基于决策树的最优生产方案问题

摘要

在我国经济飞速发展的今天，质量水平的高低不仅能反映一家企业的信誉，更能反映一个国家的综合经济实力。所以对于高效率的抽样检测方案研究成为当今热门方向。本问题聚焦于企业在生产过程中遇到的不同情况，建立模型求解最优检验方案。

针对问题一，围绕如何通过最小的抽检次数确定是否采购零配件，探讨了基于**假设检验**的**抽样检测**优化方法。样本服从**二项分布**，并通过**正态分布**近似计算。为解决抽检次数的优化问题，提出了一种基于 ***Monte Carlo* 模拟**的**自适应抽检方案**。该方案通过动态调整抽检样本量，以在固定置信度内**最小化抽检次数**；讨论了在两种置信度下(**95%**和 **90%**)的自适应抽检模型，通过模拟求得最优抽检次数在置信度为 95%下是 **235 次**，在置信度为 90%下是 **209 次**，并结合**国家标准**计算**接受域**$\boldsymbol{A_{c1}=11, A_{c2}=20}$与**拒绝域**$\boldsymbol{R_{c1}=11, R_{c2}=20}$。

针对问题二，本文提取出了企业在产品生产中的**四个**关键决策环节，提出了一种基于**决策树**的分析方法，旨在确保产品质量并最大化利润。利用**决策树模型**，分析并扩展策略为 **24 种**决策组合，详见**表 6**。结合各类成本参数建立了**利润最大化**的目标函数给出了**最优决策组合，详见表 7**。

针对问题三，本文提出了一种基于**熵权法**的**评价体系**，以优化成本与次品率之间的平衡，选取**成本**和**次品率**为**核心决策参数**，建立了基于熵权法的多指标优化模型。通过对 8 个零配件、3 个半成品和 3 道工序的成本与次品率数据进行计算，本文利用信息熵确定各指标的**权重**，综合评价每个策略组合的表现。最终在模型求解后，得出本题所提出的对应情景的最优策略组合，具体决策方案见**表 8**。

针对问题四，本文通过**自适应抽样**和 ***Monte Carlo* 模拟**推导真实次品率，并以此优化企业生产决策的过程。通过**点估计**和 *Monte Carlo* 模拟对 5%、10%、20%三个检测次品率的样本进行真实次品率推断，本文得出相应的点估计结果分别为 **6.51%、12.46%、22.02%**。然后，将推导出的真实次品率代替问题二和问题三中的样本次品率，重新计算生产过程中的优化决策模型。结果表明，部分情境下最佳策略组合发生了变化，优化后的方案能够为企业在复杂条件下的生产决策提供了可靠依据。

关键词： 假设检验；自适应抽样；*Monte Carlo* 模拟；决策树；质量检测

一、问题分析

本题给出了一个工业经济中非常常见的场景，即对于需要装配以及需要出厂产品的质量检验工作。本文对本场景下提出的四个问题做出如下分析。

1.1 问题一的分析

针对问题一，需要我们以假设检验的方式设计检验次数尽可能少的抽样方案。由于本场景中没有给出具体的初始样本数n_0，所以我们有两个方面需要考虑：

①总体样本遵循什么分布。

②在此基础上提出的抽样检测方案如何受到初始样本n_0的影响以及在这种影响下如何进行优化以求解出最少样本量 n。

考虑到本题每个样品检测情况有且仅有合格或者不合格两种，我们认为初始样本服从二项分布。基于统计质量控制中的抽样检测理论[1]，传统连续随机抽样 *CSP* 对过程控制的能力较弱，无法完成以抽检次数最少为目标的优化方案。所以我们将 *CSP* 方案改进，加入自适应抽样方案，用于基于已有的检测结果动态调整样本量，并通过蒙特卡洛模拟多次运行改进的自适应抽样过程，并记录下每次满足置信度的样本数。最后通过所有可接收批次的样本数的期望估计最优的抽样方案。

1.2 问题二的分析

针对问题二，是一个复杂决策问题。需要我们根据题目中所给出的决策过程针对六种不同的情景做出不同决策，使企业的利润最大。对于决策的每一步，都需要我们归纳整合决策因素以及此决策对后续决策产生的影响，最后根据决策组合给企业带来的收益评估该策略的效果。根据本题中的表 1 提供的信息，我们发现每一步的成本是影响决策的最大因素。我们可以进一步整合出每一步决策对应的影响因素。

对于零配件 1 以及零配件 2 的检测，企业需要考虑每种零配件的检测成本，由于零配件 1 和零配件 2 的成本有所不同，所以我们认为企业需要对两种零配件是否检测分别做出不同的决策。若不检测，企业也需要考虑对后续成品装配次品率的影响。

对于成品装配和检测，即使两种零配件都表现良好，在成品的装配过程中也会存在一定的次品率，企业是否选择在出厂前排除不合格品也是非常重要的节点。企业需要考虑选择检测导致的检测成本、装配成本以及可能的拆卸成本的损失，以及如果次品流入市场后调换费用以及企业信誉的损失。

对于不合格的成品，企业也需要对是否回收做出判断，决策因素包括拆解过程产生的拆解费用和丢弃后产生的资源浪费的权衡。

在本题的情境中，我们还需要考虑用户的调换次数是否会对最后的决策产生影响。所以我们通过建立决策树，计算每种情景下所有树枝的利润，并最终选择利润最大的决策组合作为每种情景的决策方案。

1.3 问题三的分析

在本问题的情境中，由于决策树分支过于庞大，无法准确的追踪每种情景的销售额和利润。所以本题需要我们聚焦于如何选择有效的决策依据。我们沿用问题二的评

价步骤，对于企业而言，需要尽可能多的成品以及尽可能少的成本。即本题需要优化的目标是成本与次品率的平衡，同时需要考虑 8 个零配件、3 个半成品和 3 道工序的成本和次品率数据。将成本和次品率作为核心决策参数，建立目标优化函数，以实现成本最小和次品率最低为目标。我们利用熵权法计算成本和次品率的权重，熵值越大，权重越小，用此来表明该指标在评价体系中的重要性成都。将问题三中的 8 个零配件、3 个半成品和 3 道工序的成本和次品率数据进行组合，计算每个决策组合对应的总成本和次品率。对每个决策组合的总成本和次品率进行加权求和，得出每个组合的综合评价值。找到其中评价值最小的组合作为最优解策略。

1.4 问题四的分析

问题四的核心在于扩展问题二和问题三，并且通过基于问题一模型的抽样检测结果推导出原批次的真实次品率，再利用该真实次品率重新求解问题二和问题三的决策问题。

所以我们需要根据已知的样本次品率，通过点估计推导原批次的真实次品率。再利用推导出的真实次品率替代问题二、问题三中的样本次品率，重新计算最优决策组合。

二、模型假设

1、由于题中未给出初始样本量，根据客观事实和实际生活。假定初始样本量$n_0 = 1000$。

2、对于问题二，假设用户换货一次会重新进行一次当前决策组合的生产流程，所以用户重新得到的产品仍然有次品的可能。

三、符号说明

符号	意义
$\boldsymbol{n_0}$	初始样本数
$\boldsymbol{n}$	抽取的样本个数
$\boldsymbol{c_0}$	接收上限, 即在本样品中可以接受的最大样品数
$\boldsymbol{\hat{p}}$	样本中的次品率
$\boldsymbol{p_0}$	零配件次品率的标称值
$\boldsymbol{Z}$	正态检验统计量
$\boldsymbol{\alpha_i}$	自适应抽样中样本置信度
$\boldsymbol{\beta_1}$	情景 1 的标称置信度，本题所给 0.05
$\boldsymbol{\beta_2}$	情景 2 的标称置信度，本题所给 0.10
$\boldsymbol{E(n_i)}$	最优抽检次数
$\boldsymbol{A_c}$	接受数
$\boldsymbol{R_c}$	拒绝数
$\boldsymbol{x_i}$	第 i 步决策的决定，取值为 1 和 0
$\boldsymbol{Profit}$	总利润
$SALES$	总销售额
$\boldsymbol{C_{total}}$	总成本取值

四、模型建立与求解

4.1 问题一模型建立与求解

4.1.1 问题一模型的准备

问题一考虑企业需要用最小检验次数的抽样检测方法决定是否从供应商处购买这批零配件。由于零配件的初始个数n_0在每次采购中存在变化，不能确定，所以我们围绕初始样本量n_0以及最小抽检样本量 n，将问题聚焦探究以下三个部分：

①初始样本量的分布规律：

假设企业需要检测一批零配件次品率是否超过标称值p_0，根据本问题描述，可以用假设检验来实现：

零假设H_0：检验出样本的次品率$p \leq p_0$

备择假设H_1：检验出样本的次品率$p > p_0$

工厂在对产品质量抽检时，每一个样品都只有两种情况：合格或者不合格，符合二项分布的定义。所以我们使用二项分布描述不合格品的检测过程。设 n 为样本量，X 为抽样中次品的个数，则 X 服从参数为 n 和 p 的二项分布：

$$X \sim B(n,p) \tag{1}$$

当样本量 n 较大时，根据 *DeMoivre-Laplace* 中心极限定理，对满足$Y_n \sim B(n,p)$的任意实数 x,有：

$$\lim_{n\to\infty} P\left(\frac{Y_n - np}{\sqrt{np(1-p)}} \le x\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{t^2}{2}}\,dt \tag{2}$$

$$Y_n \sim N[np, np(1-p)], n \to \infty \tag{3}$$

即可以用正态分布近似二项分布，从而简化计算。由上式(2)可知，二项分布可以用均值np_0和方差$np_0(1p_0)$的正态分布近似：

$$Z = \frac{\hat{p}p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}} \tag{4}$$

其中，$\hat{p}$是样本中的次品率；p_0是标称值，Z是正态检验统计量。

②抽样检测的优化方法：

本问题需要我们在抽样检测的基础上考虑抽样检测抽样次数方面的成本问题。随着工业经济以及计算机技术的发展，平衡生产能达到预期经济效益以及用户需求的质量控制技术层出不穷，其中以连续抽样检验(*CSP*)为典型代表。允许以预定的间隔检查连续流动的产品或材料，并对其进行一系列质量测试。在我国 2002 年修订后的国家连续抽样检验标准 *GB/T 8052-2002* 中[2]，详细阐述了连续抽样检测的过程质量控制方案(*Continuous Sampling Plan,CSP*)。但是 *CSP* 方案有一定的局限性，只能满足唯一质量约束(*AOQL*)[3]。为了适应本问题，在无法固定抽检样本量的情况下，我们将 *CSP* 方案进行优化，在假设检验置信区间的约束下，使检验数最小化为新抽检方案的目标。

4.1.2 问题一模型的建立

根据假设 1，我们可以得到假设的默认样本总量n_0=1000，如果采用 *CSP* 方案，所得出的抽检方案不能具有良好的稳定性，且在对于抽检次数的优化问题上没有很好的表现。所以我们在 *CSP* 方案的基础上做出改进，以达到根据初始样本量以及固定置信度调整抽检样本量的方案。

①自适应抽样检测方案的介绍[4]

在本题中，我们需要根据假设检验置信度的约束，动态规划出适合的抽检样本量。所以我们选择自适应算法：

在给定置信度后，自适应算法可在运行的过程中根据置信度对自身进行修改，使算法具有更高的精度。

自适应算法通过随机采样得到逼近真实结果的采样值，为了提高传统自适应算法的精确度，我们需要做大量的自适应随机采样模拟，我们采用 *Monte Carlo* 方法优化后的自适应方案，且样本次数越多越精确[7]：

$$Approximation(Average(X) = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{5}$$

Step1:

对初始样本总量n_0进行模拟随机抽样，并且用贝叶斯方法[6]计算每一次随机抽样的样本接受零假设H_0的置信度，

Step2:

比较样本置信度α与标称置信度β，若：

$$\alpha \leq \beta \tag{6}$$

则保留此时的抽检样本量n_i。

Step3:

在进行M次随机采样后，标定所有被记录的n_M的期望$E(n_M)$为最优抽检方案。

$$E(n_M) = \sum_{M=1}^{M}(n_1 + n_2 + \cdots + n_M) \tag{7}$$

具体步骤如下图 1 所示：

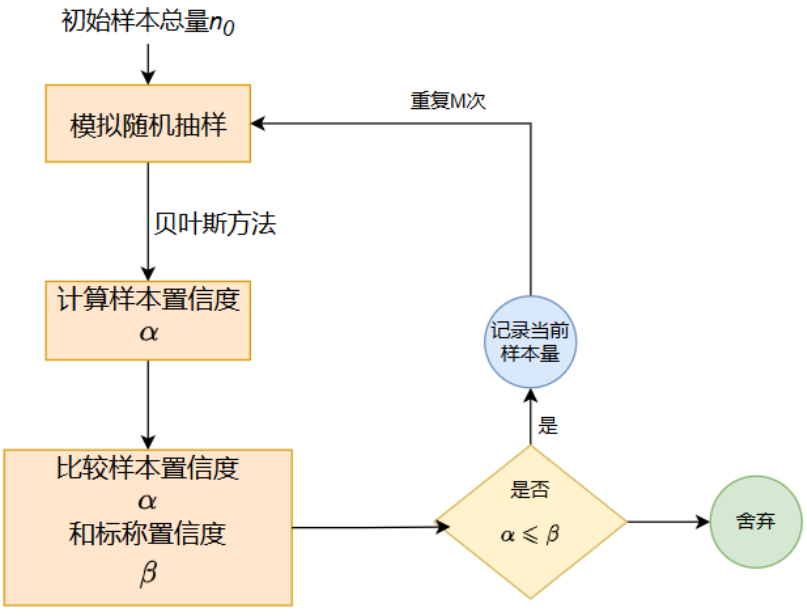

图 1 基于蒙特卡洛模拟的自适应算法步骤

4.1.3 问题一模型的求解

①情况一

在本题的情境下，第一种情况，需要我们求解出，在 95%的置信度下拒绝原假设H_0。即$\beta = 0.05$。我们用 *Python* 进行基于蒙特卡洛模拟的自适应抽样，可以得到如下表 1 所示结果。具体代码见附件一 Q1.py。

表 1 情况一蒙特卡洛模拟结果

情况一		
	置信度α_1	平均样本数$E(n_i)$
第一次模拟	95%	239.49
验证集	95%	232.66

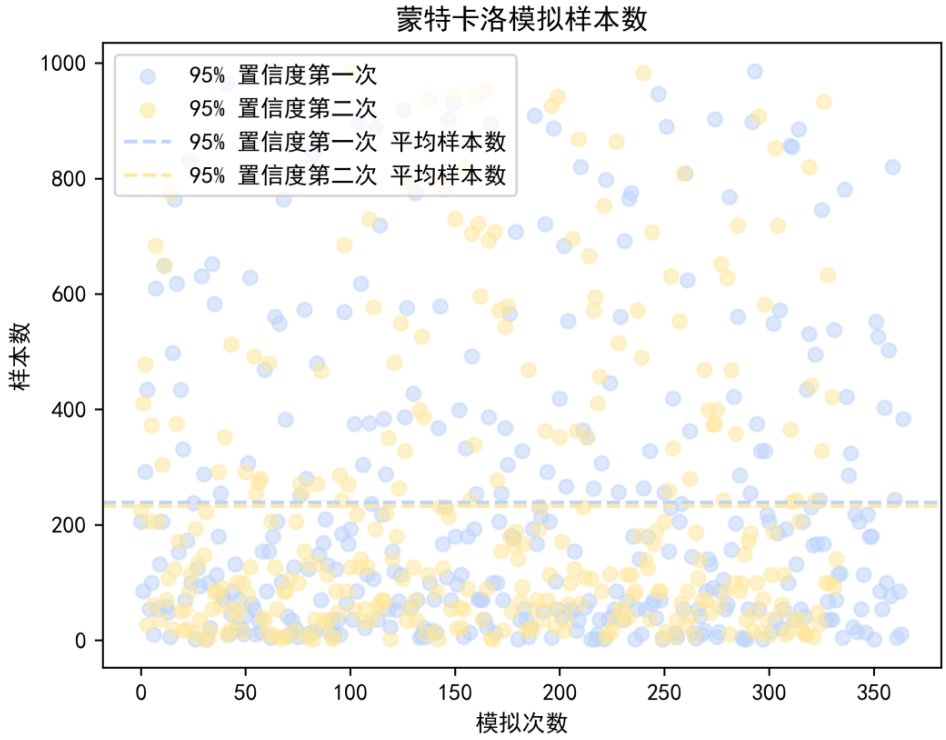

图 2 情况一蒙特卡洛模拟结果

由上表 1，可以得到，在 1000 次蒙特卡洛模拟结束后，可以得到最优抽检次数 *n* 约为 235 次左右。

在国家标准 *GB/T2828.1-2003* 中(完整表格见附录)，抽样检验标准表给出了不同检验水平以及不同样本量对应的接受数A_c和拒绝数R_c。我们也可以得到通过样本量和接受质量限(*AQL*)计算对应接收数A_c和拒绝数R_c的公式如下式 0 所示：

$$\begin{cases} C = n \times AQL \\ A_c = \lfloor C \rfloor \\ R_c = \lceil C \rceil \end{cases} \tag{8}$$

所以，我们可以得到样本量 *n*=235、接收质量限为 0.05 时的接受域A_c和拒绝域R_c：

$$\begin{cases} C_1 = n \times AQL = 235 \times 0.05 = 11.75 \\ A_{c1} = \lfloor C_1 \rfloor = 11 \\ R_{c1} = \lceil C_1 \rceil = 12 \end{cases} \tag{9}$$

即，当样本中检测出小于等于 11 个次品时，我们应该接受该批产品；当样本中检测出大于等于 12 个次品时，我们就必须拒绝该批产品。

②情况二

第二种情况，需要我们考虑在 90%的置信度下接受原假设H_0。即$\beta = 0.1$。同样，我们用 *Python* 进行基于蒙特卡洛模拟的自适应抽样，可以得到如下表 2 所示结果。具体代码见附录 Q1.py。

表 2 情况二蒙特卡洛模拟结果

情况二		
	置信度	平均样本数
第一次模拟	90%	210.19
验证集	90%	207.15

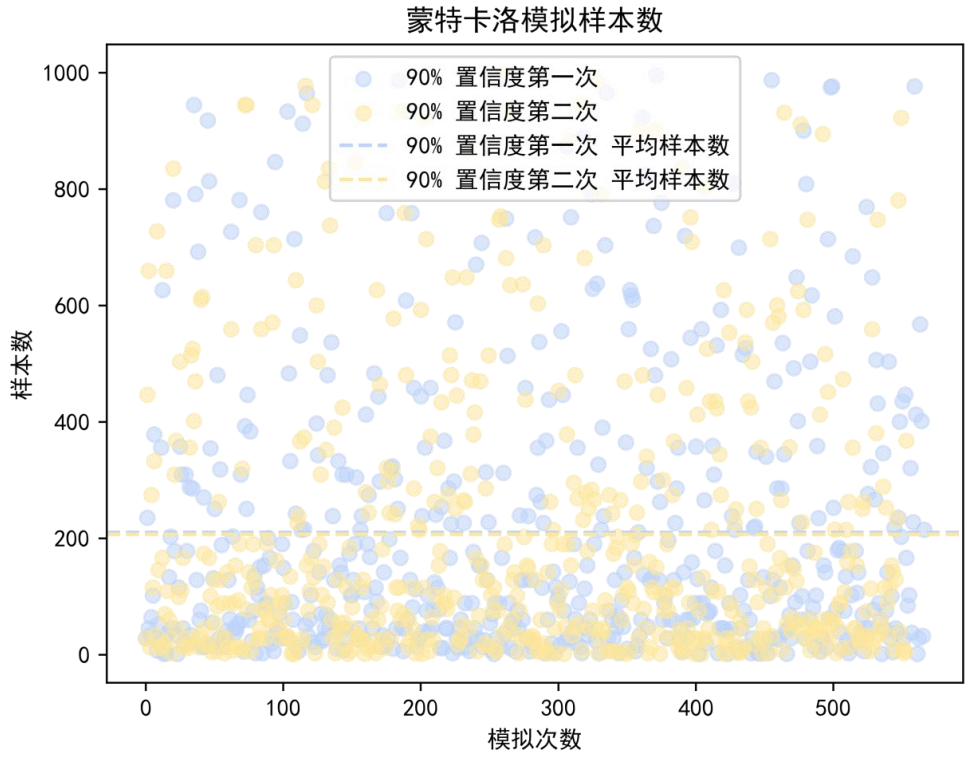

图 3 情况二蒙特卡洛模拟结果

由上表 2，我们可以得到，在 1000 次蒙特卡洛模拟结束后，可以得到最优抽检次数 n 约为 210 次左右。

同样，我们根据公式 0，可以得出样本量 n=210、接收质量限为 0.10 时的接受域A_c和拒绝域R_c：

$$\begin{cases} C_2 = n \times AQL = 209 \times 0.10 = 20.9 \\ A_{c2} = \lfloor C \rfloor = 20 \\ R_{c2} = \lceil C \rceil = 21 \end{cases} \tag{10}$$

即，当样本中检测出小于等于 20 个次品时，我们应该接受该批产品；当样本中检测出大于等于 21 个次品时，我们就必须拒绝该批产品。

③小结

在本节中，我们通过改进后的基于蒙特卡洛模拟的自适应算法，在初始样本是二项分布的情景下，求解出了最优的抽样检测数在两种情况中的结果，并通过国家标准的样本与接受数、拒绝数的关系公式，求解出在对应置信度下企业如何做出对应的决策。在上文中，我们通过计算式(9)和计算式(10)计算得到了两种情况的接受域和拒绝域，最后，将两种情况最后的抽检次数最少的方案总结在下表 3。

表 3 n_0=1000 情况最优抽检方案

	β	A_c	R_c	n(个)
情况一	0.05	11	12	235
情况二	0.1	20	21	209

4.2 问题二模型建立与求解

在问题二中，题目所给的情境进一步细化，需要我们对成品在检验、装配以及销售中的四种场景做出决策。为了确保产品质量并最小化成本，企业需要在产品制造的四个阶段进行全面分析以及做出相应决策。

4.2.1 问题二模型的准备

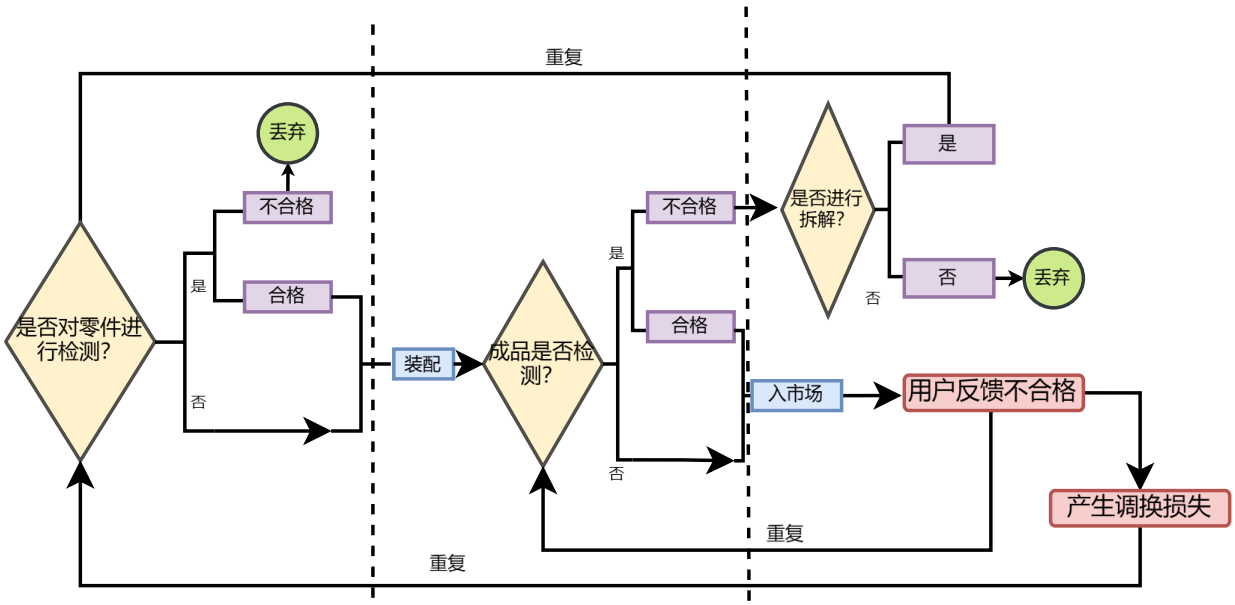

图 4 四个决策环节

根据上图 3，我们可以较为清晰的看到四个决策环节。

①企业需要考虑是否对零配件进行检测。

在生产过程中，零配件的质量是基础，所以对零配件的质量检测环节的决策尤为重要。本题中给出两种零配件，所以企业需要做两步决策。如果企业选择对零配件 1 或零配件 2 进行检测，都会产生一定的检测成本，并且：

合格，可以进入装配环节，且减少装配环节的成本。

不合格，直接丢弃，产生无法回收的制造成本。

如果不进行检测，则所有零配件可以直接进入装配环节，虽然节省了检测费用，但是会增加后续过程的次品率，产生不必要的装配成本和检测成本。

②企业需要考虑是否对成品进行检测。

在装配过程中也会产生由装配失误产生的损耗。此时也需要企业做出是否检测的决策。如果企业选择对装配成功的成品进行检测，同样有两种情况：

合格，会被送入市场，但是如果样本较大时，检测成本较大。

不合格，进入第三层决策，也许会产生较大的拆解费用以及重新检测费用。

如果企业选择不检测，不合格的产品可能会流入市场。如果永和购买到不合格的产品，不仅会产生调换损失和承担赔付用户的产品损失，更重要的是会对企业声誉造成不小的影响。

③企业需要考虑对于检测出的不合格品如何处理。

对于检测出的不合格品，企业可以选择拆解回收。拆解出的零配件在经过检测合格后仍然可以重新利用，避免资源浪费。当然，不可避免会产生拆解成本以及第二次的检测成本。此外，企业也可以选择直接丢弃不合格品，如果这样选择，加大对零配件的消耗，可能未来会面临采购成本增加。

4.2.2 问题二模型的建立

①决策变量以及决策参数的确定

我们需要考虑上述 4 个决策变量产生的决策组合：

$$Dec\_com = 2^r, r = 4 \tag{11}$$

其中，*Dec_com* 为所有决策组合，*r* 为决策变量。根据上式**错误!未找到引用源。**，可以得到一共有 16 种决策组合，每个组合对应以下 4 个决策变量：

表 4 四个取值变量的含义

含义	变量	说明
零配件 1 是否检测	x_1	x_1=1 代表检测，x_1=0 代表不检测
零配件 2 是否检测	x_2	x_2=1 代表检测，x_2=0 代表不检测
成品是否检测	x_3	x_3=1 代表检测，x_3=0 代表不检测
是否拆解不合格成品	x_4	x_4=1 代表检测，x_4=0 代表不检测

在本定义下的决策组合中，当x_3=0 时，未经过检测的成品直接流入市场，用户有概率会收到不合格品从而需要调换，被退回的不合格品需要重复步骤三做出是否拆解的决策，也就是需要重新决定x_4的取值，即，即使x_3=0，也会重新做决策x_4，所以所有情景都成立，决策树不需要进行剪枝处理。所以共有 16 种决策组合，如下表 5 所示

表 5 决策树所有决策组合

策略	检测零配件 1	检测零配件 2	检测成品	拆解不合格成品
策略 1	是	是	是	是
策略 2	是	是	是	否
策略 3	是	是	否	是
策略 4	是	是	否	否
策略 5	是	否	是	是
策略 6	是	否	是	否
策略 7	是	否	否	是
策略 8	是	否	否	否
策略 9	否	是	是	是
策略 10	否	是	是	否
策略 11	否	是	否	是
策略 12	否	是	否	否
策略 13	否	否	是	是
策略 14	否	否	是	否
策略 15	否	否	否	是
策略 16	否	否	否	否

由上表 5 可知，策略 3、策略 7、策略 11、策略 15 在对于是否检测成本的决策中选择不检测直接流入市场，但是在决策四的选择中选择了拆解，说明该成品为不合格成品，被用户退换后进行决策四且选择了拆解；

策略 1、策略 5、策略 9、策略 13 在决策三的检测中检测出不合格产品，并在决策四中进行拆解；

上表 5 中x_4=1 的 8 种策略在拆解后零配件 1 和零配件 2 会重新进入决策树进行选择，所以在这 8 种决策组合中，每种决策不可避免会多产生多种情景。为了简化计算同时节省成本，对于第二轮零配件 1 和零配件 2 检测决策为第一轮零配件 1 和零配件 2 决策取反(若零配件在第一轮检测中已经被检测合格，我们在第二轮检测中便不再进行检验;若该零配件在第一轮检测中未被检验，则我们在第二轮检测中检验是否合格)。同时，我们发现，在拆解一次后的检验中，次品率普遍在 2%以下，再检测为不合格品的可能性很低，所以我们选择忽略多次拆解再检测的情景，只考虑一次拆解。所以，可以将上表 5 中x_4=1 的 8 种策略扩展为 16 种，如下表 6 所示：

表 6 扩展策略组合

策略	检测零配件 1	检测零配件 2	检测成品	拆解不合格品	拆后检测零配件 1	拆后检测零配件 2	拆后检测成品	拆解不合格成品
策略 1	是	是	是	是	否	否	是	否
					否	否	否	否
策略 3	是	是	否	是	否	否	是	否
					否	否	否	否
策略 5	是	否	是	是	否	是	是	否
					否	是	否	否
策略 7	是	否	否	是	否	是	是	否
					否	是	否	否
策略 9	否	是	是	是	是	否	是	否
					是	否	否	否
策略 11	否	是	否	是	是	否	是	否
					是	否	否	否
策略 13	否	否	是	是	是	是	是	否
					是	是	否	否
策略 15	否	否	否	是	是	是	是	否
					是	是	否	否

以上每种策略对应两种扩展策略，如上表所示，我们根据假设 3，为了简化计算以及降低成本，我们不允许再拆解已经拆解过的不合格品，即所有策略只允许拆解并重新检验一次。

根据表 5 和表 6 综合来看，总共有 24 种决策组合。

由题目中的表 1 我们可以提取出企业决策的依据，并将它们转化成决策树参数：

-零配件的次品率
-采购价格
-检测费用(包括零配件和成品)
-装配费用
-市场价格
-产品调换损失
-拆解成本
-成品的次品率

② 成本定义和利润计算

Step1: 利润计算

在生产过程中，每一个决策点都伴随一定成本支出，包括检测成本、拆解成本、装配成本、调换损失等。本题需要通过利润来选择每种情景的最佳决策，即最大化利润函数：

$$\max \quad Profit = sales - C_{total} \tag{12}$$

其中，$Profit$表示利润，$sales$表示销售额取值，C_{total}表示总成本取值。根据公式0，如果需要利润最大，则需要在每种情景的决策中，得到销售额尽可能大以及总成本尽可能小的决策组合作为最终企业采取的决策。

Step2: 成本计算

根据上图 5.2 总结的决策步骤，我们将总成本定义分成 6 个方面：

$$C_{total} = C_{perchase} + C_{parts\_inspect} + C_{assemble} + C_{finished\_inspect} + C_{exchange\_loss} + C_{disa} \tag{13}$$

在上式 0 中，$C_{perchase}$表示所有零配件的采购成本，其具体计算公式如下式(14)所示:

$$C_{perchase} = n \times C_{parts\_inspect} \tag{14}$$

其中，n 表示采购零配件的数量，$C_{perchase}$表示每个零配件的购买单价，具体取值来自于本题目中的表 1。

$C_{parts\_inspect}$、$C_{finished_inspect}$分别表示零配件检测成本、成品检测成本，其具体计算公式如下式(15)(16)所示：

$$C_{parts\_inspect} = n \times C_{f1} \times x_a, a = 1,2 \tag{15}$$

$$C_{finished\_inspectt} = n \times C_{f2} \times x_3 \tag{16}$$

其中，C_{f1}、C_{f2}分别表示单个零配件检测成本、单个成品检测成本，其具体取值同样来源于本题的表 1;x_1表示决策变量 1,x_2表示决策变量 2。

$C_{assemble}$表示装配成本，是第三步装配成品的决策中的重要依据，每种情景装配成本不变，单个成品的装配成本为 6 元。

$C_{exchange\_loss}$、C_{disa}表示总的调换损失和拆解费用。用户一旦收到不合格的产品就会选择调换成另一个产品，由假设 2，我们可以得到调换一个产品会有另一个产品按照当前决策被生产，所以第二次用户同样由概率收到次品进行调换。设用户进行调换的次数为 i。所以调换成本和拆解成本的公式如下式(17)(18)所示。

$$C_{exchange\_loss} = C_s \times i + C_{products} \tag{17}$$

$$C_{disa} = n_{unqualified} \times C_d \tag{18}$$

其中，C_s表示单个调换成本，$C_{products}$表示调换产品是企业赔付给用户的产品损失，$n_{unqualified}$表示检测出不合格品的个数，C_d表示单个成品拆解费用。

4.2.3 问题二模型的求解

我们发现当成品经过检测不合格后，若企业选择拆解，零配件 1 和零配件 2 重新进行决策，这一分支下还有多种情景。如上文中表 5 的总结，由于本文假定初始样本量为一个定值 1000，所以在拆后检查中，如果两个零配件有其中一个检查不合格，则

将两个零配件都丢弃；如果两个零配件都合格但是装配步骤仍然出现问题，我们依然选择不再拆解直接丢弃。这样的假设既简化了此复杂决策树本身决策点的个数避免进入不必要的死循环，也符合客观生产规律。例如策略 5 的扩展策略，若在拆后检测零配件 2 不合格，则零配件 1 一同丢弃，决策结束。

将决策树所有树枝导入 *python* 中，对于题目中所提出的 6 种情景进行求解，最终以利润最小作为决策依据挑选最优决策组合作为对应情境分别的决策方案。具体代码如附录所示 Q2.py。

表 7 六种情景最佳策略组合以及利润

	最佳策略组合	利润(元)
情景 1	1101-0000	15462.00
情景 2	1101-0000	7496.00
情景 3	0011-1110	13985.46
情景 4	1111-0010	10528.00
情景 5	0111-1010	9333.38
情景 6	0000	18586.75

上表中策略组合的第 i 个位置取值为 1 表示第 i 步决策执行，取值为 0 表示第 i 步决策不执行。“-”表示拆解后的重复决策。

结合题目中企业在生产中遇到的情景表格，情景 1 和情景 2 仅有次品率的差别，在最佳策略组合的选择中每一步做出的决策一致，但情景 2 利润小于情景 1，符合题意；

情景 3 和情景 4 的调换损失高出情景 1 和情景 2 三倍，所以应尽量避免产生调换费用，故在成品合格检验时选择检验使流入市场的产品都为正品。且情景 4 次品率高于情景 3，若仅进行成品检测，检测出的不合格品大大增加，成本增大，故情景 4 每一步都选择检测，符合题意；

情景 5 零配件 1 的检测费用增大，若避免检测零配件 1 可节约成本，所以情景 5 选择不检测零配件 1，符合题意；

情景 6 拆解费用非常高，但次品率很低，所以应该避免不合格品进入拆解步骤，所以情景 6 选择不检验，符合题意。

我们计算出了每个情景所有的成本、销售额、次品率等关键信息，具体数值见附录所示。下图为将数值可视化后的结果。

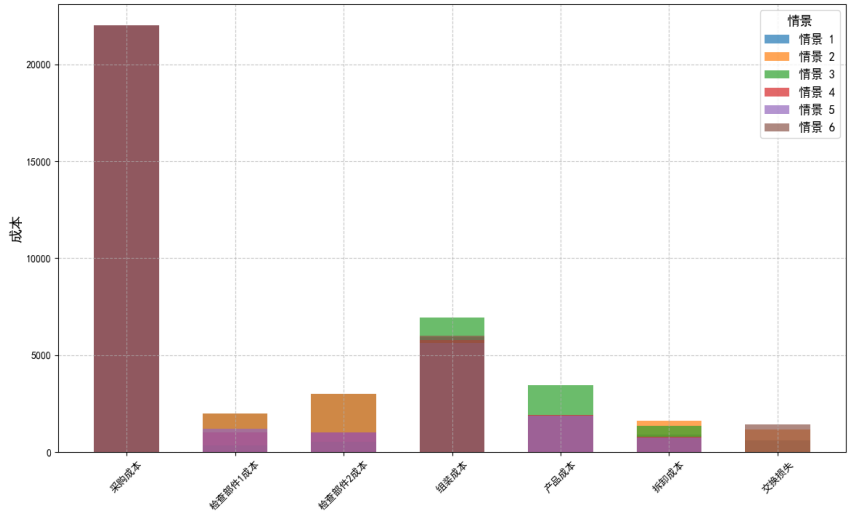

图 5 最佳决策的成本组成图

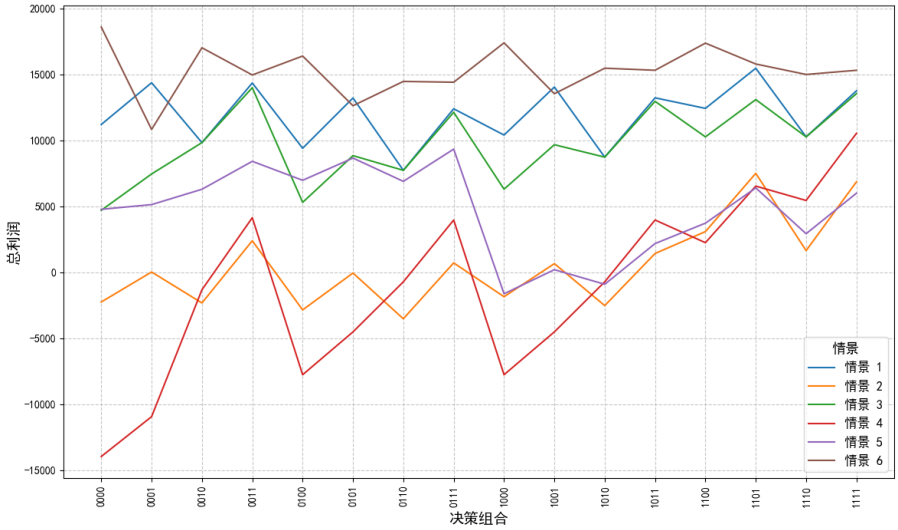

图 6 不同情景下总利润对比

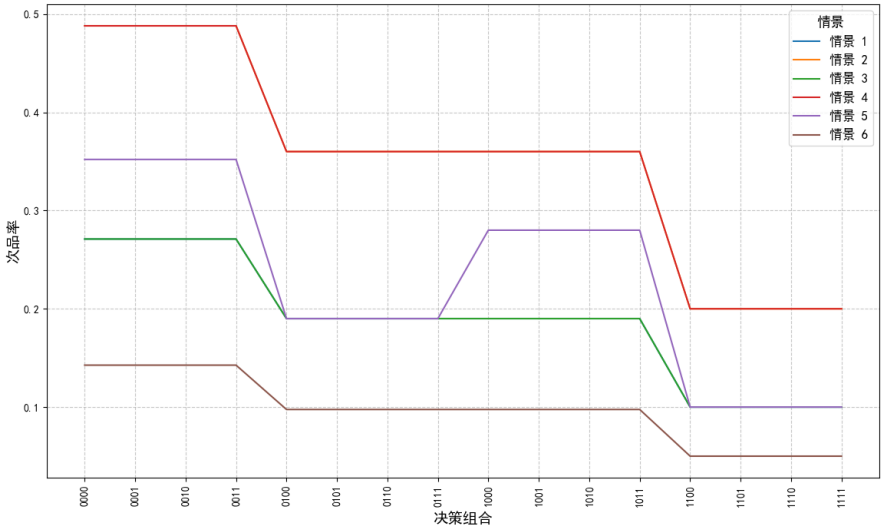

图 7 不同决策的次品率对比

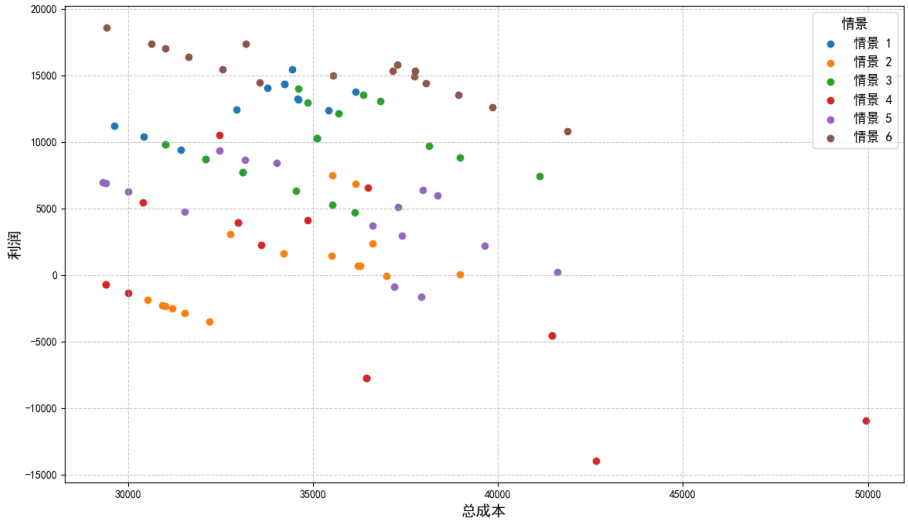

图 8 每种情景的总利润与中成本关系

4.3 问题三模型建立与求解

问题三将问题二的情景进行了扩展，将零配件种类增多、拼装工序变复杂。共 8 个零配件、3 个半成品，3 道工序。在此情境下，若需要用决策树求解出所有树枝的利润，会产生以下三个方面的问题：

①由决策树本身性质决定[6]，海量样本从计算到存储，开销非常高。

②树枝太多导致决策树太过庞大，通过决策树追踪每个树枝的利润时间复杂度太高；且决策有回溯部分，每个工序的次品也有它的价值，无法准确预估其带来的收益。

③无法进行决策树的剪枝操作，导致决策树太高、分支过多，产生过拟合[8]。

所以本问题需要在问题二的基础上进行优化改进。无法再用利润作为评价体系，需要重新选定决策参数以及决策依据。

4.3.1 问题三模型的建立

①选择决策参数

在问题二对于以利润为评价指标的评价体系建立的过程中，我们将利润分为销售金额和成本，并在后续的计算中求解销售金额和成本关系。将此关系的推导方法映射到问题三中，我们希望成品尽可能多且成本尽可能少，即次品率尽可能小同时成本尽可能小。所以对于问题三评价体系的建立，我们通过次品率和成本之间的关系建立目标优化函数$f(x,y)$：

$$\begin{cases} f(x,y)=ax+by \\ min_{(x,y)\in D} \quad f(x,y), D=\{\text{所有}x,y\text{的}2^{16}\text{种组合}\} \end{cases} \tag{19}$$

上式(19)中，x 为成本变量，y 为次品率变量。两个变量随着决策组合的改变而改变，最终目标优化的结果是提取出$f(x,y)$最小时 x 和 y 的取值以及对应的决策组合。a、b为指标 x、y 的权重，且必须满足a+b=1。

每个决策环节成本由一部分固定成本(如购买成本)和一部分可变成本(如检测成本)组成，将各个部分成本计算式总结如下：

Step1:8 个零配件

问题三存在 8 个零配件，其成本统一包括两个部分：(1)购买成本，(2)检测成本：

$$C_{parts} = C_{perchase} + C_{inspect} \tag{20}$$

上式(20)中，C_{parts}表示零配件的成本，$C_{perchase}$表示购买成本，$C_{inspect}$表示检测成本。

规定零配件总购买量为 N,购买单价为P_{parts},检测单价为I_{parts}，则

$$C_{perchase} = N \times P_{parts} \tag{21}$$

$$C_{inspect} = N \times I_{parts} \times x_i \tag{22}$$

其中，购买成本为固定成本，检测成本为可变成本，x_i表示每一步决策,i = 1,2,3,⋯,8。当进行检测时x_i=1，此时检测成本$C_{inspect}$不为零，否则，检测成本$C_{inspect}$恒为零。

Step2:3 个半成品

半成品在整个工艺中具有承上启下的作用，它的成本同样由两个部分组成：(a)组装成本，(b)检测成本。

$$C_{semi\_manu} = C_{Assemble} + C_{inspect} \tag{23}$$

其中C_{semi_manu}表示半成品的成本，$C_{Assemble}$表示组装成本。

规定半成品的总合成量为 N,组装单价为P_{ass},检测单价为I_{semi_manu},则：

$$C_{Assemble} = N \times P_{ass} \tag{24}$$

$$C_{inspect} = N \times I_{semi\_manu} \times x_i \tag{25}$$

其中，组装成本为固定成本，检测成本为可变成本，x_i表示每一步决策,i = 9,10,11。当进行检测时x_i=1，此时检测成本$C_{inspect}$不为零，否则，检测成本$C_{inspect}$恒为零。

Step3:成品

上文中，详细分析了零配件以及半成品的成本组成。成本由 8 个零配件合成 3 个半成品组成，成品的成本也包括两个部分：(1)装配成本，(2)检测成本。

$$C_{products} = C_{Assemble\_pro} + C_{inspect} \tag{26}$$

其中，$C_{products}$表示成品成本，$C_{Assemble_pro}$表示成本装配成本，$C_{inspect}$表示检测成本。

规定成品的合成量为 N，装配单价为 A，检测单价为$I_{products}$,则：

$$C_{Assemble\_pro} = N \times A \tag{27}$$

$$C_{inspect} = N \times I_{products} \times x_{15} \tag{28}$$

其中，装配成本为固定成本，检测成本为可变成本，当进行检测时x_{15}=1，此时检测成本$C_{inspect}$不为零，否则，检测成本$C_{inspect}$恒为零。

②确定权重

由上式(19)，我们可以得到一个基于成本和次品率的评价体系。同时我们加入了每个指标的权重用来衡量两个指标的重要程度。我们选择用熵权法确定指标权重。

③熵权法求解步骤

对于成本以及次品率，可以用熵值来判断它们的离散程度，其信息熵 *H(x)*值越小，指标的离散程度越大，对于综合评价的影响就越大。信息熵 *H(x)*的计算公式如下式所示[9]：

$$H(x) = -\sum_{i=1}^{q} p(x_i)\, log(p(x_i)) \tag{29}$$

其中 *H* 是信息熵，q 是信源消息格式，$p(x_i)$是消息x_i出现的概率。

Step1：数据标准化

这里采用极差标准化

$$x_0 = \frac{x - x_{min}}{x_{max} - x_{min}} \qquad y_0 = \frac{y - y_{min}}{y_{max} - y_{min}} \tag{30}$$

Step2:计算熵值

$$E_i = -\frac{1}{\ln(n)} \sum_{i=1}^{n} p_{ij} \ln(p_{ij}) \tag{31}$$

其中 *n* 是指标的个数，在本题中指标为成本 *x* 以及次品率 *y*，所以 n=2；

Step3:计算权重

通过熵值，计算每个指标的权重。熵值越大，权重越小，表明该指标在评价体系中的重要性较低。

$$w_i = \frac{1 - E_i}{\sum_{j=1}^{n}(1 - E_j)} \tag{32}$$

其中，E_i是第 i 个指标的熵值，通过式(31)计算得出。

Step4:计算综合评价值

求解上述两步后，代码根据各个指标的权重，对标准化后的数据加权求和，得出每个策略的综合评价值，即计算目标优化方程组(19)的值，并得出评价值最小的策略组合。

4.3.2 问题三模型的求解

①指标评价体系的求解

我们整理出本问题中 8 种零配件、3 种半成品以及最后拼装后的成品基本信息(如次品率、购买单价、检测成本、装配成本等)，枚举出所有决策组合后，将影响决策的参数放入 python 中，计算各个组成部分的成本和次品率，并根据决策组合的不同，考

虑半成品和成品的拆解费用对成本的影响。最后，我们可以得到所有决策组合对应的成本以及次品率，即x_i,y_i。

带入基于熵权法的指标评价体系模型中，用 python 遍历决策组合计算每个决策组合对应的权重a_i, b_i，通过x_i,y_i加权求和得到指标评价得分模型$f(x,y)$，取$f(x,y)$的最小值即为本题企业遇到情况在此指标评价体系模型下的最优解。具体代码见附录。

我们将得到的最优解用下表 8 展示。

表 8　问题三最优策略展示

最优检测结果		
熵值	x=0. 99445784	y=0. 99703556
权重	x=0. 65151324	y=0. 34848676
总成本	129. 589194	
成品次品率	0. 56473	
评价值	0. . 209837	

最优解策略编号为 256(全部策略表及带有评分的策略表见支撑材料)，详细决策如下图 9 所示：

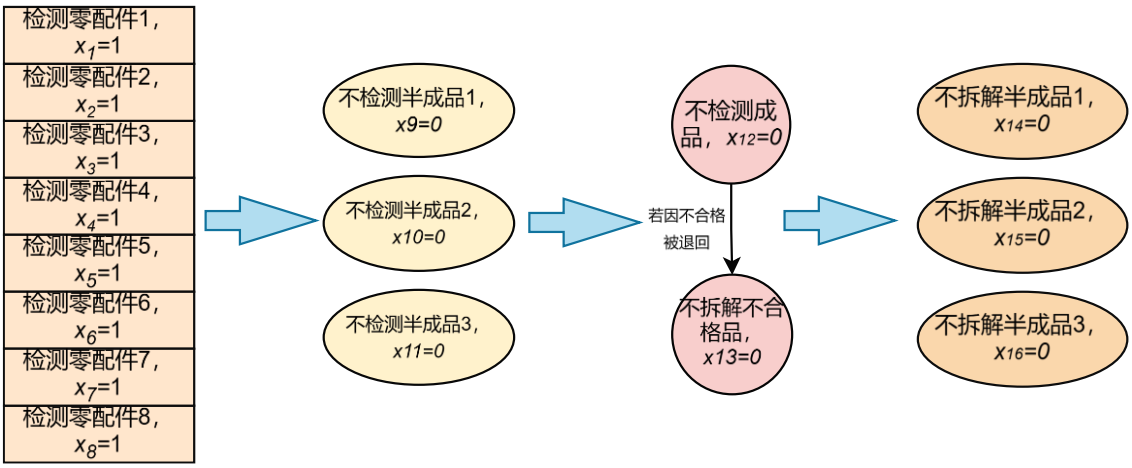

图 9 最优策略流程

我们将所有策略的总成本和成品次品率做可视化对比，得到如下图所示：

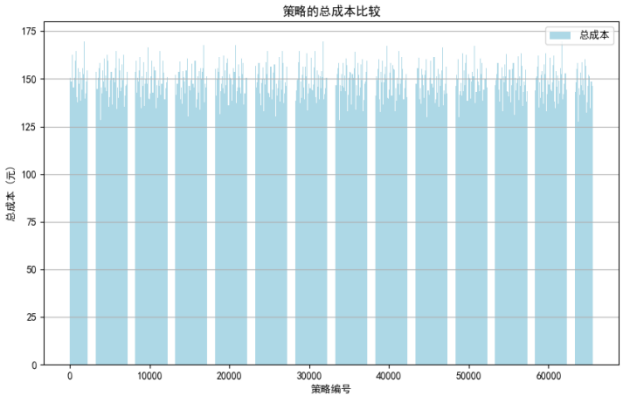

图 10 策略的总成本比较

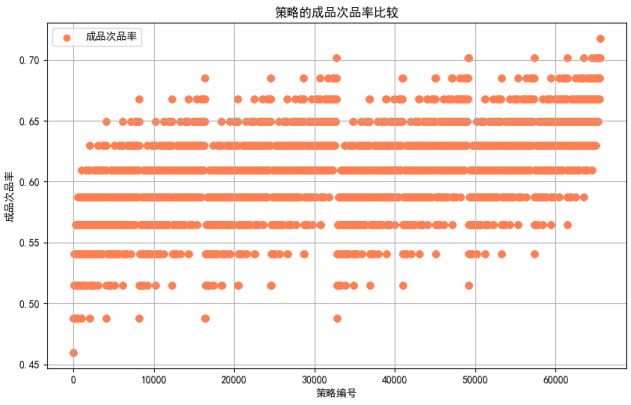

图 11 策略的成品次品率比较

4.4 问题四模型建立与求解

问题四在问题二、三的基础上，需要我们根据以问题一的模型抽样出来的概率推导原批次的真实概率后，用推导出的真实概率替代原有样本概率重新求解问题二和问题三。

4.4.1 问题四模型的建立

由问题二、问题三的数据，我们可以得到抽样检测的结果有三种概率：5%、10%、20%，需要我们根据此三个后验概率反向模拟出真实的概率。我们通过点估计来推断真实次品率。

①点估计解决过程

与问题一自适应抽样类似，本问题也需要完成在样本次品率的约束下求解总体次品率的任务。我们可以通过样本次品率约束下的自适应抽样来检测一批产品的次品率。在每次抽样的过程中，根据当前样本中的次品率进行判断，是否继续抽样，最终收敛的带每批次样本的次品率。然后，我们通过蒙特卡罗模拟多次运行，从而估计不同次品率下的真实值。

②具体实现步骤

Step1:自适应抽样检测

根据检测到的次品率进行抽样，每次抽样后计算当前次品率，并根据置信区间判断是否继续抽样。共有三个检测到的次品率：5%、10%、20%。

Step2:蒙特卡洛模拟

为了减小误差，找出普遍规律，通过蒙特卡洛算法多次模拟来收集每个批次的次品率估计值。结果是一个次品率的点估计分布，我们通过这些点估计来确定检测到的次品率对应的真实次品率。

Step3:结果可视化

最后，我们绘制出直方图来展示来展示不同检测次品率下的估计分布，并计算这些次品率对应的平均真实次品率。

我们在 python 中编写模拟程序，得到结果如下表 9 以及图 12 所示，具体代码见附录 Q4.py。

表 9 样品率对应的真实次品率

样本次品率	对应的真实次品率
5%	6.51%
10%	12.46%
20	22.02%

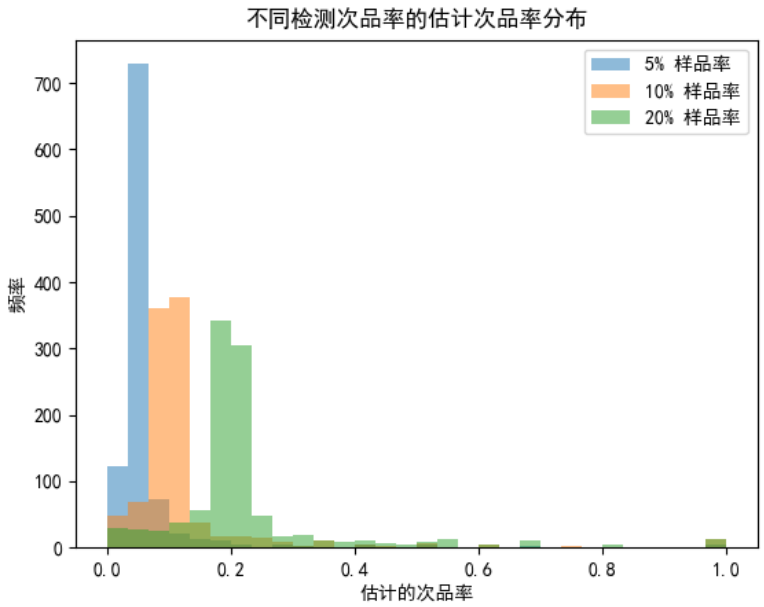

图 12 不同检测次品率的估计次品率分布

4.4.2 问题四模型的求解

①在问题四的条件下问题二的求解

我们将问题二中的所有次品率用真实次品率代替后，重新求解问题二的模型，计算出 6 种情景分别的最优决策组合如下表 10 所示。

表 10 问题四重解问题二后的最优策略

	最佳策略组合	利润(元)
情景 1	1101-0000	13527.73
情景 2	1101-0000	5868.06
情景 3	**1111-0010**	**11855.67**
情景 4	1111-0010	9080.73
情景 5	0111-1010	6679.01
情景 6	0000	15931.12

在上表中，由于次品率变化，情景 3 的最佳策略组合从 0011-1110 转变为 1111-0010。在对于零配件一和零配件二是否检测的决策上做出改变。

②在问题四的条件下问题三的求解

同样将问题三的所有次品率用真实次品率代替后，重新求解问题三的指标评价模型，计算出当前情景的最优决策组合如下表 11 所示。

表 11 问题四重解问题三后的最优检测结果

最优检测结果		
熵值	x=0.99454067	y=0.99725712
权重	x=0.6655926	y=0.32344074
总成本	132.959018	
成品次品率	0.648975	
评价值	0.214687	

同样，问题三的最优策略如图 5 所示。在经过问题四的重解后，虽然检测结果数值小幅改变，但是问题三的最优策略组合没有发生变化。

五、模型的评价与推广

5.1 模型的优缺点

5.1.1 模型的优点

①在问题一中，我们考虑到了传统连续随机抽样 *CSP* 的局限性，针对本题情景对自适应算法进行智能优化，用蒙特卡洛模拟进行多次随机采样后保留有效值，用此方法优化的优点有：

-能够比较准确逼真地模拟出企业在质量检测过程中对样本的随机抽取。

-蒙特卡洛模拟具有同时计算多个方案的能力，在本题中需要考虑在置信度的约束下抽检次数最小的情景，用蒙特卡洛模拟能更好地求出最优方案[7]。

②本文对于问题二的情景考虑全面，决策依据选取准确。既考虑到用户调换货的新产品的生产成本，又对于决策的每种情景做出详细解释。决策变量和参数具有明确的现实意义。

③问题二中对于被用户退回的不合格品，我们考虑了在拆解后重新进行决策一和决策二，且结合第一次检测的结果，通过取反操作简化了计算。

5.1.2 模型的缺点

①对于自适应抽样模型，在进行蒙特卡洛模拟时，为了节约时间，只进行了 1000 次模拟，得到的结果可能不具有普适性。

②对于决策树模型，本文为了简化建模，高效得到最优解只考虑了一次调换和拆解，实际情景往往更加复杂。同时，决策树分支过于庞大，计算成本高。

5.2 模型的推广

在模型二中，我们假设初始样本量为 1000，在对于不合格品拆解后的处理中，若零配件 1 损坏但零配件 2 合格，在本题的解决过程中我们将零配件 1 和零配件 2 同时丢弃。但是实际生产生活中，工厂一般是流水线作业，所以合格的零配件 2 可以进入下一批零配件加工生产。这样的方案不仅减少资源浪费，而且节约成本。

六、参考文献

[1].佟珈锐,刘政.抽样技术与方法在抽样检验中的应用[J].电声技术,2023,47(09):114-117.DOI:10.16311/j.audioe.2023.09.034.

[2].国家标准《单水平和多水平计数连续抽样检验程序及表》 [EB/OL]. 国家标准 - 全国标准信息公共服务平台 (samr.gov.cn)2024/9/6

[3].张道观.连续抽样检验最优方案的研究与应用[D].江西财经大学,2023.DOI:10.27175/d.cnki.gjxcu.2023.000607.

[4].张东.自适应抽样算法及其 R 包开发[D].华东师范大学,2017.

[5].《贝叶斯可信区间与频率置信区间的区别》 [EB/OL]. 杂记——贝叶斯可信区间与

频率置信区间的区别_置信区间和可信区间的区别-CSDN 博客 2024/9/7
[6].《通俗易懂的讲解决策树》 [EB/OL].【非常详细】通俗易懂的讲解决策树(Decision Tree) - 知乎 (zhihu.com)2024/9/7.
[7].《蒙特卡洛(Monte Carlo)方法的介绍和应用》 [EB/OL]. 蒙特卡洛(Monte Carlo)方法的介绍和应用-CSDN 博客 2024/9/7.
[8].汪靖翔.决策树算法的原理研究和实际应用[J].电脑编程技巧与维护,2022,(08):54-56+72.DOI:10.16184/j.cnki.comprg.2022.08.043.
[9].最常用的客观赋权方法——熵权法 [EB/OL]. 最常用的客观赋权方法——熵权法_客观赋权法-CSDN 博客.2024/9/7.

七、附录

7.1 环境描述

-操作系统：Windows 11 23H2

-运行环境：PyCharm 2024.2 (Professional Edition)

7.2 相关结果

7.2.1 问题二六种情景最优决策的详细结果

最佳策略	决策	总成本	销售额	利润	次品率	采购成本	零件1检测成本	零件2检测成本	装配成本	成品检测成本	拆解成本	换货损失
情景一	1101|0000	34434	49896	15462	10.00%	22000	2000	3000	5940	0	900	594
情景二	1101|0000	35512	43008	7496	20.00%	22000	2000	3000	5760	0	1600	1152
情景三	0011|1110	34595.1	48580.56	13985.46	27.10%	22000	342	513	6923.4	3461.7	1355	0
情景四	1111|0010	32480	43008	10528	20.00%	22000	1000	1000	5760	1920	800	0
情景五	0111|1010	32470.4	41803.78	9333.38	19.00%	22000	1216	1000	5620.8	1873.6	760	0
情景六	0000	29426.25	48013	18586.75	14.26%	22000	0	0	6000	0	0	1426.25

7.2.2 国家标准质量检测表

正常检验一次抽样方案

表2

样本量字码	样本量	接收质量限(AQL)																												
		0.010		0.015		0.025		0.040		0.065		0.10		0.15		0.25		0.40		0.65		1.0		1.5		2.5		4.0		6.5	
		Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re
A	2	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1
B	3	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩	
C	5	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩	
D	8	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2
E	13	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3
F	20	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3
G	32	⇩		⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6
H	50	⇩		⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8
J	80	⇩		⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11
K	125	⇩		⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15
L	200	⇩		⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22
M	315	⇩		⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	⇧	
N	500	⇩		⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	⇧		⇩	
P	800	⇩		0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	⇧		⇩		⇩	
Q	1250	0	1	⇧		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	⇧		⇩		⇩		⇩	
R	2000	⇧		⇧		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	⇧		⇩		⇩		⇩		⇩	

样本量字码	样本量	接收质量限(AQL)																												
		10		15		25		40		65		100		150		250		400		650		1000								
		Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re	Ac	Re							
A	2	⇩		⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	30	31							
B	3	⇩		1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	30	31	44	45							
C	5	1	2	2	3	3	4	5	6	7	8	10	11	14	15	21	22	30	31	44	45	⇧								
D	8	⇩		2	3	3	4	5	6	7	8	10	11	14	15	21	22	30	31	44	45	⇧								
E	13	2	3	3	4	5	6	7	8	10	11	14	15	21	22	30	31	44	45	⇧										
F	20	3	4	5	6	7	8	10	11	14	15	21	22	⇧		⇧		⇧		⇧										
G	32	5	6	7	8	10	11	14	15	21	22	⇧		⇧		⇧		⇧		⇧										
H	50	7	8	10	11	14	15	21	22	⇧		⇧		⇧		⇧		⇧		⇧										
J	80	10	11	14	15	21	22	⇧		⇧		⇧		⇧		⇧		⇧		⇧										
K	125	14	15	21	22	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
L	200	21	22	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
M	315	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
N	500	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
P	800	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
Q	1250	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										
R	2000	⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧		⇧										

⇩ —— 使用箭头下面的第一个抽样方案。如果样本量等于或超过批量，则执行100%全检。

⇧ —— 使用箭头上面的第一个抽样方案。

Ac —— 接收数。

Re —— 拒收数。

7.3 支撑材料

re 带评价值的策略结果.xlsx
re 策略结果.xlsx
带评价值的策略结果.xlsx
策略结果.xlsx

7.4 相关代码

7.4.1 问题一代码

Q1.py

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.stats.proportion import proportion_confint
from tqdm import tqdm
from matplotlib.font_manager import FontProperties

plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False    # 正常显示负号


# 进行蒙特卡洛模拟函数
def adaptive_sampling(p0, alpha, max_samples=1000):
    n = 0    # 当前抽取的样本数
    defective_count = 0    # 次品数量

    while n < max_samples:
        sample = np.random.binomial(1, p0)
        defective_count += sample
        n += 1
        current_defective_rate = defective_count / n

        ci_lower, ci_upper = proportion_confint(defective_count, n, alpha=alpha, method='beta')

        if ci_upper < p0:
            return "接收该批次", n, current_defective_rate
        elif ci_lower > p0:
            return "拒绝该批次", n, current_defective_rate

    return "无法确定", n, current_defective_rate


# 蒙特卡洛模拟
def monte_carlo_simulation(p0, alpha, max_samples=1000, num_simulations=1000):
    samples_list = []
    for _ in tqdm(range(num_simulations)):
        decision, samples, _ = adaptive_sampling(p0, alpha, max_samples)
        if decision != "无法确定":
            samples_list.append(samples)
    return samples_list


# 可视化
def plot_simulation_results(simulation1, simulation2, color1, color2, label1, label2, filename):
```

```
    fig, ax = plt.subplots()

    # 绘制两次蒙特卡洛模拟的散点图
    x1 = np.arange(len(simulation1))
    x2 = np.arange(len(simulation2))
    ax.scatter(x1, simulation1, color=color1, label=label1, alpha=0.5)
    ax.scatter(x2, simulation2, color=color2, label=label2, alpha=0.5)

    # 计算平均值
    avg_simulation1 = np.mean(simulation1)
    avg_simulation2 = np.mean(simulation2)

    # 输出拟合值到控制台
    print(f'{label1} 平均样本数: {avg_simulation1:.2f}')
    print(f'{label2} 平均样本数: {avg_simulation2:.2f}')

    # 绘制拟合线
    ax.axhline(y=avg_simulation1, color=color1, linestyle='--', label=f'{label1} 平均样本数')
    ax.axhline(y=avg_simulation2, color=color2, linestyle='--', label=f'{label2} 平均样本数')

    # 图表细节
    ax.set_xlabel('模拟次数')
    ax.set_ylabel('样本数')
    ax.set_title('蒙特卡洛模拟样本数')
    ax.legend()

    # 设置背景透明
    fig.patch.set_alpha(0)

    # 保存高分辨率透明图片
    plt.savefig(filename, dpi=500, transparent=True)  # 高分辨率 dpi=500


# 参数设定
p0 = 0.1  # 标称次品率
num_simulations = 1000  # 模拟次数
max_samples = 1000

# 在 95% 的置信度下，拒收次品率超过标称值的批次
alpha_for_reject = 0.05
simulation1_1 = monte_carlo_simulation(p0, alpha_for_reject, max_samples, num_simulations)  # 第一次模拟
simulation1_2 = monte_carlo_simulation(p0, alpha_for_reject, max_samples, num_simulations)  # 第二次模拟
```

```
# 在 90% 的置信度下，接收次品率不超过标称值的批次
alpha_for_accept = 0.10
simulation2_1 = monte_carlo_simulation(p0, alpha_for_accept, max_samples, num_simulations)  # 第一次模拟
simulation2_2 = monte_carlo_simulation(p0, alpha_for_accept, max_samples, num_simulations)  # 第二次模拟

# 绘制并保存两种情况下的图
plot_simulation_results(simulation1_1, simulation1_2, '#BDD4FC', '#FFE89C', '95% 置信度第一次', '95% 置信度第二次',
                        'monte_carlo_simulation_reject.png')
plot_simulation_results(simulation2_1, simulation2_2, '#BDD4FC', '#FFE89C', '90% 置信度第一次', '90% 置信度第二次',
                        'monte_carlo_simulation_accept.png')
```

7.4.2 问题二代码

Q2.py

```
import itertools

# 定义所有决策的组合 (16 种可能的组合)
decisions_combinations = list(itertools.product([0, 1], repeat=4))

# 根据表格中的数据重新定义情景参数
scenarios_corrected = [
    {'defective_rate_1': 0.10, 'defective_rate_2': 0.10, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 6, 'disassembly_cost': 5, 'product_failure_rate': 0.10},

    {'defective_rate_1': 0.20, 'defective_rate_2': 0.20, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 6, 'disassembly_cost': 5, 'product_failure_rate': 0.20},

    {'defective_rate_1': 0.10, 'defective_rate_2': 0.10, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 30, 'disassembly_cost': 5, 'product_failure_rate': 0.10},

    {'defective_rate_1': 0.20, 'defective_rate_2': 0.20, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 1, 'inspect_2': 1, 'assemble_cost': 6, 'inspect_product': 2,
     'market_price': 56, 'exchange_loss': 30, 'disassembly_cost': 5, 'product_failure_rate': 0.20},

    {'defective_rate_1': 0.10, 'defective_rate_2': 0.20, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 8, 'inspect_2': 1, 'assemble_cost': 6, 'inspect_product': 2,
     'market_price': 56, 'exchange_loss': 10, 'disassembly_cost': 5, 'product_failure_rate': 0.10},
```

```
    {'defective_rate_1': 0.05, 'defective_rate_2': 0.05, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 10, 'disassembly_cost': 40, 'product_failure_rate': 0.05}
]

# 成本计算函数，计算利润、次品率，并返回详细的成本分项
def calculate_profit_and_defective_rate_detailed(scenario, decision, quantity_part1=1000,
quantity_part2=1000, sub_decision_str=None):
    d1, d2 = scenario['defective_rate_1'], scenario['defective_rate_2']
    c1, c2 = scenario['purchase_1'], scenario['purchase_2']
    i1, i2 = scenario['inspect_1'], scenario['inspect_2']
    ac = scenario['assemble_cost']
    ip = scenario['inspect_product']
    mp = scenario['market_price']
    el = scenario['exchange_loss']
    dc = scenario['disassembly_cost']
    product_failure_rate = scenario['product_failure_rate']

    # 解析决策组合 (d1 检测, d2 检测, 成品检测, 拆解)
    detect_part1, detect_part2, detect_product, disassemble = decision

    # 1. 固定采购成本：无论是否检测，所有零配件都先买回来
    purchase_cost_part1 = c1 * quantity_part1
    purchase_cost_part2 = c2 * quantity_part2
    purchase_cost = purchase_cost_part1 + purchase_cost_part2

    # 2. 零配件 1 检测成本
    if detect_part1:
        inspect_cost_part1 = i1 * quantity_part1  # 检测所有的零配件，无论是否合格
        part1_defective_rate = 0  # 检测后次品率变为 0
        good_quantity_part1 = quantity_part1 * (1 - d1)  # 只留下合格的零配件 1
        # 检测的零配件直接丢弃
        quantity_part1 = good_quantity_part1  # 更新数量
    else:
        inspect_cost_part1 = 0  # 不检测时无检测费用
        part1_defective_rate = d1  # 不检测次品率保持原样

    # 3. 零配件 2 检测成本
    if detect_part2:
        inspect_cost_part2 = i2 * quantity_part2  # 检测所有的零配件，无论是否合格
        part2_defective_rate = 0  # 检测后次品率变为 0
        good_quantity_part2 = quantity_part2 * (1 - d2)  # 只留下合格的零配件 2
        # 检测的零配件直接丢弃
```

```
            quantity_part2 = good_quantity_part2  # 更新数量
        else:
            inspect_cost_part2 = 0  # 不检测时无检测费用
            part2_defective_rate = d2  # 不检测次品率保持原样

        # 4. 装配逻辑：根据两个零配件的质量情况决定成品次品率
        if detect_part1 and detect_part2:
            total_quantity = min(good_quantity_part1, good_quantity_part2)  # 两个零配件都合格时，用最少的
            product_defective_rate = product_failure_rate  # 仅考虑装配失败的次品率
        elif detect_part1 and not detect_part2:
            total_quantity = good_quantity_part1  # 零配件 1 合格的数量
            product_defective_rate = part2_defective_rate + (1 - part2_defective_rate) * product_failure_rate  # 考虑未检测零配件的次品率
        elif not detect_part1 and detect_part2:
            total_quantity = good_quantity_part2  # 零配件 2 合格的数量
            product_defective_rate = part1_defective_rate + (1 - part1_defective_rate) * product_failure_rate  # 考虑未检测零配件的次品率
        else:
            total_quantity = min(quantity_part1, quantity_part2)  # 两个零配件都可能次品
            # 如果任意一个零配件次品，成品必然是次品
            product_defective_rate = 1 - (1 - part1_defective_rate) * (1 - part2_defective_rate)
            # 如果两个零配件都是合格的，考虑装配失败的次品率
            product_defective_rate += (1 - product_defective_rate) * product_failure_rate

        # 5. 装配成本：无论装配是否生成次品，装配都需要费用
        assemble_cost = total_quantity * ac  # 实际装配成本

        # 6. 成品检测逻辑
        if detect_product:
            # 如果成品被检测，次品进入拆解
            product_cost = ip * total_quantity  # 检测所有成品
            defective_quantity = total_quantity * product_defective_rate  # 检测出次品的数量
            sold_quantity = total_quantity * (1 - product_defective_rate)  # 售出的正品数量
            total_sales = sold_quantity * mp  # 只有合格成品进入市场
            product_sale_loss = 0  # 检测后无销售损失
        else:
            # 如果不检测，次品直接进入市场
            product_cost = 0  # 不检测时无检测费用
            sold_quantity = total_quantity * (1 - product_defective_rate)  # 售出的正品数量
            total_sales = sold_quantity * mp  # 售出的正品销售额
            # 换货损失 = 调换损失(次品流入市场)
            product_sale_loss = product_defective_rate * el * total_quantity  # 换货损失=调换损失
            defective_quantity = total_quantity * product_defective_rate  # 流入市场的次品
```

```
    # 打印主序列中的正品售出数量和对应的销售额
    print(f"主序列售出正品数量: {sold_quantity:.2f}，对应销售额: {total_sales:.2f}")

    # 7. 计算进入子序列的正品零配件：正品零配件总数 - 已生产的正品成品个数
    total_good_part1 = quantity_part1 * (1 - part1_defective_rate)  # 零配件 1 的总正品数量
    total_good_part2 = quantity_part2 * (1 - part2_defective_rate)  # 零配件 2 的总正品数量
    # 未被使用的正品零配件个数
    unused_good_part1 = total_good_part1 - sold_quantity
    unused_good_part2 = total_good_part2 - sold_quantity
    # 进入子序列的正品零配件数为两个正品零配件中较少的那个
    good_parts_for_subsequence = min(unused_good_part1, unused_good_part2)

    # 8. 换货后的拆解逻辑：如果选择拆解
    if disassemble == 1:
        disassembly_cost = defective_quantity * dc  # 拆解所有次品的费用
        # 如果不检测，流入市场的次品也进入拆解
        if not detect_product:
            disassembly_cost += product_defective_rate * total_quantity * dc  # 流入市场的次品拆
解
    else:
        disassembly_cost = 0  # 没有拆解则无拆解费用

    # 9. 拆解后的子决策：子序列继承主序列的零配件检测和装配结果
    if disassemble == 1 and defective_quantity > 0:
        # 根据主序列的检测结果判断子序列中的零配件
        if detect_part1 and detect_part2:
            # 主序列检测了零配件，子序列只有因装配失败的正品零配件
            sub_defective_rate_1 = 0  # 子序列中的零配件都是正品
            sub_defective_rate_2 = 0
        else:
            # 主序列未检测，子序列可能有次品零配件
            sub_defective_rate_1 = d1  # 继承主序列的次品率
            sub_defective_rate_2 = d2

        # 拆解后的子决策：零部件检测决策相反，成品检测决策保持原决策或取反
        new_decision_part1 = 1 - detect_part1
        new_decision_part2 = 1 - detect_part2

        # 子决策，保留成品检测决策不变或取反
        new_decision_combinations = [(new_decision_part1, new_decision_part2, detect_product, 0),
# 保持成品检测决策
                                     (new_decision_part1,      new_decision_part2,      1      -
detect_product, 0)]  # 取反成品检测决策
```

```
        sub_results = []
        for sub_decision in new_decision_combinations:
            sub_decision_str = f"{''.join(map(str, decision))}|{''.join(map(str, sub_decision))}"
            # 子决策计算时根据主序列继承次品率
            sub_result = calculate_profit_and_defective_rate_detailed(
                {
                    'defective_rate_1': sub_defective_rate_1,
                    'defective_rate_2': sub_defective_rate_2,
                    'purchase_1': c1,
                    'purchase_2': c2,
                    'inspect_1': i1,
                    'inspect_2': i2,
                    'assemble_cost': ac,
                    'inspect_product': ip,
                    'market_price': mp,
                    'exchange_loss': el,
                    'disassembly_cost': dc,
                    'product_failure_rate': product_failure_rate
                },
                sub_decision,
                quantity_part1=good_parts_for_subsequence,  # 使用计算出的剩余正品零配件数量
                quantity_part2=good_parts_for_subsequence,
                sub_decision_str=sub_decision_str
            )
            sub_result['purchase_cost'] = 0  # 子决策没有采购成本

            # 子决策中的逻辑调整：
            # 如果子决策再次检测，次品被丢弃；如果不检测，次品进入市场，产生换货损失
            if sub_decision[2]:  # 成品再次检测
                sub_result['exchange_loss'] = 0  # 检测后不会有次品进入市场
            else:
                # 不检测，次品进入市场，计算换货损失(包括调换损失)
                defective_rate = sub_result['product_defective_rate']
                sub_result['exchange_loss'] = defective_rate * el * good_parts_for_subsequence  # 只算调换损失

            sub_results.append(sub_result)

        # 打印子序列中的正品售出数量和对应的销售额
        for sub_result in sub_results:
            sub_decision_str = sub_result.get('decision_str')
            print(f"子序列 {sub_decision_str} -> 售出正品数量: {sub_result['total_sales'] / mp:.2f},
```

```
对应销售额: {sub_result['total_sales']:.2f}")

            # 获取子分支中最佳利润方案
            best_sub_result = max(sub_results, key=lambda x: x['profit'])

            # 将子分支的成本、销售和利润与主决策相加
            total_sales += best_sub_result['total_sales']  # 加上子分支的销售收入
            disassembly_cost += best_sub_result['disassembly_cost']  # 加上子分支的拆解费用
            assemble_cost += best_sub_result['assemble_cost']  # 加上子分支的装配费用
            inspect_cost_part1 += best_sub_result['inspect_cost_part1']  # 加上子分支的零配件 1 检测费
用
            inspect_cost_part2 += best_sub_result['inspect_cost_part2']  # 加上子分支的零配件 2 检测费
用
            product_cost += best_sub_result['product_cost']  # 加上子分支的成品检测费用
            product_sale_loss += best_sub_result['exchange_loss']  # 子分支的换货损失
            sub_decision_str = best_sub_result.get('decision_str', sub_decision_str)

        # 总成本：采购成本 + 检测成本 + 装配成本 + 拆解成本 + 换货损失
        total_cost = (purchase_cost + inspect_cost_part1 + inspect_cost_part2 +
                      assemble_cost + product_cost + disassembly_cost + product_sale_loss)

        # 利润 = 销售收入 - 成本
        profit = total_sales - total_cost

        # 返回详细的成本分项
        return {
            'profit': profit,
            'product_defective_rate': product_defective_rate,
            'total_cost': total_cost,
            'total_sales': total_sales,
            'purchase_cost': purchase_cost,
            'inspect_cost_part1': inspect_cost_part1,
            'inspect_cost_part2': inspect_cost_part2,
            'assemble_cost': assemble_cost,
            'product_cost': product_cost,
            'disassembly_cost': disassembly_cost,
            'exchange_loss': product_sale_loss,
            'decision_str': sub_decision_str or ''.join(map(str, decision))
        }

# 对每个情景计算 24 种决策的详细结果
detailed_results = []
for index, scenario in enumerate(scenarios_corrected):
    print(f"\n 情景 {index + 1} 的 24 种决策详情:")
```

```
    scenario_result = []
    for decision in decisions_combinations:
        result = calculate_profit_and_defective_rate_detailed(scenario, decision)
        decision_str = result['decision_str']
        print(f"决策 {decision_str} -> 总成本: {result['total_cost']:.2f}, 销售额: {result['total_sales']:.2f}, 利润: {result['profit']:.2f}, 次品率: {result['product_defective_rate']:.2%}")
        print(f"详细成本: 采购成本: {result['purchase_cost']:.2f}, 零配件 1 检测成本: {result['inspect_cost_part1']:.2f}, "
              f"零配件 2 检测成本: {result['inspect_cost_part2']:.2f}, 装配成本: {result['assemble_cost']:.2f}, "
              f"成品检测成本: {result['product_cost']:.2f}, 拆解成本: {result['disassembly_cost']:.2f}, "
              f"换货损失: {result['exchange_loss']:.2f}")
        scenario_result.append(result)
    detailed_results.append(scenario_result)

# 打印最佳策略
print("\n 最佳策略:")

optimal_profit_decisions = []
for result in detailed_results:
    best_decision = max(result, key=lambda x: x['profit'])
    print(
        f"情景 {best_decision['decision_str']} -> 总成本: {best_decision['total_cost']:.2f}, 销售额: {best_decision['total_sales']:.2f}, 利润: {best_decision['profit']:.2f}, 次品率: {best_decision['product_defective_rate']:.2%}")
    print(f"详细成本: 采购成本: {best_decision['purchase_cost']:.2f}, 零配件 1 检测成本: {best_decision['inspect_cost_part1']:.2f}, "
          f"零配件 2 检测成本: {best_decision['inspect_cost_part2']:.2f}, 装配成本: {best_decision['assemble_cost']:.2f}, "
          f"成品检测成本: {best_decision['product_cost']:.2f}, 拆解成本: {best_decision['disassembly_cost']:.2f}, "
          f"换货损失: {best_decision['exchange_loss']:.2f}")
    optimal_profit_decisions.append(best_decision)
```

7.4.3 问题三代码

Q3.py

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
import pandas as pd
from matplotlib import rcParams

# 配置中文字体
```

```
rcParams['font.sans-serif'] = ['SimHei']
rcParams['axes.unicode_minus'] = False

# 零配件和半成品及成品数据
parts_data = {
    '零配件 A': {'次品率': 0.1, '单价': 2, '检测费': 1},
    '零配件 B': {'次品率': 0.1, '单价': 8, '检测费': 1},
    '零配件 C': {'次品率': 0.1, '单价': 12, '检测费': 2},
    '零配件 D': {'次品率': 0.1, '单价': 2, '检测费': 1},
    '零配件 E': {'次品率': 0.1, '单价': 8, '检测费': 1},
    '零配件 F': {'次品率': 0.1, '单价': 12, '检测费': 2},
    '零配件 G': {'次品率': 0.1, '单价': 8, '检测费': 1},
    '零配件 H': {'次品率': 0.1, '单价': 12, '检测费': 2},
}

sub_assemblies = {
    '组件 A': {'次品率': 0.1, '组装费': 8, '检测费': 4, '拆解费': 6},
    '组件 B': {'次品率': 0.1, '组装费': 8, '检测费': 4, '拆解费': 6},
    '组件 C': {'次品率': 0.1, '组装费': 8, '检测费': 4, '拆解费': 6},
}

finished_product = {
    '产品': {'次品率': 0.1, '组装费': 8, '检测费': 6, '拆解费': 10, '售价': 200, '退换损失': 40}
}

# 成本计算函数
def estimate_total_cost(parts, subs, final, inspect_parts, inspect_subs, inspect_final, dismantle_final,
dismantle_subs):
    cost_sum = 0
    combined_defective_rate = 1

    # 计算零配件成本
    for idx, (part, part_info) in enumerate(parts.items()):
        part_cost = part_info['单价']
        defect_rate = part_info['次品率']
        if inspect_parts[idx]:
            defect_rate *= 0.5
            part_cost += part_info['检测费']

        cost_sum += part_cost
        combined_defective_rate *= (1 - defect_rate)

    # 计算半成品成本
    for idx, (sub, sub_info) in enumerate(subs.items()):
```

```
            sub_cost = sub_info['组装费']
            defect_rate = sub_info['次品率']
            if inspect_subs[idx]:
                defect_rate *= 0.5
                sub_cost += sub_info['检测费']

            cost_sum += sub_cost
            combined_defective_rate *= (1 - defect_rate)

            if dismantle_subs[idx]:
                cost_sum += sub_info['拆解费']

        # 计算成品成本
        final_defective_rate = final['产品']['次品率'] * (0.5 if inspect_final else 1)
        final_cost = final['产品']['组装费']

        if inspect_final:
            final_cost += final['产品']['检测费']

        cost_sum += final_cost
        combined_defective_rate = 1 - (combined_defective_rate * (1 - final_defective_rate))

        # 添加调换损失和拆解费用
        replacement_loss = final['产品']['退换损失'] * combined_defective_rate
        cost_sum += replacement_loss

        if dismantle_final:
            cost_sum += final['产品']['拆解费']

        return cost_sum, combined_defective_rate

# 生成所有策略的组合
part_options = list(product([True, False], repeat=8))
sub_options = list(product([True, False], repeat=3))
final_options = list(product([True, False], repeat=2))
dismantle_sub_options = list(product([True, False], repeat=3))

strategy_info = []
outcomes = []

# 描述策略
def describe_strategy(inspect_parts, inspect_subs, inspect_final, dismantle_final, dismantle_subs):
    desc = ""
    for idx, inspect in enumerate(inspect_parts):
```

```
        desc += f"检测零配件 {idx + 1}，" if inspect else f"不检测零配件 {idx + 1}，"
    for idx, inspect in enumerate(inspect_subs):
        desc += f"检测组件 {idx + 1}，" if inspect else f"不检测组件 {idx + 1}，"
    desc += "检测成品，" if inspect_final else "不检测成品，"
    desc += "拆解成品，" if dismantle_final else "不拆解成品，"
    for idx, dismantle in enumerate(dismantle_subs):
        desc += f"拆解组件 {idx + 1}，" if dismantle else f"不拆解组件 {idx + 1}，"
    return desc

# 计算每种策略的结果
strategy_id = 1
for part_comb in part_options:
    for sub_comb in sub_options:
        for final_comb in final_options:
            for dismantle_sub_comb in dismantle_sub_options:
                inspect_parts = part_comb
                inspect_subs = sub_comb
                inspect_final = final_comb[0]
                dismantle_final = final_comb[1]
                dismantle_subs = dismantle_sub_comb

                total_cost, defective_rate = estimate_total_cost(parts_data, sub_assemblies, finished_product,
                                                                 inspect_parts, inspect_subs, inspect_final,
                                                                 dismantle_final, dismantle_subs)

                strategy_desc = describe_strategy(inspect_parts, inspect_subs, inspect_final, dismantle_final,
                                                  dismantle_subs)
                outcomes.append([strategy_id, strategy_desc, total_cost, defective_rate])
                strategy_id += 1

# 转换为 DataFrame
df_strategies = pd.DataFrame(outcomes, columns=['策略编号', '策略描述', '总成本', '成品次品率'])

# 保存到 Excel
df_strategies.to_excel('策略结果.xlsx', index=False)
print("策略已保存到文件 '策略结果.xlsx'。")

# 绘制成本和次品率图表
plt.figure(figsize=(10, 6))
plt.bar(df_strategies['策略编号'], df_strategies['总成本'], color='lightblue', label="总成本")
```

```
plt.xlabel("策略编号")
plt.ylabel("总成本（元)")
plt.title("策略的总成本比较")
plt.grid(axis='y')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(df_strategies['策略编号'], df_strategies['成品次品率'], color='coral', label="成品次品率")
plt.xlabel("策略编号")
plt.ylabel("成品次品率")
plt.title("策略的成品次品率比较")
plt.grid(True)
plt.legend()
plt.show()
```

Q3_do.py

```
import numpy as np
import pandas as pd

# 读取策略结果文件(假设前面的代码已生成了策略结果)
# 如果你有已生成的结果可以直接从 df 中读取，否则用 pd.read_excel('策略结果.xlsx') 读取
df = pd.read_excel('策略结果.xlsx')

# 提取总成本和成品次品率两列
data = df[['总成本', '成品次品率']].values


# 第一步：数据标准化(这里采用极差标准化)
def normalize(data):
    min_vals = np.min(data, axis=0)
    max_vals = np.max(data, axis=0)
    norm_data = (data - min_vals) / (max_vals - min_vals)
    return norm_data


norm_data = normalize(data)


# 第二步：计算熵值
def calculate_entropy(norm_data):
    # 防止计算时出现 log(0)，将 0 替换为一个极小值
    norm_data = np.where(norm_data == 0, 1e-12, norm_data)
```

```
    # 计算每个策略在各指标下的占比
    proportion = norm_data / np.sum(norm_data, axis=0)

    # 计算熵值
    k = 1 / np.log(len(norm_data))
    entropy = -k * np.sum(proportion * np.log(proportion), axis=0)

    return entropy


entropy = calculate_entropy(norm_data)


# 第三步：计算权重
def calculate_weights(entropy):
    d = 1 - entropy  # 熵越大，d 值越小，权重越小
    weights = d / np.sum(d)  # 归一化权重
    return weights


weights = calculate_weights(entropy)

# 打印熵值和权重
print("各个指标的熵值为：", entropy)
print("各个指标的权重为：", weights)


# 第四步：计算综合评价值(加权得分)
def calculate_evaluation(norm_data, weights):
    evaluation = np.dot(norm_data, weights)
    return evaluation


evaluation_scores = calculate_evaluation(norm_data, weights)

# 将评价值添加到 DataFrame 中
df['评价值'] = evaluation_scores

# 根据评价值排序，评价值越小越好
df = df.sort_values(by='评价值')

# 保存新的策略结果到 Excel
df.to_excel('带评价值的策略结果.xlsx', index=False)
```

```
print("策略结果已保存到 '带评价值的策略结果.xlsx' 文件中。")

# 打印最优策略
best_strategy = df.iloc[0]  # 获取评价值最小的策略
print("最优策略为：")
print(best_strategy)
```

7.4.4 问题四代码

Q4.py

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.stats.proportion import proportion_confint
from tqdm import tqdm

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei']  # 使用黑体字体显示中文
plt.rcParams['axes.unicode_minus'] = False  # 正常显示负号


# 自适应抽样检测函数，用于直接点估计次品率
def adaptive_sampling(true_defective_rate, alpha, max_samples=1000):
    n = 0  # 当前抽取的样本数
    defective_count = 0  # 次品数量

    while n < max_samples:
        sample = np.random.binomial(1, true_defective_rate)
        defective_count += sample
        n += 1
        current_defective_rate = defective_count / n

        ci_lower, ci_upper = proportion_confint(defective_count, n, alpha=alpha, method='beta')

        # 判断是否满足接受或拒绝的条件
        if ci_upper < true_defective_rate:
            return "接收该批次", n, current_defective_rate
        elif ci_lower > true_defective_rate:
            return "拒绝该批次", n, current_defective_rate

    return "无法确定", n, current_defective_rate


# 蒙特卡洛模拟函数，使用点估计
def monte_carlo_simulation(detected_defective_rate, alpha, max_samples=1000, num_simulations=1000):
    estimated_defective_rate_list = []
```

```
    for _ in tqdm(range(num_simulations)):
        # 使用检测到的次品率作为 "真实次品率" 来生成数据
        decision, _, estimated_defective_rate = adaptive_sampling(detected_defective_rate, alpha, max_samples)

        # 记录每次模拟中的次品率估计
        estimated_defective_rate_list.append(estimated_defective_rate)

    return estimated_defective_rate_list


# 可视化结果
def plot_simulation_results(defective_rate_list_5, defective_rate_list_10, defective_rate_list_20, filename):
    fig, ax = plt.subplots()

    # 绘制三种次品率的直方图
    ax.hist(defective_rate_list_5, bins=30, alpha=0.5, label="5% 样品率")
    ax.hist(defective_rate_list_10, bins=30, alpha=0.5, label="10% 样品率")
    ax.hist(defective_rate_list_20, bins=30, alpha=0.5, label="20% 样品率")

    # 输出平均值
    avg_5 = np.mean(defective_rate_list_5)
    avg_10 = np.mean(defective_rate_list_10)
    avg_20 = np.mean(defective_rate_list_20)

    print(f'5% 样品率对应的真实次品率: {avg_5:.4f}')
    print(f'10% 样品率对应的真实次品率: {avg_10:.4f}')
    print(f'20% 样品率对应的真实次品率: {avg_20:.4f}')

    # 设置图表细节
    ax.set_xlabel('估计的次品率')
    ax.set_ylabel('频率')
    ax.set_title('不同检测次品率的估计次品率分布')
    ax.legend()

    # 保存图像
    plt.savefig(filename, dpi=500, transparent=True)  # 高分辨率保存图片
    plt.show()


# 参数设定
num_simulations = 1000  # 模拟次数
max_samples = 1000
```

```
# 设置后验检测到的次品率
detected_defective_rates = [0.05, 0.10, 0.20]
alpha = 0.05  # 95% 的置信度

# 分别进行蒙特卡洛模拟
estimated_defective_rate_list_5 = monte_carlo_simulation(detected_defective_rates[0], alpha, max_samples,
                                                         num_simulations)
estimated_defective_rate_list_10 = monte_carlo_simulation(detected_defective_rates[1], alpha, max_samples,
                                                          num_simulations)
estimated_defective_rate_list_20 = monte_carlo_simulation(detected_defective_rates[2], alpha, max_samples,
                                                          num_simulations)

# 绘制结果
plot_simulation_results(estimated_defective_rate_list_5, estimated_defective_rate_list_10,
                        estimated_defective_rate_list_20, 'point_estimate_real_defective_rate.png')
```

Q4_reQ2.py

```
import itertools

# 定义所有决策的组合 (16 种可能的组合)
decisions_combinations = list(itertools.product([0, 1], repeat=4))

# 根据表格中的数据重新定义情景参数
scenarios_corrected = [
    {'defective_rate_1': 0.1246, 'defective_rate_2': 0.1246, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 6, 'disassembly_cost': 5, 'product_failure_rate': 0.1246},

    {'defective_rate_1': 0.2202, 'defective_rate_2': 0.2202, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 6, 'disassembly_cost': 5, 'product_failure_rate': 0.2202},

    {'defective_rate_1': 0.1246, 'defective_rate_2': 0.1246, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 30, 'disassembly_cost': 5, 'product_failure_rate': 0.1246},

    {'defective_rate_1': 0.2202, 'defective_rate_2': 0.2202, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 1, 'inspect_2': 1, 'assemble_cost': 6, 'inspect_product': 2,
     'market_price': 56, 'exchange_loss': 30, 'disassembly_cost': 5, 'product_failure_rate': 0.2202},
```

```
    {'defective_rate_1': 0.1246, 'defective_rate_2':0.2202, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 8, 'inspect_2': 1, 'assemble_cost': 6, 'inspect_product': 2,
     'market_price': 56, 'exchange_loss': 10, 'disassembly_cost': 5, 'product_failure_rate': 0.1246},

    {'defective_rate_1': 0.0651, 'defective_rate_2': 0.0651, 'purchase_1': 4, 'purchase_2': 18,
     'inspect_1': 2, 'inspect_2': 3, 'assemble_cost': 6, 'inspect_product': 3,
     'market_price': 56, 'exchange_loss': 10, 'disassembly_cost': 40, 'product_failure_rate': 0.0651}
]

# 成本计算函数，计算利润、次品率，并返回详细的成本分项
def calculate_profit_and_defective_rate_detailed(scenario, decision, quantity_part1=1000,
quantity_part2=1000, sub_decision_str=None):
    d1, d2 = scenario['defective_rate_1'], scenario['defective_rate_2']
    c1, c2 = scenario['purchase_1'], scenario['purchase_2']
    i1, i2 = scenario['inspect_1'], scenario['inspect_2']
    ac = scenario['assemble_cost']
    ip = scenario['inspect_product']
    mp = scenario['market_price']
    el = scenario['exchange_loss']
    dc = scenario['disassembly_cost']
    product_failure_rate = scenario['product_failure_rate']

    # 解析决策组合 (d1 检测, d2 检测, 成品检测, 拆解)
    detect_part1, detect_part2, detect_product, disassemble = decision

    # 1. 固定采购成本：无论是否检测，所有零配件都先买回来
    purchase_cost_part1 = c1 * quantity_part1
    purchase_cost_part2 = c2 * quantity_part2
    purchase_cost = purchase_cost_part1 + purchase_cost_part2

    # 2. 零配件 1 检测成本
    if detect_part1:
        inspect_cost_part1 = i1 * quantity_part1  # 检测所有的零配件，无论是否合格
        part1_defective_rate = 0  # 检测后次品率变为 0
        good_quantity_part1 = quantity_part1 * (1 - d1)  # 只留下合格的零配件 1
        # 检测的零配件直接丢弃
        quantity_part1 = good_quantity_part1  # 更新数量
    else:
        inspect_cost_part1 = 0  # 不检测时无检测费用
        part1_defective_rate = d1  # 不检测次品率保持原样

    # 3. 零配件 2 检测成本
    if detect_part2:
```

```
            inspect_cost_part2 = i2 * quantity_part2  # 检测所有的零配件，无论是否合格
            part2_defective_rate = 0  # 检测后次品率变为 0
            good_quantity_part2 = quantity_part2 * (1 - d2)  # 只留下合格的零配件 2
            # 检测的零配件直接丢弃
            quantity_part2 = good_quantity_part2  # 更新数量
        else:
            inspect_cost_part2 = 0  # 不检测时无检测费用
            part2_defective_rate = d2  # 不检测次品率保持原样

        # 4. 装配逻辑：根据两个零配件的质量情况决定成品次品率
        if detect_part1 and detect_part2:
            total_quantity = min(good_quantity_part1, good_quantity_part2)  # 两个零配件都合格时，用最少的
            product_defective_rate = product_failure_rate  # 仅考虑装配失败的次品率
        elif detect_part1 and not detect_part2:
            total_quantity = good_quantity_part1  # 零配件 1 合格的数量
            product_defective_rate = part2_defective_rate + (1 - part2_defective_rate) * product_failure_rate  # 考虑未检测零配件的次品率
        elif not detect_part1 and detect_part2:
            total_quantity = good_quantity_part2  # 零配件 2 合格的数量
            product_defective_rate = part1_defective_rate + (1 - part1_defective_rate) * product_failure_rate  # 考虑未检测零配件的次品率
        else:
            total_quantity = min(quantity_part1, quantity_part2)  # 两个零配件都可能次品
            # 如果任意一个零配件次品，成品必然是次品
            product_defective_rate = 1 - (1 - part1_defective_rate) * (1 - part2_defective_rate)
            # 如果两个零配件都是合格的，考虑装配失败的次品率
            product_defective_rate += (1 - product_defective_rate) * product_failure_rate

        # 5. 装配成本：无论装配是否生成次品，装配都需要费用
        assemble_cost = total_quantity * ac  # 实际装配成本

        # 6. 成品检测逻辑
        if detect_product:
            # 如果成品被检测，次品进入拆解
            product_cost = ip * total_quantity  # 检测所有成品
            defective_quantity = total_quantity * product_defective_rate  # 检测出次品的数量
            sold_quantity = total_quantity * (1 - product_defective_rate)  # 售出的正品数量
            total_sales = sold_quantity * mp  # 只有合格成品进入市场
            product_sale_loss = 0  # 检测后无销售损失
        else:
            # 如果不检测，次品直接进入市场
            product_cost = 0  # 不检测时无检测费用
            sold_quantity = total_quantity * (1 - product_defective_rate)  # 售出的正品数量
```

```
        total_sales = sold_quantity * mp  # 售出的正品销售额
        # 换货损失 = 调换损失(次品流入市场)
        product_sale_loss = product_defective_rate * el * total_quantity  # 换货损失=调换损失
        defective_quantity = total_quantity * product_defective_rate  # 流入市场的次品

    # 打印主序列中的正品售出数量和对应的销售额
    print(f"主序列售出正品数量: {sold_quantity:.2f}，对应销售额: {total_sales:.2f}")

    # 7. 计算进入子序列的正品零配件：正品零配件总数 - 已生产的正品成品个数
    total_good_part1 = quantity_part1 * (1 - part1_defective_rate)  # 零配件 1 的总正品数量
    total_good_part2 = quantity_part2 * (1 - part2_defective_rate)  # 零配件 2 的总正品数量
    # 未被使用的正品零配件个数
    unused_good_part1 = total_good_part1 - sold_quantity
    unused_good_part2 = total_good_part2 - sold_quantity
    # 进入子序列的正品零配件数为两个正品零配件中较少的那个
    good_parts_for_subsequence = min(unused_good_part1, unused_good_part2)

    # 8. 换货后的拆解逻辑：如果选择拆解
    if disassemble == 1:
        disassembly_cost = defective_quantity * dc  # 拆解所有次品的费用
        # 如果不检测，流入市场的次品也进入拆解
        if not detect_product:
            disassembly_cost += product_defective_rate * total_quantity * dc  # 流入市场的次品拆
解
    else:
        disassembly_cost = 0  # 没有拆解则无拆解费用

    # 9. 拆解后的子决策：子序列继承主序列的零配件检测和装配结果
    if disassemble == 1 and defective_quantity > 0:
        # 根据主序列的检测结果判断子序列中的零配件
        if detect_part1 and detect_part2:
            # 主序列检测了零配件，子序列只有因装配失败的正品零配件
            sub_defective_rate_1 = 0  # 子序列中的零配件都是正品
            sub_defective_rate_2 = 0
        else:
            # 主序列未检测，子序列可能有次品零配件
            sub_defective_rate_1 = d1  # 继承主序列的次品率
            sub_defective_rate_2 = d2

        # 拆解后的子决策：零部件检测决策相反，成品检测决策保持原决策或取反
        new_decision_part1 = 1 - detect_part1
        new_decision_part2 = 1 - detect_part2

        # 子决策，保留成品检测决策不变或取反
```

```
            new_decision_combinations = [(new_decision_part1, new_decision_part2, detect_product, 0),
# 保持成品检测决策
                                         (new_decision_part1, new_decision_part2, 1 -
detect_product, 0)]  # 取反成品检测决策

            sub_results = []
            for sub_decision in new_decision_combinations:
                sub_decision_str = f"{''.join(map(str, decision))}|{''.join(map(str, sub_decision))}"
                # 子决策计算时根据主序列继承次品率
                sub_result = calculate_profit_and_defective_rate_detailed(
                    {
                        'defective_rate_1': sub_defective_rate_1,
                        'defective_rate_2': sub_defective_rate_2,
                        'purchase_1': c1,
                        'purchase_2': c2,
                        'inspect_1': i1,
                        'inspect_2': i2,
                        'assemble_cost': ac,
                        'inspect_product': ip,
                        'market_price': mp,
                        'exchange_loss': el,
                        'disassembly_cost': dc,
                        'product_failure_rate': product_failure_rate
                    },
                    sub_decision,
                    quantity_part1=good_parts_for_subsequence,  # 使用计算出的剩余正品零配件数
量
                    quantity_part2=good_parts_for_subsequence,
                    sub_decision_str=sub_decision_str
                )
                sub_result['purchase_cost'] = 0  # 子决策没有采购成本

                # 子决策中的逻辑调整：
                # 如果子决策再次检测，次品被丢弃；如果不检测，次品进入市场，产生换货损失
                if sub_decision[2]:  # 成品再次检测
                    sub_result['exchange_loss'] = 0  # 检测后不会有次品进入市场
                else:
                    # 不检测，次品进入市场，计算换货损失(包括调换损失)
                    defective_rate = sub_result['product_defective_rate']
                    sub_result['exchange_loss'] = defective_rate * el * good_parts_for_subsequence  #
只算调换损失

                sub_results.append(sub_result)
```

```
            # 打印子序列中的正品售出数量和对应的销售额
            for sub_result in sub_results:
                sub_decision_str = sub_result.get('decision_str')
                print(f"子序列 {sub_decision_str} -> 售出正品数量: {sub_result['total_sales'] / mp:.2f},
对应销售额: {sub_result['total_sales']:.2f}")

            # 获取子分支中最佳利润方案
            best_sub_result = max(sub_results, key=lambda x: x['profit'])

            # 将子分支的成本、销售和利润与主决策相加
            total_sales += best_sub_result['total_sales']  # 加上子分支的销售收入
            disassembly_cost += best_sub_result['disassembly_cost']  # 加上子分支的拆解费用
            assemble_cost += best_sub_result['assemble_cost']  # 加上子分支的装配费用
            inspect_cost_part1 += best_sub_result['inspect_cost_part1']  # 加上子分支的零配件 1 检测费
用
            inspect_cost_part2 += best_sub_result['inspect_cost_part2']  # 加上子分支的零配件 2 检测费
用
            product_cost += best_sub_result['product_cost']  # 加上子分支的成品检测费用
            product_sale_loss += best_sub_result['exchange_loss']  # 子分支的换货损失
            sub_decision_str = best_sub_result.get('decision_str', sub_decision_str)

    # 总成本：采购成本 + 检测成本 + 装配成本 + 拆解成本 + 换货损失
    total_cost = (purchase_cost + inspect_cost_part1 + inspect_cost_part2 +
                  assemble_cost + product_cost + disassembly_cost + product_sale_loss)

    # 利润 = 销售收入 - 成本
    profit = total_sales - total_cost

    # 返回详细的成本分项
    return {
        'profit': profit,
        'product_defective_rate': product_defective_rate,
        'total_cost': total_cost,
        'total_sales': total_sales,
        'purchase_cost': purchase_cost,
        'inspect_cost_part1': inspect_cost_part1,
        'inspect_cost_part2': inspect_cost_part2,
        'assemble_cost': assemble_cost,
        'product_cost': product_cost,
        'disassembly_cost': disassembly_cost,
        'exchange_loss': product_sale_loss,
        'decision_str': sub_decision_str or ''.join(map(str, decision))
    }
```

```
# 对每个情景计算 24 种决策的详细结果
detailed_results = []
for index, scenario in enumerate(scenarios_corrected):
    print(f"\n 情景 {index + 1} 的 24 种决策详情:")
    scenario_result = []
    for decision in decisions_combinations:
        result = calculate_profit_and_defective_rate_detailed(scenario, decision)
        decision_str = result['decision_str']
        print(f" 决 策 {decision_str} -> 总 成 本 : {result['total_cost']:.2f}, 销 售 额 : {result['total_sales']:.2f}, 利润: {result['profit']:.2f}, 次品率: {result['product_defective_rate']:.2%}")
        print(f" 详 细 成 本 : 采 购 成 本 : {result['purchase_cost']:.2f}, 零 配 件 1 检 测 成 本 : {result['inspect_cost_part1']:.2f}, "
              f" 零 配 件 2 检 测 成 本 : {result['inspect_cost_part2']:.2f}, 装 配 成 本 : {result['assemble_cost']:.2f}, "
              f" 成 品 检 测 成 本 : {result['product_cost']:.2f}, 拆 解 成 本 : {result['disassembly_cost']:.2f}, "
              f"换货损失: {result['exchange_loss']:.2f}")
        scenario_result.append(result)
    detailed_results.append(scenario_result)

# 打印最佳策略
print("\n 问题四条件下的最佳策略:")

optimal_profit_decisions = []
for result in detailed_results:
    best_decision = max(result, key=lambda x: x['profit'])
    print(
        f"情景 {best_decision['decision_str']} -> 总成本: {best_decision['total_cost']:.2f}, 销售额: {best_decision['total_sales']:.2f}, 利 润 : {best_decision['profit']:.2f}, 次 品 率 : {best_decision['product_defective_rate']:.2%}")
    print(f" 详 细 成 本 : 采 购 成 本 : {best_decision['purchase_cost']:.2f}, 零 配 件 1 检 测 成 本 : {best_decision['inspect_cost_part1']:.2f}, "
          f" 零 配 件 2 检 测 成 本 : {best_decision['inspect_cost_part2']:.2f}, 装 配 成 本 : {best_decision['assemble_cost']:.2f}, "
          f" 成 品 检 测 成 本 : {best_decision['product_cost']:.2f}, 拆 解 成 本 : {best_decision['disassembly_cost']:.2f}, "
          f"换货损失: {best_decision['exchange_loss']:.2f}")
    optimal_profit_decisions.append(best_decision)
```

Q4_reQ3.py

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
```

```
import pandas as pd
from matplotlib import rcParams

# 配置中文字体
rcParams['font.sans-serif'] = ['SimHei']
rcParams['axes.unicode_minus'] = False

# 零配件和半成品及成品数据
parts_data = {
    '零配件 A': {'次品率': 0.1246, '单价': 2, '检测费': 1},
    '零配件 B': {'次品率': 0.1246, '单价': 8, '检测费': 1},
    '零配件 C': {'次品率': 0.1246, '单价': 12, '检测费': 2},
    '零配件 D': {'次品率': 0.1246, '单价': 2, '检测费': 1},
    '零配件 E': {'次品率': 0.1246, '单价': 8, '检测费': 1},
    '零配件 F': {'次品率': 0.1246, '单价': 12, '检测费': 2},
    '零配件 G': {'次品率': 0.1246, '单价': 8, '检测费': 1},
    '零配件 H': {'次品率': 0.1246, '单价': 12, '检测费': 2},
}

sub_assemblies = {
    '组件 A': {'次品率': 0.1246, '组装费': 8, '检测费': 4, '拆解费': 6},
    '组件 B': {'次品率': 0.1246, '组装费': 8, '检测费': 4, '拆解费': 6},
    '组件 C': {'次品率': 0.1246, '组装费': 8, '检测费': 4, '拆解费': 6},
}

finished_product = {
    '产品': {'次品率': 0.1246, '组装费': 8, '检测费': 6, '拆解费': 10, '售价': 200, '退换损失': 40}
}

# 成本计算函数
def estimate_total_cost(parts, subs, final, inspect_parts, inspect_subs, inspect_final, dismantle_final,
dismantle_subs):
    cost_sum = 0
    combined_defective_rate = 1

    # 计算零配件成本
    for idx, (part, part_info) in enumerate(parts.items()):
        part_cost = part_info['单价']
        defect_rate = part_info['次品率']
        if inspect_parts[idx]:
            defect_rate *= 0.5
            part_cost += part_info['检测费']

        cost_sum += part_cost
```

```
            combined_defective_rate *= (1 - defect_rate)

        # 计算半成品成本
        for idx, (sub, sub_info) in enumerate(subs.items()):
            sub_cost = sub_info['组装费']
            defect_rate = sub_info['次品率']
            if inspect_subs[idx]:
                defect_rate *= 0.5
                sub_cost += sub_info['检测费']

            cost_sum += sub_cost
            combined_defective_rate *= (1 - defect_rate)

            if dismantle_subs[idx]:
                cost_sum += sub_info['拆解费']

        # 计算成品成本
        final_defective_rate = final['产品']['次品率'] * (0.5 if inspect_final else 1)
        final_cost = final['产品']['组装费']

        if inspect_final:
            final_cost += final['产品']['检测费']

        cost_sum += final_cost
        combined_defective_rate = 1 - (combined_defective_rate * (1 - final_defective_rate))

        # 添加调换损失和拆解费用
        replacement_loss = final['产品']['退换损失'] * combined_defective_rate
        cost_sum += replacement_loss

        if dismantle_final:
            cost_sum += final['产品']['拆解费']

        return cost_sum, combined_defective_rate

    # 生成所有策略的组合
    part_options = list(product([True, False], repeat=8))
    sub_options = list(product([True, False], repeat=3))
    final_options = list(product([True, False], repeat=2))
    dismantle_sub_options = list(product([True, False], repeat=3))

    strategy_info = []
    outcomes = []
```

```
# 描述策略
def describe_strategy(inspect_parts, inspect_subs, inspect_final, dismantle_final, dismantle_subs):
    desc = ""
    for idx, inspect in enumerate(inspect_parts):
        desc += f"检测零配件 {idx + 1}，" if inspect else f"不检测零配件 {idx + 1}，"
    for idx, inspect in enumerate(inspect_subs):
        desc += f"检测组件 {idx + 1}，" if inspect else f"不检测组件 {idx + 1}，"
    desc += "检测成品，" if inspect_final else "不检测成品，"
    desc += "拆解成品，" if dismantle_final else "不拆解成品，"
    for idx, dismantle in enumerate(dismantle_subs):
        desc += f"拆解组件 {idx + 1}，" if dismantle else f"不拆解组件 {idx + 1}，"
    return desc

# 计算每种策略的结果
strategy_id = 1
for part_comb in part_options:
    for sub_comb in sub_options:
        for final_comb in final_options:
            for dismantle_sub_comb in dismantle_sub_options:
                inspect_parts = part_comb
                inspect_subs = sub_comb
                inspect_final = final_comb[0]
                dismantle_final = final_comb[1]
                dismantle_subs = dismantle_sub_comb

                total_cost, defective_rate = estimate_total_cost(parts_data, sub_assemblies,
finished_product,
                                                                 inspect_parts,
inspect_subs, inspect_final,
                                                                 dismantle_final,
dismantle_subs)

                strategy_desc = describe_strategy(inspect_parts, inspect_subs, inspect_final,
dismantle_final,
                                                  dismantle_subs)
                outcomes.append([strategy_id, strategy_desc, total_cost, defective_rate])
                strategy_id += 1

# 转换为 DataFrame
df_strategies = pd.DataFrame(outcomes, columns=['策略编号', '策略描述', '总成本', '成品次品率'])

# 保存到 Excel
df_strategies.to_excel('re 策略结果.xlsx', index=False)
print("策略已保存到文件 're 策略结果.xlsx'。")
```

```
# 绘制成本和次品率图表
plt.figure(figsize=(10, 6))
plt.bar(df_strategies['策略编号'], df_strategies['总成本'], color='lightblue', label="总成本")
plt.xlabel("策略编号")
plt.ylabel("总成本 (元)")
plt.title("策略的总成本比较")
plt.grid(axis='y')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(df_strategies['策略编号'], df_strategies['成品次品率'], color='coral', label="成品次品率")
plt.xlabel("策略编号")
plt.ylabel("成品次品率")
plt.title("策略的成品次品率比较")
plt.grid(True)
plt.legend()
plt.show()
```

Q4_reQ3_do.py

```
import numpy as np
import pandas as pd

# 读取策略结果文件(假设前面的代码已生成了策略结果)
# 如果你有已生成的结果可以直接从 df 中读取，否则用 pd.read_excel('策略结果.xlsx') 读取
df = pd.read_excel('re 策略结果.xlsx')

# 提取总成本和成品次品率两列
data = df[['总成本', '成品次品率']].values


# 第一步：数据标准化(这里采用极差标准化)
def normalize(data):
    min_vals = np.min(data, axis=0)
    max_vals = np.max(data, axis=0)
    norm_data = (data - min_vals) / (max_vals - min_vals)
    return norm_data


norm_data = normalize(data)
```

```
# 第二步：计算熵值
def calculate_entropy(norm_data):
    # 防止计算时出现 log(0)，将 0 替换为一个极小值
    norm_data = np.where(norm_data == 0, 1e-12, norm_data)

    # 计算每个策略在各指标下的占比
    proportion = norm_data / np.sum(norm_data, axis=0)

    # 计算熵值
    k = 1 / np.log(len(norm_data))
    entropy = -k * np.sum(proportion * np.log(proportion), axis=0)

    return entropy


entropy = calculate_entropy(norm_data)


# 第三步：计算权重
def calculate_weights(entropy):
    d = 1 - entropy  # 熵越大，d 值越小，权重越小
    weights = d / np.sum(d)  # 归一化权重
    return weights


weights = calculate_weights(entropy)

# 打印熵值和权重
print("各个指标的熵值为：", entropy)
print("各个指标的权重为：", weights)


# 第四步：计算综合评价值(加权得分)
def calculate_evaluation(norm_data, weights):
    evaluation = np.dot(norm_data, weights)
    return evaluation


evaluation_scores = calculate_evaluation(norm_data, weights)

# 将评价值添加到 DataFrame 中
df['评价值'] = evaluation_scores

# 根据评价值排序，评价值越小越好
```

```
df = df.sort_values(by='评价值')

# 保存新的策略结果到 Excel
df.to_excel('re 带评价值的策略结果.xlsx', index=False)
print("策略结果已保存到 're 带评价值的策略结果.xlsx' 文件中。")

# 打印最优策略
best_strategy = df.iloc[0]  # 获取评价值最小的策略
print("最优策略为：")
print(best_strategy)
```