

$$N > \frac{Z_{1-\alpha}^2 p_0(1-p_0)}{\epsilon^2} \tag{4}$$

同理可得，在 $\beta = 0.10$ 的信度下认定零配件次品率超过标称值接收这批零配件，即:

$$P\{\frac{\hat{p}-p_0}{\sqrt{\frac{p_0(1-p_0)}{N}}} < Z_{1-\beta}\} = 1-\beta \tag{5}$$

则情形 2 中 N 的表达式为:

$$N > \frac{Z_{1-\beta}^2 p_0(1-p_0)}{\epsilon^2} \tag{6}$$

在上述分析中，已推导出样本中次品数量的分布及其相关的拒绝概率。为了有效评估样本的合格性，设定合理的阈值是至关重要的，由此在 $[0.01, 0.10]$ 的范围对 ϵ 进行讨论（即阈值 $\in [11\%, 20\%]$）

拒绝零假设的概率可以表示为:

$$P(X > k) = 1 - P(X \leq k) = 1 - \sum_{x=0}^{k} P(X = x) \tag{7}$$

结合二项分布的概率质量函数:

$$P(X = x) = \binom{n}{x} p_0^x (1-p_0)^{n-x} \tag{8}$$

因此，拒绝的概率可以写为:

$$P(X > k) = 1 - \sum_{x=0}^{k} \binom{n}{x} p_0^x (1-p_0)^{n-x} \tag{9}$$

通过对比不同阈值下特定样本大小的拒绝零假设概率变化，我们能够观察到为满足所要求的信度，在不同次品率设定下所需的样本规模。此外，这一分析还揭示了不同次品率对样本大小变化的敏感程度，从而为实际质控决策提供了重要参考[2]。

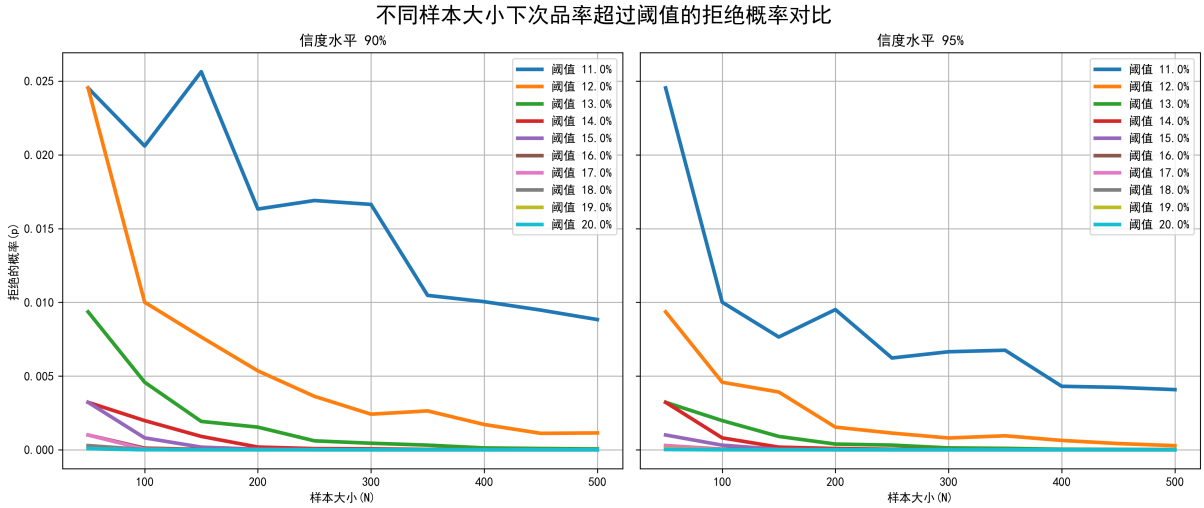

图 1

综合考虑所要求信度下样本规模大小和曲线收敛速度，我们选择阈值 15%（即 $\epsilon = 0.05$）代入式（4）计算，计算得：情况（1）中检测次数最小为 98（97.3996），情况（2）中检测次数最小为 60（59.1255）

根据正态分布近似条件，$np \geq 5$ 和 $n(1-p) \geq 5$ 时，二项分布可近似看作正态分布，考虑到标称值 10% 是供应商对产品质量的最低承诺，即这批零配件的理想次品率会远低于 10%。据此，我们应使用二项分布分析计算，且式子中 p_0 也应根据行业标准调整；同时考虑实际生产场景是连续系列批次抽样检测，故抽样策略应引入一定动态优化策略。

根据上述粗略计算和现实工业场景分析，我们提出基于电子行业标准的动态迭代优化模型。

5.1.2 动态迭代优化模型建立

按照电子产品行业标准，一般良品率平均为 98%，且次品个数服从二项分布，据此生成总体数据，为后续进行蒙特卡洛模拟准备。

根据《中华人民共和国国家标准》计数抽样检验程序中第 1 部分：按接收质量限（AQL）检索的逐批检验抽样计划，逐批检验抽样计划适用于连续系列批次，满足规则：连续批次质量较好可转移到放宽检验，连续批次质量较差可转移到加严检验。

其中，一次抽样方案为：当接受数大于等于 2 时，如果 AQL 加严一级后该批接收，则转移得分加 3，否则转移得分重新设为 0；当接受数为 0 或时，如果该批接收，则给转移得分加 2，否则将转移得分重新设为 0。

5.2 问题一的模型求解

Step1：生成训练数据模拟动态迭代优化过程

状态转移规则简图

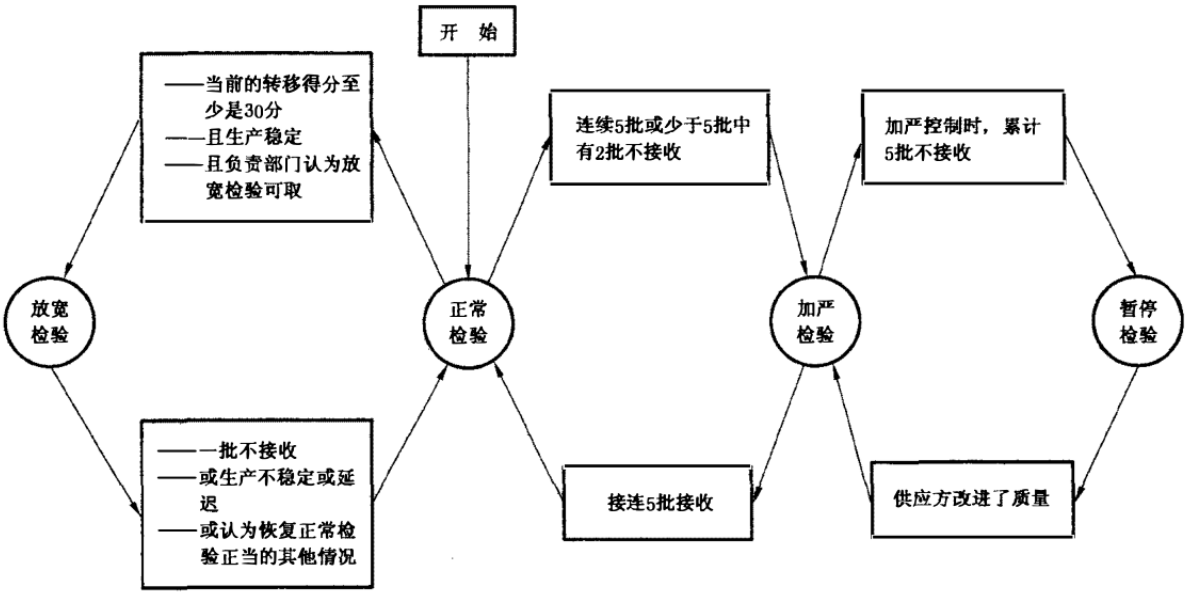

图 2

Step2：构建样本大小更新公式：

$$n_{\text{next}} = \begin{cases} n - k \cdot \Delta n & \text{if status} = 1 \quad (\text{放宽检验}) \\ n & \text{if status} = 2 \quad (\text{正常检验}) \\ n + k \cdot \Delta n & \text{if status} = 3 \quad (\text{加严检验}) \end{cases} \tag{10}$$

其中，$\Delta n = n - \text{accept\_count}$，表示步长根据本轮检验中拒收的样本数量动态调整。状态转移逻辑：状态 1：放宽检验状态 2：正常检验状态 3：加严检验

Step3：根据国标书设计状态转移条件，做出两点微调：1. 考虑到本问题估计的理想次品率和标称值之间的差异，将正常检验转移为放宽检验的转移得分下限修改为 10 分，以达到收敛。2. 参考国标书中对暂停检验的定义，考虑在理想次品率分布下容易出现样本数量 N 正反馈地增长和减小的现象，设置连续 5 次放宽/加严后回归正常检验的策略。

具体规则如下：

正常检验转移到放宽检验：当转移得分大于等于 10 时；

正常检验转移到加严检验：当最近 5 次的拒收计数之和大于等于 2 时；

加严检验转移到正常检验：当最近拒收计数之和为 0，或连续 5 次进行加严检验时；

放宽检验转移到正常检验：当本次检验拒收，或连续 5 次进行放宽检验时；

同时，若发生拒收，则重置转移得分为 0；

当接受数大于等于 2 时，如果 AQL 加严一级后当前状态为接收，则转移得分增加 3；

当接受数大于等于 2 时，如果 AQL 加严一级后当前状态为拒收，则重置转移得分为 0；

当接受数为 0 或 1 时，且当前状态为接收，则转移得分增加 2；
当接受数为 0 或 1 时，且当前状态为拒收，则重置转移得分为 0；

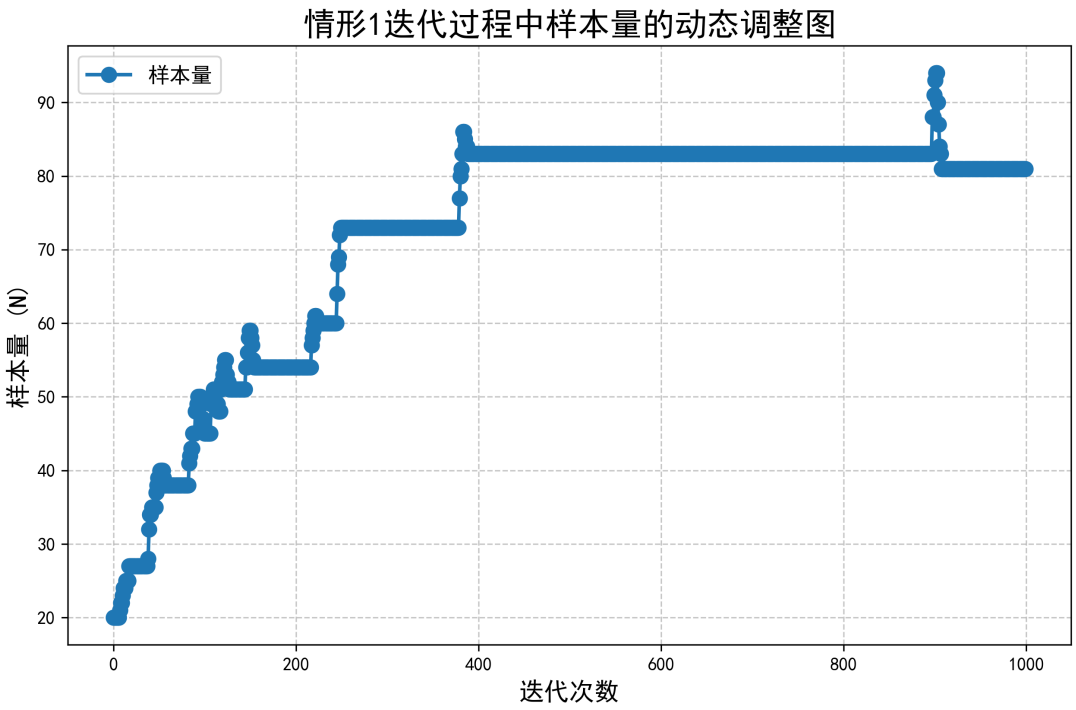

图 3

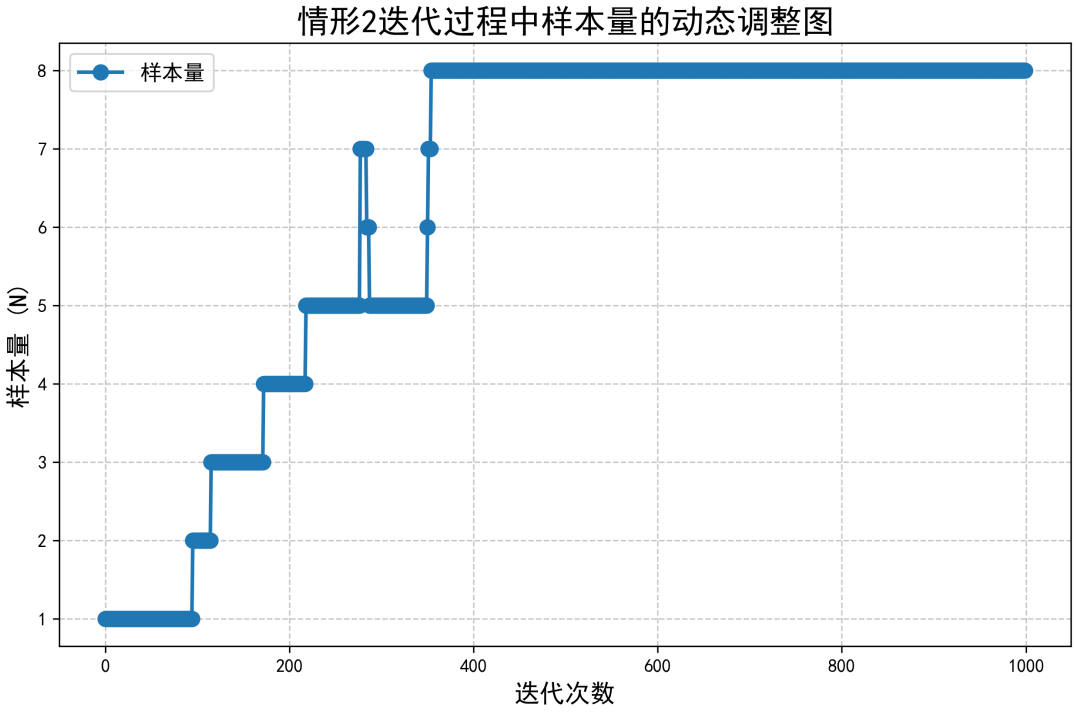

图 4

在良品率为 98% 的条件下，通过动态优化模型计算得：情况（1）中检测次数最小值收敛于 83，情况（2）中检测次数最小值收敛于 8。

5.3 问题二的模型建立

问题二场景中，成品由两个零件构成，企业需要在生产过程中对零件检测、成品检测以及成品拆解进行选择决策，而不同的决策方案会导致不合格品数量、调换损失费等变量的变化。本文通过设置二元决策变量，建立整数规划模型，并计算在不同决策选择下企业最终利润。对于不合格品流入市场导致的企业信誉的降低，题目中将其包含在调换损失费中，因此模型将企业最终利润作为决策衡量标准。

问题二决策流程

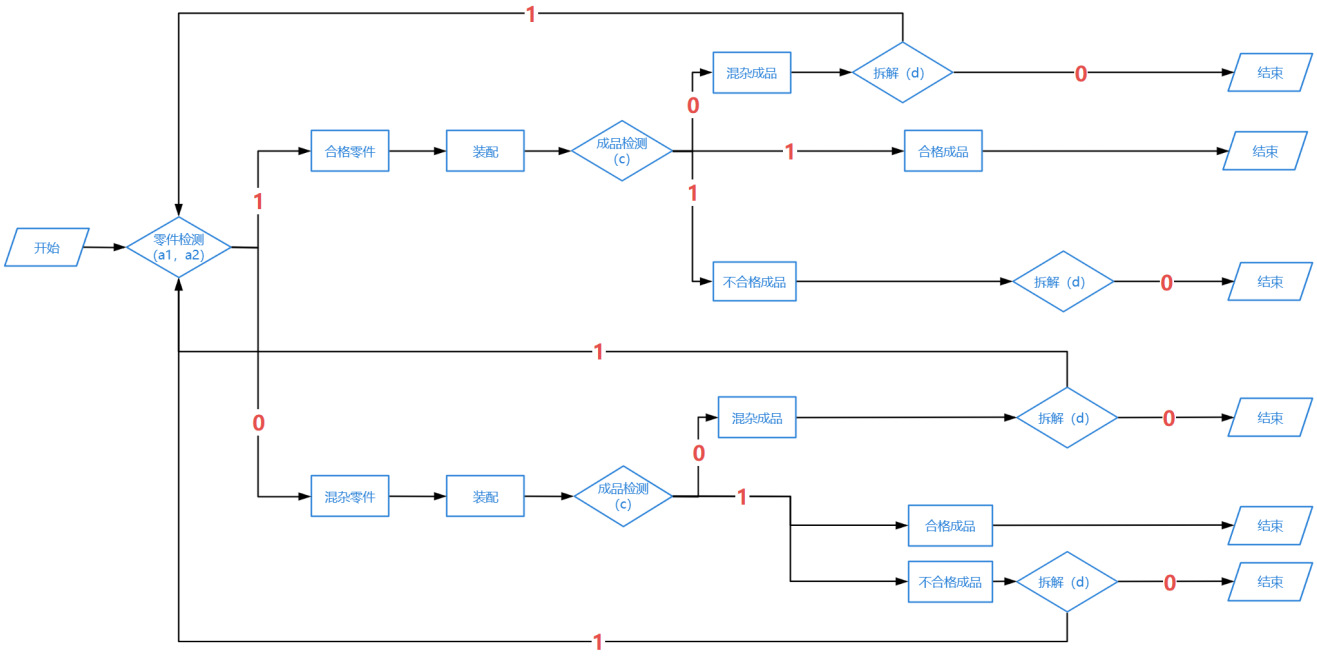

图 5

5.3.1 整数规划下支出成本模型

在生产过程中，企业需考虑以下过程所产生的成本费用：零件购买、零件检测、零件装配、成品检测、成品拆解、不合格品调换损失以及不合格品替换。由于决策时只有两种选择，即“采纳”和“不采纳”，因此使用二元整数变量作为决策参数。特别的，由于生产过程为企业的流水线生产，且次品率保持不变，因此认为各个阶段零件数恒定为一个较大的常数，不会因为拆解或丢弃的选择而发生变化。下面对各个部分的成本进行单独讨论。

1. 零件购买成本

对构成成品的两个零件单独购买，有购买成本：

$$cost_1 = n_1 \cdot (x_1 + x_2) \tag{11}$$

其中 n_1 为零件数量，x_1 为零件一购买单价，x_2 为零件二购买单价

2. 零件检测成本

检测成本取决于是否决策对某个零件进行检测，因此可得零件检测费用：

$$cost_2 = n_1 \cdot (a_1 y_1 + a_2 y_2) \tag{12}$$

其中 a_1 为“零件一是否进行检测”的决策变量，a_2 为“零件二是否进行检测”决策变量，y_1 为零件一的检测费用，y_2 为零件二的检测费用。

过程中对零件一、二的检测决策会对装配产品数量产生变化，由于流水线上会不断生产新的零件，因此流入装配环节的零件数量应当为：

$$n_2 = \max\left(n_1\left(1 - a_1 p_1\right), n_1\left(1 - a_2 p_2\right)\right) \tag{13}$$

其中 p_1、p_2 分别为零件一和零件二的次品率。

3. 零件装配成本

由上一阶段所求“流入装配环节零件数量”可得装配费用为：

$$cost_3 = n_2 \cdot \alpha_1 \tag{14}$$

其中 α_1 为装配费.

4. 成品检测成本

由上一阶段所求“流入装配环节零件数量”可得成品检测费用为：

$$cost_4 = c \cdot n_2 \cdot \alpha_2 \tag{15}$$

其中 α_2 为成品检测费用。

5. 成品拆解成本

成品中包含合格品及次品，其中次品包含两种类型。第一种次品由不合格的零件装配而成，且由于流水线零件数量为一恒定较大值，因此可认为该种次品是由一个不合格零件和一个合格零件装配而成。第二种次品由合格的零件装配后，因成品存在一定次品率而产生，即该种次品由两个合格零件装配而成。因此成品中合格品数量为：

$$k_{pass} = \left(n_2 - \left[(1 - a_1) n_1 p_1 + (1 - a_2) n_1 p_2\right]\right) p_3 \tag{16}$$

由此可得成品中次品数量为：

$$k_{fail} = n_2 - k_{pass} \tag{17}$$

其中 p_3 为成品的次品率。

拆解过程只针对成品中的次品，由（17）可得成品拆解费用为：

$$cost_5 = c \cdot d \cdot k_{fail} \cdot \alpha_3 \tag{18}$$

其中 α_3 为成品拆解费用

6. 调换损失及替换成本

当用户购买到不合格成品时，企业需支付调换损失费并为顾客替换商品，即支付成品的零件成本及装配费。根据（16），调换损失及替换成本费为：

$$cost_6 = (1-c) \cdot k_{pass} \cdot (x_1 + x_2 + \alpha_1 + \alpha_4) \tag{19}$$

其中 α_4 为调换损失费。

5.3.2 基于二元决策变量的收入模型

生产过程中收入主要由出售成品及不合格成品拆解后的合格零件回收产生。通过建立针对是否检测和是否拆解的二元决策变量 c、d，分别计算上述收入，得到流水线收入模型。

1. 成品出售收入

首先由（13）和（16），可得流入市场的成品数量为：

$$n_3 = (1 - cp_3) \cdot n_2 - c \cdot k_{fail} \tag{20}$$

其中 p_3 为成品的次品率。由（20），可得成品出售收入：

$$income_1 = n_3 \cdot \alpha_5 \tag{21}$$

其中 α_5 为成品售价

2. 零件回收收入

在零件装配中，由于生产线零件较多，因此认为不合格零件一般与合格零件装配，组成不合格成品。同时，两个合格零件装配而成的成品也可能成为次品，因此需分别对两中情况的零件回收费用进行计算。

针对由两个合格零件组成的次品，零件回收收入为：

$$income_2 = c \cdot d \cdot (x_1 + x_2) \cdot (1 - p_3) \cdot k_{pass} \tag{22}$$

针对由一个不合格零件和一个合格零件组成的次品，零件回收收入为：

$$income_3 = x_1 \cdot (1 - a_1) \cdot n_1 \cdot p_1 + x_2 \cdot (1 - a_2) \cdot n_1 \cdot p_2 \tag{23}$$

由支出模型及收入模型，确定最终收入为：

$$profit = \sum_{i=1}^{3} income_i + \sum_{i=1}^{6} cost_i \tag{24}$$

5.4 问题二的模型求解

模型建立了具有多决策变量的适应性函数，并需要对函数值进行最优化处理。对此本题采用模拟退火算法，将整体优化问题变为多个局部最优化问题，并根据结果选择最优化方案。首先随机生成决策变量值，并设置迭代次数阈值 1000。在当前解的邻域内随机选择一组新的解。并计算前后两组解产生的函数值之差。若差值大于零，则保留原解；若差值小于零，则更新解为新的一组决策变量值。当求得局部最优值或迭代次数达到阈值时，算法结束。具体步骤如下：

Step1: 随机生成决策变量值 $[a_1, a_2, c, d]$，并设定迭代次数阈值 1000

Step2: 生成解析域，对上一步中的解进行更新。

Step3: 根据利润判断是否接受新的解。若 $E_{new} > E$，则更新解析域 $X = X_{new}$; 若新解利润较差 $E_new < E$，则根据 Metropolis 准则决定是否接受; 若 $E - E_{new} <= 500$，则以一定概率 exp$((E - E_{new})/T)$ 接受新解。

Step4: 温度退火，重复第二步和第三步 100 次后，降低温度 $T = T \cdot alpha$。温度越低，接受较差解的概率越小，逐步逼近全局最优解。

Step5: 检查终止条件：当温度 T 降到 T_{min} 时，算法停止

5.4.1 问题二模型求解结果

根据所建立模型及求解算法，针对不同情况最终得到下列决策方案

表 1　问题二求解结果

	零件一检测	零件二检测	成品检测	成品拆解
情况一	0	0	1	0
情况二	0	0	1	0
情况三	0	0	1	0
情况四	0	0	1	0
情况五	0	0	1	0
情况六	0	0	1	0

表中“1”代表执行决策，“0”代表不执行决策。从整体结果的对比来看，此模型及对应的决策方案具有稳定性和准确性。

如图所示，当初始零件数量为 100 时，在不同情况下采取不同决策方案，企业获得利润不同。

不同情况下的求解结果 ($n_1 = 100$)

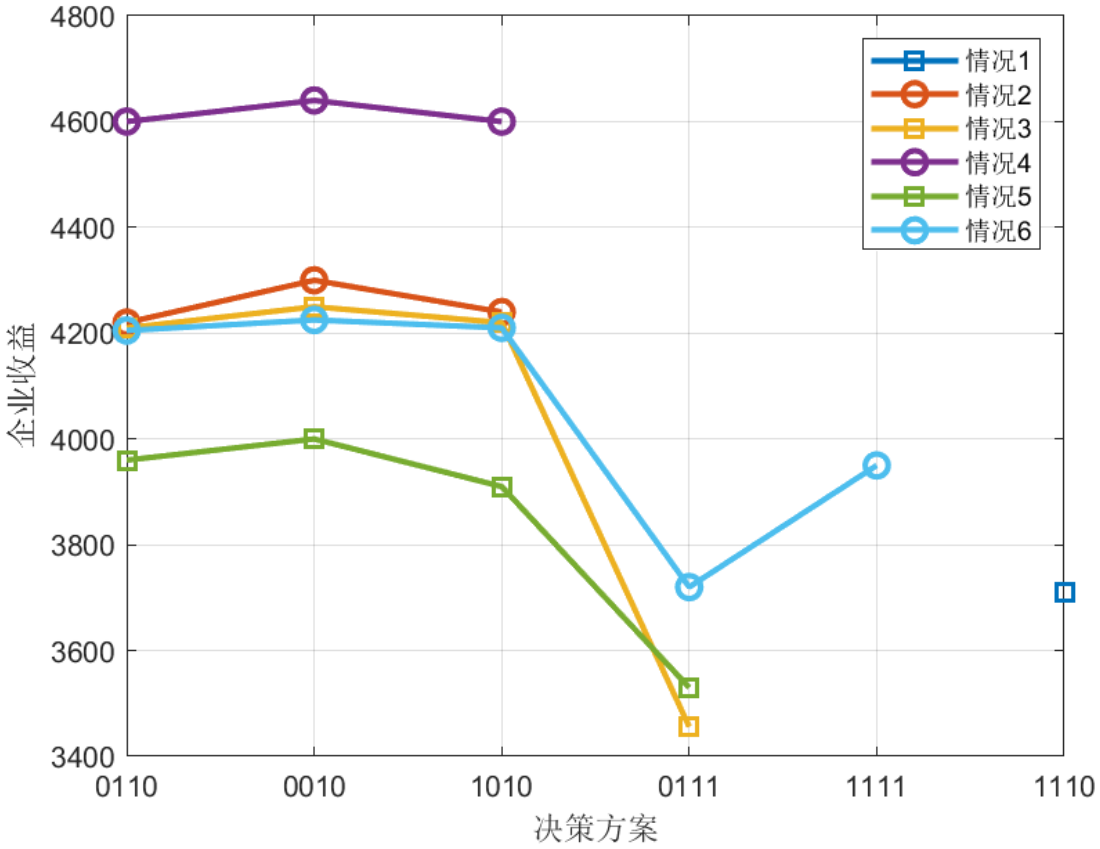

图 6

5.5 问题三的模型建立

问题三在上一问题的基础上对生产流程及零件数量进行一般化处理，随着工序和零件种类的改变，决策变量、半成品数等都会相应变化。为化归问题，需要对整个流程进行分段处理，将整体流程分为多个类似问题二的单工序步骤（零件数量不同），最后根据每一段间的递推关系将模型进行数学表示。其中，为确定流入下一阶段的半成品数量，需对半成品中合格品数量及不合格品数量单独建模。[3]

相比于问题二，问题三的生产流程更加复杂，除企业盈利额外还需考虑盈利效率，即能否在单位时间内做出更多的商品。因此将决策标准多元化，使最终模型在满足单个成品收益最大化的同时，尽量减少检测、拆解次数以保证生产效率，并尽量降低成品的不合格率，维护企业信誉。由此可以建立多目标优化模型

5.5.1 阶段加工品数目的递推求解

对问题三建模的关键在于确定每一道工序中不合格品 (由不合格的零部件合成) 和合格品 (所有合成零部件都是合格的) 分别的个数，分别用 A_{ij} 和 B_{ij} 表示 (i, j 分别表示所在工序阶段和加工品编号)。由零件合成的要求可以得到，A_{ij} 和 B_{ij} 只与前一阶段的不合格品和合格品数量有关，也就是 A_{i-1k} 和 B_{i-1k}(k 代表合成该加工品的上一阶段半成品编号)，服从递推关系。要得到递推关系式，需要将工序分为两种阶段，即零件到半成品和半成品到半成品阶段 (半成品到成品阶段的递推式和后者相同)。

为方便计算，问题三中用到的符号如下，未做特殊说明下标 i 都，从 0 开始:

- k, m, n 分别为零件数量，工序数，零件种类数
- p_{ij} 为第 i 道工序时编号为 j 的加工品次品率
- X_{ij}, Y_{ij} 分别决定是否在 i 道工序对编号为 j 的加工品进行检修和拆解，为 0/1 决策变量
- d_{ij}, c_{ij}, q_{ij} 分别为在 i 道工序编号 j 的加工品的检修费，装载费和拆解费
- ti 为第 i 个零件的购买费用
- Q 为加工品分组数
- M_{ij} 为拆解第 i 道工序编号为 j 的加工品后能省下的零件购买费
- $halfnumi$ 为第 i 阶段的半成品数目
- $sale\ change$ 分别为成品售卖总费用和调换总费用

与问题二不同的是，问题三中需要确定加工品的分组数 Q，即多少个零件合成半成品。根据工序数和零件种类的约束，可以得到关系式

$$Q = \lceil n^{\frac{1}{m}} \tag{25}$$

首先计算零件到半成品阶段的递推式，半成品中的不合格品只能由不合格的零件合成。考虑一般情况，不合格半成品数由最大次品率的零件编号决定:

$$A_{1i} = \max(p_{0j} \cdot (1 - Y_{0j})) \cdot k, \quad j \in [Q(i-1), Qi) \tag{26}$$

同理，半成品中的合格品全部由合格的零件合成，其个数由最小次品率的零件编号决定:

$$B_{1i} = \min(1 - p_{0j} Y_{0j}) \cdot k, \quad j \in [Q(i-1), Qi) \tag{27}$$

再计算半成品到半成品阶段的递推式，下一阶段的不合格半成品由上一阶段的不合格半成品以及合格半成品合成后产生的新次品合成组成，同样也是由上一阶段产生最多次品的编号决定，则第 k 阶段的递推式如下:

$$A_{ki} = \max(A_{k-1j}(1 - Y_{k-1j}) + B_{k-1j} p_{k-1j}(1 - Y_{k-1j})), \quad j \in [Q(i-1), Qi) \tag{28}$$

同理，下一阶段半成品中的合格品全部由合格的上一阶段半成品合成，第 k 阶段的递推式如下:

$$B_{ki} = \min(B_{k-1j}(1 - p_{k-1}j)), \quad j \in [Q(i-1), Qi) \tag{29}$$

5.5.2 阶段费用的递推求解

求解费用首先需要明确费用的组成，我们有如下文字表达式，其中拆解剩下费用主要由合格零件省下的额外零件购买费构成:

总收入 = 成品售卖费 + 拆解省下费用 - (零件购买费 + 装配费 + 检修费 + 拆解费 + 调换损失费)

同样的，各费用可以分成两个阶段求解 (半成品到成品阶段只会额外多出调换损失费和成品售卖费，且拆解省下的费用和前面有些区别，会单独阐明)。

对于零件组装成为半成品的阶段，零件购买费和检修费都能直接计算 (装配费和后面的阶段一起计算)，分别如下:

$$buycost = \sum_{i=0}^{n} t_i \cdot k \tag{30}$$

$$checkcost = \sum_{i=0}^{n} d_i Y_{0i} \cdot k \tag{31}$$

这一阶段的拆解能省下的费用是指拆成这些零件能够省下的总零件购买费用，由两方面构成。一是所有合格的零件组成的半成品拆解省下的费用，另一种是有不合格的零件组成的半成品拆解省下的费用。前者是构成半成品的所有零件购买费用，后者是构成半成品的所有合格零件的购买费用。前一种情况公式如下:

$$allgoodfee = B_{1i} p_{1i} Y_{1i} \cdot \sum_{j=Q(i-1)}^{Qi} t_j \tag{32}$$

针对后一种情况，需要求出组成半成品的合格零件。拆解后每一种零件能省下的购买费应该如下求得: 取 B_{1i} 和对应零件剔除次品后产生的合格零件中的较小值。求和后即能得到总费用，即:

$$partgoodfee = \sum_{j=Q(i-1)}^{Qi} \min(B_{1i}, kp_{0j}(1 - Y_{0j})) \cdot t_j \tag{33}$$

最后将二者相加即可得到该阶段拆解能省下的总费用，作为递推的基础:

$$M_{1i} = (allgoodfee + partgoodfee) \cdot X_{1i} \tag{34}$$

对于半成品到半成品的阶段，首先需要求得该阶段的半成品种类数。为简化问题，假设每一步的分组数目都相同，即 $m^{\frac{1}{n}}$ 为整数 (实际代码中可以为任何数)，可以求得第 i 阶段的半成品数为:

$$halfnumi = Q^{n-i} \tag{35}$$

在 5.5.1 中求得不合格品和合格品数目后，检修费、装配费和拆解费计算如下，假设处于第 j 个工序阶段:

$$checkcost = \sum_{i=1} halfnumi(A_{ji} + B_{ji}) \cdot d_{ji}Y_{ki} \tag{36}$$

$$loadcost = \sum_{i=1} halfnumi(A_{ji} + B_{ji}) \cdot c_{ij} \tag{37}$$

$$separatecost = \sum_{i=1} halfnumi(A_{ji} + B_{ji}p_{ji}) \cdot X_{ji}Y_{ji} \tag{38}$$

拆解节省费用可以由递推关系求解，注意到其仍然由两部分组成，即上述的 $allgoodfee$ 和 $partgoodfee$，前者计算较为简单，只需要找到组成该半成品的零件编号即可:

$$allgoodfee = B_{ki}p_{ki}Y_{ki} \sum_{j=iQ^k-Q^k}^{iQ^k} t_j \tag{39}$$

后者利用递推关系，即该节省费用为组成其所有半成品能节省费用的总和，由于 M_{1i} 已求解，故可以求得在第 k 阶段:

$$partgoodfee = \sum_{j=Q(i-1)}^{Qi} M_{ki} \tag{40}$$

拆解总节省费用即二者相加:

$$M_{ki} = (allgoodfee + partgoodfee) \cdot X_{ki} \tag{41}$$

最后需要额外计算成品阶段的售卖费和调换费:

$$sale = (A_{m1} + B_{m1})(1 - p_{m1}Y_{m1}) \cdot price \tag{42}$$

$$change = (A_{m1} + B_{m1}p_{m1})(1 - Y_{m1}) \cdot price \tag{43}$$

最后的总费用即根据总收入公式即可求得。

5.5.3 多目标优化

多目标中的总费用已经求得，还需得到成品中的合格品数量 T 和检修和拆解总次数 G。由之前的分析，容易得到 $T = B_{m1} \cdot (1 - p_{m1})$; 拆解总次数后续通过编程直接得到结果，伪代码如下:

Algorithm 1 Calculate Total Check

$total\_check \leftarrow acc\_num \times \sum_{k=1}^{n} \text{individual}[: type\_acc] + \sum_{k=n-1}^{n} \text{individual}_k$
for $i \leftarrow 1$ **to** $step\_num - 1$ **do**
    $half\_num\_i \leftarrow group\_num^{\text{step\_num}-i}$
    for $j \leftarrow 1$ **to** $half\_num\_i$ **do**
        $step \leftarrow$ calculate_past_half_num$(i, group_num, step_num)$
        $total\_check \leftarrow total\_check + (A[i][j] + B[i][j]) \times \text{individual}[type\_acc + step + j - 1]$
        $total\_check \leftarrow total\_check + (A[i][j] + B[i][j] \times \text{new\_half\_badrate}[calculate\_past\_half\_num(i, group\_num, step\_num) + j]) \times \text{individual}[type\_acc + step + half\_num + j - 1]$
    end for
end for

5.6 问题三的模型求解

问题三通过遗传算法求解，python 代码中使用了 DEAP 包的进化算法框架。每一个个体都是一个 0/1 组成的列表，代表决策变量，交叉操作即对两个个体的部分决策变量进行交换，具体使用了 DEAP 库自带的 tools.cxTwoPoint 方法；变异操作即对个体的某几个决策变量进行翻转，使用了 tools.mutFlipBit 方法。

个体的适应度 fitness 属性由三个元素组成，分别是成品合格数，利润和检测拆解总次数，每一次进行子代选择时，相比于传统的多目标优化解法将其通过加权转化为单目标，该算法中使用了智能优化算法 NSGA-II，通过非支配排序选择子代，并为了维持种群的多样性，引入了拥挤距离的概念。拥挤距离衡量了个体在目标空间中的分布密度，算法倾向于选择拥挤距离较大的个体，以避免解的聚集。

最后将演化代数设置为 1000 代，每代由 10 个个体组成即可解决不同 m,n 值对应的生产策略 (次品率为定值)。

当 $m = 2, n = 8$ 时，对应的最优生产策略如下 (零件个数设置为 100 个，对结果没有影响): [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1]，即只对半成品 1 进行检修和拆解，成品进行拆解，其余不做操作。此时的 fitness 是 (81.0, 68160.0, 221.0)。

Step1: 利用决策为“进行”的比例评估决策方案：在遗传算法中，每个个体由一组决策变量构成，具体包括：

- 零件检验决策（n）
- 半成品检验决策（n_{half}）
- 半成品拆分决策（n_{half}）
- 成品检验和拆分决策（各 1 个）

根据 m 道工序、n 个零配件, 零件分组的数量 n_{group} 可表示为:

$$n_{group} = \lceil n^{1/m} \rceil \tag{44}$$

半成品的数量 n_{half} 可表示为:

$$n_{half} = \sum_{i=1}^{m-1} n_{group}^{i} \tag{45}$$

Step2: 对不同数据量 m、n 下决策方案比较分析 g1 代表小数据量零配件数、小数据量工序数（n=8,m=2）g2 代表大数据量零配件数、大数据量工序数（n=500,m=5）g3 代表小数据量零配件数、大数据量工序数（n=32,m=5）g4 代表小数据量零配件数、小数据量工序数（n=500,m=2）

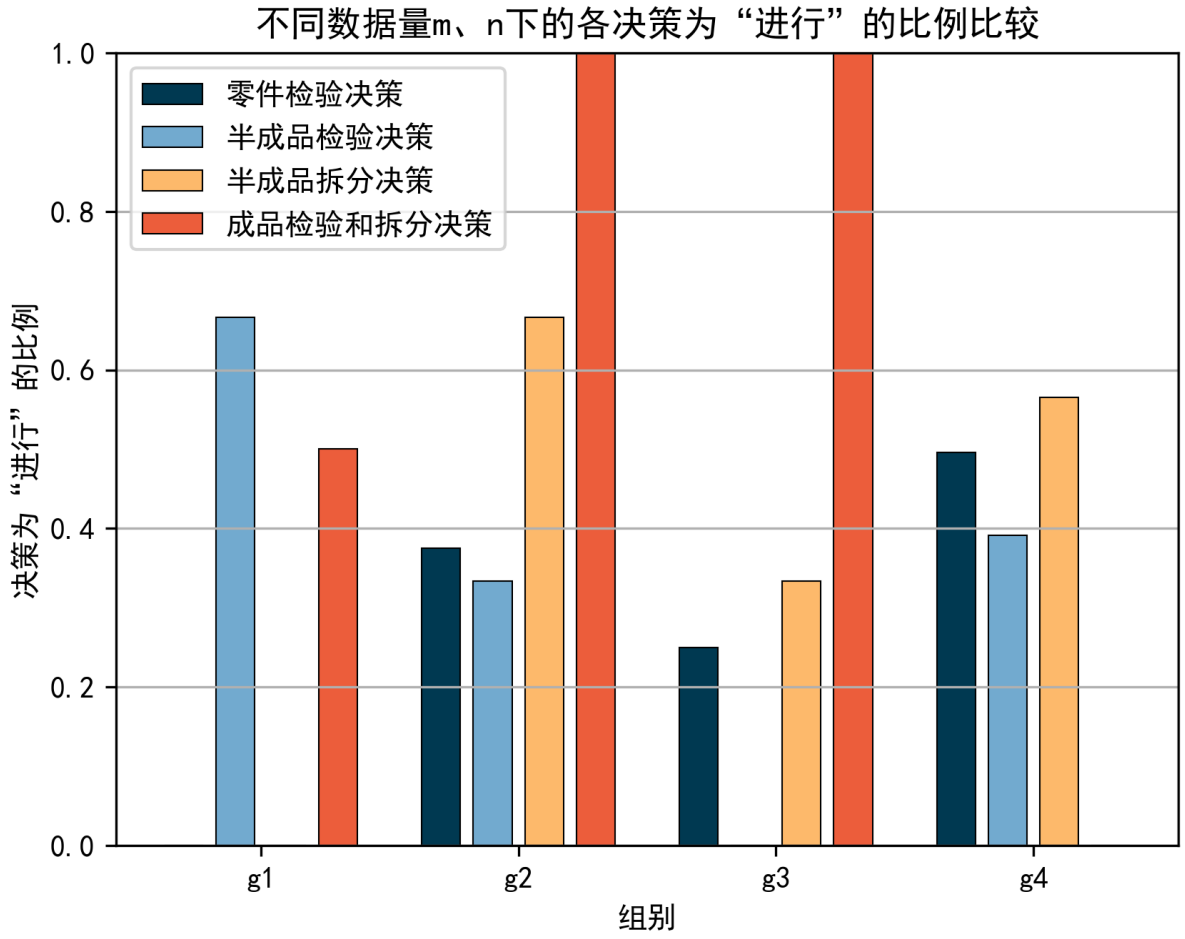

图 7

零件检验决策: 在小数据量和小工序数的情况下（如 g1 和 g4），零件检验决策显得重要。而在复杂产品（如 g2）的情况下，零件检验的相对重要性下降。

成品检验决策: 随着工序数的增加，成品检验的重要性提升，尤其是在复杂产品中（如 g2 和 g3），这表明在多工序的环境下，成品的整体质量控制显得更加重要。拆分决

策: 在大数据量和复杂工序的情况下（如 g2），拆分决策的比例可能出现上升，反映出在处理复杂产品时，决策的多样性和灵活性。

5.7 问题四的模型建立

次品率数据通过抽样检测得到，这会带来估计的不确定性，这种不确定性可能影响生产过程中的的检测、装配和拆解决策。为了得出更好的决策方案，我们构建鲁棒性模型处理这些不确定性[4]，优化目标函数，在各种可能的次品率下寻求最优决策策略；同时，与问题二、问题三相同，决策过程同样需要对多个阶段的检测、装配和拆解决策进行全局优化，以确保目标函数达到最值。据此，通过遗传算法并结合随即规划的思想即可对问题求解。

在问题四中，考虑生产过程中连续系列批次抽样，可将抽样总体视为无穷，次品率近似服从正态分布，即次品率的估算基于以下公式:

$$P(X \leq x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{46}$$

为模拟不确定性，采用拉丁超立方采样进行不确定样本的采样。拉丁超立方采样（Latin Hypercube Sampling, LHS）是一种统计学中的采样技术，用于生成近似均匀分布的样本。它特别适用于计算机模拟和多维参数空间的优化问题。LHS 旨在通过确保每个参数的范围内都有样本点，来提高采样的效率。每一个样本对应其相应的概率 p_i，采样样本总数为 samplenum, 最后目标函数 (新的适应度) 即变为，其中 p_i 通过上述正态分布公式即可求解:

$$\max(\sum_{i=1} samplenum fitness \cdot p_i) \tag{47}$$

5.8 问题四的模型求解

Step0: 改变问题三代码中的次品率, 通过拉丁超立方采样抽取 1000 个样本, 即 1000 种不同次品率的组合，得到新的次品率后代入目标函数，更新每一个个体的适应度，该适应度即鲁棒优化的目标函数的期望。

Step1：采用遗传算法（GA）初始化种群，使用 DEAP 库随机生成 100 个个体，个体的每一位表示不同的决策变量。计算每个个体的适应度，进行适应度评估包括总合格产品数量、利润和总检验次数。适应度函数通过以下步骤实现：计算零件到半成品的成本，包括购买成本、检验成本和拆分成本。递推计算半成品到成品的合格率和相关成本。计算总费用并与销售收入进行比较，最终得出个体的适应度值。

Step2：完成包括选择、交叉和变异的进化过程：选择操作：使用非支配排序选择（NSGA-II），选择适应度较高的个体进入下一代。交叉操作：采用双点交叉方法（cxTwo-Point）生成新的个体。变异操作：使用位翻转变异（mutFlipBit）对个体进行随机变异。

Step3：迭代与收敛：算法运行设定为 1000 代，每代输出包括最小、最大、平均适应度及标准差。通过监控 Pareto 前沿，评估算法的收敛性和解的多样性。

问题二重做后结果和原来保持一致，仍然为 [0,0,1,0] 即只对成品进行检修，与预期保持一致 (其鲁棒性很高)。

问题三的具体案例当 $m=2, n=8$ 时经过演化后得到最终结果为

Best individual (Generation 999): [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1], Fitness: (96.2190780453189, 9046.9056248712, 117.75670792339709)

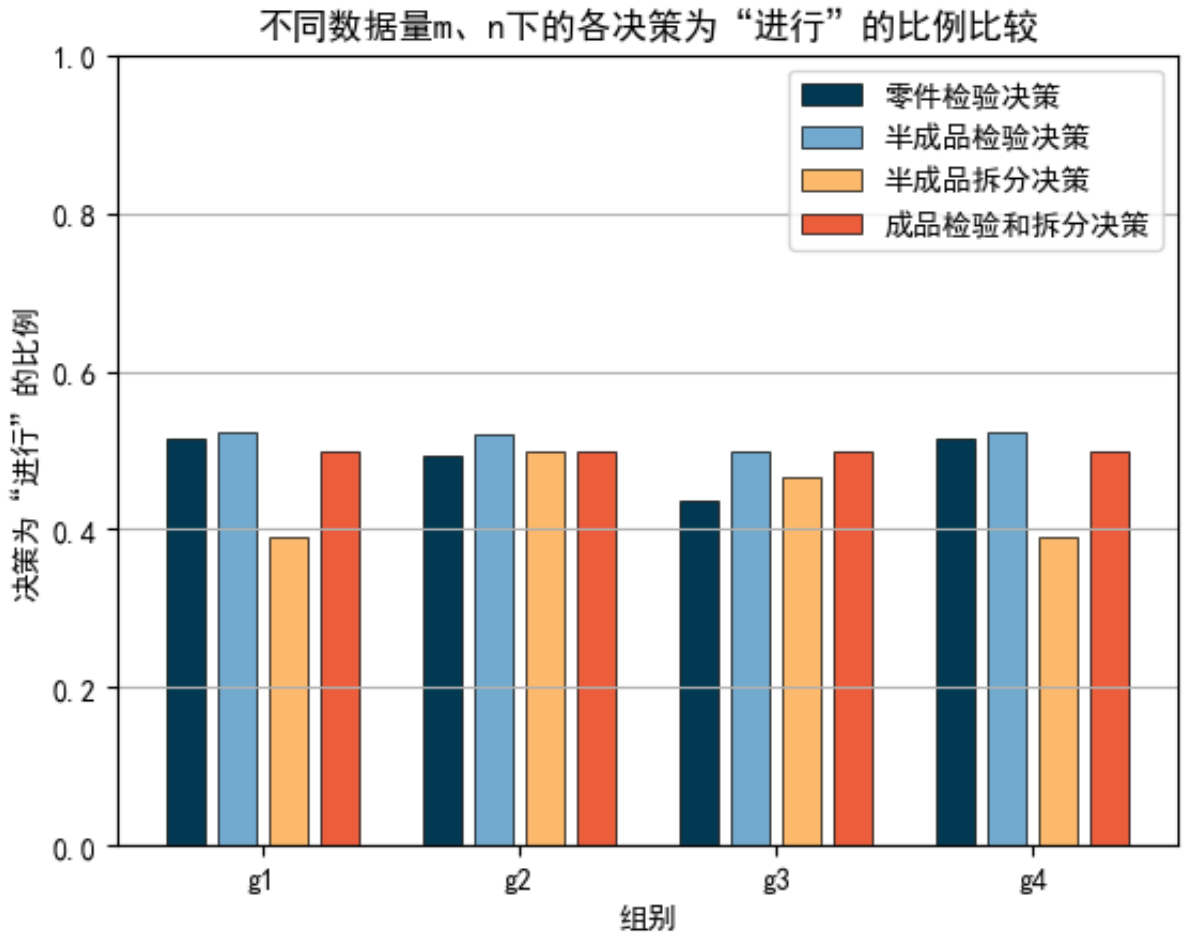

图 8

对不同数据量 m、n 下的决策方案进行比较，发现不同组别下成品检验都稳定地起到显著作用，模型的鲁棒性增强

六、 模型评价

6.1 规划模型的稳定性检验

针对单一流程的规划模型，对所给的相关产品数据在小范围内进行波动（10%），并与数据稳定时的决策方案进行对比。由结果可看出，当数据值发生一定波动时，决策并未发生变化。

规划模型的稳定性分析结果

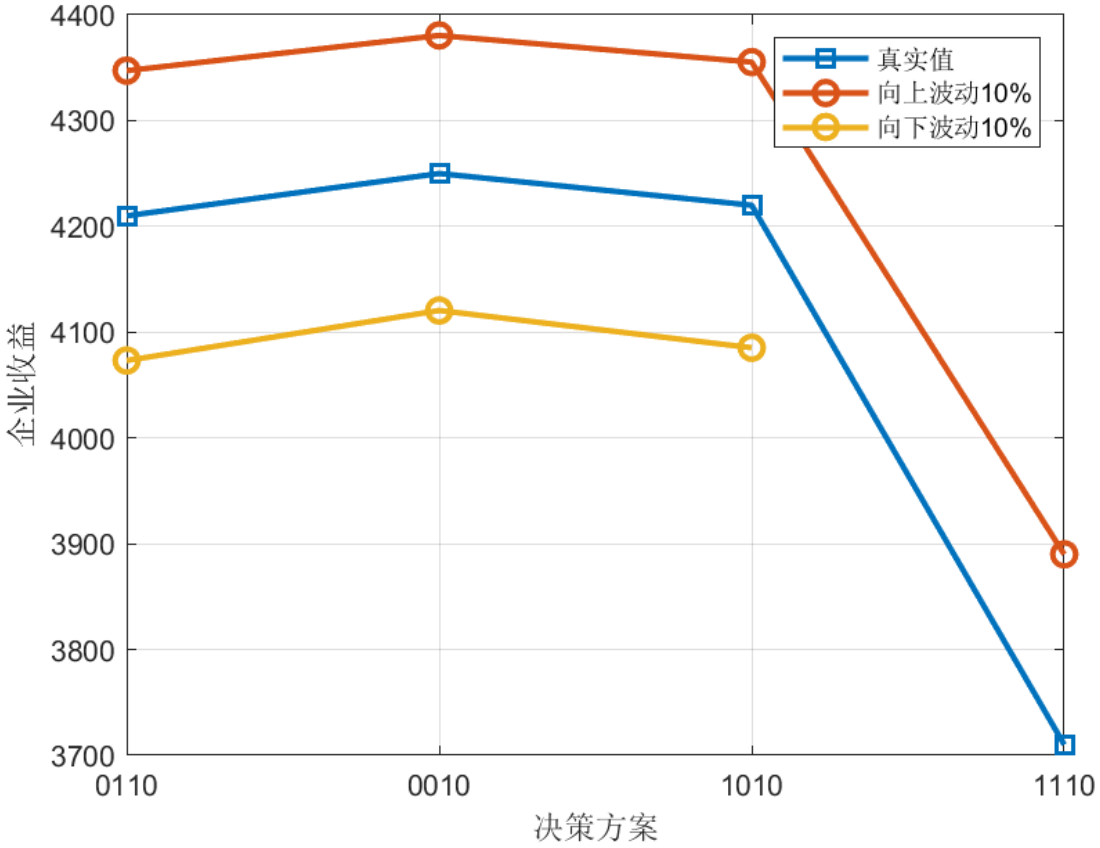

图 9

6.2 对零配件数目 n 和工序数 m 的敏感性分析

通过对电子产品生产实际调研，发现工序数 m 和零件数 n 的范围如下：

- 简单产品：通常需要 5-10 个工序，涉及基本的组装和测试步骤。
- 中等复杂度的产品：可能需要 10-30 个工序，包括各种组装、焊接、测试和质量检查。
- 复杂产品：通常需要 30-100 个工序，涵盖多种制造和测试过程，如自动化组装、功能测试、质量控制等。

- 简单电子产品（如基本的电路板）：可能需要 20-50 个零件。
- 中等复杂度的产品（如智能家居设备或小型家电）：可能需要 50-200 个零件。
- 复杂产品（如智能手机或计算机）：可能需要 200-1000 个零件，甚至更多。

为分析工序数 m 和零件数 n 对工业生产过程中决策方案的影响，我们选择了 n=32 代表简单产品、n=512 代表复杂产品，比较他们在不同工序数取值下的决策方案变化

根据问题二中 n=2，m=1 的情况和问题三 n=8，m=2 的情况，我们发现在这两个场景中零件检验决策和成品检验对目标优化起到更显著的作用

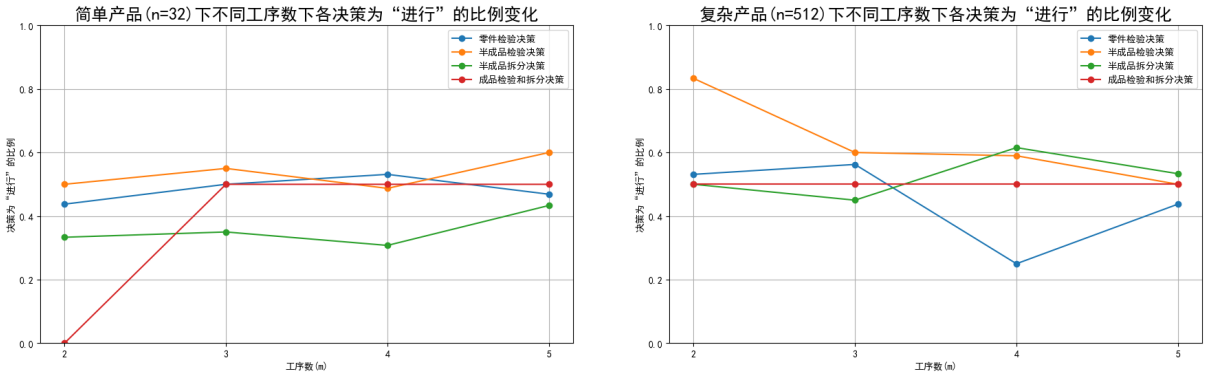

图 10

根据问题二中 n=2，m=1 的情况和问题三 n=8，m=2 的情况，我们发现在这两个场景中零件检验决策和成品检验对目标优化起到更显著的作用在对零配件数量 n 的敏感性分析中，发现成品检验稳定地起到显著作用，且存在随着 n 增大成品检测重要性增强的趋势，推测这是因为随着工序数的增加，产品的制造过程变得更加复杂。这可能导致每个工序的潜在缺陷数量增加，使得成品检验在最终产品质量保证中的作用变得更加重要；在复杂产品（n=512）中对零配件检测重要性随工序数的增加有减少的趋势，推测这是因为在复杂产品的制造中，整体系统的集成性提高，可能使得成品检验能够更有效地捕捉到零配件问题。

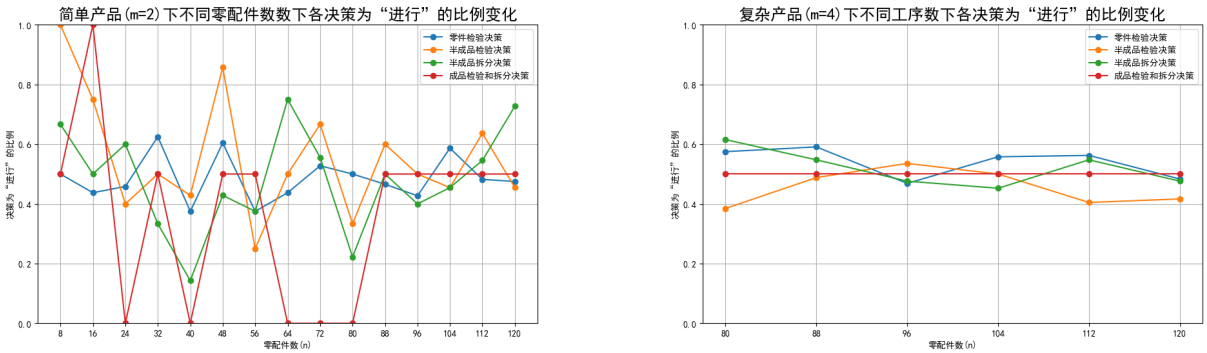

图 11

在对工序数 m 的敏感性分析中，发现成品检验依旧稳定地起到显著作用。在简单

产品（m=2）中，决策随零配件数波动很大，而在复杂产品（m=4）中则较为平稳，推测这是因为简单性导致的脆弱性，缺乏冗余和容错机制。

6.3 模型的推广

1. 应用于其他制造业领域：本文所提出的模型不仅适用于电子制造业的决策，对于同样使用流水线生产，需同时处理多道工序协同决策的相关制造领域如汽车制造业、金工也等同样适用。在不同生产过程中，通过改变模型预设参数即可帮助相关领域生产过程做出决策。
2. 引入随机变量进行决策预测：在实际生产中，销售策略及制造要求需根据市场实时变化。在已知当前和过去市场情况的条件下，可以设置随机变量对市场波动进行模拟，帮助企业根据市场预测决策方案变化，保证企业能够在生产过程中保持成本控制及制造效率的优势

参考文献

[1] 刘超. 基于强化学习的流程工业运行指标优化决策方法 [D]. 东北大学,2020.DOI:10.27007/d.cnki.gdbeu.2020.002303.

[2] GB/T2828.1-2008 计数抽样检验程序第 1 部分按接收质量限（AQL）检索的逐批检验抽样计划案 (ISO 2859-1:1985,NEQ)

[3] Blanche117. 2020.7.18. 数学建模——多目标规划模型（智能优化算法 NSGA-II）. Blanche117. https://blog.csdn.net/weixin_45745854/article/details/107433168

[4] 章宇, 张静, 刘文欣, 等. 管理决策中的分布鲁棒优化 [J/OL]. 中国科学基金,1-10[2024-09-08].https://doi.org/10.16262/j.cnki.1000-8217.20240729.001.

附录 A　问题一抽样方案动态迭代优化模型-python 源程序

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as stats
from collections import deque

# 设置 matplotlib 以支持中文字体显示
plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定默认字体
plt.rcParams['axes.unicode_minus'] = False # 解决保存图像时负号出现方块的问题

def Threshold():
    nominal_defect_rate = 0.10 # 标称次品率
    threshold_rates = [0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2] # 不同的阈值
    sample_sizes = np.arange(50, 501, 50) # 不同的样本量
    confidence_levels = [0.90, 0.95] # 不同的信度水平

    # 创建图表
    fig, axes = plt.subplots(nrows=1, ncols=len(confidence_levels), figsize=(14, 6),
        sharey=True)

    for idx, confidence_level in enumerate(confidence_levels):
        ax = axes[idx]
        for jdx, threshold_defect_rate in enumerate(threshold_rates):
            # 计算 Z 值
            z_critical = stats.norm.ppf(1 - (1 - confidence_level) / 2)

            # 计算拒绝的临界值
            critical_values = [sample_size * threshold_defect_rate + z_critical *
                np.sqrt(sample_size * threshold_defect_rate * (1 - threshold_defect_rate)) for
                sample_size in sample_sizes]

            # 计算拒绝概率
            probabilities = [1 - binom.cdf(critical_value, sample_size, nominal_defect_rate) for
                sample_size, critical_value in zip(sample_sizes, critical_values)]
            ax.plot(sample_sizes, probabilities, linestyle='-', label=f'阈值
                {threshold_defect_rate * 100:.1f}%', linewidth=3)

        ax.set_xlabel("样本大小(N)")
        ax.set_title(f'信度水平 {int(confidence_level * 100)}%')
        ax.grid(True)
        ax.legend()

    axes[0].set_ylabel('拒绝的概率(p)')
    plt.tight_layout(rect=[0, 0, 1, 0.95])
```

```
    plt.suptitle("不同样本大小下次品率超过阈值的拒绝概率对比", fontsize=18)
    plt.savefig("阈值对比.png", dpi=300, bbox_inches='tight')
    plt.close()


def Calculation_n():
    # 定义参数
    alpha = 0.05 # 拒收风险
    beta = 0.1  # 接受风险
    p = 0.1     # 标称次品率
    d = 0.05    # 允许的误差

    # 计算 z 值
    z_alpha = stats.norm.ppf(1 - alpha) # 单尾检验
    z_beta = stats.norm.ppf(1 - beta) # 单尾检验

    # 计算样本大小
    n1 = (z_alpha ** 2 * p * (1 - p)) / (d ** 2) # 拒收标准
    n2 = (z_beta ** 2 * p * (1 - p)) / (d ** 2) # 接收标准

    print(n1, n2)

def generate_training_data(n, p):
    """生成 num_samples 组良品率为 p 的二项分布数据"""
    return np.random.binomial(n, p)

def Dynamic_1():
    nominal_defect_rate = 0.1 # 标称次品率
    p_good = 0.98        # 良品率
    n_initial = 20        # 初始样本大小
    k = 1
    n = n_initial

    total_batches = 0 # 总批次数
    transfer_score = 0
    n_values = []            # 记录每次迭代的样本大小
    recent_reject_counts = deque(maxlen=5) # 滑动窗口
    status_counts = deque(maxlen=5) # 滑动窗口
    status = 2 # 1-放宽 2-正常 3-加严
    current = 0


    while True:
        status_counts.append(status)
        # 生成训练数据
        defects = generate_training_data(n, p_good)
        accept_count = defects
```

```
delta_n = n-accept_count

# 计算拒收和接收的信度
p_accept = stats.binomtest(n-accept_count, n, nominal_defect_rate,
    alternative='greater').pvalue

# 判断接收或拒收
if p_accept >= 0.95:
    current = 0 # 接收
else:
    current = 1 # 拒收
recent_reject_counts.append(current)
# print(current)
total_batches += 1
n_values.append(n) # 记录当前样本大小

# 根据转移得分和拒收计数更新样本大小
if status==2 and transfer_score >= 10: #原版为30
    # 正常检验->放宽检验
    status = 1
elif status==2 and sum(recent_reject_counts) >= 2:
    # 正常检验->加严检验
    status = 3
elif status==3 and sum(recent_reject_counts) == 0 or sum(status_counts)==15:
    # 加严检验—>正常检验
    status = 2
elif status==1 and current==1 or sum(status_counts)==5:
    # 放宽检验—>正常检验
    status = 2
    # 发生拒收时，转移得分加2
    transfer_score = 0
elif sum(recent_reject_counts)<=2 and status==3 and current==0: # 接收数为2后加严检验通过
    transfer_score += 3
elif sum(recent_reject_counts)<=2 and status==3 and current==1: # 接收数为2后加严检验通过
    transfer_score = 0
elif sum(recent_reject_counts)>=3 and current==0:
    transfer_score += 2
else:
    status = status

if status==1:
    n = n - k * delta_n
elif status==2:
    n = n
else:
    n = n + k * delta_n
```

```
        # 判断是否满足约束
        if total_batches >= 1000:
            break

    # 绘制折线图
    plt.figure(figsize=(10, 6)) # 设置图形大小
    plt.plot(n_values, marker='o', linestyle='-', markersize=8, linewidth=2, label='样本量') #
        设定线条和标记样式

    # 添加标题和标签
    plt.title('情形1迭代过程中样本量的动态调整图', fontsize=18, fontweight='bold') # 设置标题
    plt.xlabel('迭代次数', fontsize=14) # 设置X轴标签
    plt.ylabel('样本量 (N)', fontsize=14) # 设置Y轴标签

    # 添加网格
    plt.grid(True, linestyle='--', alpha=0.7) # 设置网格线样式和透明度

    # 添加图例
    plt.legend(fontsize=12)

    # 保存图像
    plt.savefig("n1.png", dpi=300, bbox_inches='tight')
    plt.close()

def Dynamic_2():
    nominal_defect_rate = 0.1 # 标称次品率
    p_good = 0.98        # 良品率
    n_initial = 1        # 初始样本大小
    k = 1
    n = n_initial

    total_batches = 0 # 总批次数
    transfer_score = 0
    n_values = []            # 记录每次迭代的样本大小
    recent_reject_counts = deque(maxlen=5) # 滑动窗口
    status_counts = deque(maxlen=5) # 滑动窗口
    status = 2 # 1-放宽 2-正常 3-加严
    current = 0

    while True:
        status_counts.append(status)
        # 生成训练数据
        defects = generate_training_data(n, p_good)
        accept_count = defects
        delta_n = n-accept_count

        # 计算拒收和接收的信度
```

```
    p_reject = stats.binomtest(n-accept_count, n, nominal_defect_rate,
        alternative='less').pvalue
    # 判断接收或拒收
    if p_reject >= 0.90:
        current = 1 # 拒收
    else:
        current = 0 # 接收
    recent_reject_counts.append(current)
    # print(current)
    total_batches += 1
    n_values.append(n) # 记录当前样本大小

    # 根据转移得分和拒收计数更新样本大小
    if status==2 and transfer_score >= 10:

        # 正常检验->放宽检验
        status = 1
    elif status==2 and sum(recent_reject_counts) >= 2:
        # 正常检验->加严检验
        status = 3
    elif status==3 and sum(recent_reject_counts) == 0 or sum(status_counts)==15:
        # 加严检验—>正常检验
        status = 2
    elif status==1 and current==1 or sum(status_counts)==5:
        # 放宽检验—>正常检验
        status = 2
        # 发生拒收时，转移得分加2
        transfer_score = 0
    elif sum(recent_reject_counts)<=2 and status==3 and current==0: # 接收数为2后加严检验通过
        transfer_score += 3
    elif sum(recent_reject_counts)<=2 and status==3 and current==1: # 接收数为2后加严检验通过
        transfer_score = 0
    elif sum(recent_reject_counts)>=3 and current==0:
        transfer_score += 2
    else:
        status = status

    # print(status)

    if status==1:
        n = n - k * delta_n
    elif status==2:
        n = n
    else:
        n = n + k * delta_n

    # 判断是否满足约束
```

```
        if total_batches >= 1000:
            break

    # 绘制折线图
    plt.figure(figsize=(10, 6)) # 设置图形大小
    plt.plot(n_values, marker='o', linestyle='-', markersize=8, linewidth=2, label='样本量') #
        设定线条和标记样式

    # 添加标题和标签
    plt.title('情形2迭代过程中样本量的动态调整图', fontsize=18, fontweight='bold') # 设置标题
    plt.xlabel('迭代次数', fontsize=14) # 设置X轴标签
    plt.ylabel('样本量 (N)', fontsize=14) # 设置Y轴标签

    # 添加网格
    plt.grid(True, linestyle='--', alpha=0.7) # 设置网格线样式和透明度

    # 添加图例
    plt.legend(fontsize=12)

    # 保存图像
    plt.savefig("n2.png", dpi=300, bbox_inches='tight')
    plt.close()


if __name__=="__main__":
    Threshold()
    # Calculation_n()
    # Dynamic_1()
    # Dynamic_2()
```

附录 B　问题二退火模拟算法-python 源程序

```
import random
import math
num_accessory = 100
bad_rate = [0.1,0.1,0.1]
cost_list_accessory = [2,3]
product_list = [6,3]
failed_product_list = [6,5]
sale = 56
buy_list = [4,18]

def profit(X):
    buy_accessory = num_accessory * (cost_list_accessory[0] + cost_list_accessory[1])
    check_1 = X[0] * num_accessory * bad_rate[0] + X[1] * num_accessory * bad_rate[1]
```

```
    left_1 = max(1-X[0]*bad_rate[0],1-X[1]*bad_rate[1]) * num_accessory #一阶段剩下的
    check_2 = product_list[0] * left_1# 装配费用
    check_3 = X[2] * left_1 * product_list[1] #检验成品
    no_product = left_1 - (1 - bad_rate[2]) * (left_1 -((1-X[0]) * num_accessory * bad_rate[0]
        + (1-X[1]) * num_accessory * bad_rate[1]))
    part = X[2] * X[3] * no_product * failed_product_list[1] #拆解费用
    exchange = (1 - X[2]) * (left_1 -((1-X[0]) * num_accessory * bad_rate[0] + (1-X[1]) *
        num_accessory * bad_rate[1]) ) * (failed_product_list[0] + buy_list[0] + buy_list[1] +
        product_list[0] )
    n_3 = left_1 * (1 - bad_rate[2] * X[3]) - (1- bad_rate[2]) * X[3] * ((1-X[0]) *
        num_accessory * bad_rate[0] + (1-X[1]) * num_accessory * bad_rate[1]) #流入市场的产品数量
    income = sale * n_3
    part_income = (cost_list_accessory[0] + cost_list_accessory[1]) * (1 - bad_rate[2]) * X[2]
        * X[3] * (left_1 -((1-X[0]) * num_accessory * bad_rate[0] + (1-X[1]) * num_accessory *
        bad_rate[1])) #合格拆解利润
    part_income_ = (cost_list_accessory[0] * (1-X[0]) * num_accessory * bad_rate[0] +
        cost_list_accessory[1] * (1-X[1]) * num_accessory * bad_rate[1])
    return income - (buy_accessory + check_1 + check_2 + check_3 + part + exchange) +
        part_income + part_income_

def SA():
    T = 1000
    T_min = 1
    alpha = 0.9
    X = [0,0,0,0]
    for j in range(4):
        X[j] = random.randint(0,1)
    while T > T_min:
        for i in range(100):
            E = profit(X)
            X_new = X.copy()
            X_new[random.randint(0,3)] = 1 - X_new[random.randint(0,3)]
            E_new = profit(X_new)
            if E_new > E:
                X = X_new
            else:
                if E - E_new <= 500:
                    if random.random() < math.exp((E - E_new) / T):
                        X = X_new
            print(X)
        print(profit(X))
        T = T * alpha
    return X

print(SA())
```

附录 C　问题三多目标动态规划-python 源程序

```
import matplotlib.pyplot as plt
import numpy as np
import random
from deap import base
from deap import creator
from deap import tools
import math
# Basic figures
acc_num = 100 #零件个数
type_acc = 2 #零件种类
step_num = 1 #工序数
group_num = math.ceil(math.pow(type_acc,1/step_num))
acc_bad_rate = np.random.randint(5,21,size=type_acc)/100
acc_buy_price = np.random.randint(1,21,size=type_acc)
acc_check_price = np.random.randint(1,11,size=type_acc)
group_num

def calculate_half_num(step_num,group_num):
    result = 0
    for i in range(1, step_num):
        result += group_num**i
    return result

half_num = calculate_half_num(step_num,group_num)
half_check_price = np.random.randint(1,6,size=half_num+1) #半成品检测价,
    是否需要增加数组长度方便计算
half_check_price
half_load_price = np.random.randint(7,16,size=half_num+1)
half_seperate_price = np.random.randint(7,10,size=half_num+1)
half_bad_rate = np.random.randint(5,21,size=half_num+1)/100
product_sale_price = np.random.randint(50,101)
product_check_price = np.random.randint(1,6)
product_bad_rate = np.random.randint(5,21)/100
product_seperate_price = np.random.randint(7,10)
product_load_price = np.random.randint(7,16)
product_change_price = np.random.randint(1,6)
# Fitness Function
creator.create("FitnessMulti",base.Fitness,weights=(0.02,0.8,-0.13))
creator.create("Individual",list,fitness=creator.FitnessMulti)

def calculate_individual_size():
    return type_acc + half_num*2 + 2
        #individual构成[零件检修type_acc,半成品检修half_num，半成品拆分half_num，成品检修1,成品拆分1]
```

```
IND_SIZE = calculate_individual_size()
toolbox = base.Toolbox()
toolbox.register("attr_binary",random.randint,0,1)
toolbox.register("individual",tools.initRepeat,creator.Individual,toolbox.attr_binary,n=IND_SIZE)
#check
ind1 = toolbox.individual()
print(ind1)
IND_SIZE
# Population
toolbox.register("population",tools.initRepeat,list,toolbox.individual)
pop = toolbox.population(n=5)
print(pop)
# Evaluation
def evaluate(individual):
# 零件到半成品
    acc2half_buy_cost = acc_num * sum(acc_buy_price)
    acc2half_check_cost = 0
    for i in range(type_acc):
        acc2half_check_cost += acc_check_price[i] * individual[i]
    acc2half_check_cost *= acc_num
    A = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段的不合格品
    B = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段的合格品
    M = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段拆解可以节省的费用
    half_num_1 = group_num ** (step_num - 1)
    for i in range(1,half_num_1 + 1):
        start_index = i * group_num - group_num
        end_index = i * group_num
        max_result = -math.inf
        min_result = math.inf
        for j in range(start_index,end_index):
            #有可能最后一个半成品合成零件数不够
            if j >= type_acc:
                break
            max_result = max(max_result,acc_bad_rate[j] * (1 - individual[j]))
            min_result = min(min_result,1 - acc_bad_rate[j] * individual[j])
        A[1][i] = max_result * acc_num
        B[1][i] = min_result * acc_num
    acc2half_load_cost = 0
    acc2half_seperate_cost = 0
    half1_check_cost = 0
    for i in range(1, half_num_1 + 1):
        acc2half_load_cost += half_load_price[i] * (A[1][i] + B[1][i])
        acc2half_seperate_cost += individual[type_acc + i - 1] * half_seperate_price[i] *
            individual[type_acc + half_num + i - 1] * (A[1][i] + B[1][i] * half_bad_rate[i])
        half1_check_cost += half_check_price[i] * individual[type_acc + half_num + i - 1] *
            (A[1][i] + B[1][i])
    # 零件到半成品，拆解能省下的费用
```

```
    acc2half_save_cost = 0
    for i in range(1, half_num_1 + 1):
        start_index = i * group_num - group_num
        end_index = i * group_num
        acc_sum = 0
        for j in range(start_index,end_index):
            #有可能最后一个半成品合成零件数不够
            if j >= type_acc:
                break
            acc_sum += acc_buy_price[j]
            bad_ones = min(B[1][i],acc_num * acc_bad_rate[j] * (1 - individual[j])) *
                acc_buy_price[j]
        M[1][i] = (B[1][i] * half_bad_rate[i] * acc_sum + bad_ones ) * individual[type_acc + i -
            1]
        acc2half_save_cost += M[1][i] * individual[type_acc + half_num + i - 1]
# 半成品到半成品,递推关系

    def calculate_past_half_num(i,group_num, step_num):
        result = 0
        for j in range(1,i):
            result += group_num ** (step_num - j)
        return result


    half_check_sum = 0
    half_separate_sum = 0
    half_load_sum = 0
    half_save_cost = 0

    for i in range(2,step_num):
        half_num_i = group_num ** (step_num - i)
        past_half_num = calculate_past_half_num(i - 1,group_num,step_num)
            #上一步已经计算的半成品数
        temp_check_sum = 0
        temp_load_sum = 0
        temp_separate_sum = 0
        temp_save_cost = 0
        for j in range(1,half_num_i + 1):
            start_index = j * group_num - group_num
            end_index = j * group_num
            max_result = -math.inf
            min_result = math.inf
            M_sum = 0
            for k in range(start_index,end_index):
                if k >= half_num_i * group_num:
                    break
                max_result = max(max_result, (A[i-1][k] + B[i-1][k] * half_bad_rate[past_half_num
```

```
                + k] ) * (1 - individual[type_acc + past_half_num + k - 1]))
            min_result = min(min_result, B[i-1][k] * (1 - half_bad_rate[past_half_num + k]))
            M_sum += M[i-1][k]
        A[i][j] = max_result
        B[i][j] = min_result
        temp_check_sum += (A[i][j] + B[i][j]) * half_check_price[past_half_num + half_num_i
            * group_num + j] * individual[type_acc + past_half_num + half_num_i * group_num
            + j - 1]
        temp_load_sum += (A[i][j] + B[i][j]) * half_load_price[past_half_num + half_num_i *
            group_num + j]
        temp_separate_sum += individual[type_acc + past_half_num + half_num_i * group_num +
            j - 1] * half_seperate_price[past_half_num + half_num_i * group_num + j] *
            individual[type_acc + past_half_num + half_num_i * group_num + half_num + j - 1]
            * (A[i][j] + B[i][j] * half_bad_rate[past_half_num + half_num_i * group_num + j])
        start_index = j * group_num ** i - group_num ** i
        end_index = j * group_num ** i
        tt_sum = 0
        for k in range(start_index,end_index):
            if k >= type_acc:
                break
            tt_sum += acc_buy_price[k]
        M[i][j] = (M_sum + tt_sum * B[i][j] * half_bad_rate[past_half_num + half_num_i *
            group_num + j] * individual[type_acc + past_half_num + half_num_i * group_num +
            j - 1])
        temp_save_cost += M[i][j] * individual[type_acc + past_half_num + half_num_i *
            group_num + half_num + j - 1]
    half_check_sum += temp_check_sum
    half_load_sum += temp_load_sum
    half_separate_sum += temp_separate_sum
    half_save_cost += temp_save_cost

# 半成品到成品
past_half_num = calculate_past_half_num(step_num - 1,group_num,step_num)
max_result = -math.inf
min_result = math.inf
M_sum = 0
for i in range(1, group_num + 1):
    max_result = max(max_result, (A[step_num - 1][i] + B[step_num - 1][i] *
        half_bad_rate[past_half_num + i]) * (1 - individual[type_acc + past_half_num + i -
        1]))
    min_result = min(min_result, B[step_num - 1][i] * (1 - half_bad_rate[past_half_num + i]))
    M_sum += M[step_num - 1][i]
A[step_num][1] = max_result
B[step_num][1] = min_result
product_check_cost = (A[step_num][1] + B[step_num][1]) * product_check_price *
    individual[-2]
product_load_cost = (A[step_num][1] + B[step_num][1]) * product_load_price
```

```
    product_separate_cost = individual[-2] * product_seperate_price * individual[-1] *
        (A[step_num][1] + B[step_num][1] * product_bad_rate)
    product_save_cost = (M_sum + acc2half_buy_cost * B[step_num][1] * product_bad_rate *
        individual[type_acc + past_half_num] ) * individual[-1]
    product_change_cost = product_change_price * individual[-2] * (A[step_num][1] +
        B[step_num][1] * product_bad_rate)
    product_sale = product_sale_price * (A[step_num][1] + B[step_num][1]) * (1 -
        product_bad_rate * individual[-1])
    print(A)
    print(M)


# 总费用
    total_cost = acc2half_buy_cost + acc2half_check_cost + acc2half_load_cost +
        acc2half_seperate_cost + half1_check_cost - acc2half_save_cost + half_check_sum +
        half_load_sum + half_separate_sum - half_save_cost + product_check_cost +
        product_load_cost + product_separate_cost - product_save_cost + product_change_cost
    profit = product_sale - total_cost

# 总检验次数
    total_check = acc_num * sum(individual[:type_acc]) + sum(individual[-2:])
    for i in range(1,step_num):
        half_num_i = group_num ** (step_num - i)
        for j in range(1,half_num_i + 1):
            step = calculate_past_half_num(i,group_num,step_num)
            total_check += (A[i][j] + B[i][j]) * individual[type_acc + step + j - 1] + (A[i][j]
                + B[i][j] * half_bad_rate[calculate_past_half_num(i,group_num,step_num) + j]) *
                individual[type_acc + step + half_num + j - 1]

# 成品合格总数
    total_product = B[step_num][1] * (1 - product_bad_rate)
    return total_product, profit, total_check



print(evaluate(ind1))
print(ind1.fitness.valid)
ind1
num_population = 10
N_GEN = 100
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selNSGA2)
toolbox.register("evaluate", evaluate)

pop = toolbox.population(n=num_population)
```

```
fitnesses = list(map(toolbox.evaluate, pop))
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

CXPB, MUTPB = 0.5, 0.2

for g in range(N_GEN):
    # 选择并克隆个体
    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, offspring))
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values
    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    if invalid_ind:
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

    # 将父代与子代合并，选择下一代
    pop = toolbox.select(offspring + pop, num_population)

    fits = [ind.fitness.values[0] for ind in pop]
    length = len(pop)
    mean = sum(fits) / length
    sum2 = sum(x*x for x in fits)
    std = abs(sum2 / length - mean**2)**0.5
    print("Generation %i: Min %s Max %s Avg %s Std %s" % (g, min(fits), max(fits), mean, std))
    pareto_front = tools.sortNondominated(pop, len(pop), first_front_only=True)[0]
    print("Pareto Front (Generation %i):" % g)
    for ind in pareto_front:
        print(ind, ind.fitness.values)
    best_individual = max(pop, key=lambda ind: ind.fitness.values)
    print("Best individual (Generation %i): %s, Fitness: %s" % (g, best_individual,
        best_individual.fitness.values))
```

附录 D　问题四遗传算法-python 源程序

```
import matplotlib.pyplot as plt
import numpy as np
import random
from deap import base
from deap import creator
from deap import tools
import math
from pyDOE import lhs
from scipy.stats import norm

# CDF of bad rate

def normal_dist_calculate(lower_bound, upper_bound):
    sigma = 5.10
    mu = 10
    normal_dist = norm(loc=mu, scale=sigma)
    prob = normal_dist.cdf(upper_bound) - normal_dist.cdf(lower_bound)
    return prob

prob = []

for i in range(21):
    prob.append(normal_dist_calculate(i,i+2))



def calculate_probability_bad_rate(x):
    return prob[math.floor(x)]


# Basic figures
acc_num = 100 #零件个数d
type_acc = 8 #零件种类
step_num = 2 #工序数
group_num = math.ceil(math.pow(type_acc,1/step_num))
# acc_bad_rate = np.random.randint(5,21,size=type_acc)/100
acc_buy_price = np.random.randint(1,21,size=type_acc)
acc_check_price = np.random.randint(1,11,size=type_acc)
group_num

def calculate_half_num(step_num,group_num):
    result = 0
    for i in range(1, step_num):
        result += group_num**i
    return result
```

```
half_num = calculate_half_num(step_num,group_num)
half_check_price = np.random.randint(1,6,size=half_num+1) #半成品检测价,
    是否需要增加数组长度方便计算
half_check_price
half_load_price = np.random.randint(7,16,size=half_num+1)
half_seperate_price = np.random.randint(7,10,size=half_num+1)
# half_bad_rate = np.random.randint(5,21,size=half_num+1)/100
product_sale_price = np.random.randint(50,101)
product_check_price = np.random.randint(1,6)
# product_bad_rate = np.random.randint(5,21)/100
product_seperate_price = np.random.randint(7,10)
product_load_price = np.random.randint(7,16)
product_change_price = np.random.randint(1,6)
bad_rate_num = half_num + type_acc + 1

# Fitness Function
creator.create("FitnessMulti",base.Fitness,weights=(0.02,0.8,-0.13))
creator.create("Individual",list,fitness=creator.FitnessMulti)

def calculate_individual_size():
    return type_acc + half_num*2 + 2
        #individual构成[零件检修type_acc,半成品检修half_num, 半成品拆分half_num, 成品检修1,成品拆分1]

IND_SIZE = calculate_individual_size()
toolbox = base.Toolbox()
toolbox.register("attr_binary",random.randint,0,1)
toolbox.register("individual",tools.initRepeat,creator.Individual,toolbox.attr_binary,n=IND_SIZE)

# Population
toolbox.register("population",tools.initRepeat,list,toolbox.individual)

# Evaluation
def evaluate(individual, new_acc_badrate, new_half_badrate, new_product_badrate):
    #需要修改里面涉及次品率的值
# 零件到半成品
    acc2half_buy_cost = acc_num * sum(acc_buy_price)
    acc2half_check_cost = 0
    for i in range(type_acc):
        acc2half_check_cost += acc_check_price[i] * individual[i]
    acc2half_check_cost *= acc_num
    A = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段的不合格品
    B = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段的合格品
    M = np.zeros((type_acc + 1, type_acc + 1)) #代表每个阶段拆解可以节省的费用
    half_num_1 = group_num ** (step_num - 1)
    for i in range(1,half_num_1 + 1):
        start_index = i * group_num - group_num
        end_index = i * group_num
```

```
        max_result = -math.inf
        min_result = math.inf
        for j in range(start_index,end_index):
            #有可能最后一个半成品合成零件数不够
            if j >= type_acc:
                break
            max_result = max(max_result,new_acc_badrate[j] * (1 - individual[j]))
            min_result = min(min_result,1 - new_acc_badrate[j] * individual[j])
        A[1][i] = max_result * acc_num
        B[1][i] = min_result * acc_num
    acc2half_load_cost = 0
    acc2half_seperate_cost = 0
    half1_check_cost = 0
    for i in range(1, half_num_1 + 1):
        acc2half_load_cost += half_load_price[i] * (A[1][i] + B[1][i])
        acc2half_seperate_cost += individual[type_acc + i - 1] * half_seperate_price[i] *
            individual[type_acc + half_num + i - 1] * (A[1][i] + B[1][i] * new_half_badrate[i])
        half1_check_cost += half_check_price[i] * individual[type_acc + half_num + i - 1] *
            (A[1][i] + B[1][i])
    # 零件到半成品，拆解能省下的费用
    acc2half_save_cost = 0
    for i in range(1, half_num_1 + 1):
        start_index = i * group_num - group_num
        end_index = i * group_num
        acc_sum = 0
        for j in range(start_index,end_index):
            #有可能最后一个半成品合成零件数不够
            if j >= type_acc:
                break
            acc_sum += acc_buy_price[j]
            bad_ones = min(B[1][i],acc_num * new_acc_badrate[j] * (1 - individual[j])) *
                acc_buy_price[j]
        M[1][i] = (B[1][i] * new_half_badrate[i] * acc_sum + bad_ones ) * individual[type_acc +
            i - 1]
        acc2half_save_cost += M[1][i] * individual[type_acc + half_num + i - 1]
# 半成品到半成品,递推关系

    def calculate_past_half_num(i,group_num, step_num):
        result = 0
        for j in range(1,i):
            result += group_num ** (step_num - j)
        return result


    half_check_sum = 0
    half_separate_sum = 0
    half_load_sum = 0
```

```
half_save_cost = 0

for i in range(2,step_num):
    half_num_i = group_num ** (step_num - i)
    past_half_num = calculate_past_half_num(i - 1,group_num,step_num)
        #上一步已经计算的半成品数
    temp_check_sum = 0
    temp_load_sum = 0
    temp_separate_sum = 0
    temp_save_cost = 0
    for j in range(1,half_num_i + 1):
        start_index = j * group_num - group_num
        end_index = j * group_num
        max_result = -math.inf
        min_result = math.inf
        M_sum = 0
        for k in range(start_index,end_index):
            if k >= half_num_i * group_num:
                break
            max_result = max(max_result, (A[i-1][k] + B[i-1][k] *
                new_half_badrate[past_half_num + k] ) * (1 - individual[type_acc +
                past_half_num + k - 1]))
            min_result = min(min_result, B[i-1][k] * (1 - new_half_badrate[past_half_num +
                k]))
            M_sum += M[i-1][k]
        A[i][j] = max_result
        B[i][j] = min_result
        temp_check_sum += (A[i][j] + B[i][j]) * half_check_price[past_half_num + half_num_i
            * group_num + j] * individual[type_acc + past_half_num + half_num_i * group_num
            + j - 1]
        temp_load_sum += (A[i][j] + B[i][j]) * half_load_price[past_half_num + half_num_i *
            group_num + j]
        temp_separate_sum += individual[type_acc + past_half_num + half_num_i * group_num +
            j - 1] * half_seperate_price[past_half_num + half_num_i * group_num + j] *
            individual[type_acc + past_half_num + half_num_i * group_num + half_num + j - 1]
            * (A[i][j] + B[i][j] * new_half_badrate[past_half_num + half_num_i * group_num +
            j])
        start_index = j * group_num ** i - group_num ** i
        end_index = j * group_num ** i
        tt_sum = 0
        for k in range(start_index,end_index):
            if k >= type_acc:
                break
            tt_sum += acc_buy_price[k]
        M[i][j] = (M_sum + tt_sum * B[i][j] * new_half_badrate[past_half_num + half_num_i *
            group_num + j] * individual[type_acc + past_half_num + half_num_i * group_num +
            j - 1])
```

```
            temp_save_cost += M[i][j] * individual[type_acc + past_half_num + half_num_i *
                group_num + half_num + j - 1]
        half_check_sum += temp_check_sum
        half_load_sum += temp_load_sum
        half_separate_sum += temp_separate_sum
        half_save_cost += temp_save_cost

# 半成品到成品
    past_half_num = calculate_past_half_num(step_num - 1,group_num,step_num)
    max_result = -math.inf
    min_result = math.inf
    M_sum = 0
    for i in range(1, group_num + 1):
        max_result = max(max_result, (A[step_num - 1][i] + B[step_num - 1][i] *
            new_half_badrate[past_half_num + i]) * (1 - individual[type_acc + past_half_num + i
            - 1]))
        min_result = min(min_result, B[step_num - 1][i] * (1 - new_half_badrate[past_half_num +
            i]))
        M_sum += M[step_num - 1][i]
    A[step_num][1] = max_result
    B[step_num][1] = min_result
    product_check_cost = (A[step_num][1] + B[step_num][1]) * product_check_price *
        individual[-2]
    product_load_cost = (A[step_num][1] + B[step_num][1]) * product_load_price
    product_separate_cost = individual[-2] * product_seperate_price * individual[-1] *
        (A[step_num][1] + B[step_num][1] * new_product_badrate)
    product_save_cost = (M_sum + acc2half_buy_cost * B[step_num][1] * new_product_badrate *
        individual[type_acc + past_half_num] ) * individual[-1]
    product_change_cost = product_change_price * individual[-2] * (A[step_num][1] +
        B[step_num][1] * new_product_badrate)
    product_sale = product_sale_price * (A[step_num][1] + B[step_num][1]) * (1 -
        new_product_badrate * individual[-1])


# 总费用
    total_cost = acc2half_buy_cost + acc2half_check_cost + acc2half_load_cost +
        acc2half_seperate_cost + half1_check_cost - acc2half_save_cost + half_check_sum +
        half_load_sum + half_separate_sum - half_save_cost + product_check_cost +
        product_load_cost + product_separate_cost - product_save_cost + product_change_cost
    profit = product_sale - total_cost

# 总检验次数
    total_check = acc_num * sum(individual[:type_acc]) + sum(individual[-2:])
    for i in range(1,step_num):
        half_num_i = group_num ** (step_num - i)
        for j in range(1,half_num_i + 1):
            step = calculate_past_half_num(i,group_num,step_num)
```

```
            total_check += (A[i][j] + B[i][j]) * individual[type_acc + step + j - 1] + (A[i][j]
                + B[i][j] * new_half_badrate[calculate_past_half_num(i,group_num,step_num) + j])
                * individual[type_acc + step + half_num + j - 1]

    # 成品合格总数
        total_product = B[step_num][1] * (1 - new_product_badrate)
        return total_product, profit, total_check



num_population = 100
N_GEN = 1000
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selNSGA2)
toolbox.register("evaluate", evaluate)

pop = toolbox.population(n=num_population)

#适应度计算
sample_num = 2

def generate_new_bad_rate(ind):
    #拉丁超立方采样得到新的次品率
    samples = lhs(bad_rate_num, samples=sample_num)
    param_ranges = [0,20]
    scaled_samples = samples.T * (param_ranges[1] - param_ranges[0]) + param_ranges[0]
    sample_prob_list = []
    fit_list = []
    for i in range(sample_num):
        new_acc_badrate = scaled_samples[:type_acc,i] / 100
        new_half_badrate = scaled_samples[type_acc:type_acc + half_num,i] / 100
        new_product_badrate = scaled_samples[-1,i] / 100
        temp_multi = 1
        for j in range(bad_rate_num):
            temp_multi *= calculate_probability_bad_rate(scaled_samples[j][i])
        # print(temp_multi)
        sample_prob_list.append(temp_multi)
        new_half_badrate_list = new_half_badrate.tolist()
        new_half_badrate_list.insert(0,0)
        temp_fit = toolbox.evaluate(ind, new_acc_badrate, new_half_badrate_list,
            new_product_badrate)
        fit_list.append(temp_fit)
    # 重新计算概率
    sum_pro = sum(sample_prob_list)
    fit = [0,0,0]
    for i in range(sample_num):
```

```
        sample_prob_list[i] = sample_prob_list[i] / sum_pro
        for j in range(3):
            fit[j] += sample_prob_list[i] * fit_list[i][j]
    # print(sample_prob_list)
    return fit[0],fit[1],fit[2]

CXPB, MUTPB = 0.5, 0.2

# 生成父代个体
for ind in pop:
    ind.fitness.values = generate_new_bad_rate(ind)


for g in range(N_GEN):
    # 选择并克隆个体
    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, offspring))
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values
    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    if invalid_ind:
        fitnesses = map(generate_new_bad_rate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
                ind.fitness.values = fit

    # 将父代与子代合并，选择下一代
    pop = toolbox.select(offspring + pop, num_population)

    fits = [ind.fitness.values[0] for ind in pop]
    length = len(pop)
    mean = sum(fits) / length
    sum2 = sum(x*x for x in fits)
    std = abs(sum2 / length - mean**2)**0.5
    print("Generation %i: Min %s Max %s Avg %s Std %s" % (g, min(fits), max(fits), mean, std))
    pareto_front = tools.sortNondominated(pop, len(pop), first_front_only=True)[0]
    print("Pareto Front (Generation %i):" % g)
    for ind in pareto_front:
        print(ind, ind.fitness.values)
    best_individual = max(pop, key=lambda ind: ind.fitness.values)
```

```
    print("Best individual (Generation %i): %s, Fitness: %s" % (g, best_individual,
        best_individual.fitness.values))
```