

多轮次多工序生产最优决策方案的研究

摘要

在电子产品的生产过程中,质量控制和生产效率直接影响企业的竞争力和盈利,建立有效的决策模型对于企业运营具有重要的现实意义。本文建立了针对多轮次多工序生产过程的成本-收益模型,运用 Wald 序贯概率比检验(SPRT)、深度优先搜索、贪心算法、遗传算法等方法构建多轮次多工序生产的最优决策模型。

对于问题一,本文采用**序贯抽样**方法,运用 Wald 序贯概率比检验(SPRT)确定样本量。根据给定的显著性水平 α ,在一定的第 II 类误差概率 β 和次品率的可接受误差范围 δ 下,建立两种情形的假设。首先考虑简单随机抽样的比率检测问题,可通过数学推导求解样品量;其次考虑通过序贯抽样,通过 SPRT 方法最小化样本量,将二者求解的样本量进行比较,确定抽样检测方案为序贯抽样,并根据两种情形分别计算得出最小样本量。

对于问题二,本文对于**多轮次单工序生产过程建立成本-收益模型**作为评估方法,基于**深度优先搜索(DFS)**求最优解。首先确定决策指标为投资产出比,决策目标为利润最大化,决策变量为是否检测零件/成品,是否进入拆解,建立目标函数。其次,推导得出成品修正次品率的公式,以及生产循环中零配件次品率的递推公式,建立多轮次生产过程的成本-收益模型。最后,基于深度优先搜索算法在决策变量的解空间中对目标函数求解最优解。以此模型求解出题中所给六种情况的最优决策方案以及相应指标结果。

对于问题三,本文在问题二的基础上,完善了在**多轮次多工序情况下的成本-收益模型**,并基于**贪心算法和遗传算法**求解。首先通过分析生产过程的简化子结构推导出所需参数在**每轮次内部的递推公式**,建立单轮次多工序的成本-收益模型。其次,构建**多轮生产中修正次品率的递推公式**,建立多次轮次的成本-收益模型。最后,通过遗传算法求解单次循环内的最优决策,通过贪心算法求解循环过程的最优决策,求解生产过程中的最优决策方案。以此模型求解表 2 情况,共进行**六轮循环**,投资产出比为**78%**。

对于问题四,本文通过**区间估计**,以**最大化最小投资产出比**为优化目标,基于问题二和问题三的决策模型进行改进。由于实际次品率位置,选用置信区间上界作为初始次品率,带入问题二和问题三的决策模型中,得到最小投资产出比最大的决策方案,即为抽样次品率背景下的最优决策方案。

关键词: 抽样检测, 决策模型, 深度优先搜索, 贪心算法, 遗传算法

一、问题重述

1.1 问题背景

在电子产品制造业中，检测与质量控制是保证电子产品性能和质量的关键。只有严格控制成品的质量，减少残次品流入市场，才能更好地增强企业在市场中的竞争力^[1]。

在生产过程中，企业需购买多种零配件，零配件的质量将影响半成品和成品的性能。为此，必须设计适当的抽样检测方案评估零配件质量。由于装配工艺或其他因素，即使零配件合格，仍可能装配出不合格的半成品或成品。同事，企业还需考虑市场反馈，对于消费者购买的次品，需要提供相应的调换服务，这将产生额外的成本。因此，为在利润最大化和质量保证之间取得平衡，需要构建适当的数学模型，为企业做出最优决策。

1.2 待解决问题

为使该企业在电子产品的生产过程中做出合理的决策，实现利润最大化，现需建立数学模型解决以下问题：

问题 1：已知标称值为 10%，设计一种在样本量最小化条件下的抽样检测方案，以判断是否接收零配件，并在题目给定的两种信度和假设条件下，应用此方案，给出具体结果。

问题 2：针对一道工序、两种零配件，已知两种零配件和成品的次品率，选择适当的决策依据和指标，为生产过程构建合理的决策模型。并对表 1 中的具体情形应用该决策模型，给出具体的决策方案及相应指标结果。

问题 3：该问是在问题 2 的基础上，将生产过程推广至 m 道工序、 n 种零配件，已知各零配件、半成品、成品的次品率，构造适当的决策模型。并对表 2 中的具体数值应用此模型，给出具体决策方案、决策依据及相应指标结果。

问题 4：依据问题 1 中的抽样检测方案，得出零配件、半成品、成品的次品率，并选择决策的依据和目标，构造合理的生产过程决策模型。并对问题 2、问题 3 中的具体情形应用该模型，给出具体的决策方案和指标结果。

二、问题分析

2.1 问题一的分析

在问题一中，要根据给定的显著性水平和决策方法，计算出最小的样本量，以确保在给定的置信度下，能够准确判断零配件的次品率是否超过标称值。

首先考虑**简单随机抽样**，这属于**比率检测**问题^[2]¹⁵⁴。在大样本场合下，依据大数定律下的近似正态分布，考虑两类错误发生的概率，可推导出简单随机抽样

的最小样本量的一般表达式。在给定第 II 类误差概率和次品率的可接受误差范围后，结合显著性水平和标称值的具体参数值，便可求出最小的样本量。

再考虑**序贯抽样**。由于该方法不预设样本量，可根据初步样本的结果调整后续策略，在样品次品率超过标称值时立即停止抽样，因此抽样次数明显小于简单随机抽样。通过似然比检验，计算每次抽样后该时刻的对数似然比，将其与决策临界值比较，直到超出决策临界时终止。终止时的样本量 n 便是想要求的最小的样本量。二者进行比较得到更优的最小样本量。

2.2 问题二的分析

问题二要求根据给定的参数选择最优的生产流程，涉及到两类决策变量的优化：是否检测零件/成品，是否拆解成品。是否检测决定了单轮生产流程的利润，而是否拆解成品决定了实现最大利润的轮数。由于题目只给出了零配件的初始次品率，且成品的次品率是基于零配件均为正品情况的次品率，因此需要根据决策变量计算出**成品的修正次品率**与下一轮生产开始前**零件的修正次品率(即实际次品数量/实际总数)**。以**多轮生产后的最大利润**为优化目标，通过**成本-收益分析**构建模型并利用**深度优先(DFS)**搜索最优解，以确定最优决策方案。

2.3 问题三的分析

问题三要求将问题二的情形扩展至包含 n 个零件， m 道工序的更复杂情况。因此，首先分析生产流程中的一个简化子结构，推导出修正次品率、产品数量、次品贡献率的**递推公式**，并将此过程推广至 m 道工序的生产流程，以建立单轮生产的成本-收益模型。通过**遗传算法**进行优化，以利润最大化为目标，求解得到单轮生产中的最优决策方案。同时，构建不同轮次之间零配件修正次品率的递推模型，并基于**贪心思想**求解多轮生产的最优决策方案。最终将表 2 中的特例带入模型求解。

2.4 问题四的分析

问题四的次品率并非真实次品率，而是基于抽样检测得到的统计量，是一个随机变量。如果直接将抽样次品率直接作为实际次品率应用到问题二和问题三的模型中，将会导致得到一个非最优的决策。由于实际次品率未知，应当考虑存在一种决策方案，使得在该种决策下，达到最大化最小投资产出比的目标。

对于抽样检测，除点估计外，在给定显著性水平的条件下，可以进行区间估计。通常情况下，在置信度足够高时，不含实际次品率的置信区间是很少见的。在置信区间内，置信上界对应的次品率组合会得到最小投资产出比。因此，选用置信上界作为修正的初始次品率。将修正的初始次品率带入问题二和问题三的模型中，便可得到抽样次品率背景下的最优决策方案。

三、模型假设

- (1) 假设零配件、半成品、成品在其各自的样本空间中独立且同分布；
- (2) 假设每次抽样均在大样本条件下进行；
- (3) 假设应用于生产的零配件的次品率与总体次品率相同；
- (4) 假设生产流程固定，即每一道工序的半成品数量确定，且零配件、各级半成品之间的对应关系明确且唯一；
- (5) 假设各级半成品和成品只能拆解为零配件；
- (6) 假设生产过程中各零件投入数量相等。

四、符号说明

符号	说明
p_i, r_i	第 <i>i</i> 个零件的购买单价、检测成本
a, r, p	成品的装配成本、检测成本、市场售价
l, d	不合格成品的调换损失、拆解费用
W_j	第 <i>j</i> 轮产生的利润
M_j	第 <i>j</i> 轮生产的成品数
Z_j	第 <i>j</i> 轮生产的正品数
S_{ij}	第 <i>j</i> 轮开始前，第 <i>i</i> 个零件的数量
n_{ij}	第 <i>j</i> 轮开始前，第 <i>i</i> 个零件的修正次品率
q_j	第 <i>j</i> 轮结束后的成品的修正次品率 (q_0 为题中成品的初始次品率)
$totalcost_j$	第 <i>j</i> 轮产生的成本

五、模型的建立与求解

5.1 问题一的模型的建立与求解

5.1.1 模型的建立

问题一中关注的次品率 p 是一种比率，可看做两点分布 $b(1, p)$ 中的参数。对于次品率 p 的假设检验可以采用比率推断的方法。

Step 1: 初始化参数和确定假设

p_0 : 可接受质量水平设定为标称值，即为 $p_0 = 0.1$

α : 显著性水平，情况 1 中 $\alpha_1 = 0.05$ ，情况 2 中 $\alpha_1 = 0.10$

写出原假设和备择假设：

$$(1) H_0: p > p_0 \text{ vs } H_1: p \leq p_0$$

$$(2) H_0: p \leq p_0 \text{ vs } H_2: p > p_0$$

Step 2: 确定检验统计量

假设企业抽样检测的样本量为 n ，并假定 X_1, X_2, \dots, X_n 是来自两点分布 $b(1, p)$ 的样本。令 $T = \sum_{i=1}^n X_i \sim b(n, p)$ ，是 p 的充分统计量，故理论上采用 T 作为检验统计量是一种好的选择。

由于企业零件的生产量较大，在本题中可认为样本量 n 较大，使用二项分布的正态近似，即当 $T \sim b(n, p)$ ， $E(T) = np$ ， $Var(T) = np(1-p)$ 时，由中心极限定理，当样本量 n 较大时，当 $p = p_0$ 时，有

$$u = \frac{T - np_0}{\sqrt{np_0(1-p_0)}} = \frac{\hat{p} - p_0}{\sqrt{p_0(1-p_0)/n}} \sim N(0,1)$$

其中， $\hat{p} = T/n$ ，代表抽样检测中不合格品的比率。

综上所述，我们采取为 u 检验统计量。

Step 3: 控制犯第 II 类错误概率以及概率差 δ 确定最小样本量

(1) 情形一

该情况中原假设与备择假设为 $H_0: p > p_0 \text{ vs } H_1: p \leq p_0$ 。在原假设 H_0 为真时，对给定的显著性水平 $\alpha_1 = 0.05$ ，其拒绝域为 $W = \{u \leq u_{\alpha_1}\}$ ，犯第一类错误的概率不超过 α_1 ；在原假设 H_0 为真时，若检验统计量落入接受域 $\bar{W} = \{u > u_{1-\alpha_1}\}$ ，

则犯第 II 类错误的发生概率 $\beta_1 = P_{p_1}(u > u_{1-\alpha_1})$ 。其中 p_1 为 H_1 中某个点，为确定起见，设 $p_1 = p_0 - \delta_1$ ($\delta_1 > 0$) 为 H_1 中某个点。利用标准正态分布的分位数，可解得最小样本数，

$$n = \left\lceil \frac{(u_{1-\alpha_1} \sqrt{p_0(1-p_0)} + u_{1-\beta_1} \sqrt{p_1(1-p_1)})^2}{(p_1 - p_0)^2} \right\rceil$$

(2)情形二

情形二中原假设与备择假设为 $H_0: p \leq p_0$ vs $H_2: p > p_0$ ，给定的显著性水平 $\alpha_2 = 0.10$ 。与情形一类似的，可解得最小样本数

$$n = \lceil \frac{(u_{1-\alpha_2}\sqrt{p_0(1-p_0)} + u_{1-\beta_2}\sqrt{p_2(1-p_2)})^2}{(p_2 - p_0)^2} \rceil$$

Step 4: 已知最小样本量 n 计算最大不合格品数 m

$$m = \lceil np_0 + u_{1-\alpha}\sqrt{np_0(1-p_0)} \rceil$$

5.1.2 Wald 序贯概率比检验 (SPRT)

Wald 序贯概率比检验(SPRT)是一种基于序贯抽样的统计方法，已知显著性水平和标称值的情况下，SPRT 允许在抽样过程中不断评估所抽取的数据，可以在样品次品率超过标称值时立即停止抽样，进而减少抽样次数，降低抽样成本。

Step 1:初始化参数和确定假设

p_0 : 可接受质量水平设定为标称值，即为 $p_0 = 0.1$

p_1 : 次品率上限，超过时拒收，设 $p_1 = 0.15$

α : 显著性水平，情况 1 中 $\alpha_1 = 0.05$ ，情况 2 中 $\alpha_1 = 0.10$

写出原假设和备择假设：

$$(1)H_0: p > p_0 \text{ vs } H_1: p \leq p_0$$

$$(2)H_0: p \leq p_0 \text{ vs } H_2: p > p_0$$

Step 2: 确定似然比函数和临界值

定义似然比函数为：

$$LRT(x) = \frac{P(x_1, x_2, \dots, x_n | H_1)}{P(x_1, x_2, \dots, x_n | H_0)} = \frac{p_1^x (1 - p_1)^{n-x}}{p_0^x (1 - p_0)^{n-x}}$$

其中， n 为当前的样本量， x 是观测到的次品数量，似然比函数 LRT 用于检验是否接受或拒绝 H_0 。

根据显著性水平 α 和检验功效 β ，假设 $\beta = 0.1$ ，确定两个临界值：

$$\text{上界: } B = \frac{1 - \beta}{\alpha}$$

$$\text{下界: } A = \frac{\beta}{1 - \alpha}$$

为简化实际计算，可取对数似然比函数，得到：

$$\ln L(x) = x \ln \left(\frac{p_1}{p_0} \right) + (n - x) \ln \left(\frac{1 - p_1}{1 - p_0} \right)$$

相应地，决策上下界变为：

$$\ln A = \ln \left(\frac{1 - \beta}{\alpha} \right)$$

$$\ln B = \ln \left(\frac{\beta}{1 - \alpha} \right)$$

Step 3: 抽样

从总体中随机抽取一个样品进行检测,记录当前的样本量 n 和不合格品数量 x , 并计算当前对数似然比。

Step 4: 决策

记录当前对数似然比 $\ln L(x)$, 比较 $\ln L(x)$ 和对数上下界。

(1)情形一

- 如果 $\ln L(x) \leq \ln A$, 拒绝 H_0
- 如果 $\ln L(x) \geq \ln B$, 接受 H_0
- 如果 $\ln A < \ln L(x) < \ln B$, 则继续抽样

(2)情形二

- 如果 $\ln L(x) \geq \ln B$, 拒绝 H_0
- 如果 $\ln L(x) \leq \ln A$, 接受 H_0
- 如果 $\ln A < \ln L(x) < \ln B$, 则继续抽样

Step 5: 重复 step3 至 step4, 直到做出最终决策, 得到尽量小的样本量 n

5.1.3 模型的求解

(1) 比率的检验

抽样检测方案需给出最小样本量 n 以及最大不合格品数 m 。假设情形一、二中的犯第 II 类错误概率均为 $\beta = 0.1$, 概率差 $\delta = 0.05$, 根据上述模型求解。

情形一抽样检测方案: 从这一批零件中随机抽取样本量为 $n = 176$ 的零配件进行检测, 若不合格品数量不超过 $m = 23$, 则接受这批零件

情形二抽样检测方案: 从这一批零件中随机抽取样本量为 $n = 362$ 的零配件进行检测, 若不合格品数量超过 $m = 46$, 则拒绝这批零件。

(2) SPRT 检验

利用 python 程序进行模型求解, 为保证结果的可靠性, 将 1000 次实验的所得到的样本量 n 的平均值作为最终结果。假设情形一、二中的犯第 II 类错误概率均为 $\beta = 0.1$, 根据上述模型求解, 结果如下:

情形一: 样本量为 $n = 188$;

情形二: 样本量为 $n = 167$

5.1.4 模型的对比

将简单随机抽样与 SPRT 算法对比, SPRT 能够明显减少抽样次数, 降低抽样检测成本, 且保证了检测功效, 所以选择 SPRT 抽样检测方案。

设定第 II 类错误概率均为 $\beta = 0.1$ 。

最优的抽样检测方案为: 从这一批零件中随机抽取一个零配件进行检测, 记录对数似然比, 若 $\ln L(x) \geq \ln B$, 拒绝这批零件, 若 $\ln L(x) \leq \ln A$, 接受这批零件。

(1)在情形一下, 抽样检测次数最终为 $n = 188$;

(2)在情形二下, 抽样检测次数最终为 $n = 167$ 。

5.2 问题二的模型的建立与求解

5.2.1 符号说明

表 1：问题二中的符号说明

符号	说明
x_{1j}, x_{2j}	第 j 轮是否检测零件 1 或 2; 若是, 取值为 1, 反之取值为 0
x_{3j}, x_{4j}	第 j 轮是否检测成品/是否拆解不合格成品; 若是, 取值为 1, 反之取值为 0

5.2.2 模型的建立

Step 1: 初始条件设置

- (1) S_{11}, S_{21} 是零件 1 和 2 的初始数量, 人为给定;
- (2) n_{11}, n_{21} 是零件 1 和 2 的初始次品率, 由题中表格给定;
- (3) q_0 是初始的成品次品率, 由题中表格给定;
- (4) 假设当 $j = 0$ (即第一轮开始前) 时, 设 $Z_0 = 0, M_0 = 0, totalcost_0 = S_{11} \cdot p_1 + S_{21} \cdot p_2$

Step 2: 构建决策指标, 确定决策依据

企业在生产中的目标是 $\{ROI\}$ (投资产出比) 的最大化, $ROI = \frac{\text{总利润}}{\text{初始成本}}$, 因此问题二的决策指标 $\{ROI\}$, 决策依据是 $\max \{ROI\}$ 。

在第 j 轮中, 第 j 轮的利润由本轮的正品销售额与生产成本作差得到, 其中正品销售额可表达为成品的市场售价与正品数的乘积。因此, 问题二中的目标函数表示为:

$$\max \left\{ \frac{\sum_{j=0}^t (p \cdot Z_j - totalcost_j)}{totalcost_0} \right\} \quad (1)$$

其中, t 表示该情况下的总轮次。

Step 3: 模型推导

第 j 轮中产生的成本 $totalcost_j$ 则由若干部分组成, 具体包括: (1) 零件 1 或 2 的检测费用 $r_1 \cdot S_{1j} \cdot x_{1j} + r_2 \cdot S_{2j} \cdot x_{2j}$; (2) 成品的装配费用 $M_j \cdot a$; (3) 成品的检测费用 $M_j \cdot x_{3j} \cdot r$; (4) 不合格成品的调换损失 $M_j \cdot (1 - x_{3j}) \cdot q_j \cdot l$; (5) 不合格成品的拆解费用 $M_{j-1} \cdot x_{4,j-1} \cdot q_{j-1} \cdot d$ 。整理得到:

$$totalcost_j = r_1 \cdot S_{1j} \cdot x_{1j} + r_2 \cdot S_{2j} \cdot x_{2j} + M_j(a + x_{3j} \cdot r + (1 - x_{3j})q_j \cdot l) + M_{j-1} \cdot x_{4,j-1} \cdot q_{j-1} \cdot d$$

成品的修正次品率 = 实际生产次品 / 实际生产总数。考虑三种情况: 当检测全

部零件时，成品的修正次品率= q_0 ；当零件全部不检测时，成品的修正次品率= $1 - \text{正品率} = 1 - (1 - q_0)(1 - n_{1j})(1 - n_{2j})$ ；当只检测一个零件时，成品的修正次品率= $1 - \text{正品率} = 1 - (1 - q_0)(1 - n_{1j})$ 或 $1 - (1 - q_0)(1 - n_{2j})$ 。

即，第 j 轮结束后的修正成品次品率为 q_j ，

$$q_j = 1 - \prod_{i=1}^2 (1 - n_{ij})^{1-x_{ij}} (1 - q_0)$$

第 j 轮中生成的正品数为 Z_j ，其中 q_0 为最初的成品次品率：

$$Z_j = \min \{S_{1j} \cdot (1 - x_{1j} \cdot n_{1j}), S_{2j} \cdot (1 - x_{2j} \cdot n_{2j})\} \times (1 - q_j)$$

第 j 轮开始前，第 i 个零件的数量为 S_{ij} ，后一轮的零件数与前一轮零件数相比减少了正品数和若检测该零件丢掉的次数：

$$S_{i(j+1)} = S_{ij} - Z_j - S_{ij} \cdot x_{ij} \cdot n_{ij} \quad (2)$$

第 j 轮开始前，第 i 个零件的修正次品率为 n_{ij} 。若第 i 个零件被检测过，不合格的零件被丢弃，修正次品率变为零；若第 i 个零件未被检测过，不合格零件数仍然等于初始数量，与本轮开始前的零件总数作比即可得到修正次品率：

$$n_{ij} = \frac{(1 - \sum_{k=1}^{j-1} x_{ik}) \cdot S_{i1} \cdot n_{i1}}{S_{ij}} \quad (3)$$

由上述式(2)-(3)即可完成一轮生产中成本-收益模型的构建。

Step 4: 生产的终止条件

当第 j 轮不再盈利或不再有成品生成时，循环终止，生产过程结束。

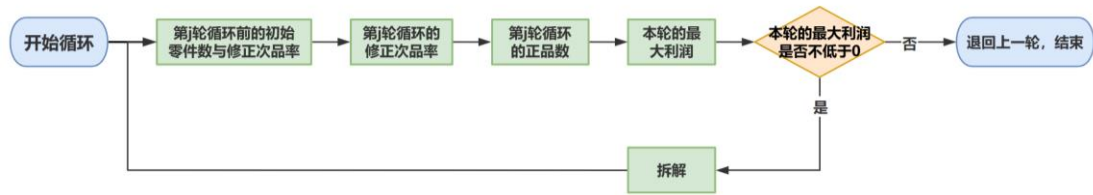


图 1：问题二模型的生产流程图

5. 2. 3 模型的求解

基于目标函数(1)以及循环条件(2)和(3)，假定两种零件等比例投入，即 $S_{11}:S_{21} = 1:1$ ，将每种情况下的零件 1 和 2 的初始次品率 n_{11}, n_{21} ，以及初始的成品次品率 q_0 带入模型。考虑到题目二的计算复杂度低，使用深度优先搜索 (DFS) 算法对目标函数求最优解。

DFS 算法伪代码如下：

```
DFS(deep)://制定搜索树的最大深度，本题目将 deep 设定为 5
    Q = empty_stack
    For index in all_test_case://将第一层的所有可能的检测操作入栈(共 8 种)
        Q.push((index,level=1))
    While(!Q.empty()):
        N = Q.pop()
        If(N.now_level>deep):
            Record_max_benefit()//判断是否应该更新全局解
            Continue
        Benefit = count_benefit(N.index)//计算本轮利润
        If(Benefit<=0):
            Record_max_benefit()//判断是否更新全局解
        Else:
            For index in all_test_case:
                Q.push((index,N.level+1))//更新栈
```

对 6 种情况下分别作出决策，具体求解结果如下：

表 2：决策结果汇总

情况与轮次		决策				依据
情况	轮次	检测零件 1	检测零件 2	检测成品	拆解不合格品	$\max \{ROI\}$
1	1	-	-	-	+	76%
	2	+	+	-	+	
	3	-	-	-	+	
	4	-	-	-	-	
2	1	+	+	-	+	43%
	2	-	-	-	+	
	3	-	-	-	+	
	4	-	-	-	+	
	5	-	-	-	-	
3	1	-	-	+	+	68%
	2	+	+	+	+	
	3	-	-	+	+	
	4	-	-	+	-	
4	1	+	+	+	+	53%
	2	-	-	+	+	
	3	-	-	+	+	
	4	-	-	+	+	
	5	-	-	+	-	
5	1	-	-	+	+	39%
	2	+	+	-	+	
	3	-	-	-	+	
	4	-	-	-	-	
6	1	-	-	-	-	84%

注：+ 代表检测或拆解，-则不进行任何操作。

5.3 问题三的模型的建立与求解

问题三在问题二的基础上，将生产过程推广至 m 道工序、 n 种零配件。

5.3.1 符号说明

表 3：问题三中的符号说明

符号	说明
N_{tjk}	第 j 轮循环的第 k 道工序中生产的第 t 种生产材料数量
H_{tjk}	第 j 轮循环的第 k 道工序中生产的第 t 种生产材料合格数
g_{tjk}	第 j 轮循环的第 k 道工序生产结束后的第 t 种生产材料修正次品率
x_{tjk}	在第 j 轮循环的第 k 道工序中，是否检测第 t 种生产材料；若是，取值为 1，反之取值为 0
y_{tjk}	在第 j 轮循环的第 k 道工序中，是否拆解第 t 种不合格生产材料；若是，取值为 1，反之取值为 0
c_{tjk}	第 j 轮循环中第 k 道工序生产的第 t 种生产材料对第 $k+1$ 道工序的生产材料的修正次品率的贡献率
b_k	在第 k 道工序结束后的生产材料种类数

此处生产材料包括零配件、半成品、成品。因此，有 $N_{1jm} = M_j$, $H_{1jm} = Z_j$, $g_{1jm} = q_j$, $g_{10m} = q_0$, $N_{ij0} = S_{ij}$, $g_{ij0} = n_{ij}$ 。

5.3.2 单次生产循环过程中的递推模型

设在第 j 轮循环的第 k 道工序中生产的第 t 种生产材料，由第 $k-1$ 道工序中的 a 种次级生产材料构成，如图 2 所示。

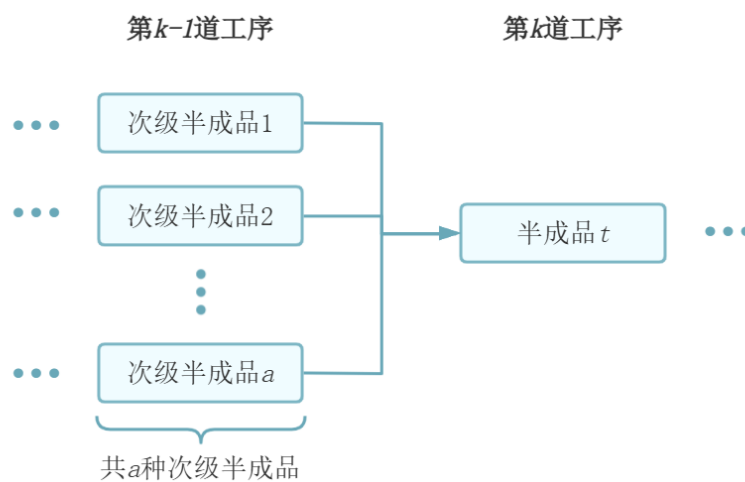


图 2 部分生产过程

(1) 修正次品率递推公式

Step 1: 确定初始条件

已知第 j 轮循环初始 n 种零配件的次品率 n_{1j} 、 $n_{2j} \cdots n_{nj}$ ，即在第 j 轮循环的第 0 道工序中的 g_{1j0} 、 $g_{2j0} \cdots g_{nj0}$ 。

Step 2:建立递推公式

已知第 j 轮循环第 $k-1$ 道工序中的 a 种次级半成品的修正次品率和第 t 种半成品的次品率 g_{t0k} ，计算第 k 道工序中第 t 种半成品的修正次品率 g_{tjk} 。

若对某个次级半成品进行检验($x_{tj(k-1)} = 1$)，对半成品 t 的修正次品率的贡献为 0；若不进行检测($x_{tj(k-1)} = 0$)，对半成品 t 的修正次品率的贡献为 $g_{tj(k-1)}$ 。基于以上逻辑，半成品 t 的修正次品率 g_{tjk} ：

$$g_{tjk} = 1 - (1 - g_{t0k}) \times \prod_{t=1}^a (1 - g_{tj(k-1)})^{1-x_{tj(k-1)}}$$

Step 3:验证公式

以问题 2 为例，成品的修正次品率符合递推公式结果，因此递推公式正确。

(2) 生产材料数量递推公式

Step 1:确定初始条件

已知第 j 轮循环初始 n 种零配件的数量 S_{1j} 、 $S_{2j} \cdots S_{nj}$ ，即在第 j 轮循环的第 0 道工序中的 N_{1j0} 、 $N_{2j0} \cdots N_{nj0}$ 。

Step 2:建立递推公式

已知第 j 轮循环第 $k-1$ 道工序中的 a 种次级半成品的数量和第 t 种半成品的次品率 g_{t0k} ，计算第 k 道工序中第 t 种半成品的数量 N_{tjk} 。

若对某个次级半成品进行检验($x_{tj(k-1)} = 1$)，进入装配过程的数量不变为 $N_{tj(k-1)}$ ；若不进行检测($x_{tj(k-1)} = 0$)，进入装配过程的数量为 $N_{tj(k-1)} \times (1 - g_{tj(k-1)})$ 。基于以上逻辑，半成品 t 的数量 N_{tjk} ：

$$N_{tjk} = \min (N_{1j(k-1)} \times (1 - x_{1j(k-1)} \times g_{1j(k-1)}), \cdots N_{nj(k-1)} \times (1 - x_{nj(k-1)} \times g_{nj(k-1)}))$$

Step 3:验证公式

以问题 2 为例，成品的数量符合递推公式结果，因此递推公式正确。

(3) 修正次品率贡献率递推公式

Step 1:确定初始条件

已知第 j 轮循环初始 n 种零配件的次品率 n_{1j} 、 $n_{2j} \cdots n_{nj}$ ，可以通过计算得到 c_{1j0} 、 $c_{2j0} \cdots c_{nj0}$ 。

Step 2:建立递推公式

已知第 j 轮循环第 $k-1$ 道工序中的 a 种次级半成品的修正次品率和第 t 种半成品的次品率 g_{t0k} ，计算第 $k-1$ 道工序中某个次级半成品对第 k 道工序的第 t 种半成品修正次品率的贡献率 $c_{tj(k-1)}$ 。

若对某个次级半成品进行检验($x_{tj(k-1)} = 1$)，对半成品 t 的修正次品率的贡

献为 0；若不进行检测($x_{tj(k-1)} = 0$)，对半成品 t 的修正次品率的贡献为 $g_{tj(k-1)}$ 。基于以上逻辑，对修正次品率的贡献率 $c_{tj(k-1)}$ ：

$$c_{tj(k-1)} = (1 - x_{1j(k-1)}) \times \frac{g_{tj(k-1)}}{1 - (1 - g_{t0k}) \prod_{i=1}^a (1 - g_{tj(k-1)})}$$

Step 3:验证公式

以问题 2 为例，2 种零配件对成品修正次品率的贡献率符合递推公式结果，因此修正次品率贡献率递推公式正确。

5.3.3 模型的建立

Step 1: 循环的初始条件设置

- (1) S_{i1} 是第 i 种零件的初始数量，人为给定；
- (2) n_{i1} 是第 i 种零件的初始次品率，由题中表格给定；
- (3) g_{t0k} 是初始的各级半成品和成品次品率，由题中表格给定；
- (4)假设当 $j = 0$ 时，设 $Z_0 = 0$ ， $totalcost_0 = \sum_{i=1}^n S_{i1}p_i$ 。

Step 2: 构建决策指标，确定决策依据

问题 3 是在问题 2 基础上的推广，因此沿用问题二的决策指标 ROI，决策依据是 $\max \{ROI\}$ 。目标函数表示为：

$$\max \left\{ \frac{\sum_{j=0}^t (p \cdot Z_j - totalcost_j)}{totalcost_0} \right\} \quad (4)$$

其中， t 表示该情况下的总循环次数。

Step 3:模型推导

基于单次生产循环中的递推公式，可得单次循环内的任意 N_{tjk} 、 g_{tjk} 、 $c_{tj(k-1)}$ 、 H_{tjk} 。

第 j 轮中产生的成本 $totalcost_j$ 则由若干部分组成，具体包括：（1）零配件的检测费用；（2）各级半成品和成品的装配费用；（3）各级半成品和成品的检测费用；（4）不合格成品的调换损失；（5）不合格的各级半成品和成品的拆解费用。整理得到：

$$\begin{aligned} totalcost_j = & \sum_{i=1}^n S_{ij}x_{ij0} d_{i0} \\ & + \sum_{k=1}^{m-1} \sum_{t=1}^{b_k} N_{tjk} (a_{tk} + x_{tjk} \cdot r_{tk} + (1 - x_{tjk})g_{tjk} \cdot l_{tk} + y_{tjk} \\ & \cdot g_{tjk} \cdot d_{tk}) + y_{1(j-1)m} \cdot q_{j-1} \cdot M_j \cdot d_{tk} \end{aligned}$$

第 j 轮循环开始前，第 i 个零件的数量为 S_{ij} ，后一轮的零件数与前一轮零件数相比减少了正品数和每次检测中该零件丢掉的次数：

$$S_{i(j+1)} = S_{ij} - Z_j - \sum_{k=1}^{m-1} N_{tjk} g_{tjk} x_{tjk} (1 - y_{tjk}) - M_j q_j (1 - y_{1jm}) \quad (5)$$

第 j 轮循环开始前，第 i 个零件的修正次品率为 n_{ij} 。若第 i 个零件被检测过，则不合格的零件被丢弃，修正次品率变为零；若第 i 个零件未被检测过，会在其组成的半成品的决策中被丢弃一部分，可以得到修正次品率：

$$n_{ij} = \frac{\sum_{k=1}^{m-1} (x_{tjk} y_{tjk} N_{tjk} g_{tjk} \prod_{h=1}^k c_{tj(k-1)}) + y_{1jm} M_j q_j \prod_{k=1}^m c_{tj(k-1)}}{S_{ij}} \quad (6)$$

由上述式(5)-(6)即可完成循环内部的构建。

Step 4: 循环的终止条件

当第 j 轮不再盈利或不再有成品生成时，循环终止，生产过程结束。

5.3.4 模型的求解

基于目标函数(4)以及循环条件(5)和(6)，假定 n 种零件的初始数量配比为 1:1:⋯:1，将表 2 中所给数据带入上述模型，在 2 道工序、8 个零配件的情况下，可以得到如下循环条件：

$$S_{i(j+1)} = S_{ij} - Z_j - x_{tj} (1 - y_{tj}) N_{tj} g_{tj} - (1 - y_{1j2}) M_j q_j$$

$$n_{ij} = \frac{x_{tj} y_{tj} N_{tj} g_{tj} \prod_{k=1}^2 c_{tj(k-1)} + y_{1j2} M_j q_j n_{i(j-1)}}{S_{ij}}$$

基于贪心算法和遗传算法，提出了一种混合优化策略，用于解决问题 3 中多次循环的复杂优化问题。在每次循环内部，利用遗传算法的全局搜索能力，得到单次循环中的最优决策。在循环之间，应用贪心算法以选择局部最优解，并通过贪心选择的逐步优化提高解的整体质量。通过这种结合优化策略，遗传算法在每次循环中提供了最优解，而贪心算法确保在多次循环的过程中快速收敛到较优解，从而兼顾全局搜索和局部优化的优势。

遗传算法建立：

Step 1: 确定优化目标

在单次循环内部，目标是使目标函数 ROI 最大化：

$$\theta = \arg \max_{\theta \in \omega} ROI$$

其中， ω 为可行解空间。

Step 2: 初始化

将优化问题的解惊醒编码，可表示为向量 $\theta = (x_{1j0}, \dots, x_{1jm}, y_{1j0}, \dots, y_{1jm})$ 。

生成初始种群 $P(0)$ ，种群大小为 100，基因数量为 12，变异率为 0.01。

Step 3: 循环迭代

使用轮盘赌选择策略进行选择, 设置最大迭代次数为 50, 对于 $t = 0, 1, 2, \dots, 50$, 执行选择、交叉、变异和更新, 生成新种群 $P(t + 1)$

Step 4: 目标函数最大化

根据目标函数 ROI 评估个体优劣, 选择利润率最高的个体作为最优解, 输出算法搜索结果。

Step 5: 输出结果

当超过最大迭代次数或者得到全局最优解时结束循环, 输出算法搜索结果。

贪心算法建立:

根据图 3, 在前三次生产循环中, 每轮循环的利润呈现显著的梯度下降趋势, 因此, 在多轮循环中, 基于贪心算法求得的局部最优解与全局最优解具有较高的一致性。

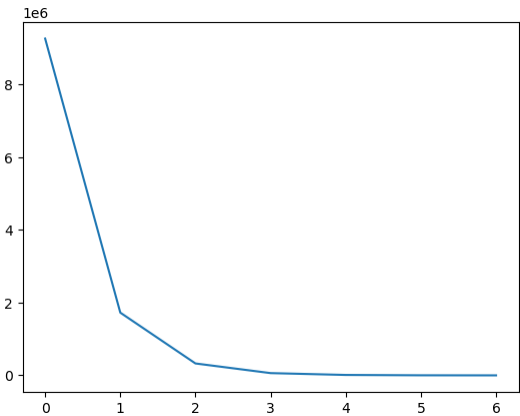


图 3 每轮生产利润折线图

使用 Python 编写贪心算法和遗传算法对上述模型求解, 即可得到表 2 中情形的具体决策方案: 共进行六轮循环, $\max\{ROI\} = 78\%$ 。

表 4 决策过程

轮次	检测零件	检测半成品	拆解不合格半成品	检测成品	拆解成品不合格品
1	检测 8 种	检测 3 种	拆解 3 种	不检测	拆解
2-5	不检测	检测 3 种	拆解 3 种	不检测	拆解
6	不检测	检测 3 种	不拆解	不检测	不拆解

5.4 问题四的模型的建立与求解

5.4.1 模型的建立

Step 1：分析由抽样检测获得的次品率

采用简单随机抽样的方法获得次品率。设样本量为 n ， X_1, X_2, \dots, X_n 独立同分布且是来自于两点分布 $b(1, q)$ 的一个样本，其中 q 为零件（半成品、成品）的真实次品率， $0 < q < 1$ 。总体均值 $E(X) = q$ ，总体方差存在且 $Var(X) = q(1 - q)$ ，总体标准差 $\sigma(X) = \sqrt{Var(X)}$ 。

根据简单随机抽样的结果，由抽样检测获得的次品率 $\hat{q} = \frac{\sum_{i=1}^n X_i}{n}$ 。样本均值 $\bar{X} = \frac{\sum_{i=1}^n X_i}{n}$ ， $\hat{q} = \bar{X}$ 。计算出样本标准差 S ，

$$S = \sqrt{\frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1}} = \sqrt{\frac{\sum_{i=1}^n X_i^2 - n\bar{X}^2}{n-1}} = \sqrt{\frac{n\hat{q}(1-\hat{q})}{n-1}}$$

假定随机抽样获得的样本是大样本。在独立同分布场合下，该分布的总体均值与方差均存在^{[2]127}。根据中心极限定理，样本均值 \bar{X} 有渐近正态分布，即

$$\bar{X} \sim N(q, \frac{q(1-q)}{n})$$

由此可以得到总体均值 q 的近似 $1 - \alpha$ 等尾置信区间： $q \pm u_{1-\alpha/2} \cdot \sigma(X)/\sqrt{n}$ ，

其中 $u_{1-\alpha/2}$ 是标准正态分布的 $1 - \alpha/2$ 分位数。由于其中的总体标准差 $\sigma(X)$ 未知，用 $\sigma(X)$ 的相合估计样本标准差 S 替代，即：

$$\max_{q=\hat{q}^U} \{ROI\}$$

对此置信区间的解释是：得到的置信区间能盖住未知参数 q 的概率至少为 $1 - \alpha$ 。这个区间会随着样本观测值的变化而变化，但 100 次运用这个区间估计，约有 $100(1 - \alpha)$ 个区间能盖住 q ， 100α 个区间不含 q 。当选取较高的置信水平时，不含 q 的区间的个数是非常少的。因此，在问题四后续的计算中，用近似 $1 - \alpha$ 等尾置信区间来估计 q 是十分合理的。并记置信上界为 \hat{q}^U ，置信下界为 \hat{q}^L 。

Step 2：确定决策指标与决策依据

根据上一步的分析，各个零件（半成品、成品）的次品率都可以得到相应的置信区间 $[\hat{q}^L, \hat{q}^U]$ 。当各个零件、半成品以及成品的次品率都在 $[\hat{q}^L, \hat{q}^U]$ 中变动时，就会产生不同的次品率的组合。对于一个给定的检测方案，不同的次品率的组合会对应不同的 ROI （投资产出比）。将所有 ROI 的取值进行比较，最小值应当是在各个零件（半成品、成品）的次品率均取其置信上界 \hat{q}^U 时得到的。

在问题四中，仍然沿用问题二、三种的 ROI 作为决策指标，以 $\max\{ROI\}$ 作为决策依据。由于次品率 q 的不确定性，不同次品率组合所对应的 $\max\{ROI\}$ 有所不同。为了使投资产出比尽可能少的受到 q 的不确定性的影响，应当最大化最小的 ROI ，也即决策依据依据为：

$$\max_{q=\hat{q}^U} \{ROI\}$$

Step 3: 将各个零件（半成品、成品）的置信上界 \hat{q}^U 作为其初始次品率以此带入问题二及问题三的模型中，重新求解上述两个问题。

5.4.2 模型的求解

将各零件（半成品、成品）的置信上界 \hat{q}^U 带入问题二及问题三的模型，得到具体决策方案。

表 5 问题二的决策

情况与轮次		决策				依据
情况	轮次	检测零件 1	检测零件 2	检测成品	拆解不合格品	$\max \{ROI\}$
1	1	+	+	-	+	69%
	2	-	-	-	+	
	3	-	-	-	+	
	4	-	-	-	-	
2	1	+	+	-	+	37%
	2	-	-	-	+	
	3	-	-	-	+	
	4	-	-	-	+	
	5	-	-	-	-	
3	1	-	-	+	+	61%
	2	+	+	+	+	
	3	-	-	+	+	
	4	-	-	+	+	
	5	-	-	+	-	
4	1	+	+	+	+	48%
	2	-	-	+	+	
	3	-	-	+	+	
	4	-	-	+	+	
	5	-	-	+	-	
5	1	-	-	+	+	30%
	2	+	+	-	+	
	3	-	-	-	+	
	4	-	-	-	+	
	5	-	-	-	-	
6	1	-	-	-	-	75%

注：+ 代表检测或拆解，-则不进行任何操作。

表 6 问题三的决策

轮次	检测零件	检测半成品	拆解不合格半成品	检测成品	拆解成品不合格品
1	检测 8 种	检测 3 种	拆解 3 种	不检测	拆解
2-5	不检测	检测 3 种	拆解 3 种	不检测	拆解
6	不检测	检测 3 种	不拆解	不检测	不拆解

分析上述结果发现，当基于抽样检测给出次品率时：

(1)在问题二的六种情况中，情况 2，4，6 的决策方法不发生改变，问题三的决策方法也不发生改变。

(2)投资产出比虽略有下降，但总体而言受到的影响较小。

因此可以表明，这种次品率的修正方法时较为合适的。

六、模型的分析与检验

6.1.1 问题一的灵敏度分析

表 7 灵敏度分析

β	情形 1 样本量	情形 2 样本量	δ	情形 1 样本量	情形 2 样本量
0.1	188	167	0.05	188	167
0.05	255	227	0.04	284	250
0.08	201	190	0.03	481	420
0.12	175	155	0.02	985	902
0.15	152	133	0.01	3870	3435

灵敏度分析显示，不同的第 II 类错误发生概率 β 和次品率的最大允许误差 δ 都会对最小样本量产生影响。 δ 越小所需样本量越大，这是因为 δ 越小越难辨别。

6.1.2 问题二、三的灵敏度分析

零件和成品的次品率，检测成本，成品的转配成本，拆解成本，调换成本都会影响策略的选择。对于这些变量问题二的各种情况已经给出例子。在灵敏度分析中，我们对于零件次品率相同与零件次品率不同的情况，测试了零件配比 1:1 ,1:1.25 ,1:1.5 ,1:2 ,1.25:1 ,1.5:1 ,2:1 的情况下最优策略与最优投入产出比的变化。

我们采用问题 2 中情况 1 的参数作为次品率相同情况下的初始条件，采用问题 2 中情况 5 的参数作为次品率不同情况下的初始条件，得到如下表格：

表 8 灵敏度分析

零件配比 (零件 1: 零件 2)	$\max \{ROI\}$ (次品率相同)	$\max \{ROI\}$ (次品率不同)
1:1	76%	39%
1:1.25	47%	44%
1.25:1	68%	29%
1:1.5	19%	22%
1.5:1	58%	23%
1.2	-12%	-6%
2:1	42%	14%

我们发现，对于零件次品率相同的情况，在相同条件下零件数量比不同会减少利润率，购买单价更贵的零件增多对最优投资产出比的影响更大。对于零件次品率不同的情况，适量增加次品率高的零件的比例使最优投入产出比增大，但是增加过多仍然会造成最优投资产出比的下降。

七、模型的评价、改进与推广

7.1 模型的优点

(1)本模型的推导较为严格的遵循概率论、数理统计等统计学知识,利用点估计、区间估计、假设检验等理论对实际生产过程进行抽象,从而指导实践过程。

(2)在决策产品是否符合标称值时,采用序贯抽样算法对简单随机抽样进行优化。这一优化算法充分利用了抽样过程中实时更新的信息,通常比简单随机抽样需要更少的样本,因此节省了时间和成本。

(3)模型中引入修正次品率的概念,考虑了先前决策对后续决策产生的影响,增强了模型的普遍适用性。

(4)求解最优化目标的过程中,本模型采用了多种算法:深度优先算法可以探索所有的路径,以达到全局最优的目的;贪心算法简单高效且复杂度低;遗传算法则适用于各种类型的优化问题。

7.2 模型的缺点

(1)与简单随机抽样相比,序贯抽样历经多次迭代,需要程序辅助制定抽样计划,制定过程较为复杂;另外,在特定情况下序贯抽样可能会无限循环,导致无休止的抽样。

(2)求解最优化目标的过程中,贪心算法与遗传算法无法避免陷入局部最优,而深度优先搜索在深度很大时可能导致栈溢出,且效率较低。

(3)当次品率来自抽样检测时,给定的置信区间不一定覆盖真实的次品率,因此生产过程中的决策未必真正纳入了真实次品率的信息;并且决策过程中基于若干假定,这在生产实践中很难实现。

7.3 模型的推广

本模型基于次品率的迭代来优化生产决策,适用于对产品质量要求较高的行业,如电子产品制造、医疗器械生产、航空航天等行业。本模型能够在保证产品质量的基础上优化生产成本,帮助企业在检测、装配和拆解等环节中做出更加精确的决策。



八、参考文献

- [1] 秦媛,电子产品检测及质量控制研究[J],电子世界,(12):48,2017
- [2] 吕晓玲,黄丹阳,数据科学概率基础,北京:中国人民大学出版社,2021,124-157

附录

附录 1

支撑材料的文件列表

-  problem 1.ipynb
-  problem 2-4.ipynb

附录 2

该代码是 python 语言编写的，用于问题一中序贯抽样的迭代

```
import numpy as np

def sequential_probability_ratio_test(p0, p1, alpha, beta, max_samples=50000):

    # 上下界
    A = beta / (1 - alpha)
    B = (1 - beta) / alpha

    # 统计变量初始化
    n = 0 # 抽样次数
    log_LRT = 0 # 似然比的对数

    while n < max_samples:
        # 随机生成次品或正品（假设真实次品率为 p0）
        part = np.random.binomial(1, p0) # 1 表示次品，0 表示正品

        # 更新似然比的对数
        if part == 1: # 发现次品
            log_LRT += np.log(p1 / p0)
        else: # 发现正品
            log_LRT += np.log((1 - p1) / (1 - p0))

        # 增加抽样次数
        n += 1

    # 序贯检验决策
    if log_LRT >= np.log(B): # 拒绝 H0
        return n, "拒绝 H0（次品率较高）"
    elif log_LRT <= np.log(A): # 接受 H0
        return n, "接受 H0（次品率符合标准）"
```

```

# 如果到达最大样本数
return n, "未能做出明确决策，达到最大样本数"

def run_multiple_trials(trials, p0, p1, alpha, beta):

    total_samples = 0

    for _ in range(trials):
        n_samples, _ = sequential_probability_ratio_test(p0, p1, alpha, beta)
        total_samples += n_samples

    average_samples = total_samples / trials
    return average_samples

# 参数设置
p0 = 0.1 # 假设的次品率
p1 = 0.15 # 备择假设次品率
alpha = 0.05 # 显著性水平
beta = 0.1 # 检验功效
trials = 1000 # 运行 1000 次试验

# 运行 1000 次序贯抽样实验，计算平均抽样次数
average_samples = run_multiple_trials(trials, p0, p1, alpha, beta)
print(f'1000 次试验的平均抽样次数: {average_samples:.2f}')

```

```

import numpy as np

def sequential_probability_ratio_test(p0, p1, alpha, beta, max_samples=50000):

    # 上下界
    A = beta / (1 - alpha)
    B = (1 - beta) / alpha

    # 统计变量初始化
    n = 0 # 抽样次数
    log_LRT = 0 # 似然比的对数

    while n < max_samples:
        # 随机生成次品或正品（假设真实次品率为 p0）
        part = np.random.binomial(1, p0) # 1 表示次品，0 表示正品

```

```

    # 更新似然比的对数
    if part == 1: # 发现次品
        log_LRT += np.log(p1 / p0)
    else: # 发现正品
        log_LRT += np.log((1 - p1) / (1 - p0))

    # 增加抽样次数
    n += 1
    log_LRT <= np.log(A)
    # 序贯检验决策
    if log_LRT <= np.log(A): # 拒绝 H0
        return n, "拒绝 H0 (次品率较高)"
    elif log_LRT >= np.log(B): # 接受 H0
        return n, "接受 H0 (次品率符合标准)"

    # 如果到达最大样本数
    return n, "未能做出明确决策, 达到最大样本数"

def run_multiple_trials(trials, p0, p1, alpha, beta):

    total_samples = 0

    for _ in range(trials):
        n_samples, _ = sequential_probability_ratio_test(p0, p1, alpha, beta)
        total_samples += n_samples

    average_samples = total_samples / trials
    return average_samples

# 参数设置
p0 = 0.1 # 假设的次品率
p1 = 0.15 # 备择假设次品率
alpha = 0.1 # 显著性水平
beta = 0.1 # 检验功效
trials = 1000 # 运行 1000 次试验

# 运行 1000 次序贯抽样实验, 计算平均抽样次数
average_samples = run_multiple_trials(trials, p0, p1, alpha, beta)
print(f'1000 次试验的平均抽样次数: {average_samples:.2f}')

```

附录 3

该代码是 python 语言编写的, 用于问题二、三、四中获得最佳策略

```

import numpy as np
from itertools import product
import copy

# 效用函数示例，这里我们使用一个简单的逻辑函数作为示例
def utility_function(bitstring):
    # 这里是一个示例效用函数，实际应用中需要替换为具体问题的效用函数
    return np.sum(bitstring)

# 遗传算法参数
POPULATION_SIZE = 100 # 种群大小
GENES = 12 # 基因数量（即 0,1 串的长度）
MUTATION_RATE = 0.01 # 变异率
GENERATIONS = 500 # 迭代次数

# 初始化种群
def initialize_population(size, genes):
    return np.random.choice([0, 1], size=(size, genes))

# 评估种群
def evaluate_population(population, utility_func):
    return np.array([utility_func(individual) for individual in population])

# 选择（轮盘赌选择法）
def selection(population, fitness):
    fitness_sum = np.sum(fitness)
    probabilities = fitness / fitness_sum
    indices = np.random.choice(population.shape[0], size=population.shape[0],
replace=True, p=probabilities)
    return population[indices]

# 交叉（单点交叉）
def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, GENES) # 随机选择交叉点
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2

# 变异
def mutate(individual, mutation_rate):
    for i in range(GENES):
        if np.random.rand() < mutation_rate:
            individual[i] = 1 - individual[i]
    return individual

```

```

# 遗传算法主函数
def genetic_algorithm(utility_func):
    population = initialize_population(POPULATION_SIZE, GENES)
    best_solution = None
    best_fitness = -np.inf

    for generation in range(GENERATIONS):
        fitness = evaluate_population(population, utility_func)
        fitness = np.clip(fitness, a_min=1, a_max=None)
        best_idx = np.argmax(fitness)
        if fitness[best_idx] > best_fitness:
            best_fitness = fitness[best_idx]
            best_solution = population[best_idx].copy()

        selected = selection(population, fitness)
        next_generation = []
        for i in range(0, POPULATION_SIZE, 2):
            parent1, parent2 = selected[i], selected[i+1]
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, MUTATION_RATE)
            child2 = mutate(child2, MUTATION_RATE)
            next_generation.extend([child1, child2])
        population = np.array(next_generation)

    return best_solution, best_fitness

# 运行遗传算法
best_bitstring, best_fitness = genetic_algorithm(utility_function)
print("Best Bitstring:", best_bitstring)
print("Best Fitness:", best_fitness)

#流程图中的节点
class Node(object):
    def __init__(self, loss_rate, test_cost, attribute, number, is_tested) -> None:
        self.loss_rate = loss_rate#次品率, 固有属性
        self.test_cost = test_cost#检测成本, 固有属性
        self.attribute = attribute#三种类型之一
        self.number = number#个数
        self.is_tested = is_tested#是否被检查
#表示零件的节点
class Component(Node):
    def __init__(self, loss_rate, test_cost, buy_cost, number, is_tested) -> None:
        super().__init__(loss_rate, test_cost, "component", number, is_tested)

```



```

        self.buy_cost = buy_cost # 购买单价
        self.real_loss_rate = loss_rate
# 表示半成品的节点
class Semi_finished(Node):
    def __init__(self, loss_rate,
test_cost, number, product_cost, destory_cost, is_tested) -> None:
        super().__init__(loss_rate, test_cost, "semi_finifshed", number, is_tested)
        self.product_cost = product_cost
        self.forward = np.array([])
        self.real_loss_rate = self.loss_rate # 真实次品率, 即考虑是否检查后生产的
半成品的次品率
        self.destory_cost = destory_cost # 拆解成本
        def add_component(self, com: Component):
            self.forward = np.append(self.forward, com) # 添加生产半成品的零件类的
信息, 用于计算产量和次品率
            """
            计算真实次品率
            """
        def count_real_lossrate(self):
            if len(self.forward) == 2:
                x1 = self.forward[0].is_tested
                x2 = self.forward[1].is_tested
                n1 = self.forward[0].real_loss_rate
                n2 = self.forward[1].real_loss_rate
                n3 = self.loss_rate
                self.real_loss_rate = x1*x2*n3 + (1-x1)*(1-x2)*(1-(1-n1)*(1-n2)*(1-
n3)) + x1*(1-x2)*(1-(1-n3)*(1-n2)) + x2*(1-x1)*(1-(1-n3)*(1-n1))
            if len(self.forward) == 3:
                # 对第三题有用, 需要给出公式再完善
                x1 = self.forward[0].is_tested
                x2 = self.forward[1].is_tested
                x3 = self.forward[2].is_tested
                n1 = self.forward[0].real_loss_rate
                n2 = self.forward[1].real_loss_rate
                n3 = self.forward[2].real_loss_rate
                n4 = self.loss_rate
                self.real_loss_rate = 1-pow((1-n1),1-x1)*pow((1-n2),1-x2)*pow((1-
n3),1-x3)*pow((1-n1),1-x1)*(1-n4)
            """
            计算本轮的真实数量, 并在 number 里面更新
            """
        def count_real_number(self):
            num = np.array([])
            for i in self.forward:

```

```

        num = np.append(num,i.number*(1-i.is_tested*i.real_loss_rate))
        self.number = num.min()
'''
计算正品数量
'''
def positive_object(self):
    self.positive = self.number*(1-self.real_loss_rate)
class Final_product(Node):
    def __init__(self, loss_rate, test_cost,
number,product_cost,destory_cost,change_cost, price,is_tested) -> None:
        super().__init__(loss_rate, test_cost,"Final_product", number, is_tested)
        self.product_cost = product_cost
        self.forward = np.array([])
        self.real_loss_rate =self.loss_rate
        self.price =price#价格
        self.destory_cost =destory_cost#拆除成本
        self.change_cost = change_cost#更换成本呢
    def add_component(self,com):
        self.forward =np.append(self.forward,com)
'''
计算真实次品率
'''
def count_real_lossrate(self):
    if(len(self.forward)==2):
        x1 = self.forward[0].is_tested
        x2 =self.forward[1].is_tested
        n1 =self.forward[0].real_loss_rate
        n2 = self.forward[1].real_loss_rate
        n3 =self.loss_rate
        self.real_loss_rate =x1*x2*n3+(1-x1)*(1-x2)*(1-(1-n1)*(1-n2)*(1-
n3))+x1*(1-x2)*(1-(1-n3)*(1-n2))+x2*(1-x1)*(1-(1-n3)*(1-n1))

    if(len(self.forward)==3):
        if(len(self.forward)==3):
            #对第三题有用，需要给出公式再完善
            x1 = self.forward[0].is_tested
            x2 =self.forward[1].is_tested
            x3 = self.forward[2].is_tested
            n1 = self.forward[0].real_loss_rate
            n2 = self.forward[1].real_loss_rate
            n3 = self.forward[2].real_loss_rate
            n4 = self.loss_rate
            self.real_loss_rate = 1-pow((1-n1),1-x1)*pow((1-n2),1-x2)*pow((1-
n3),1-x3)*pow((1-n1),1-x1)*(1-n4)

```

```

"""
计算本轮的真实数量,并在 number 里面更新
"""
def count_real_number(self):
    num = np.array([])
    for i in self.forward:
        num =np.append(num,i.number*(1-i.is_tested*i.real_loss_rate))
    self.number = num.min()
"""
计算正品数量
"""
def positive_object(self):
    self.positive = self.number*(1-self.real_loss_rate)

"""
这个字典用于对题目二的不同检测方法进行索引，用于深度优先搜索
"""
test_dict = {1:np.array([0,0,0]),2:np.array([0,0,1]),3:np.array([0,1,0]),4:np.array([1
,0,0]),5:np.array([0,1,1]),6:np.array([1,0,1]),7:np.array([1,1,0]),8:np.array([1,1,1])}
"""
用于解决第二问的深度优先搜索树的节点结构，记录了历史检测方法的操作索引，
本轮检测方法的索引，当前层次，当前的累计盈利，上一层到这一层的转化成本
"""
class DFS_NODE(object):
    def
__init__(self,history_test:np.array,now_test:int,now_level:int,now_benefit:float,change_cost:float) -> None:
    self.history_test =history_test
    self.now_test =now_test
    self.now_level =now_level
    self.now_benefit =now_benefit
    self.change_cost = change_cost

class Produce_loop(object):
    """
    level_count:工艺层数
    final:成品节点
    node_list:半成品节点
    """
    def __init__(self,level_count:int,final:Final_product,node_list:np.array) ->
None:
    self.level_count = level_count
    self.final = final

```

```

self.node_list = node_list
self.origin_num = []#初始的零件数量
self.Total_benefit = 0#累计利润
self.buy_cost = 0#购买成本
self.X = np.array([])#最优选择
self.destory_choice = np.array([])#拆解内容的选择
self.component_num = 0#零件数量
self.benefit_each_batch = np.array([])#每一轮的利润
if(self.level_count==3):
    for i in self.node_list:
        for j in i.forward:
            self.component_num+=1
            self.buy_cost+=j.number*j.buy_cost
            self.origin_num.append(j.number)
else:
    for i in self.final.forward:
        self.component_num+=1
        self.buy_cost+=i.number*i.buy_cost
        self.origin_num.append(i.number)
'''

```

向前计算每一层的数量，真实次品率和正品个数(需要先规定 is_tested，即确定策略)

```

'''
def cal_forward(self):
    if(self.level_count==3):
        for i in self.node_list:
            i.count_real_lossrate()
            i.count_real_number()
            i.positive_object()
        self.final.count_real_lossrate()
        self.final.count_real_number()
        self.final.positive_object()
'''

```

向后计算，用于调整零件的数量和次品率开启新的循环：输入:X 本轮对于每一个可拆解的组件(半成品/成品)是否拆解的选择

```

'''
def backward(self,X:np.array):
    #先看成品是否拆解
    all_product = []#记录零件下一轮的总数
    for i in range(self.component_num):
        if(X[-1]==1):#拆解成品
            all_product.append(self.final.number-self.final.positive)#所有的次
            都被拆成了零件
        else:

```

```

        all_product.append(0)
    if(self.level_count==3):#存在半成品的情况
        tag =0
        for i in range(len(self.node_list)):
            if(X[i]==1):#需要拆解
                for j in self.node_list[i].forward:
                    #累加零件的数量
                    all_product[tag]+= self.node_list[i].number-
self.node_list[i].positive
                    if(j.is_tested==1):#如果零件被检测则没有次品
                        j.real_loss_rate= 0
                        j.number = all_product[tag]
                    else: #如果没有被检验则次品是过程开使时的次品数
                        j.real_loss_rate =
j.number*j.real_loss_rate/all_product[tag]
                        j.number = all_product[tag]
                        tag+=1
                else:#不需要拆解则不需要增加
                    for j in self.node_list[i].forward:
                        if(j.is_tested==1):#如果零件被检测则没有次品
                            j.real_loss_rate= 0
                            j.number = all_product[tag]
                        else: #如果没有被检验则次品是过程开使时的次品数
                            try:
                                j.real_loss_rate =
j.number*j.real_loss_rate/all_product[tag]
                            except:
                                j.real_loss =0#证明没有新的零件进入循环
                                j.number = all_product[tag]
                                tag+=1
            else:#不存在半成品的情况
                tag =0
                for i in self.final.forward:
                    if(i.is_tested==1):
                        i.real_loss_rate=0
                        i.number = all_product[tag]
                    else:
                        i.real_loss_rate = i.number*i.real_loss_rate/all_product[tag]
                        i.number =all_product[tag]
                tag+=1
'''
    修改是否检测零件，半成品，成品的策略(为了方便，这里面的 x 是从成品
    到半成品到零件的顺序)
'''

```

```

def change_choice(self,X:np.array):
    self.final.is_tested = X[0]
    if(self.level_count==2):
        tag =1
        for i in self.final.forward:
            i.is_tested = X[tag]
            tag+=1
    else:
        tag =1
        for i in self.final.forward:
            i.is_tested =X[tag]
            tag+=1
        for i in self.final.forward:
            for j in i.forward:
                j.is_tested =X[tag]
                tag+=1
'''
得到一轮中的盈利
'''

def benefit_one_batch(self):
    if(self.final.number<=1):return-1
    return self.final.positive*self.final.price
'''
得到一轮中的成本
'''

def cost_one_batch(self):
    if(self.final.number<=1): return -1
    cost =0
    cost+=self.final.number*(self.final.is_tested*self.final.test_cost+self.final.pr
product_cost)+(1-
self.final.is_tested)*self.final.number*self.final.change_cost*self.final.real_loss_rat
e#产品检测成本和生产成本,调换成本
    if(self.level_count==3):
        for i in self.node_list:
            cost+=i.number*(i.is_tested*i.test_cost+i.product_cost)
            for j in i.forward:
                cost+=j.number*j.is_tested*j.test_cost
    else:
        for i in self.final.forward:
            cost+=i.is_tested*i.number*i.test_cost
    return cost
'''
得到拆解成本,X 为记录成品/半成品是否拆解的变量,从半成品到成品记录
'''

```

```

def destory_cost(self,X:np.array):
    cost =0
    if(X[-1]==1):#需要拆解成品
        cost+=(self.final.number-self.final.positive)*self.final.destory_cost
    if(self.level_count==3):
        tag =0
        for i in self.node_list:
            cost+=X[tag]*(i.number-i.positive)*i.destory_cost
            tag+=1
    return cost
#选择最优的策略
def optimizer(self):
    if(self.level_count==2): choose_number =3
    else: choose_number =12
    max_benefit =0
    X_max =np.full(choose_number,-1)
    if(choose_number==3):#遍历的方法，第二问只有三层循环
        for a1,a2,a3 in product(range(2), repeat=3):
            X =np.array([a1,a2,a3])
            self.change_choice(X)#改变策略
            self.cal_forward()#更新参数
            benefit = self.benefit_one_batch()-self.cost_one_batch()#计算成本
            if(benefit>max_benefit):
                X_max =X.copy()
                max_benefit =benefit
    if(choose_number==12):
        """
        遗传算法
        """
        #def benefit_for_genetic(X:np.array):
        #    #self.change_choice(X)
        #    #self.cal_forward()
        #    #return self.benefit_one_batch()-self.cost_one_batch()
        #X_max,max_benefit = genetic_algorithm(benefit_for_genetic)
        """
        暴力遍历
        """
        for a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12 in product(range(2),
repeat=12):
            X =np.array([a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12])
            self.change_choice(X)#改变策略
            self.cal_forward()#更新参数
            benefit = self.benefit_one_batch()-self.cost_one_batch()#计算成本
            if(benefit>max_benefit):

```

```

        X_max=X.copy()
        max_benefit =benefit
    return (X_max,max_benefit)
'''
根据历史路径去恢复当前系统状态（结合 DFS 使用）
'''
def recover(self,history:np.array):
    #首先对零件参数进行恢复（先只写题目二的）
    if(self.level_count==2):
        tag =0
        for i in self.final.forward:
            i.number = self.origin_num[tag]#恢复数量
            i.real_loss_rate =i.loss_rate#恢复真实次品率
            tag+=1
        for i in history:#然后对每一层进行推理
            self.change_choice(test_dict[i])
            self.cal_forward()
            self.backward(np.array([1]))
'''
利用深度优先搜索求问题二的全局最优解,规定深度
'''
def DFS(self,deep:int):
    #初始化节点栈,并加入第一层操作
    q = []
    for index in test_dict.keys():
        q.append(DFS_NODE(np.array([]),index,1,0,0))
    MAX_BENEFIT =0#记录最大利润
    MAX_BENEFIT_NODE=None#记录最大利润对应的操作序列
    while(len(q)!=0):
        now_node = q.pop()#得到当前节点
        #重新更新流程网络的状态并计算当前选择的利润
        if(now_node.now_level>deep):#如果当前层次大于设定的最深层次则
            不再更新,只判断是否更新 MAX_BENEFIT
            if(now_node.now_benefit>MAX_BENEFIT):#如果大于增更新策略
                MAX_BENEFIT = now_node.now_benefit
                MAX_BENEFIT_NODE = now_node.history_test
            continue
        self.recover(now_node.history_test)#恢复系统状态
        self.change_choice(test_dict[now_node.now_test])#按照本轮选择更新
        self.cal_forward()#更新参数
        benefit =self.benefit_one_batch()-self.cost_one_batch()#计算成本
        if(benefit-now_node.change_cost<=0):#如果本轮成本为负检查进行
            本轮前的利润是否大于最大值

```



```

        if(now_node.now_benefit>MAX_BENEFIT):#如果大于增更新策略
            MAX_BENEFIT = now_node.now_benefit
            MAX_BENEFIT_NODE = now_node.history_test
        else:#如果本轮仍能盈利则计算转化成本并且将下一层的节点添加到栈中
            ds = self.destory_cost(np.array([1]))#计算进入下一轮需要的拆解成本
            #将下一轮的所有节点添加进栈中
            for index in test_dict.keys():
                q.append(DFS_NODE(np.append(now_node.history_test,now_node.now_test),index,now_node.now_level+1,now_node.now_benefit+benefit-now_node.change_cost,ds))
            return MAX_BENEFIT,MAX_BENEFIT_NODE
'''
主循环得到整体的策略
'''
def max_loop(self):
    CHANGE_COST =0#拆解成本
    if(self.level_count==2):#题目二情况
        while(1):
            X,benefit = self.optimizer()
            if(X[0]==-1): break
            if(benefit-CHANGE_COST<=0): break
            self.X = np.append(self.X,X)#记录这种状态
            self.destory_choice = np.append(self.destory_choice,np.array([1]))
            self.Total_benefit+=benefit-CHANGE_COST#增加利润
            self.change_choice(X)#按照最大的情况并更新参数
            self.cal_forward()
            CHANGE_COST =self.destory_cost(np.array([1]))#计算这种情况下的拆解成本
            self.backward(np.array([1]))#更新拆解后的情况
        else:#题目三情况
            #第一轮循环单独进行
            X,benefit = self.optimizer()
            self.X = np.append(self.X,X)#记录这种状态
            #self.destory_choice = np.append(self.destory_choice,DESTORY)
            self.Total_benefit+=benefit-CHANGE_COST#增加利润
            self.benefit_each_batch = np.append(self.benefit_each_batch,benefit-CHANGE_COST)
            self.change_choice(X)#按照最大的情况并更新参数
            self.cal_forward()
            while(1):
                MAXBENEFIT =np.full(16,-1)

```

```

MAX_X = np.full(16, None)
destory_case = np.full(16, None)
i = 0
for a1, a2, a3, a4 in product(range(2), repeat=4): #生成拆解的选择并
进行计算
    if ((a1+a2+a3+a4==0) or (((a1+a2+a3)<3) and (a4==0))): continue #
都不拆解或者在成品不拆解的情况下半成品只拆解一部分
    destory = np.array([a1, a2, a3, a4])
    new_case = copy.deepcopy(self) #得到此类的深拷贝后的副本
    CHANGE_COST = new_case.destory_cost(destory) #计算这种
情况的成本
    new_case.backward(destory) #更新拆解后的情况
    X, benefit = new_case.optimizer()
    if (X[0]==-1): continue
    if (benefit-CHANGE_COST<=0): continue
    MAXBENEFIT[i] = benefit-CHANGE_COST
    MAX_X[i] = X
    destory_case[i] = destory #记录情况
    i += 1
    print("one_case_calculated\n")
    #选择最大利润对应的操作并且更新类的状态
    max_index = np.argmax(MAXBENEFIT, axis=0) #最大值对应的索引
    if MAXBENEFIT[max_index]==-1: break #无法继续盈利，退出
    else:
        self.X = np.append(self.X, MAX_X[max_index]) #记录状态
        self.destory_choice =
np.append(self.destory_choice, destory_case[max_index])
        self.Total_benefit += MAXBENEFIT[max_index]
        self.benefit_each_batch =
np.append(self.benefit_each_batch, MAXBENEFIT[max_index])
        #将对象的状态更新:
        self.backward(destory_case[max_index])
        self.change_choice(MAX_X[max_index])
        self.cal_forward()
    print("*****one_batch_finish*****\n")

Q1 = 0.1
Q2 = 0.2
Q3 = 0.1
u = 1.64
n = 1000
def new_q(q):
    return q + u * pow(q * (1 - q) / (n - 1), 0.5)

```

```

c1 =Component(Q1,8,4,10000,0)
c2 =Component(Q2,1,18,10000,0)
f1= Final_product(Q3,3,0,6,5,10,56,0)
f1.add_component(c1)
f1.add_component(c2)
p = Produce_loop(2,f1,None)
max_benefit,choose = p.DFS(5)
#p.max_loop()
#p.X,p.Total_benefit/p.buy_cost
max_benefit/p.buy_cost,choose

LOSS_RATE =new_q(0.1)
NUM =100000
IS_TESTED =0
c2_1 = Component(LOSS_RATE,1,2,NUM,IS_TESTED)
c2_2 =Component(LOSS_RATE,1,8,NUM,IS_TESTED)
c2_3 = Component(LOSS_RATE,2,12,NUM,IS_TESTED)
c2_4 = Component(LOSS_RATE,1,2,NUM,IS_TESTED)
c2_5 = Component(LOSS_RATE,1,8,NUM,IS_TESTED)
c2_6 = Component(LOSS_RATE,2,12,NUM,IS_TESTED)
c2_7 = Component(LOSS_RATE,1,8,NUM,IS_TESTED)
c2_8 = Component(LOSS_RATE,2,12,NUM,IS_TESTED)
s2_1 = Semi_finished(LOSS_RATE,4,8,8,6,IS_TESTED)
s2_2 = Semi_finished(LOSS_RATE,4,8,8,6,IS_TESTED)
s2_3 = Semi_finished(LOSS_RATE,4,8,8,6,IS_TESTED)
s2_1.add_component(c2_1)
s2_1.add_component(c2_2)
s2_1.add_component(c2_3)
s2_2.add_component(c2_4)
s2_2.add_component(c2_5)
s2_2.add_component(c2_6)
s2_3.add_component(c2_7)
s2_3.add_component(c2_8)
f2_1= Final_product(LOSS_RATE,6,NUM,8,10,40,200,IS_TESTED)
f2_1.add_component(s2_2)
f2_1.add_component(s2_1)
f2_1.add_component(s2_3)
p =Produce_loop(3,f2_1,np.array([s2_1,s2_2,s2_3]))
p.max_loop()
p.X,p.destory_choice,p.Total_benefit/p.buy_cost,p.benefit_each_batch

from matplotlib import pyplot as plt
plt.plot(p.benefit_each_batch)
plt.show()

```

