生产过程抽样检测与优化方案的设计及分析

摘 要

本研究主要围绕企业在生产过程中对零配件及成品的质量检测与成本控制问题展开，通过设计抽样检测方案和构建决策模型，旨在帮助企业实现经济高效的生产管理。

针对问题一：设计经济高效的抽样检测方案，以准确评估零配件次品率，并据此决定特定情形下是否接收整批零配件。针对此问题，本文提出两种抽样检测方案：方案一采用**序贯抽样检验**，逐步抽样并动态调整策略以做出决策；方案二采用**无限总体的一次检验**，利用**不完全Beta函数**，通过**经验公式**估算初始样本量。本文通过**预设零配件真实次品率**，综合比较两种方案的经济性、精确性和效率性。结果发现**方案二所需样本量更少**，尤其在次品率接近标称值时优势显著。最终得到结论：**基于无限总体的一次抽样为更优检测方案**。结合该方案，本文针对两种特定情形为企业提供了最优决策组合。

针对问题二：针对企业不同生产情况制定最优决策，以最小化生产过程中所承担的成本。企业在生产流程中共涉及四个关键决策节点，共形成 **16 种**潜在的决策方案。为确定最优方案，首先针对生产流程的各个环节构建**成本模型**，推导出**总成本函数**作为求解问题的核心目标。随后，采用**动态规划与强化学习相结合的综合模型**进行求解。最终，为企业在不同情境下提供最优的检测、装配、拆解等策略组合，从而实现总成本的最小化。最终，本文得出不同情况下的最优决策组合。通过比对不同情况的结果可得**规律：成品检测**会导致**成品单位成本增加**；而**不拆解不合格成品**导致成品**单位成本最高**；在不检测成品的前提下，**适度检测零配件**能够**减少成品单位成本**。

针对问题三：问题三在问题二的基础上调整了企业生产流程，显著提升了复杂度，其主要体现在增加了半成品生产工序以及零配件的种类的多样性。此生产流程中，共形成 **8192 种**潜在的决策方案。基于此问题背景，本文首先在问题二所建模型基础上，构建出**总体成本函数**作为求解问题的核心目标。随后，采用**加权算法融合模型**进行求解。该模型结合了**动态规划**、**Q-Learning 强化学习**和**贪心算法**三种方法，并引入**动态调整机制**来求取算法权重，从而求解出最优的决策方案。经过求解得到，最优决策方案为：**对零配件不进行检验、对半成品不进行检验、对成品进行检验、以及对不合格成品进行拆解**。其**成品单位成本则为 34.02 元**。从特殊情况引申出**一般规律**：随着零配件个数 n 的增多，装配和检测成本呈现出明显的上升趋势。同时，工序 m 的增加导致生产成本的直接提升。

针对问题四：重新求解了问题2与问题3，其中零配件、半成品及成品的次品率均基于抽样检测数据。该问题难点在于真实次品率难以直接测定，对此，本文采用**不完全Beta分布**进行估计，并利用构建的**无限总体的一次抽样模型**计算次品率的置信区间。通过**蒙特卡洛模拟**精确更新真实次品率后，在问题二与问题三的建模基础上，分别求解对应情况下的最优决策组合，以实现总成本最小化。经过求解，问题二结果见表；问题三最佳方案为：**对零配件不进行检验、对半成品不进行检验、对成品进行检验、以及对不合格成品进行拆解**，其**总成本为10498.58元**，成品单位成本则为**10.50元**。

关键词：序贯抽样检验；无限总体的一次检验；动态规划；强化学习；加权算法融合模型

一、问题重述

1.1 问题背景

某企业生产需要分别购买两种零配件（零配件1和零配件2），并将两个零配件装配成成品。在装配的成品中，若其中一个零配件不合格，则成品一定不合格；若两个零配件均合格，则成品也不一定合格。对于不合格成品，企业面临报废与拆解，或者对其进行拆解，拆解过程不会对零配件造成损坏，但需要承担拆解费用。

1.2 解决问题

问题一：供应商提供一批零配件（零配件1或零配件2），声称其次品率不会超过标称值。企业需设计出检测次数尽可能少的抽样检测方案，以决定是否接收该批零配件。检测费用由企业自行承担。

如果标称值为10%，根据抽样检测方案，针对以下两种情形，分别给出具体结果：

(1) 在95%的信度下认定零配件次品率超过标称值，则拒收这批零配件；

(2) 在90%的信度下认定零配件次品率不超过标称值，则接收这批零配件。

问题二：已知两种零配件和成品次品率，请为企业生产过程的各个阶段做出相应决策：

(1) 对零配件（零配件 1 和/或零配件 2）是否进行检测，如果对某种零配件不检测，这种零配件将直接进入到装配环节；否则将检测出的不合格零配件丢弃；

(2) 对装配好的每一件成品是否进行检测，如果不检测，装配后的成品直接进入到市场；否则只有检测合格的成品进入到市场；

(3) 对检测出的不合格成品是否进行拆解，如果不拆解，直接将不合格成品丢弃；否则对拆解后的零配件，重复步骤(1)和步骤(2)；

(4) 对用户购买的不合格品，企业将无条件予以调换，并产生一定的调换损失（如物流成本、企业信誉等）。对退回的不合格品，重复步骤(3)。

请根据所做的决策，对所提供情形给出具体的决策方案，并给出决策的依据及相应的指标结果。

问题三：对 m 道工序、n 个零配件，已知零配件、半成品和成品的次品率，重复问题2，给出生产过程的决策方案。结合所提供的2道工序、8个零配件的组装情况与具体数值，给出具体的决策方案，以及决策的依据及相应指标。

问题四：假设问题2和问题3中零配件、半成品和成品的次品率均通过问题一中提及的抽样检测方法得到，请重新完成问题2和问题3。

二、问题分析

2.1 问题整体分析

本题聚焦于企业生产过程中的成本控制与质量管理环节，核心问题在于如何通过有效的检测策略和决策优化，确保零配件及成品的质量，同时控制生产成本。

其中，问题一核心目标在于抽样检测方案设计，根据供应商提供的次品率标称值，设计抽样检测方案以验证其真实性。问题二至问题四核心目标在于生产过程的决策优化，旨在通过明确决策变量（如零配件检测、成品检测、不合格品处理等），构建精细的决策模型，以针对各种特定情境实现成本效益的最大化。

通过这一系列问题的求解，本文旨在为企业提供一套完整的解决方案，以优化生产过程中的质量管理成本控制，确保企业的持续发展和竞争力。

2.2 具体问题分析

2.2.1 问题一分析

问题一要求设计出一种经济高效的抽样检测方案，以评估零配件次品率，并基于此判断是否接收该批零配件。该问题的核心在于制定抽样策略，其既能保证统计学意义又能尽量减少检测次数。为此，本文提出两种方案：方案一采用序贯抽样检测方法，通过逐步增加样本量和依据似然比及决策规则动态调整策略，直至做出决策或达到停止规则，旨在通过优化抽样过程来确定最佳样本量。方案二基于不完全 Beta 函数，先利用经验公式估计初始样本量，再通过不完全 Beta 函数计算置信区间，最后迭代增大样本量直至满足误差控制条件，旨在通过精确计算来确定最佳样本量。针对每种方案，将求解预设真实次品率下所需的样本量，并综合比较经济性、精确性和效率性，确定最优抽样检测方案。最终，结合最优方案为企业作出决策。

2.2.2 问题二分析

问题二要求针对企业的不同生产情况，为其提供最优决策，使其在生产过程中，最终所承担的成本最小。本题核心问题在于企业需在四个关键决策点上做出选择：是否对零配件1进行检测、是否对零配件2进行检测、是否对成品进行检测、以及是否对不合格成品进行拆解，这四个决策点各自包含“执行”与“不执行”两种选项，组合起来共形成16种可能的决策方案。为求解出最优决策，首先需要针对生产流程的各个环节构建详细的成本模型，并在此基础上推导出总成本函数，作为求解问题的核心目标。随后，将采用一个融合了动态规划与强化学习的综合求解模型。这个模型以16种决策方案为基础，通过定义系统状态、可执行的动作以及状态转移函数，同时设计考虑关键成本因素的奖励函数，来系统地探索并寻找最优策略。最终，为企业在不同情境下明确指出最优的检测、装配、拆解等策略，从而实现总成本的最小化。

2.2.3 问题三分析

问题三在问题二的基础上，生产模式由简单的零配件组装成品转变为零配件先组合成半成品，再由半成品生成成品，丰富了决策组合空间，提升了寻找最优决策的难度，并影响了整体生产流程的总成本结构和各阶段成本计算的复杂性。针对此背景，需在问题二模型基础上，全面补充与调整各生产环节的成本构成，构建出问题三在更广泛背景下的总体成本函数作为核心目标函数。为求解最优解，本文采用加权算法融合模型，结合历史数据误差反馈灵活调整动态规划、Q-Learning强化学习和贪心算法的权重，全面评估各算法计算出的成本，最终求解出不同情况下的最优决策方案。

2.2.4 问题四分析

在问题四的设定中，假设问题 2 与问题 3 涉及的零配件、半成品及成品的次品率均源自抽样检测。鉴于真实次品率难以直接测定，本研究采用不完全 Beta 分布，基于抽样数据估计真实值。为精确描述次品率分布，首先利用构建的无限总体的一次抽样模型计算次品率的置信区间，并假定抽样过程遵循不完全 Beta 分布而非均匀分布。随后，应用蒙特卡洛模拟，基于不完全 Beta 分布抽样，以精确更新真实次品率。最后，结合问题二与问题三的总成本函数，分别采用前文构建的动态规划-强化学习联合求解模型与加权算法融合模型，求解最优决策组合，以最小化总成本。

三、模型假设

为简化模型，我们作出以下假设：

1.零配件次品率假定：本文假定零配件 1 与零配件 2 的实际次品率分别是 0.08、

0.09、0.11、0.12、0.13、0.14、0.15 和 0.16；

2.拆解折旧假设：不合格成品拆解后形成的零配件存在折旧：问题二中，折旧率为 0.4；问题三中，折旧率提高至 0.5。

3.半成品合格性假定：只要零配件中的任何一个不合格，则半成品必定不合格；即使零配件均合格，半成品仍可能存在其他原因导致其不合格；

4.半成品不可拆解性：对于检验出的不合格半成品，企业无法选择拆解处理，只能选择丢弃；

5.检验过程的独立性：各类零配件的检验环节是相互独立的。

四、符号说明

此处仅列出全文通用的参数设置，以便读者参考。对于各个问题中模型的具体详细参数说明，请查阅每个章节的第二节内容。

序号	符号	含义
1	N	企业生产成品数
2	n	抽样样本量
3	k	抽样检测出的次品数
4	L	置信区间下界
5	U	置信区间上界
6	W	置信区间宽度
7	E	允许误差
8	$z_{\alpha/2}$	正态分布分位数
9	L	置信区间下界
10	Λ	似然比

五、问题一模型建立与求解

5.1 建模思路

问题一旨在设计一种经济高效的抽样检测方案，用于准确评估零配件的次品率，并据此决定是否接收整批零配件。该问题的关键在于制定一个抽样策略，该策略能在保证统计学意义的同时，最大限度地减少检测次数。

针对此问题，本文提出了两种抽样检测方案：

方案一采用序贯抽样检验方法，通过逐步抽样，以实现最优决策。具体而言，通过逐步增加样本量，并依据似然比和预设决策规则动态调整抽样策略，直至做出明确的接受或拒绝决策，或者达到预设的停止规则。此方法旨在通过逐步优化抽样过程，确定最佳样本量。

方案二则基于不完全 Beta 函数进行无限总体的一次抽样检验，通过初始样本量估计和迭代计算来求解样本量大小。具体而言，首先，利用经验公式初步估计样本量；然后，通过不完全 Beta 函数计算置信区间的上下界；最后，通过迭代过程逐步增大样本量并重新计算置信区间，直至满足设定的误差控制条件。此方法旨在通过精确计算和迭代优化，确定最佳的抽样样本量。

针对每种方案，都将求解在预设的真实次品率水平下所需的抽样样本量。通过综合比较两种方案的经济性、结果精确性和效率性，本文将确定出最优的抽样检测方案。

最后，结合最优的抽样检测方案，为企业在不同情形下做出最优决策。

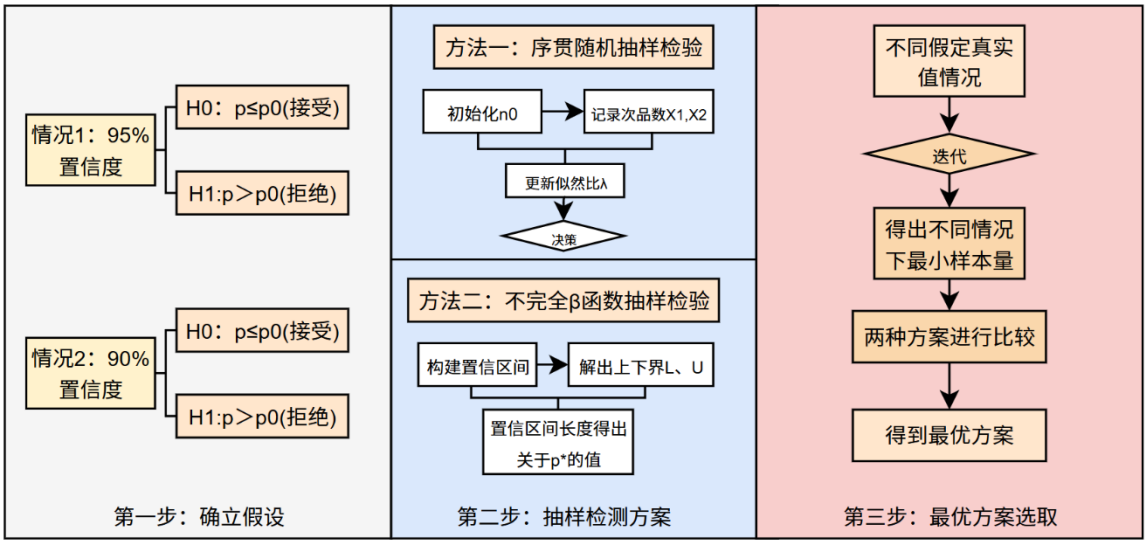

图 1 问题一流程图

5.2 模型建立

5.2.1 基本参数设定

表1 基本参数设定

序号	符号	含义
1	p	实际次品率
2	p_0	厂商标称次品率（10%）
3	k	抽样检测出的次品数
4	n	抽样样本量
5	n_0	初始抽样样本量
6	Δn	迭代时增加的固定样本量
7	n_{max}	样本量上限值
8	X_1	零配件 1 检测出的次品率
9	X_2	零配件 2 检测出的次品率
10	X	检测到的次品数
11	$\hat{p}_1$	当前抽样样本得到的零配件 1 的样本次品率
12	$\hat{p}_2$	当前抽样样本得到的零配件 2 的样本次品率
13	Λ	似然比
14	α	第一类错误概率
15	β	第二类错误概率
16	L	置信区间下界
17	U	置信区间上界
18	W	置信区间宽度
19	E	允许误差
20	$z_{\alpha/2}$	正态分布分位数

5.2.2 原假设与备择假设的设定

为了判断供应商提供的这批零配件的次品率 p 是否符合其宣称的标称值 p_0，需要进行假设检验。本文为两种不同的情形分别设定相应的原假设与备择假设：

1.针对情形一，其中次品率 p 的假设检验为如下类型：

假设Ⅰ H_0：$p \le p_0$ vs H_1：$p > p_0$

其中，H_0 为原假设（符合供应商的宣称，接收这批零配件），H_1 为备择假设（不符合供应商的宣称，拒绝这批零配件）。

2.针对情形二，其中次品率 p 的假设检验为如下类型：

假设Ⅱ H_0：$p \ge p_0$ vs H_1：$p < p_0$

其中，H_0 为原假设（不符合供应商的宣称，拒绝这批零配件），H_1 为备择假设（符合供应商的宣称，接受这批零配件）。

5.2.3 抽样检测方案设计

1.方案一：序贯抽样方法

一般而言，企业通过简单随机抽样检测方法，从供应商提供的零配件中随机抽取 n 个零配件作为样本进行检测，计算出次品数 k 。然后再依据次品数 k 求出其次品率,结合假设检验理论，通过计算出 p 值并与显著性水平比较，最终作出接收或拒收的决策。

但考虑到企业需自行承担检验费用，为保证企业检验方案的经济性，本文采用序贯抽样。序贯抽样对子样本的容量不做事先规定，而是在抽检的过程中，根据先前检验结果的概率来决定，即每次从批次中抽取一组产品，检验后按所确定的规则做出关于接收该批次或拒收该批次或再检验或一组产品的决定的抽样检验。因此，此检验方法能够充分有效地利用信息，减少不必要的抽样，从而降低成本和时间。

（1）抽样方案设计

对于每一批次的零配件1和零配件2，本文结合序贯抽样检验方法设计了一套抽样检验流程，以确保产品质量的同时，最大化检测效率。以下是详细步骤：

步骤1：初始样本抽取：设定初始抽样样本量 n_0；

步骤2：检测与计算：对每个零配件分别进行检测，检测出零配件1与零配件2的次品数分别为 X_1 与 X_2，计算出当前抽样样本次品率 $\hat{p}_1$ 与 $\hat{p}_2$；

步骤3：更新似然比：根据检测出的次品率，结分别更新零配件1和零配件2的似然比 Λ_1 与 Λ_2；

步骤4：做出判断：基于预设的决策阈值 A 与 B，对似然比进行评估，依据决策规则决定是继续抽样、接收该批次还是拒绝该批次；

步骤5：动态调整样本量：若选择继续抽样，则在上一轮次所设定的抽样样本量上增加固定数量 Δn，并继续循环步骤2——步骤4；

步骤6：停止规则：当似然比 Λ_1 和 Λ_2 达到预设临界值，或样本量达到预设上限 n_{max} 时，停止抽样。若在此过程中仍未达成明确的接收或拒绝决策，企业可以根据当前次品率数据进行风险评估，并做出最终的判断。

（2）似然比的更新

对于步骤3，零配件1与零配件2的次品率的似然比计算公式分别为：

$$\Lambda_1 = \frac{(X_1+1)\times(p^*)^{X_1}(1-p_1^*)^{n-X_1}}{(n+1)\times p_0{}^{X_1}(1-p_0)^{n-X_1}} \tag{5.1}$$

$$\Lambda_2 = \frac{(X_2+1)\times(p^*)^{X_2}(1-p_2^*)^{n-X_2}}{(n+1)\times p_0{}^{X_2}(1-p_0)^{n-X_2}} \tag{5.2}$$

其中，X_1 与 X_2 分别为零配件1与零配件2检测出的次品数；n 为抽样样本量；p_0 为标称次品率；p^* 为假设的实际次品率；p_1^* 与 p_2^* 分别为假设的零配件1与零配件2的实际次品率。

（3）决策规则

针对步骤4，本文设定了一套决策规则，以确保在抽样检验过程中能够准确判断零配件批次的质量状况。以下是详细规则及解释：

（1）接收批次

如果 $\Lambda_1 < A$ 或 $\Lambda_2 < A$，则接收该批次；

（2）拒绝批次

如果 $\Lambda_1 > B$ 或 $\Lambda_2 > B$，则拒绝该批次；

（3）继续抽样

如果 $A \le \Lambda_1 \le B$ 且 $A \le \Lambda_2 \le B$，表明当前数据不足以做出明确的接收或拒绝决策。此时，通过增加样本量可以进一步逼近真实次品率，提高决策的准确性，则继续抽样。

其中，A 和 B 是拒绝临界值，分别由风险参数 α 和 β 决定，其公式分别为：

$$A = \frac{\beta}{1-\alpha} \tag{5.3}$$

$$B = \frac{1-\beta}{\alpha} \tag{5.4}$$

其中，α 为第一类错误概率（即拒绝合格品的风险），β 是第二类错误概率（即接受不合格品的风险）。

决策规则如下图所示。

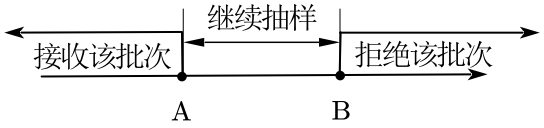

图2 决策规则

2.方案二：基于无限总体的一次抽样方法

（1）二项分布和不完全 Beta 函数的关系

此检测问题本质上是一个二分类事件，即每个零配件的检测结果仅有合格（标记为1）或不合格（标记为0）两种可能。每个零配件的合格与否均可视为独立事件，且次品率保持恒定，由此，本文选择二项分布作为检验统计量的基础。

具体而言，零配件的次品数量 k 服从二项分布，其概率质量函数(PMF)为：

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{5.5}$$

其中，X 为检测到的次品数；p 为真实次品率。

为了判断检测到的次品数 k 是否显著偏离供应商声称的次品率 p_0，需要计算出二项分布的累积分布函数(CDF),即在抽取的样本中，次品数不超过 k 的概率，其可以表示为：

$$P(X \le k) = \sum_{i=0}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{5.6}$$

然而，在零配件检验过程中，样本量的大小往往存在不确定性。单纯依赖中心极限定理进行正态近似可能引发一系列问题，如近似准确性不足、对样本量不确定性敏感、对偏度和峰度反应过度、统计推断稳健性差以及假设检验可靠性降低等。因此，本文选

择不完全 Beta 函数作为计算工具，以精确计算二项分布的累积概率，特别是在样本量较小的情况下。不完全 Beta 函数能够直接给出准确的统计结果，无需依赖正态近似。

具体地，二项分布的累积概率可以通过不完全 Beta 函数来计算：

$$P(X \le k) = I_{1-p}(n-k, k+1) \tag{5.7}$$

其中，k 为在抽样检测过程中检测到的次品数；$I_{1-p}(n-k,k+1)$ 的一般形式为 $I_x(a,b)$，是不完全 β 函数，其函数定义为：

$$I_x(a,b) = \frac{\int_0^x t^{a-1}(1-t)^{b-1}dt}{\int_0^1 t^{a-1}(1-t)^{b-1}dt} = \frac{\int_0^x t^{a-1}(1-t)^{b-1}dt}{B(a,b)} \tag{5.8}$$

其中，$B(a,b)$ 是 Beta 函数的标准化因子：

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt \tag{5.9}$$

（2）置信区间的构建

为了在给定的置信水平下判断次品率 $p_0 = 0.10$ 是否成立，需要构建基于二项分布和不完全 Beta 函数的置信区间。置信区间需满足特定条件：

$$P(L \le p \le U) = 1-\alpha \tag{5.10}$$

其中，L 为置信区间的下界；U 为置信区间的上界。

对于置信区间的上下界 L 与 U，需通过不完全 Beta 函数求解。具体而言，需找到一个次品率下界 L，使得样本中观察到不多于 k 个次品的概率等于 $\alpha/2$；同时，需找到一个次品率上界 U，使得样本中观察到不多于 k 个次品的概率等于 $1-\alpha/2$。即：

$$I_{1-L}(n-k,k+1) = \frac{\alpha}{2} \tag{5.11}$$

$$I_{1-U}(n-k,k+1) = 1-\frac{\alpha}{2} \tag{5.12}$$

（3）误差控制

为确保估计的次品率在允许误差范围内，本文设定允许误差为 E，其与假定的真实次品率 p^* 相关，可以表示为：

$$E = | p^* - p_0 | \tag{5.13}$$

其中，p^* 为假定的真实次品率；p_0 为标称值。

因此，次品率的置信区间宽度 W 需要满足：

$$W = U - L \le 2E = 2 | p^* - p_0 | \tag{5.14}$$

（4）样本量估算

此时，样本量 n 的估算目标是保证置信区间的宽度 W 不超过误差 $2E$。因此，需找到满足以下条件的最小样本量 n：

1）置信水平控制：次品率的置信区间满足给定的置信水平;

2）误差控制：置信区间的宽度不超过 $2E$。

综上所述，样本量计算步骤为：

步骤 1：初始样本量估计：通过经验公式对样本量 n 进行初步估计：

$$n_0 = \left(\frac{z_{\alpha/2} \cdot \sqrt{p_0(1-p_0)}}{| p^* - p_0 |} \right)^2 \tag{5.15}$$

其中，$z_{\alpha/2}$ 为正态分布的分位数；p_0 为供应商声称的次品率，即 10%；p^* 为假定的零配件的真实次品率；

步骤 2：置信区间的迭代计算：对于初始样本量n_0，使用不完全 Beta 函数来求解置信区间的上下界L和U，即公式(5.11)、(5.12)；

步骤 3：送代求解：如果当前置信区间的宽度W不满足公式(5.14)所述条件，则增大样本量n，并重新计算置信区间的上下界U与L，直到满足所设定的条件。

5.3 模型求解

根据题目所述，标称次品率p_0为 10%；针对情形一，显著性水平α为 0.05，针对情形二，显著性水平α为 0.10。

针对假定的次品率真实值p^*，鉴于实际生产场景中零配件的次品率通常会围绕其标称值上下波动，为了更贴合实际情况并提升研究的精确性，本文选取了八个具体的次品率数值进行分析，分别是 0.08、0.09、0.11、0.12、0.13、0.14、0.15 和 0.16。值得注意的是，若假定真实次品率恰好等于标称值，其需要抽取大量的样本加以证明，即样本量n趋近于无穷大，显然是不切实际的，因此本文并未取 0.10 作为假定的次品率真实值。

5.3.1 抽样检验求解结果

1.序贯抽样检验求解结果

由题目可得，对于情形一，第一类错误概率$\alpha=0.05$，本文根据实际情况假定第二类错误概率$\beta=0.10$，由此计算出临界值$A=0.10$、$B=18$；对于情形二，第一类错误概率$\alpha=0.10$，根据实际情况假定$\beta=0.15$，由此计算出临界值$A=0.16, B=8.5$。

依据前文设定的决策规则，最终得到在不同真实次品率及两种置信水平条件下的抽样样本量n的取值，如下表所示。

表 2 方案一 不同真实次品率 95%置信水平条件下的抽样及假设Ⅰ检验结果

p^*	样本量n	是否接受原假设
0.08	1220	0
0.09	5045	0
0.11	5354	1
0.12	1375	1
0.13	627	1
0.14	361	1
0.15	236	1
0.16	168	1

表 3 方案一 不同真实次品率 90%置信水平条件下的抽样及假设Ⅱ检验结果

p^*	样本量n	是否接受原假设
0.08	756	1
0.09	3127	1
0.11	3319	0
0.12	852	0
0.13	388	0
0.14	224	0
0.15	146	0
0.16	104	0

其中 1 代表拒绝原假设、符合标称值；0 代表接受原假设、不符合标称值。因 95%的置信水平和 90%的置信水平的原假设和备择假设相反，进而在 95%置信水平下，若符合标称值，则接受原假设，反之则拒绝；相反地，在 90%置信水平下，若符合标称值，则拒绝原假设，反之则接受。

为了更直观地展示样本量如何随真实次品率的变化而变化，本文绘制出样本量与真实次品率的关系图：

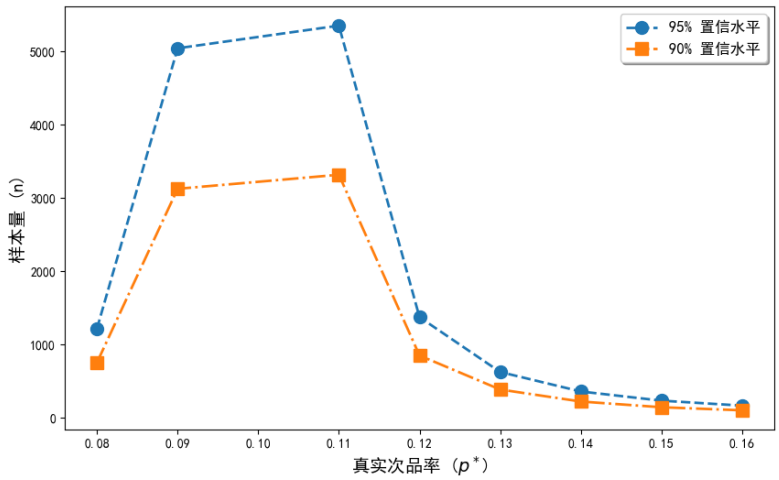

图 3 方案一 样本量与真实次品率的关系图

在此图中，横轴代表真实次品率，纵轴则代表对应的样本量。黄色散点标出了在 90%置信水平下，各个次品率所对应的样本量，而黄色虚线则是对这些散点进行近似拟合得到的折线；同理，蓝色散点标出了在 95%置信水平下，各个次品率所对应的样本量，蓝色虚线为相应的近似拟合折线。

通过观察这幅图表，可以发现一些显著的趋势：在 95%置信水平下，当真实次品率落在 0.09 至 0.11 区间内时，即接近标称值时，所需的抽样样本量达到高峰，大约在 5000 个左右；随后，随着次品率的继续提高，其所需抽样的样本量逐渐下降，最终减少到约 200 个样本。类似地，在 90%置信水平下，当真实次品率落在 0.09 至 0.11 区间内时，所需的抽样样本量达到高峰，大约在 3000 个左右；随后，随着次品率的提高，其所需抽样的样本量逐渐下降至大约 100 个样本。

2. 基于无限总体的一次抽样结果

根据上文提及的样本量计算步骤，得到在不同真实次品率及两种置信水平条件下的抽样样本量 n 的取值，如下表所示。

表 4 方案二 不同真实次品率 95%置信水平条件下的抽样及检验结果

p^*	样本量 n	是否接受原假设
0.08	681	0
0.09	2901	0
0.11	3501	1
0.12	971	1
0.13	471	1
0.14	291	1
0.15	191	1
0.16	141	1

表 5 方案二 不同真实次品率 90%置信水平条件下的抽样及检验结果

p^*	样本量 n	是否接受原假设
0.08	481	1
0.09	2101	1
0.11	2501	0
0.12	681	0
0.13	341	0
0.14	201	0
0.15	141	0
0.16	98	0

本文绘制出的样本量与真实次品率的关系图如下：

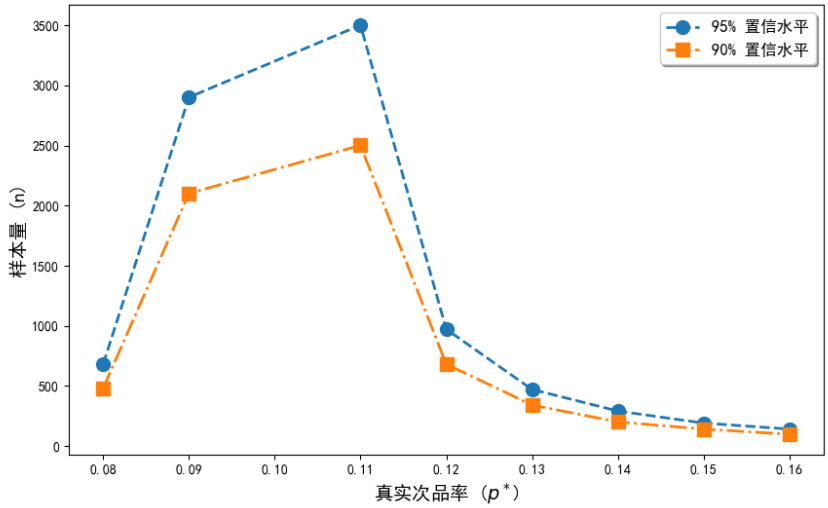

图 4 方案二 样本量与真实次品率的关系图

在此图中，横轴代表真实次品率，纵轴则代表对应的样本量。黄色散点标出了在 90%置信水平下，各个次品率所对应的样本量，而黄色虚线则是近似拟合得到的折线；同理，蓝色散点标出了在 95%置信水平下，各个次品率所对应的样本量，蓝色虚线为相应的近似拟合折线。

通过观察这幅图表，可以发现一些显著的趋势：在 95%置信水平下，当真实次品率落在 0.09 至 0.11 区间内时，即接近标称值时，所需的抽样样本量达到高峰，大约在 3000 个左右；随后，随着次品率的继续提高，其所需抽样的样本量逐渐下降，最终减少到约 200 个样本。类似地，在 90%置信水平下，当真实次品率落在 0.09 至 0.11 区间内时，所需的抽样样本量达到高峰，大约在 2000 个左右；随后，随着次品率的提高，其所需抽样的样本量逐渐下降至大约 100 个样本。

3.小结

本文采取两种方法进行抽样检测，第一种方法为序贯概率比检验(SPRT)。此方案采用动态采样策略，即每次抽样之后都会即时更新对数似然比，并据此迅速做出接收、拒收或是继续进行抽样的决策。其显著优势在于，针对某些特定情境，它能够提前终止抽样流程，进而有效缩减所需的样本数量。然而，上文实验结果表明，当假定的零配件次品率趋近于标称值时，这种方法很大概率导致样本量急剧增加，从而在一定程度上削弱了其效率优势。

第二种方法是无限总体的一次抽样方法。该方法利用二项分布的累积分布函数来构建置信区间，从而实现对误差范围的严格控制。与 SPRT 方法相比，这种方法在多数情况下能够提供更少的样本量需求，特别是当真实次品率接近标称值的情况下，其所需的

样本量表现得更为稳定且相对较低。

综合两种方法求解结果，本文绘制乐两种抽样方案在不同置信水平下的样本量对比图：

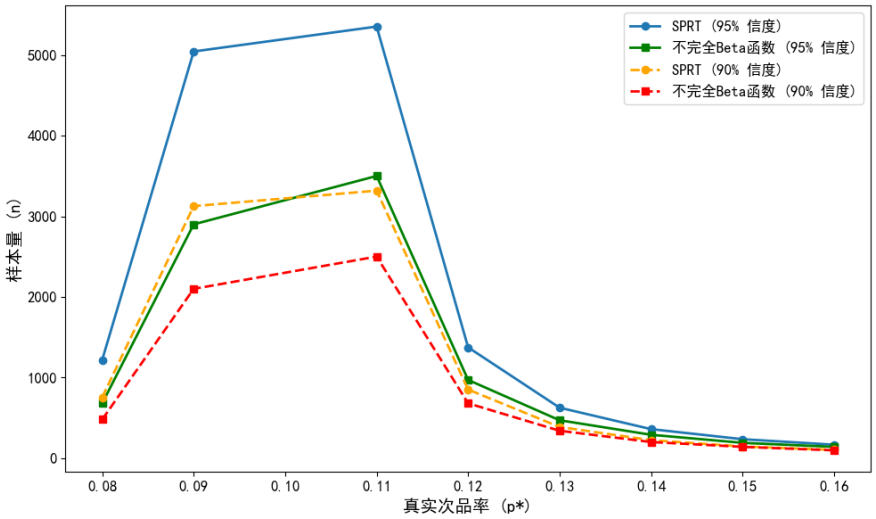

图 5 两种抽样方案在不同置信水平下的样本量对比

图中，横轴代表真实次品率，纵轴则代表对应的样本量。蓝色实线表示在 95%置信区间下，使用 SPRT 方法所得到的结果、绿色实线表示 95%置信区间下，使用基于无限总体的一次抽样方法所得到的结果；黄色虚线表示在 90%置信区间下，使用 SPRT 方法所得到的结果、红色虚线表示 90%置信区间下，使用基于无限总体的一次抽样方法所得到的结果。

通过对比图可以观察到，蓝色实线与黄色虚线均分别位于绿色实线与红色虚线之上。也就是说，在两种情形下，当真实次品率落在 $(0.08, 0.16)$ 区间内时，SPRT 方法所需的样本量均大于基于无限总体的一次抽样方法所需的样本量。

综上所述，第二种方法在进行抽样检测时更满足经济高效的检测要求，因此，选择第二种抽样检测方案，即基于无限总体的一次抽样检验，作为更优选项。

六、问题二模型建立与求解

6.1 建模思路

本问题已知两种零配件和成品次品率，旨在根据所提供情形，为企业生产过程的各个阶段做出决策，并给出决策的依据及相应的指标结果，以实现企业成本最小。

题目所提供情形如下表所示。

表6 企业在生产中遇到的情况

情况	零配件1			零配件2			成品				不合格成品	
	次品率	购买单价	检测成本	次品率	购买单价	检测成本	次品率	装配成本	检测成本	市场售价	调换损失	拆解费用
1	10%	4	2	10%	18	3	10%	6	3	56	6	5
2	20%	4	2	20%	18	3	20%	6	3	56	6	5
3	10%	4	2	10%	18	3	10%	6	3	56	30	5
4	20%	4	1	20%	18	1	20%	6	2	56	30	5
5	10%	4	8	20%	18	1	10%	6	2	56	10	5
6	5%	4	2	5%	18	3	5%	6	3	56	10	40

企业生产过程的各个阶段所做决策流程图如下图所示。

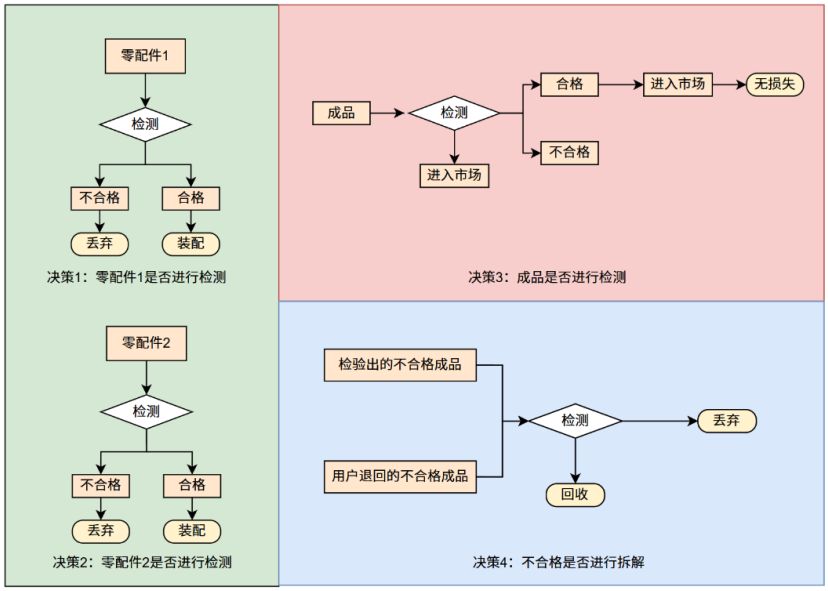

图6 问题二所述企业各阶段决策流程图

结合流程图分析，企业在生产过程中共面临四个主要决策节点，每一个决策节点均对应“执行”与“不执行”的二元选择，因此共衍生出16种不同的决策组合方案。其中，四个决策节点分别为：是否对零配件1进行检测、是否对零配件2进行检测、是否对装配后的成品进行检测、以及是否对不合格成品进行拆解。

每一项决策的不同走向，都将直接关联到企业的最终收益或损失。鉴于此，本文首先通过设立决策变量，构建出每个生产环节的成本构成，进而推导出总体的成本函数，即此问题求解的目标函数。

为寻求最优解，本文创新性地提出了一个基于动态规划和强化学习的联合求解模型。该模型以 16 种决策方案为基石，旨在不同情境下甄选出最优策略，以期实现总成本的最小化。在模型中，明确了系统状态与动作，并通过状态转移函数来描述系统状态的变化，同时设计了综合考虑关键成本因素的奖励函数。具体而言，动态规划部分通过递归的方式计算价值函数，求解出最优策略；强化学习部分采用 Q-learning 方法，通过迭代更新 Q 值，逐步逼近最优决策。最终，此模型能够在不确定性环境下确定不同状态下的最优检测、装配、拆解等策略，以实现成本最小化目标。即求解出不同情况下所对应的最优决策方案。

6.2 模型建立

6.2.1模型准备

在生产流程中，企业所制定的决策涵盖了零配件的采购、检测、装配、成品检测、不合格品的调换与拆解等多个环节，每一项决策的不同结果都将直接关联到企业的最终收益或损失。对此，本文旨在深入分析每个环节的成本构成，针对不同特定情境，为企业打造合理决策方案，以实现生产成本最小化。

1.模型参数设定

为便于记录与分析，本文特将所涉及的各类物品进行编码处理：零配件1标记为1，零配件2标记为2，成品标记为3。

表7 基本参数设定

序号	符号	含义
1	N	企业生产成品数
2	a_{1i}	零配件1\零配件2\成品的次品率，其中$i=1,2,3$
3	a_{2i}	零配件1\零配件2的采购单价，其中$i=1,2$
4	a_{3i}	零配件1\零配件2\成品的检测成本，其中$i=1,2,3$
5	c_1	单位成品的市场售价
6	c_2	单位不合格产品的调换损失（物流成本、信誉损失等）
7	c_3	单位成品的拆解费用
8	c_4	单位成品的装配成本

其中，以次品率为例，a_{11}表示零配件1的次品率，a_{12}表示零配件2的次品率，a_{13}表示成品次品率。其余参数含义与之相似。

因决策过程中涉及多个选择分支，本文引入以下决策变量：

表8 决策变量设定

序号	符号	含义
1	x_i	二值变量，取值为0（不进行检测）或1（进行检测） 其中$i=1,2,3$
2	y_1	二值变量，取值为0（对不合格的成品选择丢弃）或1（对不合格的成品进行拆解）

其中，x_i具体含义为：

$$x_1=\begin{cases}0, & \text{对零配件1不进行检测}\\ 1, & \text{对零配件1进行检测}\end{cases}$$

$$x_2=\begin{cases}0, & \text{对零配件2不进行检测}\\ 1, & \text{对零配件2进行检测}\end{cases}$$

$$x_3=\begin{cases}0, & \text{对成品不进行检测}\\ 1, & \text{对成品进行检测}\end{cases}$$

2.基于题目描述引入的多种不同决策方案

结合图1，每个阶段的决策点存在两个选择，因此共计$2^4=16$种方案组合。具体方案组合内容如下表所示。

表 9 方案组合

方案序号	检测零配件1	检测零配件2	检测成品	拆解不合格成品
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

其中，0代表不检测或者不拆解，1代表检测或者拆解。

6.2.2 成本公式

1. 零配件采购和检测成本

（1）零配件1的采购和检测成本

零配件1的采购成本可以表示为：

$$C_{零配件1采购} = N_1 \cdot a_{21} \tag{6.1}$$

其中，a_{21}表示零配件1的采购单价；N_1表示零配件1的采购数量，企业在这一环节可选择是否对零配件1进行检测，若进行检测，则检测中发现的次品将直接被丢弃，不参与后续生产，因此，零配件1采购数量可以用企业生产成品数与未接受检测时的合格品率来综合确定，公式为：

$$N_1 = \frac{N}{1-a_{11}(1-x_1)} \tag{6.2}$$

其中，N 表示企业生产成品数；a_{11}表示零配件1的次品率；x_1为二值变量，其为0时表示零配件1不接受检测，其为1时表示零配件1接受检测。

零配件1的检测成本可以表示为：

$$C_{零配件1检测} = x_1 \cdot \frac{N}{1-a_{11}} \cdot a_{31} \tag{6.3}$$

其中，a_{31}表示零配件1的检测成本。

（2）零配件2的采购和检测成本：

零配件2的采购成本可以表示为：

$$C_{零配件2采购} = N_2 \cdot a_{22} \tag{6.4}$$

其中，a_{22}表示零配件2的采购单价；N_2表示零配件2的采购数量，与零配件1采购数量公式构成一致，零配件2采购数量同样可以用企业生产成品数与未接受检测时的合格品率来综合确定，公式为：

$$N_2 = \frac{N}{1-a_{12}(1-x_2)} \tag{6.5}$$

其中，a_{12}表示零配件2的次品率；x_2为二值变量，其为0时表示零配件2不接受检

测，其为1时表示零配件2接受检测。

零配件1的检测成本可以表示为：

$$C_{\text{零配件2检测}} = x_2 \cdot \frac{N}{1-a_{12}} \cdot a_{32} \tag{6.6}$$

其中，a_{32} 表示零配件2的检测成本。

2.成品的装配和检测成本

成品的装配成本可以表示为：

$$C_{\text{装配}} = N \cdot c_4 \tag{6.7}$$

其中，c_4 为单位成品的装配成本。

当企业选择对成品进行检测时，其所支付的成品检测成本可以表示为：

$$C_{\text{成品检测}} = x_3 \cdot N \cdot a_{33} \tag{6.8}$$

其中，x_3 为二值变量，其为0时表示成品不接受检测，其为1时表示成品接受检测；a_{33} 为单位成品的检测成本。

3.成品调换损失

当企业选择对成品不进行检测时，将会造成不合格品直接流入市场，从而需要承担相应的调换损失，可以表示为：

$$C_{\text{调换损失}} = (1-x_3) \cdot N \cdot a_{13} \cdot c_2 \tag{6.9}$$

其中，a_{13} 为成品次品率；c_2 为单位不合格产品的调换损失。

4.拆解成本

当企业在生产过程中检测出不合格品时，或者收到用户退回的不合格品后，企业面临否对其进行拆解这一决策。在企业决定对不合格成品实施拆解时，需要充分考虑到拆解下来的零配件自身已经历初加工并使用了一段时间，因此会存在一定的折旧情况，这将导致其自身价值相应地有所降低。鉴于生产流程的简洁性，本文假定经过拆解后，零配件自身价值变为其售价的60%。

因此，当企业选择对不合格品进行拆解时，拆解成本包含两部分。其一，所有不合格品的拆解总费用，其二，拆解成零配件后零配件的自身价值，两者相减即拆解成本，可以表示为：

$$C_{\text{拆解}} = y_1 \cdot N \cdot (1-P_{\text{成品合格}}) \cdot c_3 - y_1 \cdot N \cdot (1-P_{\text{成品合格}}) \times 0.6 \cdot c_1 \tag{6.10}$$

其中，y_1 为二值变量，其为0时表示对不合格品作丢弃处理，其为1时表示对不合格品进行拆解；c_3 为单位成品的拆解费用；c_1 为单位成品的市场售价；$P_{\text{成品合格}}$ 为成品合格率，其计算公式为：

$$P_{\text{成品合格}} = (1-a_{11}(1-x_1)) \cdot (1-a_{12}(1-x_2)) \cdot (1-a_{13}(1-x_3)) \tag{6.11}$$

该公式中，$a_{1i}(i=1,2,3)$ 分别表示零配件1、零配件2、成品的次品率；$x_i(i=1,2,3)$ 均为二值变量，其为0时表示不进行检测，其为1时表示进行检测。

5.总成本公式

综上所述，在生产流程中，企业在关于零配件的检测、装配、成品检测、不合格品的调换与拆解多个环节上所做的决策均将直接关联到企业的最终收益或损失。由此，企业最终承担的总成本可以表示为：

$$C_{\text{总成本}} = C_{\text{零配件1 采购}} + C_{\text{零配件 2 采购}} + C_{\text{零配件 1 检测}} + C_{\text{零配件 2 检测}} + C_{\text{装配}} + C_{\text{成品检测}} + C_{\text{调换损失}} + C_{\text{拆解}} \tag{6.12}$$

6.2.3 基于动态规划和强化学习的联合求解模型

本题目标为最小化总成本，结合上文，目标函数为：

$$min \quad C_{\text{总成本}} = C_{\text{零配件1 采购}} + C_{\text{零配件 2 采购}} + C_{\text{零配件 1 检测}} + C_{\text{零配件 2 检测}} + C_{\text{装配}} + C_{\text{成品检测}} + C_{\text{调换损失}} + C_{\text{拆解}} \tag{6.13}$$

1.状态定义

在本研究中，系统状态 s_t 在每个时间步 t 包含以下关键信息：

表10 系统状态 s_t 在每个时间步 t 包含的关键信息

系统状态	时间步	包含信息
s_t	s_1	零配件1是否检测(x_1)
	s_2	零配件2是否检测(x_2)
	s_3	成品是否检测(x_3)
	s_4	不合格成品是否拆解(y_1)
	s_5	当前零配件1的次品率(a_{11})
	s_6	当前零配件2的次品率(a_{12})
	s_7	当前成品的次品率(a_{13})

2.动作定义

在给定状态 s_t 下，可选择的动作 a_t 包括：

表11 给定状态 s_t 下可选择的动作 a_t

系统状态	动作	含义
s_t	a_1	是否检测零配件1
	a_2	是否检测零配件2
	a_3	是否检测成品
	a_4	是否拆解不合格成品

这些动作将直接影响系统的后续状态及相应的成本收益。

3.状态转移

当系统在状态 s_t 下执行动作 a_t 后，其状态将依据状态转移函数 $f(s_t, a_t)$ 转移至下一个状态 s_{t+1}，可以表示为：

$$s_{t+1} = f(s_t, a_t) \tag{6.14}$$

例如，执行 a_1 会将 st_1 更新为“已检测”，执行 a_2 会将 st_2 更新为“已检测”，类似地，其他动作也会相应地更新对应的状态变量。

4. 奖励函数

在状态 s_t 下采取动作 a_t 后，本文将产生的即时奖励（或即时成本）由奖励函数 $r(a_t, a_t)$ 表示。该函数综合考虑了上文中提及的系统中的关键成本因素（即检测、装配、拆解以及调换等环节的成本），并表示为总成本的负值，即：

$$r(s_t, a_t) = -C_{\text{总成本}} \tag{6.15}$$

5.价值函数

动态规划的核心在于通过递归计算价值函数$V(s_t)$，该函数描述了从状态s_t开始，采取最优策略所能获得的最大累积收益。价值函数的递推公式为：

$$V(s_t) = \max_{a_t}[r(s_t, a_t) + \gamma V(s_{t+1})] \tag{6.16}$$

其中：

$r(s_t, a_t)$是在状态s_t采取动作a_t获得的即时奖励(或即时成本),即考虑了检测、装配、拆解以及调换等环节的成本；

γ是折扣因子，衡量未来收益的权重，$\gamma \in (0,1)$；

$V(s_{t+1})$是下一时刻状态s_{t+1}的最优价值。

通过递推公式，逐步求解从初始状态到终点的最优策略。

6. Q-learning更新

为了在不确定性环境下找到最优策略，本研究采用Q-learning方法。

Q值函数$Q(s_t, a_t)$表示在状态s_t下采取动作a_t所获得的预期累积收益。Q-learning的更新公式为：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{6.17}$$

其中：

α是学习率，控制Q值更新的幅度；

$\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$是在下一个状态$s_{t+1}$中采取最优动作所能获得的最大值。

通过反复迭代更新Q值，系统将逐步逼近最优决策，并最终确定在不同状态下的最优检测、装配、拆解等策略。

7.小结

本文提出了一个基于动态规划和强化学习的联合求解模型，旨在最小化总成本。模型具体求解步骤如下：

（1）初始化Q值：为所有状态s_t和动作a_t初始化Q值$Q(s_t, a_t)$；

（2）选择动作：根据ϵ-贪婪策略，在每个状态s_t选择当前最优的动作a_t；

（3）执行动作：执行动作a_t，并观察新的状态s_{t+1}和即时奖励$r(s_t, a_t)$（包含检测、装配、拆解、调换的成本和收益)；

（4）更新Q值：根据Q-learning更新公式，即公式(6.17)，更新Q值；

（5）状态转移：转移到新状态s_{t+1}并继续执行动作，更新Q值；

（6）最优策略：通过训练后输出的Q值，确定在每个状态下采取的最优动作，最终实现最小化总成本。

6.3 模型求解

本文设定了一个具体的企业生产场景，其中成品数量N为1000件。通过Python软件实现上述基于动态规划和强化学习的联合求解模型，揭示出六种不同情境下，针对16种潜在的决策组合，企业所需承担的总成本。为使数据结果更加直观且易于企业决策者理解，本文进一步对结果进行了单位化处理，从而得出了每个成品所对应的精确成本数额。

在此，本文选取情况一作为示例，展示在此情境下16种不同决策组合所分别对应的企业总成本。完整结果详见支撑材料。

表12 情况一对应于16种决策组合下企业所承担总成本

方案编号	总成本价	单位成本价
1	32070.44	32.07
2	24319.84	24.32
3	33444.44	33.44
4	28010.44	28.01
5	32584.44	32.58
6	27150.44	27.15
7	34444.44	34.44
8	31584.44	31.58
9	33140.00	33.14
10	27706.00	27.71
11	35000.00	35.00
12	32140.00	32.14
13	33600.00	33.60
14	30740.00	30.74
15	36000.00	36.00
16	36000.00	36.00

为了更直观地对比和分析六种不同情境下，企业针对16种潜在决策组合所需承担的总成本变化趋势，并快速识别出最优决策组合，本文绘制了折线图，从而为企业决策者提供了直观、精准的参考依据。

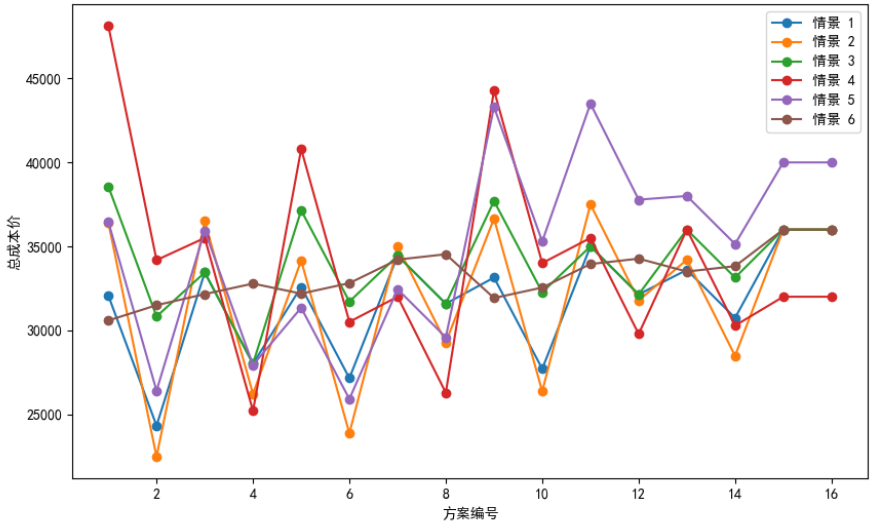

图7 六种情境下不同决策对应的总成本

在所提供的图中，横坐标表示各个决策方案的编号，纵坐标表示企业在不同情况和方案下所需承担的总体成本。通过观察可以发现，成本效益最为显著的决策大多集中在方案2至方案8这一区间内，它们的成本明显低于其他方案，且在大部分情况下，方案组合2为最优决策，如情况2、情况1。

以下是对不同情况下最优决策的梳理：

表 13 不同情况下的最优决策方案

情况序号	最优决策序号	是否检测零配件 1	是否检测零配件 2	是否检测成品	是否拆解不合格成品	总成本/元	单位成本/元
1	2	0	0	0	1	24319.84	24.32
2	2	0	0	0	1	22471.20	22.47
3	4	0	0	1	1	28010.44	28.01
4	4	0	0	1	1	25204.00	25.20
5	6	0	1	0	1	25910.44	25.91
6	1	0	0	0	0	30584.14	30.58

其中，0代表不检测或不拆解，1代表检测或拆解。

综上所述，得到以下结果：

对于情况一，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 24.32 元；

对于情况二，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 22.47 元；

对于情况三，其最优决策为：不检测零配件 1、不检测零配件 2、检测成品、拆解不合格成品，企业所承担的单位成本为 28.01 元；

对于情况四，其最优决策为：不检测零配件 1、不检测零配件 2、检测成品、拆解不合格成品，企业所承担的单位成本为 25.20 元；

对于情况五，其最优决策为：不检测零配件 1、检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 25.91 元；

对于情况六，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、不拆解不合格成品，企业所承担的单位成本为 30.58 元。

通过对比六种情况，发现成品检测会导致总成本和单位成本增加（如情况 3 和 4，单位成本上升至 28.01 元和 25.20 元），表明检测成品增加了额外成本。而不拆解不合格成品（情况 6，单位成本为 30.58 元）导致了最高的总成本，表明不拆解次品带来的调换损失显著增加了成本。相比之下，适度检测零配件 2（情况 5，单位成本 25.91 元）在减少不检测成品的前提下优化了总成本，说明零配件检测的成本控制在一定条件下能减少次品带来的影响。

七、问题三模型建立与求解

7.1 建模思路

问题三在核心任务上与问题二保持了一致性，即均旨在特定生产流程框架下，依据给定的情境，为企业的各个生产阶段制定决策，并明确决策的依据及相应的指标结果，以期达到企业成本最小化的目标。

问题三中所提供的情形如下表所示。

表14 企业在生产中遇到的情况

零配件	次品率	购买单价	检测成本	半成品	次品率	装配成本	检测成本	拆解费用
1	10%	2	1	1	10%	8	4	6
2	10%	8	1	2	10%	8	4	6
3	10%	12	2	3	10%	8	4	6
4	10%	2	1					
5	10%	8	1	成品	10%	8	6	10
6	10%	12	2					
7	10%	8	1		市场售价		调换损失	
8	10%	12	2	成品	200		40	

然而，问题三在问题二的基础上显著提升了复杂度，主要体现在引入了半成品生产工序以及增加了零配件的种类数量。具体而言，生产模式从原先的两个零配件组装成一件成品，转变为通过特定的零配件组合生成半成品，再由这些半成品按照固定组合生成最终的成品（详细组装流程参见下图）。

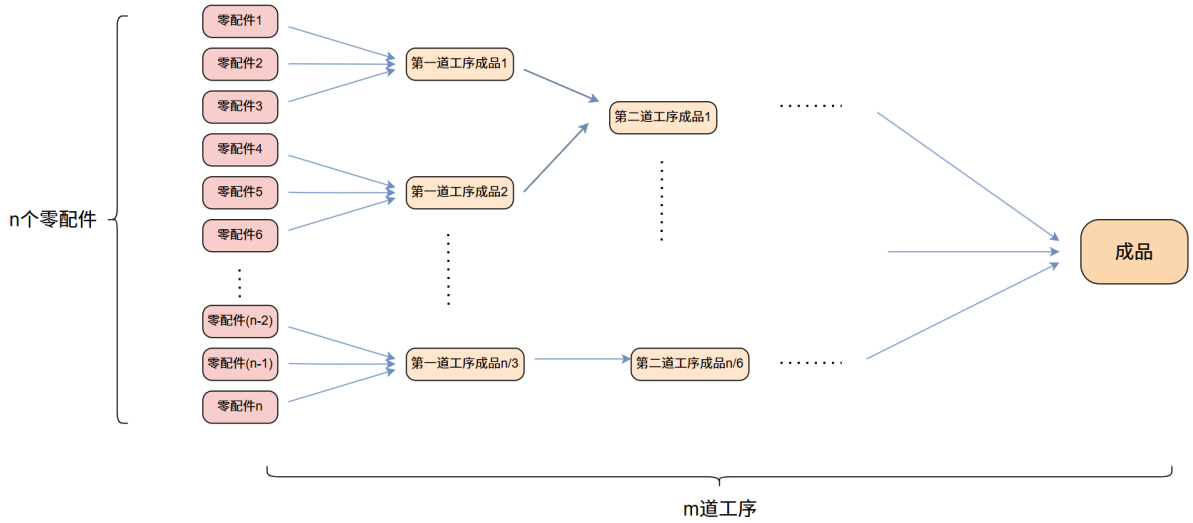

图 8 m 道工序、n 个零配件的组装情况

这一变化极大地丰富了企业的决策组合空间，使得寻找最优决策组合的难度显著提升。同时，由于生产流程的重构，直接影响了企业整体生产流程的总成本结构。此外，这一变动还间接增加了各阶段成本计算的复杂性，提高了问题求解复杂性。因此，从这个角度来看，问题二可以被视为问题三在简化条件下的特例。

在问题三中，m 道工序代表生产中的关键步骤，如零配件的检测、半成品装配和成品的最终检测。n 个零配件则表示生产中涉及的不同零部件，这些零部件可能有不同的次品率、检测成本和装配难度。通过设定 m 和 n 的值，模型可以分析在不同工序和零件组合下的最优策略，从而有效控制次品率，降低生产成本，最大化企业利润。m和n的合理设定不仅可以帮助我们讨论具体场景，还能够使模型具备更广泛的应用性和灵活性。

针对问题三的特定情况（$m=2,n=8$），本文在问题二所建立的模型基础上，对各生产环节的成本构成进行了全面的补充与调整。通过分析与推导，构建出问题三在更广泛背景下的总体成本函数，这也是本文所要求解的核心目标函数。

为探寻最优解决方案，本文采用加权算法融合模型。该模型通过灵活地调整动态规划、Q-Learning强化学习以及贪心算法这三种算法的权重，全面评估它们各自计算出的成本，旨在实现总成本的最优化。具体而言，本文首先运用动态规划方法来解决多阶段生产决策问题；其次，利用Q-Learning强化学习算法来不断更新和优化决策策略；同时，还采用贪心算法来逐步选择当前看似最优的决策，以确保每一步都能获得即时利益的最大化。

在求解出各自算法对应的成本后，本文进一步根据历史数据的误差反馈来动态地调整各算法的权重。最终，求解出不同情况下所对应的最优决策方案。

7.2 模型建立

7.2.1 模型准备

为有效控制并最小化生产成本，企业必须在检测、装配、以及可能的拆解等关键环节上做出决策。在生产流程中，企业面临着如下抉择：

零件检测：企业可以选择对零件进行检测，以筛选出次品，可以选择不进行检测，

此时次品将直接进入装配环节；

半成品检测：对于组装好的半成品，企业同样可以选择进行检测，以确保其质量，或者选择跳过检测，让半成品直接进入下一步的组装或成品生产；

成品检测：成品在出厂前也可以进行检测，以保证产品质量。若选择不检测，则次品成品将直接进入市场；

不合格成品拆解：对于检测出的不合格成品或者用户调换回的不合格成品，企业可以选择是否对其进行拆解，若选择拆解，鉴于此生产流程的复杂性，本文假设拆解后零配件的价值将按其原价值的 50%进行折旧。

1.模型参数设定

表15 基本参数设定

序号	符号	含义
1	N	企业生产成品数
2	a_{1i}	零配件次品率，其中$i=1,2,3,\cdots,8$
3	a_{19}	成品次品率
4	a_{2i}	零配件购买价格，其中$i=1,2,3,\cdots,8$
5	a_{3i}	单位零配件检测成本，其中$i=1,2,3,\cdots,8$
6	a_{4j}	单位半成品检测成本，其中 $j=1,2,3$
7	a_5	单位成品检测成品
8	c_1	单位成品市场售价
9	c_2	单位不合格品的调换损失
10	c_3	拆解单位不合格品的费用
11	c_4	单位半成品的装配成本
12	c_5	单位成品的装配成本

其中，以零配件次品率为例，a_{11} 表示零配件1的次品率，a_{12} 表示零配件2的次品率，a_{13} 表示零配件3的次品率，以此类推。其余参数含义与之相似。

因决策过程中涉及多个选择分支，本文引入以下决策变量：

表16 决策变量设定

序号	符号	含义
1	x_i	示性变量，表示是否对零配件进行检测 取值为0（不进行检测）或1（进行检测） 其中$i=1,2,3,\cdots,8$
2	y_i	示性变量，表示是否对半成品进行检测 取值为 0（不进行检测）或 1（进行检测） 其中$i=1,2,3$
3	z_1	示性变量，表示是否对成品进行检测 取值为0（不进行检测）或1（进行检测）
4	k_1	示性变量，表示是否对不合格成品进行拆解 取值为0（选择丢弃）或1（进行拆解）

2.基于题目描述引入的多种不同决策方案

在问题二和问题三中，虽然两者都是围绕企业生产流程中的检测与拆解决策展开的，但两者的决策组合数有明显的差异。问题二中，生产流程较为简单，主要涉及两个零配件、一个成品及其检测与拆解的决策，总共有$2^2\times2\times2=16$种组合。而问题三中，生产

流程中将第二问中的 2 个零配件增加到了 8 个零配件，同时引入半成品工序，增加了半成品的检测过程，进而导致成品的检测和拆解步骤增多，导致决策组合数大幅增加，达到了 $2^8 \times 2^3 \times 2^2 = 8192$ 种组合。

7.2.2 问题三的决策组合分析

在探讨问题三时，企业面临的是一个复杂的生产决策场景，其中涵盖了从零配件到半成品，再到最终成品的全方位检测与拆解策略，共涉及 8 个零部件、3 个半成品以及 1 个成品。企业的核心目标是通过精心规划的检测与拆解决策，实现生产成本的最小化。

决策组合主要来自三个方面：零配件检测、半成品检测、以及成品检测和拆解。

1.零配件检测组合的多元化分析

在零配件层面，企业需对 8 个关键零配件作出是否进行检测的决策。每个零配件都存在两种可能的选择：

检测($x_i = 1$)：此选择意味着零配件将经过严格检测，不合格品将被及时剔除，从而避免其在后续流程中造成更大的损失;

不检测($x_i = 0$)：若选择不检测，零配件将直接进入装配环节，这可能导致不合格品在后续流程中累积，增加整体成本风险。

由于零配件数量为 8 个，因此，零配件检测的组合方式多达 $2^8 = 256$ 种。

2.半成品检测组合

在半成品阶段，企业需对 3 个半成品作出是否进行检测的决策。与零配件类似，半成品也面临两种选择：

检测($x_i = 1$)：通过检测，半成品中的次品将被有效识别并剔除，确保生产流程的顺畅进行；

不检测($x_i = 0$)：若选择不检测，半成品将直接进入下一步的装配或成品环节，这同样可能带来次品累积的风险。

半成品检测的组合方式共有 $2^3 = 8$ 种。

3.成品检测与拆解组合

在成品阶段，企业的决策更加复杂，涉及成品检测(x_{12})与不合格成品拆解(y_1)两个方面。具体组合如下：

成品检测+拆解 $(x_{12} = 1, y_1 = 1)$：此组合成品经过严格检测，且不合格品将被拆解回收，实现资源的最大化利用;

成品检测+不拆解 $(x_{12} = 1, y_1 = 0)$：此组合成品经过检测，但不合格品将被直接丢弃，将会造成资源浪费;

成品不检测+拆解 $(x_{12} = 0, y_1 = 1)$：此组合成品不经过检测，但不合格品将被拆解回收，虽能回收资源，但无法确保成品质量;

成品不检测+不拆解 $(x_{12} = 0, y_1 = 0)$：此组合成品既不检测也不拆解，可能给企业带来重大损失。

综上所述，成品的检测与拆解组合共有 4 种。

7.2.3 成本公式

1.零配件的采购与检测成本

对于零配件 1 至零配件 8，企业可选择是否对其进行检测，若进行检测，则检测中发现的次品将直接被丢弃，不参与后续生产；如果不进行检测，则次品将直接进入装配

环节。因此，零配件 1 采购数量可以用企业生产成品数与未接受检测时的合格品率来综合确定，公式为：

$$N_i = \frac{N}{1-a_{1i}(1-x_i)} \tag{7.1}$$

其中，$i=1,2,3,\cdots,8$；N_i 为零配件 i 采购数量；N 为企业生产成品数；a_{1i} 为零配件 i 的次品率；x_i 为二值变量，其为0时表示零配件 i 不接受检测，其为1时表示零配件 i 接受检测。

零件i的采购成本可以表示为：

$$C_{\text{零配件i采购}} = N_i \cdot a_{2i} \tag{7.2}$$

其中，$i=1,2,3,\cdots,8$；N_i 为零配件 i 采购数量；a_{2i} 为零配件 i 的采购单价。

零配件的检测成本可以表示为：

$$C_{\text{零配件}i\text{检测}} = x_i \cdot N_i \cdot a_{3i} \tag{7.3}$$

其中，$i=1,2,3,\cdots,8$；N_i 为零配件 i 采购数量；a_{3i} 为零配件 i 的检测成本。

2.半成品的合格率、装配以及检测成本

半成品的合格率由其组成的零配件的次品率和半成品次品率所决定。

以半成品 1 为例，由零件 1、零件 2 和零件 3 装配而成，其合格率为：

$$P_{\text{半成品1}} = (1-a_{11}(1-x_1))(1-a_{12}(1-x_2))(1-a_{13}(1-x_3)) \tag{7.4}$$

其中，a_{11}、a_{12}、a_{13} 分别为零配件 1、2、3 的次品率；x_i 为示性变量，表示是否对零配件进行检测，取值为 0（表示不进行检测）或 1（表示进行检测）。

类似地，半成品 2 和半成品 3 的合格率可以通过同样的方式计算。

半成品装配成本为：

$$C_{\text{半成品装配}} = N \cdot c_4 \tag{7.5}$$

其中，N 为企业生产成品数；c_4 为单位半成品的装配成本。

半成品的检测成本为：

$$C_{\text{半成品检测}} = y_i \cdot N \cdot c_4$$

其中，y_i 为示性变量，表示是否对半成品进行检测，取值为 0（表示不进行检测）或 1（表示进行检测）

3.成品的合格率和装配成本

成品由 3 个半成品装配而成，成品的合格率由其半成品次品率和成品次品率决定，即：

$$P_{\text{成品}} = P_{\text{半成品1}} \cdot P_{\text{半成品2}} \cdot P_{\text{半成品3}} \cdot (1-a_{19}(1-z_1)) \tag{7.6}$$

其中，a_{19} 为成品次品率；z_1 为示性变量，表示是否对成品进行检测，取值为 0（表示不进行检测）或 1（表示进行检测）。

成品装配成本为：

$$C_{\text{成品装配}} = N \cdot c_5 \tag{7.7}$$

其中，N 为企业生产成品数；c_5 为单位成品的装配成本。

4.成品检测与调换损失

成品的检测成本为：

$$C_{\text{成品检测}} = z_1 \cdot N \cdot a_5 \tag{7.8}$$

其中，z_1为示性变量，表示是否对成品进行检测；N为企业生产成品数；a_5为单位成品检测成本。

如果成品未进行检测，则不合格品会进入市场，企业将会承担调换损失：

$$C_{\text{调换损失}} = (1-z_1)\cdot N\cdot(1-P_{\text{成品}})\cdot c_2 \tag{7.9}$$

其中，$P_{\text{成品}}$为成品的合格率；c_2为单位不合格品的调换损失。

5.不合格成品拆解与回收

当企业在生产过程中检测出不合格品时，或者收到用户退回的不合格品后，企业面临是否对其进行拆解这一决策。若选择拆解，需要充分考虑到拆解下来的零配件自身已经历初加工并使用了一段时间，因此会存在一定的折旧情况，这将导致其自身价值相应地有所降低。针对折旧问题，鉴于生产流程的复杂性，本文假定经过拆解后，零配件自身价值变为其售价的50%。

因此，当企业选择对不合格品进行拆解时，拆解成本包含两部分。其一，所有不合格品的拆解总费用，其二，拆解成零配件后零配件的自身价值，两者相减即拆解成本。

$$C_{\text{拆解成本}} = y_1\cdot N\cdot(1-P_{\text{成品}})\cdot c_3 \tag{7.10}$$

$$C_{\text{回收收入}} = y_1\cdot N\cdot(1-P_{\text{成品}})\times 0.5\cdot c_1 \tag{7.11}$$

其中，k_1为示性变量，表示是否对不合格成品进行拆解，取值为 0（选择丢弃）或 1（进行拆解）；$P_{\text{成品}}$为成品的合格率；c_3为拆解单位不合格品的费用；c_1为单位成品的市场售价。

6.总成本公式

综上所述，在生产流程中，企业在关于零配件的检测、装配、半成品的检测、装配、成品检测、不合格品的调换与拆解多个环节上所做的决策均将直接关联到企业的最终收益或损失。由此，企业最终承担的总成本可以表示为：

$$\begin{aligned} C_{total} = &\sum_{i=1}^{8} C_{\text{零配件}i\ \text{采购}} + \sum_{i=1}^{8} C_{\text{零配件}i\ \text{采购}} + C_{\text{半成品装配}} + C_{\text{成品装配}} \\ &+ C_{\text{成品检测}} + C_{\text{调换损失}} + C_{\text{拆解成本}} - C_{\text{回收收入}} \end{aligned} \tag{7.12}$$

7.2.4 建立加权算法融合模型

为了充分发挥三种算法各自的优势，本文创新性地提出了一种加权融合模型。该模型旨在通过融合动态规划的确定性求解能力、Q-Learning 处理随机性问题的灵活性，以及贪心算法快速收敛与局部优化能力，构建出一个更为全面和强大的决策支持系统。

此模型的核心目标是通过采用加权平均的方法，对三种算法所得出的成本进行综合考量，从而求解出总成本的最优解。通过这种方法，可以充分利用每种算法的独特优势，同时弥补它们各自的不足，进而实现更为精准和高效的决策制定。

1.加权目标函数

加权后的总成本C_{total}计算如下：

$$C_{total} = \omega_1\cdot C_{DP} + \omega_2\cdot C_{QL} + \omega_3\cdot C_{GA} \tag{7.13}$$

其中，ω_1、ω_2、ω_3是三种方法的权重，有$\omega_1+\omega_2+\omega_3=1$，权重具体数值需要根据历史表现、误差反馈和移动平均动态调整；C_{DP}为动态规划得出的总成本；C_{QL}为Q-Learning处理后得到的总成本；C_{GA}为贪心算法处理后得到的总成本。

2.动态规划(DP)部分

在多阶段生产决策中，动态规划递推公式为：

$$V(s)=\min_{a_i}\left(C_{\text{当前工序}}(s,a_i)+V(\text{下一工序状态})\right) \tag{7.14}$$

其中，$V(s)$表示在状态s下的最小总成本；a_i是工序的决策变量，表示是否检测或拆解。

3.Q-Learning 强化学习部分

Q-Learning 利用不断的学习与环境交互更新决策策略，公式如下：

$$Q(s,a)\leftarrow Q(s,a)+\alpha(r+\gamma\max_{a'}Q(s',a')-Q(s,a)) \tag{7.15}$$

其中，r为当前即时奖励(或负的成本),α为学习率，γ为折扣因子，$Q(s,a)$表示状态-动作的价值。

4.贪心算法(GA)部分

贪心算法需要定义一个收益函数，并通过逐步选择当前最优的决策（如是否检测、是否拆解）来最小化生产成本。

选择当前局部最优变化：

$$\Delta x_{best}=\arg\min_{\Delta x_i}C(x_k+\Delta x_i) \tag{7.16}$$

更新解：

$$x_{k+1}=x_k+\Delta x_{best} \tag{7.17}$$

当不存在更优的变化时停止。

5.权重动态调整机制

通过历史数据误差反馈或交叉验证，动态调整三种算法的权重，其公式如下：

$$w_i(t+1)=w_i(t)\cdot\frac{1}{1+\eta\cdot\text{Error}(i)} \tag{7.18}$$

其中，η为学习率；$Error(i)$表示方法i在当前迭代中的误差。权重变化根据每次迭代的误差反向传播，自动优化。

7.2.5 模型的最终优化目标

最终，我们的目标是通过加权融合的方式最小化总成本：

$$\min C_{total}=\omega_1\cdot C_{DP}+\omega_2\cdot C_{QL}+\omega_3\cdot C_{GA} \tag{7.19}$$

该模型能够在复杂的生产流程中，平衡三种算法的优缺点，确保生产成本的最小化，并满足成品质量的市场标准。

7.3 模型求解

7.3.1 求解结果

本文设定企业生产成品数$N=1000$。

首先，通过动态规划、强化学习、贪心算法来分别求解出各自算法对应的最优决策组合与最低总成本，如下表所示。

表 17 动态规划、强化学习、贪心算法对应求解结果

方法	决策组合				最低成本/元	最低单位成本/元
	是否检验零配件	是否检验半成品	是否检验成品	是否进行拆解		
动态规划	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	101573.62	101.57
强化学习	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	30462.50	30.46
贪心算法	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	30462.50	30.46

其中，0 代表不进行检验或者不进行拆解，1 代表进行检验或者进行拆解。

从上述结果可以观察到，动态规划所得出的解与其他两种算法所求得的最优值存在显著偏差。为了校正这一偏差，在分配权重时，应适当降低动态规划的权重，以期达到一个更为均衡的最终结果。值得注意的是，强化学习与贪婪算法所得出的最优值相等，在权重分配时应确保这两种算法的权重保持一致。

基于上述分析，并结合权重动态调整的策略，求解出 $\omega_1 = 0.05, \omega_2 = 0.475, \omega_3 = 0.475$，依据公式(7.13)对三种算法的结果进行加权融合。

最终得出的最优决策方案为对零配件不进行检验、对半成品不进行检验、对成品进行检验、以及对不合格成品进行拆解，其总成本为 34018.06 元，换算成成品单位成本则为 34.02 元。

在对比问题三与问题二的结果后，发现随着零配件个数 n 的增多，装配和检测成本呈现出明显的上升趋势。同时，工序 m 的增加不仅带来了生产成本的直接提升，还使得次品控制过程变得更为精细和复杂，进而增加了决策的难度，并导致生产周期的延长。鉴于这些变化对整体成本效益的显著影响，企业在制定生产策略时，必须综合考虑实际的成本收益比，以确保决策的科学性和合理性。

八、问题四建模与分析

在问题四的题目中，假设问题 2 和问题 3 中零配件、半成品和成品的次品率均是通过抽样检测方法得到的，即原表中所列次品率为观测值，即 $\hat{p}$。鉴于更新后的真实次品率 p_{real} 难以直接测定，本文采用不完全 Beta 分布，依据抽样检测数据来估算这一真实值 p。为了更准确地描绘次品率的分布特性，本文首先利用问题一中构建的：基于不完全 Beta 函数的抽样检测模型来计算次品率的置信区间。然后，依据问题一所建立的框架，认定此随机抽样过程应遵循不完全 Beta 函数的分布，而非均匀分布。因此，本文采用蒙特卡洛模拟方法，基于不完全 Beta 分布进行抽样，旨在更精确地更新并反映真实的次品率值。随后，再根据问题二与问题三所建立总成本函数，分别采用基于动态规划和强化学习的联合求解模型与加权算法融合模型进行求解，找到最优决策组合，以实现总成本的最小化。

8.2 模型建立

8.2.1 模型参数设定

表18 基本参数设定

序号	符号	含义
1	k	次品数
2	n	抽样样本总量
3	α	置信水平
4	$\hat{p}_i$	次品率
5	$\hat{p}_{real}$	第 i 次模拟的真实次品率
6	n_{min}	最小样本容量
7	p_{real}	蒙特卡洛更新后的真实次品率

8.2.2 次品率估计模型

通过问题一中的抽样检测，本文采用不完全 Beta 函数对零配件、半成品和成品的次品率进行精确估计。

次品率的置信区间为：

$$CI=\left[I\left(\frac{1-\alpha}{2},k+1,n-k+1\right),I\left(1-\frac{1-\alpha}{2},k+1,n-k\right)\right] \tag{8.1}$$

其中，k 为次品数；n 为样本总量；α 为置信水平。

8.2.3 蒙特卡洛模拟：

为了更准确地反映次品率的分布特性，本文在计算出更新后真实次品率 p_{real} 的置信区间后，进一步基于不完全 Beta 分布进行蒙特卡洛模拟。具体步骤如下：

1.从不完全 Beta 函数的分布中进行随机抽样

假设 k 和 n 分别为次品数和样本总量，次品率 $\hat{p}_i$ 的模拟样本来自以下 Beta 分布：

$$\hat{p}_i \sim I_{\frac{1-\alpha}{2}}(k+1,n-k+1) \tag{8.2}$$

本文通过从该分布中进行 n 次随机抽样，以全面反映次品率的概率分布：

$$\hat{p}_{real} = Beta.rvs(k+1,n-k+1,size=n) \tag{8.3}$$

其中，$\hat{p}_{real}$ 表示第 i 次模拟的真实次品率；根据问题一结果，此处最小样本量 n_{min} 为 3501。

2.通过模拟样本更新次品率

蒙特卡洛模拟得到的样本集合 $\{\hat{p}_{\text{real}},\hat{p}_{\text{real}},\ldots,\hat{p}_{\text{real}}\}$，代表了次品率的不同可能值。最终，通过模拟样本的均值来更新次品率：

$$p_{real}=\frac{1}{n}\sum_{i=1}^{n}\hat{p}_i$$

该均值即为更新后真实的次品率。

3.总成本模型

企业在生产过程中需要做出一系列决策，如是否检测零配件和成品，是否拆解不合格成品等，其核心目标是最小化总成本。总成本模型函数如下：

在问题二中，生产过程包括零配件采购、检测、成品装配、不合格品处理（包括调换和拆解），总成本函数表达式为：

$$\begin{aligned}C_{\text{总成本}} &= C_{\text{零配件1 采购}}+C_{\text{零配件 2 采购}}+C_{\text{零配件 1 检测}}+C_{\text{零配件 2 检测}}\\&\quad+C_{\text{装配}}+C_{\text{成品检测}}+C_{\text{调换损失}}+C_{\text{拆解}}\end{aligned} \tag{8.4}$$

而问题三相较于问题二，增加了多道工序和半成品的处理环节，其目标是通过多阶段的生产决策来最小化成本。总成本函数为：

$$\begin{aligned}C_{total} &= \sum_{i=1}^{8}C_{\text{零配件}i\text{ 采购}}+\sum_{i=1}^{8}C_{\text{零配件}i\text{ 采购}}+C_{\text{半成品装配}}+C_{\text{成品装配}}\\&\quad+C_{\text{成品检测}}+C_{\text{调换损失}}+C_{\text{拆解成本}}-C_{\text{回收收入}}\end{aligned} \tag{8.5}$$

总成本模型的具体成分构成与含义，请参见前文 6.2.2 与 7.2.3 部分的详细阐述。

8.2.4 问题二：基于动态规划和强化学习的联合求解模型

在问题二的求解中，本文使用了动态规划与 Q-learning 强化学习的混合模型。此模型旨在通过多阶段优化策略，找出企业生产过程中的最佳决策组合，以实现总成本的最小化。该模型的具体细节已在 6.2.3 节中详细阐述，以下仅概述其两个核心部分。

动态规划主要用于处理确定性情境。它采用自下而上的方式，通过递推方程逐步计算出每个阶段的最优决策，从而确保在生产流程中成本最小化。递推方程可表示为：

$$V(s_t) = \max_{a_t}[r(s_t, a_t) + \gamma V(s_{t+1})] \tag{8.6}$$

而 Q-learning 强化学习部分则专注于处理不确定性环境下的最优决策。Q-learning 算法通过不断迭代，更新每个状态-动作对的价值函数，以优化决策策略。其更新公式为：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{8.7}$$

在混合模型中，动态规划负责处理明确且可预期的部分，而 Q-learning 则通过探索和学习来优化处理不确定性。两者相结合，通过迭代优化，最终能够找到全局最优的检测、装配和拆解策略。此联合求解模型能够充分发挥动态规划和 Q-learning 各自的优势，从而更有效地解决复杂生产过程中的决策问题。

8.2.5 问题三：加权算法融合模型

针对问题三中生产流程的复杂性，本文采用了加权融合模型。此模型融合了动态规划、Q-learning 和贪心算法的优势，以构建一个全面的决策支持系统。该模型在 7.2.4 节中已有详细阐述，以下仅概述其核心部分。

加权融合模型通过以下方式求解最优解：

$$\min C_{total} = \omega_1 \cdot C_{DP} + \omega_2 \cdot C_{QL} + \omega_3 \cdot C_{GA} \tag{8.8}$$

其中，动态规划负责处理明确的生产流程决策，确保在多阶段生产过程中选择最优的检测和装配策略；Q-learning 在面临生产过程中不确定性时，通过学习历史数据中的经验，不断更新策略，最终在复杂环境中获得更优的决策结果；贪心算法通过逐步选择局部最优解，实现快速收敛。在每一步决策时，贪心算法会选择当前对总成本影响最小的动作，确保在快速解决问题的同时避免过度复杂的计算。

加权融合模型最终通过动态调整权重，充分利用三种方法的优势，保证在复杂多工序的生产环境中，找到最优决策组合，以实现总成本的最小化。

8.3 模型求解

8.3.1 问题二求解

首先，借助蒙特卡洛模拟出次品率。具体结果如下所示：

表 19 蒙特卡洛更新次品率（针对问题二）

情况	零配件 1 更新后的次品率	零配件 2 更新后的次品率	成品更新后的次品率
情况 1	0.100222	0.100332	0.100230
情况 2	0.200049	0.200322	0.200198
情况 3	0.100174	0.100232	0.100178
情况 4	0.200038	0.200125	0.200060
情况 5	0.100334	0.200009	0.100116
情况 6	0.050188	0.050236	0.050305

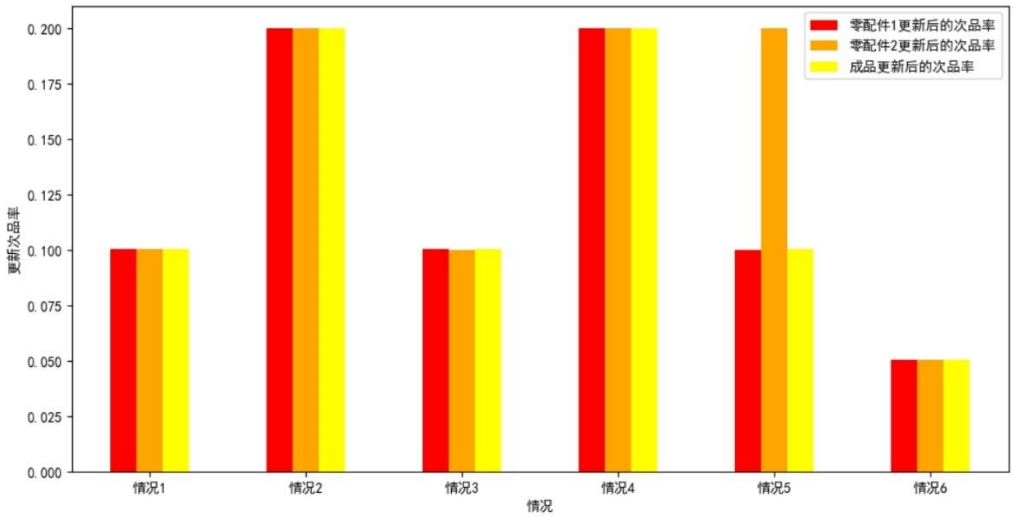

图 9 蒙特卡洛更新次品率结果图（针对问题二）

图中，横轴代表企业在生产中面临的六种情况，纵轴代表更新的次品率。红色方柱代表零配件 1 更新后的次品率，橙色方柱代表零配件 2 更新后的次品率，黄色方柱代表成品更新后的次品率。

通过对比分析，可以观察出更新后的次品率与题目中给出的真实次品率之间存在着极为细微的差异，这一发现从某种程度上有力地验证了所构建模型的准确性和可靠性。

在此基础上，本文设定企业生产成品数量为 1000 件。通过 Python 软件实现上述基于动态规划和强化学习的联合求解模型，揭示出六种不同情境下，针对 16 种潜在的决策组合，企业所需承担的总成本。并计算出最优决策组合下企业所承担的单位成本。完整结果详见支撑材料。

本文绘制了折线图，从而为企业决策者提供了直观、精准的参考依据。

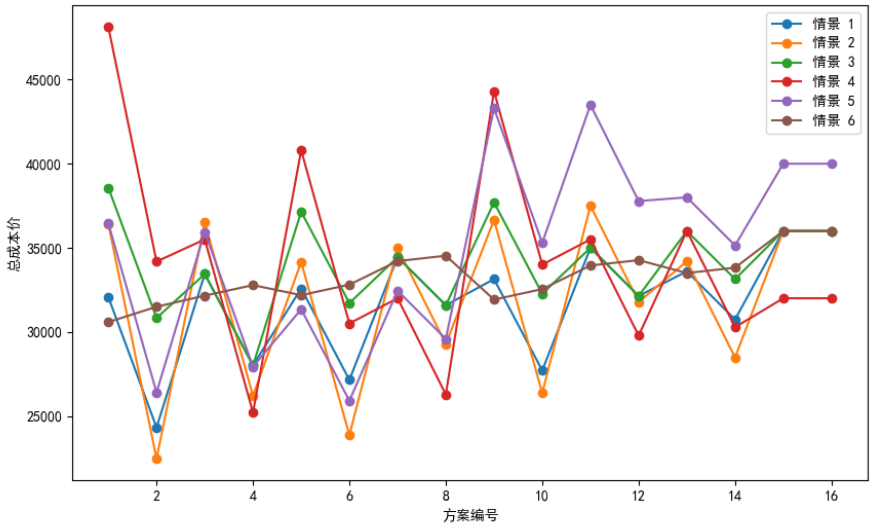

图 10 六种情境下不同决策对应的总成本

在所提供的图中，横坐标表示各个决策方案的编号，纵坐标表示企业在不同情况和方案下所需承担的总体成本。通过观察可以发现，成本效益最为显著的决策大多集中在方案 2 至方案 8 这一区间内，它们的成本明显低于其他方案。

以下是对不同情况下最优决策的梳理：

表 20 不同情况下的最优决策方案

情况序号	最优决策序号	是否检测零配件 1	是否检测零配件 2	是否检测成品	是否拆解不合格成品	总成本/元	单位成本/元
1	2	0	0	0	1	24313.97	24.31
2	2	0	0	0	1	22472.35	22.47
3	4	0	0	1	1	28006.01	28.01
4	4	0	0	1	1	25204.03	25.20
5	6	0	1	0	1	25904.56	25.90
6	1	0	0	0	0	30596.25	30.60

其中，0 代表不检测或不拆解，1 代表检测或拆解。

综上所述，得到以下结果：

对于情况一，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 24.31 元；

对于情况二，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 22.47 元；

对于情况三，其最优决策为：不检测零配件 1、不检测零配件 2、检测成品、拆解不合格成品，企业所承担的单位成本为 28.01 元；

对于情况四，其最优决策为：不检测零配件 1、不检测零配件 2、检测成品、拆解不合格成品，企业所承担的单位成本为 25.20 元；

对于情况五，其最优决策为：不检测零配件 1、检测零配件 2、不检测成品、拆解不合格成品，企业所承担的单位成本为 25.90 元；

对于情况六，其最优决策为：不检测零配件 1、不检测零配件 2、不检测成品、不拆解不合格成品，企业所承担的单位成本为 30.60 元。

8.3.2 问题三求解

首先，借助蒙特卡洛模拟出次品率。具体结果如下所示：

表 21 蒙特卡洛更新次品率（针对问题三）

项目	更新次品率
零配件 1	0.115113
零配件 2	0.213047
零配件 3	0.183772
零配件 4	0.074313
零配件 5	0.153360
零配件 6	0.056304
零配件 7	0.136059
零配件 8	0.096016
半成品 1	0.130735
半成品 2	0.184355
半成品 3	0.107215
成品	0.141291

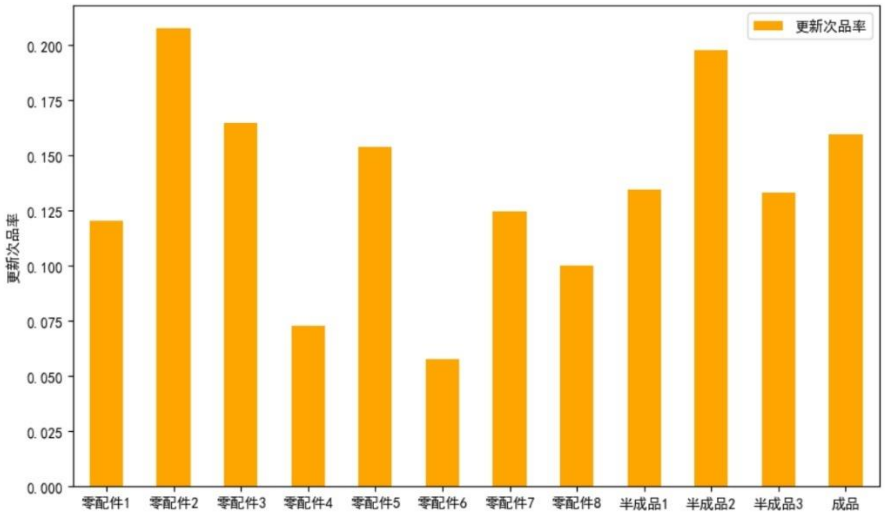

图 11 蒙特卡洛更新次品率结果图（针对问题三）

图中，横轴代表企业在生产流程中所涉及到的所有中间材料（如零配件 1、半成品 1 等）及成品，纵轴代表更新的次品率。

在此基础上，本文设定企业生产成品数 $N=1000$。通过动态规划、强化学习、贪心算法来分别求解出各自算法对应的最优决策组合与最低总成本（如下表所示），并结合权重动态调整的策略，依据公式(7.13)对三种算法的结果进行加权融合。

表 22 动态规划、强化学习、贪心算法对应求解结果

方法	决策组合				最低成本/元	最低单位成本/元
	是否检验零配件	是否检验半成品	是否检验成品	是否进行拆解		
动态规划	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	96090.10	96.10
强化学习	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	22102.28	22.10
贪心算法	(0,0,0,0,0,0,0,0)	(0,0,0)	(1)	(1)	22102.28	22.10

其中，0 代表不进行检验或者不进行拆解，1 代表进行检验或者进行拆解。

最终得出的最优决策方案为对零配件不进行检验、对半成品不进行检验、对成品进行检验、以及对不合格成品进行拆解，其总成本为 10498.58 元，换算成成品单位成本则为 10.50 元。

8.3.3 灵敏度分析

本文针对问题 2 与问题 3 中的次品率问题进行了灵敏度分析，旨在全面评估零配件、半成品及成品次品率波动对生产流程的具体影响。

分析发现，无论是在涉及单一零配件装配的简单情境（问题 2）中，还是在涵盖多道工序的复杂生产流程（问题 3）里，零配件与半成品的次品率均对最终成品的次品率产生了显著影响。

首先，针对问题 2 的分析表明，零配件次品率的任何变动都会直接影响成品的次品率。尤其当零配件次品率处于较高水平时，成品的次品率几乎呈现出一种线性增长的趋势。因此，企业应优先控制次品率较高的零配件质量，以降低成品的次品率。

其次，在问题 3 的分析中，本文进一步发现半成品次品率的波动同样对成品次品率产生了显著影响。在涉及多道工序的复杂生产流程中，半成品与零配件次品率的联动效应尤为突出。因此，通过严格控制半成品及关键零配件的质量，可以显著提升成品的合格率。

综上所述，次品率较高的零配件与半成品对成品次品率的影响显得尤为敏感。鉴于

此，企业应在检测与质量控制方面，优先将资源投入到这些关键环节。通过不断优化这些环节的质量控制措施，企业不仅能够有效提升成品质量，减少次品的产生，还能够在整体上降低生产成本，从而实现质量与效益的双重提升。

九、模型的评价与改进

9.1 模型评价

9.1.1 模型优点

1.本研究采用不完全 Beta 函数作为核心计算手段，旨在精确评估二项分布的累积概率。此方法直接输出精确的统计结果，有效规避了仅依赖中心极限定理进行正态近似所带来的诸多局限性，诸如近似精度欠缺、对样本规模变动的敏感性高、对分布偏度与峰度的过度反应、统计推断稳健性削弱以及假设检验可靠性下降等问题；

2.模型考虑了企业生产环节中零配件的自然折旧因素，使得所构建的成本模型更加贴近实际运营情况，增强了模型的现实应用价值和指导意义；

3.本研究创新性地融合动态规划与强化学习技术，提出了一种联合求解模型。该模型在探索最优决策组合时展现出更高的精确度，为复杂决策问题提供了新颖且有效的解决方案。

9.1.2 模型缺点

1.尽管模型具备高精度，但其中涉及的不完全 Beta 函数计算在处理大规模样本时，计算复杂度显著提升；

2.该模型假设每个检测过程是独立的，实际中由于批次质量波动或检测工具误差，这种假设可能并不完全成立，影响模型的准确性。

9.2 模型改进

1.动态规划与启发式算法的结合

现有模型主要使用动态规划和 Q-learning 算法进行优化，但这些算法在大规模组合决策中存在计算复杂度过高的问题。为此，可以将启发式算法与动态规划相结合，快速缩小解空间。

2.考虑实际检测过程中的误差与不确定性

目前的模型假设检测过程是准确无误的，实际应用中检测设备和环境会带来一定的误差与不确定性。可以进一步引入检测误差与不确定性因素，使得模型更加贴近现实。

十、参考文献

[1] 汪亚顺, 王振宇, 姚凯, 等. 机电设备可靠性验证试验统计方案设计方法研究[J]. 机械设计与研究, 2017, 33(4): 45-52.
[2] 宗星煜, 李慧珍, 赵学尧, 等. 基于SCORE-TCM的中医药团体标准抽样评价方案设计[J]. 中国中医基础医学杂志, 2024, 30(1): 66-72.
[3] 刘民. 基于数据的生产过程调度方法研究综述[J]. 自动化技术与应用, 2019, 38(3): 12-17.
[4] 丁明, 楚明娟, 毕锐, 石文辉. 基于序贯蒙特卡洛随机生产模拟的风电接纳能力评价方法及应用[J]. 电力自动化设备, 2016, 36(9): 76-80.
[5] 杨雪清, 张伟. 森林可燃物大样地抽样调查方法研究[J]. 森林科学研究, 2018, 31(6): 100-108.
[6] 钱琼. 生产计划问题的决策模型研究[D]. 清华大学, 2016.
[7] 赵鸣, 陈凯. 钢筋混凝土房屋混凝土质量检测抽样方案选择[J]. 建筑结构, 2018, 40(3): 102-110.
[8] 韦博成. 参数统计教程[M]. 北京: 高等教育出版社, 2010.
[9] 卯诗松, 周纪芗. 工程统计学[M]. 北京: 高等教育出版社, 2015.
[10] 张崇岐, 燕飞. 工业统计学[M]. 北京: 高等教育出版社, 2012.

附录

支撑材料清单：

1：问题一

1-1：方法一假设检验.py

1-2：方法一求解样本量.py

1-3：方法一结果画图代码.py

1-4：方法二假设检验.py

1-5：方法二求解样本量.py

1-6：方法二结果画图代码.py

1-7：方法一和方法二对比图生成代码.py

2：问题二

文件一：问题二代码

2-1：利用动态规划和强化学习求解.py

文件二：问题二支撑数据

2-1：问题二 6 个情景下的单位成本及总成本.xlsx

3：问题三

文件一：问题三代码

3-1：求解动态规划下的最优值.py

3-2：求解强化学习下的最优解.py

3-3：求解贪心算法下的最优解.py

文件二：问题三支撑数据

3-1：结果-动态规划.xlsx

3-2：结果-强化学习.xlsx

3-3：结果-贪心算法.xlsx

4：问题四

文件一：问题四代码

4-1：估计问题二的各类次品率的值.py

4-2：估计问题三的各类次品率的值.py

文件二：问题四支撑数据

4-1：问题二蒙特卡洛更新次品率.xlsx

4-2：问题三蒙特卡洛更新次品率.xlsx

4-3：更新后的次品率求解问题二的结果表.xlsx

4-4：更新后的次品率求解问题三的结果表（动态规划）.xlsx

4-5：更新后的次品率求解问题三的结果表（强化学习）.xlsx

4-6：更新后的次品率求解问题三的结果表（贪心方法）.xlsx

1.问题一
1-1:方法一假设检验 import numpy as np import math # 设定 SPRT 的边界值 def get_boundaries(alpha, beta):

```
    # SPRT 中的接受和拒绝边界
    A = beta / (1 - alpha)  # 接受 H0 的临界值
    B = (1 - beta) / alpha  # 拒绝 H0 的临界值
    return A, B
# 计算似然比
def compute_likelihood_ratio(k, n, p0, p_real, case_type):
    # k: 当前次品数
    # n: 当前抽样数
    if case_type == 1:
        # 情形 1: 原假设 H0: p <= p0, 备择假设 H1: p > p0
        return (p_real ** k * (1 - p_real) ** (n - k)) / (p0 ** k * (1 - p0) ** (n - k))
    elif case_type == 2:
        # 情形 2: 原假设 H0: p >= p0, 备择假设 H1: p < p0
        return (p0 ** k * (1 - p0) ** (n - k)) / (p_real ** k * (1 - p_real) ** (n - k))
# SPRT 检验函数
def sprt_test(p0, p_real, alpha, beta, case_type=1, max_samples=100):
    A, B = get_boundaries(alpha, beta)  # 获取接受和拒绝的临界值
    k = 0  # 初始化次品数
    for n in range(1, max_samples + 1):
        # 模拟抽样过程
        sample = np.random.binomial(1, p_real)  # 每次抽样一个零件，p_real 表示真实次品率
        k += sample  # 累积次品数
        # 计算当前的似然比
        lr = compute_likelihood_ratio(k, n, p0, p_real, case_type)
        print(f"样本数: {n}，次品数: {k}，似然比: {lr}")
        # 根据边界判断是否接受或拒绝原假设
        if lr <= A:
            if case_type == 1:
                return f"接受原假设 H0：次品率 <= {p0}，检测终止"
            else:
                return f"接受原假设 H0：次品率 >= {p0}，检测终止"
        elif lr >= B:
            if case_type == 1:
                return f"拒绝原假设 H0，接受 H1：次品率 > {p0}，检测终止"
            else:
                return f"拒绝原假设 H0，接受 H1：次品率 < {p0}，检测终止"
    return "达到最大样本量，未能做出明确判断"
# 参数设置
p0 = 0.10  # 标称次品率
p_real = 0.12  # 真实次品率（同一批次）
# 情形 1： α = 0.05, β = 0.10
alpha1 = 0.05
```

```
beta1 = 0.10
# 情形 2： α = 0.10, β = 0.15
alpha2 = 0.10
beta2 = 0.15
max_samples = 10000  # 最大抽样数
# 运行情形 1 的假设检验
print("情形 1 的检验结果：")
result1 = sprt_test(p0, p_real, alpha1, beta1, case_type=1, max_samples=max_samples)
print(result1)
# 运行情形 2 的假设检验
print("\n 情形 2 的检验结果：")
result2 = sprt_test(p0, p_real, alpha2, beta2, case_type=2, max_samples=max_samples)
print(result2)
```

```
1-2：方法一求解样本量
import math
# 设定函数参数
def estimate_sample_size(p_real, p0, alpha, beta):
    # 计算接受和拒绝界限
    A = math.log((1 - beta) / alpha)  # 拒绝界限
    B = math.log(beta / (1 - alpha))  # 接受界限
    # 对数似然比的期望值
    E_L = p_real * math.log(p_real / p0) + (1 - p_real) * math.log((1 - p_real) / (1 - p0))
    # 样本量估算
    n_reject = A / E_L  # 拒收时所需样本数
    n_accept = B / E_L  # 接收时所需样本数
    return abs(n_reject), abs(n_accept)
# 情形 1： α = 0.05, β = 0.10
alpha1 = 0.05
beta1 = 0.10
p_real = 0.08  # 真实的次品率
p0 = 0.10  # 标称次品率
# 情形 2： α = 0.10, β = 0.15
alpha2 = 0.10
beta2 = 0.15
# 估算样本数量，情形 1
n_reject_case1, n_accept_case1 = estimate_sample_size(p_real, p0, alpha1, beta1)
print(f"情形 1：拒收需要的样本数量 = {n_reject_case1}，接收需要的样本数量 =
{n_accept_case1}")
# 估算样本数量，情形 2
n_reject_case2, n_accept_case2 = estimate_sample_size(p_real, p0, alpha2, beta2)
print(f"情形 2：拒收需要的样本数量 = {n_reject_case2}，接收需要的样本数量 =
```

```
{n_accept_case2}")
```

1-3：方法一结果图

```
import matplotlib.pyplot as plt
import matplotlib as mpl
# 确保 matplotlib 能够正确显示中文
mpl.rcParams['font.sans-serif'] = ['SimHei']  # 设置字体为 SimHei 以支持中文显示
mpl.rcParams['axes.unicode_minus'] = False  # 解决负号'-'显示为方块的问题
# p*的值
p_star_values = [0.08, 0.09, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16]
# 95% 置信水平下的样本量 n
sample_sizes_95 = [1220, 5045, 5354, 1375, 627, 361, 236, 168]
# 90% 置信水平下的样本量 n
sample_sizes_90 = [756, 3127, 3319, 852, 388, 224, 146, 104]
# 调整颜色和标记样式，增加线宽
plt.figure(figsize=(10, 6))
plt.plot(p_star_values, sample_sizes_95, label='95% 置信水平', color='#1f77b4', marker='o', linestyle='-
-', markersize=10, linewidth=2)
plt.plot(p_star_values, sample_sizes_90, label='90% 置信水平', color='#ff7f0e', marker='s', linestyle='-.',
markersize=10, linewidth=2)
# 设置图像标题和标签（中文）
plt.xlabel("真实次品率 ($p^*$)", fontsize=14)
plt.ylabel("样本量 (n)", fontsize=14)
# 去掉网格线
# plt.grid(False)
# 优化图例，增加阴影效果
plt.legend(fontsize=12, loc='upper right', shadow=True)
# 显示图像
plt.show()
```

1-4：方法二假设检验

```
import numpy as np
import scipy.stats as stats
# 不完全 Beta 函数法的假设检验
def beta_binomial_test(k, n, alpha, p0, case_type=1):
    """
    基于不完全 Beta 函数的假设检验
    :param k: 次品数
    :param n: 样本数
    :param alpha: 置信水平 (1 - α)
    :param p0: 标称次品率
    :param case_type: 情形类型 (1 或 2)
```

```
    :return: 检验结果，接受或拒绝原假设
    """
    # 计算 Beta 分布的累积分布函数（CDF）
    beta_cdf = stats.betabinom.cdf(k, n, p0, 1 - p0)
    # 根据情形不同，决定如何使用不完全 Beta 函数
    if case_type == 1:
        # 情形 1: H0: p <= p0, H1: p > p0
        if beta_cdf <= alpha:
            return f"拒绝原假设 H0，接受 H1：次品率 > {p0}"
        else:
            return f"接受原假设 H0：次品率 <= {p0}"
    elif case_type == 2:
        # 情形 2: H0: p >= p0, H1: p < p0
        if beta_cdf >= (1 - alpha):
            return f"拒绝原假设 H0，接受 H1：次品率 < {p0}"
        else:
            return f"接受原假设 H0：次品率 >= {p0}"
# 模拟抽样并计算次品数
def simulate_defects(p_real, n):
    """
    根据真实次品率 p_real 和样本量 n，模拟观测到的次品数
    :param p_real: 真实次品率
    :param n: 样本量
    :return: 观测到的次品数 k
    """
    return np.random.binomial(n, p_real)
# 参数设置
p0 = 0.10  # 标称次品率
p_real = 0.09  # 假定的真实次品率
n = 50  # 样本量
# 情形 1： α = 0.05 (95% 置信水平)
alpha1 = 0.05
# 情形 2： α = 0.10 (90% 置信水平)
alpha2 = 0.10
# 模拟抽样次品数
k = simulate_defects(p_real, n)
# 运行情形 1 的假设检验
print("情形 1 的检验结果：")
result1 = beta_binomial_test(k, n, alpha1, p0, case_type=1)
print(result1)
# 运行情形 2 的假设检验
print("\n 情形 2 的检验结果：")
```

```
result2 = beta_binomial_test(k, n, alpha2, p0, case_type=2)
print(result2)
```

1-5：方法二求解样本量

```
import scipy.stats as stats
# 定义不完全 Beta 函数的计算 (CDF for Beta distribution)
def beta_cdf(x, a, b):
    return stats.beta.cdf(x, a, b)
# 计算给定样本量 n 下的置信区间
def calc_confidence_interval(n, k, alpha, p_0):
    # 下界 L，对应的是累积概率 alpha/2
    L = stats.beta.ppf(alpha / 2, k + 1, n - k)
    # 上界 U，对应的是累积概率 1 - alpha/2
    U = stats.beta.ppf(1 - alpha / 2, k + 1, n - k)
    return L, U
# 核心函数：迭代求解最小样本量 n 满足 U - L = 2E
def find_sample_size(p_0, alpha, E, initial_n=30, max_iterations=100000):
    n = initial_n  # 初始样本量
    k = int(p_0 * n)  # 假设次品数与样本量的次品率一致
    iteration = 0
    step_size = 1  # 设置动态步长
    while iteration < max_iterations:
        iteration += 1
        # 计算置信区间
        L, U = calc_confidence_interval(n, k, alpha, p_0)
        # 计算置信区间宽度
        width = U - L
        # 输出当前样本量和置信区间
        print(f"n = {n}, L = {L:.5f}, U = {U:.5f}, width = {width:.5f}")
        # 如果宽度满足要求 (U - L = 2E)，则停止迭代
        if abs(width - 2 * E) < 1e-3:  # 放宽终止条件为 1e-3
            return n
        # 动态调整步长，防止在大样本量时过慢增加
        if n > 1000:
            step_size = 100
        elif n > 100:
            step_size = 10
        # 否则，增大样本量，继续迭代
        n += step_size
        k = int(p_0 * n)  # 更新次品数与样本量匹配
    # 如果达到最大迭代次数
    print("未能在指定迭代次数内找到合适的样本量，返回当前 n")
```

```
        return n
# 参数设置
p_0 = 0.10   # 次品率 10%
E = 0.05   # 允许误差 5%
# (1) 在 95%的置信水平下的样本量计算 (拒收方案)
alpha_95 = 0.05   # 95% 置信水平
print("情形 (1): 95% 的置信度下，拒收这批零配件")
sample_size_95 = find_sample_size(p_0, alpha_95, E)
print(f"在 95%的置信水平下，最终计算得到的样本量 n: {sample_size_95}")
# (2) 在 90%的置信水平下的样本量计算 (接收方案)
alpha_90 = 0.10   # 90% 置信水平
print("\n 情形 (2): 90% 的置信度下，接收这批零配件")
sample_size_90 = find_sample_size(p_0, alpha_90, E)
print(f"在 90%的置信水平下，最终计算得到的样本量 n: {sample_size_90}")
```

1-6：方法二结果画图

```
import matplotlib.pyplot as plt
import matplotlib as mpl
# 确保 matplotlib 能够正确显示中文
mpl.rcParams['font.sans-serif'] = ['SimHei']   # 设置字体为 SimHei 以支持中文显示
mpl.rcParams['axes.unicode_minus'] = False   # 解决负号'-'显示为方块的问题
# p*的值
p_star_values = [0.08, 0.09, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16]
# 95% 置信水平下的样本量 n
sample_sizes_95 = [681, 2901, 3501, 971, 471, 291, 191, 141]
# 90% 置信水平下的样本量 n
sample_sizes_90 = [481, 2101, 2501, 681, 341, 201, 141, 98]
# 调整颜色和标记样式，增加线宽
plt.figure(figsize=(10, 6))
plt.plot(p_star_values, sample_sizes_95, label='95% 置信水平', color='#1f77b4', marker='o', linestyle='--', markersize=10, linewidth=2)
plt.plot(p_star_values, sample_sizes_90, label='90% 置信水平', color='#ff7f0e', marker='s', linestyle='-.', markersize=10, linewidth=2)
# 设置图像标题和标签（中文）
plt.xlabel("真实次品率 ($p^*$)", fontsize=14)
plt.ylabel("样本量 (n)", fontsize=14)
# 去掉网格线
# plt.grid(False)
# 优化图例，增加阴影效果
plt.legend(fontsize=12, loc='upper right', shadow=True)
# 显示图像
plt.show()
```

1-7：方法一和方法二对比图

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib as mpl
# 设置中文字体
mpl.rcParams['font.sans-serif'] = ['SimHei']  # 设置黑体
mpl.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题
# 真实次品率 p*
p_values = np.array([0.08, 0.09, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16])
# 第一种方法 (SPRT) 样本量数据
n_sprt_95 = np.array([1220, 5045, 5354, 1375, 627, 361, 236, 168])
n_sprt_90 = np.array([756, 3127, 3319, 852, 388, 224, 146, 104])
# 第二种方法 (不完全 Beta 函数) 样本量数据
n_beta_95 = np.array([681, 2901, 3501, 971, 471, 291, 191, 141])
n_beta_90 = np.array([481, 2101, 2501, 681, 341, 201, 141, 98])
# 设置图像大小和分辨率
plt.figure(figsize=(10, 6), dpi=100)
# 创建 95% 置信水平的折线图
plt.plot(p_values, n_sprt_95, label='SPRT (95% 信度)', marker='o', linestyle='-', color='#1F77B4', linewidth=2)  # 修改颜色
plt.plot(p_values, n_beta_95, label='不完全 Beta 函数 (95% 信度)', marker='s', linestyle='-', color='green', linewidth=2)
# 创建 90% 置信水平的折线图
plt.plot(p_values, n_sprt_90, label='SPRT (90% 信度)', marker='o', linestyle='--', color='orange', linewidth=2)
plt.plot(p_values, n_beta_90, label='不完全 Beta 函数 (90% 信度)', marker='s', linestyle='--', color='red', linewidth=2)
# 添加标题和标签
plt.xlabel('真实次品率 (p*)', fontsize=14)
plt.ylabel('样本量 (n)', fontsize=14)
# 设置图例位置
plt.legend(loc='upper right', fontsize=12)
# 移除背景方格线
plt.grid(False)
# 美化坐标轴
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
# 显示图形
plt.tight_layout()
plt.show()
```

2. 问题二

2-1：利用动态规划和强化学习求解

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from matplotlib import rcParams
rcParams['font.sans-serif'] = ['SimHei']
rcParams['axes.unicode_minus'] = False
# 生产产品数量和各种成本参数定义
N = 1000
a11 = [0.1, 0.2, 0.1, 0.2, 0.1, 0.05]
a12 = [0.1, 0.2, 0.1, 0.2, 0.2, 0.05]
a21 = [4, 4, 4, 4, 4, 4]
a22 = [18, 18, 18, 18, 18, 18]
a31 = [2, 2, 2, 1, 8, 2]
a32 = [3, 3, 3, 1, 1, 3]
a13 = [0.1, 0.2, 0.1, 0.2, 0.1, 0.05]
c4 = [6, 6, 6, 6, 6, 6]
a33 = [3, 3, 3, 2, 3, 3]
c1 = [56, 56, 56, 56, 56, 56]
c2 = [6, 6, 30, 30, 10, 10]
c3 = [5, 5, 5, 5, 5, 40]
# 成本计算函数
def production_cost(N, a11, a12, a13, a21, a22, a31, a32, a33, c1, c2, c3, c4, detect_part1, detect_part2,
detect_final, disassemble):
    part_quality_rate1 = 1 - a11 * (1 - detect_part1)
    part_quality_rate2 = 1 - a12 * (1 - detect_part2)
    product_quality_rate = part_quality_rate1 * part_quality_rate2 * (1 - a13 * (1 - detect_final))
    part_purchase_cost1 = N / part_quality_rate1 * a21
    part_purchase_cost2 = N / part_quality_rate2 * a22
    part_testing_cost1 = detect_part1 * N * a31
    part_testing_cost2 = detect_part2 * N * a32
    assembly_cost = N * c4
    final_testing_cost = detect_final * N * a33
    replacement_cost = (1 - detect_final) * N * (1 - product_quality_rate) * c2
    scrap_cost = disassemble * N * (1 - product_quality_rate) * c3
    scrap_revenue = disassemble * N * (1 - product_quality_rate) * 0.6 * c1
    total_cost = part_purchase_cost1 + part_purchase_cost2 + part_testing_cost1 + part_testing_cost2 +
assembly_cost + final_testing_cost + replacement_cost + scrap_cost - scrap_revenue
    return total_cost
# Q-learning 和 动态规划的参数
gamma = 0.9  # 折扣因子
```

```
alpha = 0.1  # 学习率
epsilon = 0.1  # 探索率
num_episodes = 500  # 训练轮次
num_states = 16  # 状态数量
num_actions = 4  # 动作数量
# 初始化 Q 表
Q = np.zeros((num_states, num_actions))
# 状态编码器
def encode_state(detect_part1, detect_part2, detect_final, disassemble):
    return detect_part1 * 8 + detect_part2 * 4 + detect_final * 2 + disassemble
# 探索-利用策略
def choose_action(state, epsilon):
    if np.random.uniform(0, 1) < epsilon:
        return np.random.choice(num_actions)  # 探索：随机选择动作
    else:
        return np.argmax(Q[state, :])  # 利用：选择 Q 值最大的动作
# Q-learning 训练
def q_learning_train():
    global Q
    for episode in range(num_episodes):
        state = encode_state(np.random.choice([0, 1]), np.random.choice([0, 1]), np.random.choice([0, 1]), np.random.choice([0, 1]))
        for t in range(100):  # 每轮训练最多执行 100 次动作
            action = choose_action(state, epsilon)
            detect_part1 = (action // 8) % 2
            detect_part2 = (action // 4) % 2
            detect_final = (action // 2) % 2
            disassemble = action % 2
            # 计算当前状态下的成本
            cost = production_cost(N, a11[0], a12[0], a13[0], a21[0], a22[0], a31[0], a32[0], a33[0], c1[0], c2[0], c3[0], c4[0],detect_part1, detect_part2, detect_final, disassemble)
            reward = -cost  # 成本的负值作为奖励
            next_state = encode_state(detect_part1, detect_part2, detect_final, disassemble)
            # 更新 Q 值
            Q[state, action]=Q[state, action]+alpha*(reward+gamma * np.max(Q[next_state, :]) - Q[state, action])
            state = next_state
# 动态规划（价值迭代）
def value_iteration():
    V = np.zeros(num_states)
    policy = np.zeros(num_states)
    for iteration in range(100):  # 迭代次数
```

```
            delta = 0
            for state in range(num_states):
                old_value = V[state]
                Q_values = []
                for action in range(num_actions):
                    detect_part1 = (action // 8) % 2
                    detect_part2 = (action // 4) % 2
                    detect_final = (action // 2) % 2
                    disassemble = action % 2
                    cost = production_cost(N, a11[0], a12[0], a13[0], a21[0], a22[0], a31[0], a32[0],
a33[0], c1[0], c2[0], c3[0], c4[0],detect_part1, detect_part2, detect_final, disassemble)
                    reward = -cost
                    Q_values.append(reward + gamma * V[state])  # 未来的折扣奖励
                V[state] = max(Q_values)  # 更新状态的价值
                policy[state] = np.argmax(Q_values)  # 获取最优动作
                delta = max(delta, np.abs(old_value - V[state]))
            if delta < 1e-6:  # 判断收敛
                break
        return policy, V
# 计算每个情景下的单位成本
def calculate_unit_costs():
        unit_costs = np.zeros((6, 16))  # 6 个情景，每个情景有 16 个方案
        for i in range(6):
            for j in range(16):
                detect_part1 = (j // 8) % 2
                detect_part2 = (j // 4) % 2
                detect_final = (j // 2) % 2
                disassemble = j % 2
                unit_costs[i, j] = production_cost(N, a11[i], a12[i], a13[i], a21[i], a22[i], a31[i], a32[i],
a33[i], c1[i], c2[i], c3[i], c4[i],detect_part1, detect_part2, detect_final, disassemble)
        return unit_costs
# 获取每个情景下的最优总成本
def calculate_optimal_costs(unit_costs):
        return np.min(unit_costs, axis=1)  # 每个情景下，选出最小成本的方案作为最优总成本
# 打印每个情景下详细成本信息
def print_cost_data(unit_costs):
        for i in range(unit_costs.shape[0]):
            print(f"\n 情景 {i+1} 各方案单位成本价数据:")
            print(f"{'方案编号':<10}{'单位成本价(总成本/1000)':<30}{'总成本价'}")
            for j in range(unit_costs.shape[1]):
                total_cost = unit_costs[i, j]
                unit_cost = total_cost / 1000
```

```
                print(f"{j+1:<10}{unit_cost:<30}{total_cost}")
# 生成类似你上传的图形
def plot_cost_comparison(unit_costs, optimal_costs):
    # 图 1：不同情景下各方案单位成本对比（去掉网格线）
    plt.figure(figsize=(10, 6))
    for i in range(unit_costs.shape[0]):
        plt.plot(range(1, unit_costs.shape[1] + 1), unit_costs[i, :], marker='o', label=f'情景 {i + 1}')
    plt.title('不同情景下各方案总成本价对比')
    plt.xlabel('方案编号')
    plt.ylabel('总成本价')
    plt.legend()
    plt.grid(False)  # 删除网格线
    plt.show()
    # 图 2：每种情景下的单位最优总成本（每个柱状图使用不同的颜色）
    # plt.figure(figsize=(8, 6))
    # colors = ['blue', 'green', 'red', 'purple', 'orange', 'brown']  # 颜色列表
    # plt.bar(range(1, len(optimal_costs) + 1), optimal_costs, color=colors[:len(optimal_costs)])  # 为每个柱状图指定不同的颜色
    # plt.title('每种情景下的最优总成本')
    # plt.xlabel('情景编号')
    # plt.ylabel('最优总成本')
    # plt.show()
# 主函数：计算成本并绘图
unit_costs = calculate_unit_costs()  # 计算所有情景下的方案成本
optimal_costs = calculate_optimal_costs(unit_costs)  # 计算最优总成本
# 打印每个情景下的详细成本数据
print_cost_data(unit_costs)
# 运行 Q-learning 和 动态规划
q_learning_train()  # 训练 Q-learning
policy, V = value_iteration()  # 动态规划
# 输出 Q 值表
print("训练后的 Q 表：")
print(Q)
plot_cost_comparison(unit_costs, optimal_costs)  # 绘制两种图表
```

3.问题三

3-1：求解动态规划下的最优值

```
#动态规划
import itertools
import numpy as np
import pandas as pd
# 定义生产成本计算函数
```

```
def production_cost_stage(s, a_i, N, a11_18, a21_28, a31_38, detect_parts, detect_halfproducts, detect_final):
    # 将 detect_parts 等列表转换为 NumPy 数组，确保可以进行逐元素操作
    detect_parts = np.array(detect_parts)
    detect_halfproducts = np.array(detect_halfproducts)
    detect_final = np.array(detect_final)
    # 零件 1 至 8 的采购和检测成本
    part_quality_rates = np.zeros(8)
    part_purchase_costs = np.zeros(8)
    part_testing_costs = np.zeros(8)
    for i in range(8):
        part_quality_rates[i] = 1 - a11_18[i] * (1 - detect_parts[i])
        part_quantity = N / part_quality_rates[i]
        part_purchase_costs[i] = part_quantity * a21_28[i]
        part_testing_costs[i] = detect_parts[i] * N * a31_38[i]
    # 当前阶段成本：零件采购+检测
    current_stage_cost = np.sum(part_purchase_costs) + np.sum(part_testing_costs)
    # 如果决策变量是拆解或其他，可以加入拆解成本计算
    if a_i == 1:  # 假设 1 为拆解
        disassembly_cost = np.sum(part_purchase_costs) * 0.1  # 假设拆解成本是采购成本的 10%
        current_stage_cost += disassembly_cost
    return current_stage_cost
# 定义生产成本计算函数
def production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, detect_parts,detect_halfproducts, detect_final, disassemble):
    # 将输入的列表和布尔型变量转换为 NumPy 数组以支持逐元素运算
    detect_parts = np.array(detect_parts)
    detect_halfproducts = np.array(detect_halfproducts)
    detect_final = detect_final[0]  # 解包 detect_final 元组，使其成为单个布尔值
    disassemble = disassemble[0]      # 解包 disassemble 元组，使其成为单个布尔值
    # 零件 1 至 8 的采购和检测
    part_quality_rates = np.zeros(8)
    part_purchase_costs = np.zeros(8)
    part_testing_costs = np.zeros(8)
    # 计算每个零件的采购和检测成本
    for i in range(8):
        part_quality_rates[i] = 1 - a11_18[i] * (1 - detect_parts[i])  # 零件合格率
        part_quantity = N / part_quality_rates[i]  # 需要采购的零件数量
        part_purchase_costs[i] = part_quantity * a21_28[i]  # 零件采购成本
        part_testing_costs[i] = detect_parts[i] * N * a31_38[i]  # 零件检测成本
    # 半成品 1、2、3 的合格率
    quality_halfproduct1 = np.prod(part_quality_rates[:3])  # 半成品 1 由零件 1、2、3 组成
```

```
    quality_halfproduct2 = np.prod(part_quality_rates[3:6])  # 半成品 2 由零件 4、5、6 组成
    quality_halfproduct3 = np.prod(part_quality_rates[6:8])  # 半成品 3 由零件 7、8 组成
    # 半成品检测成本
    halfproduct_testing_costs = np.sum(detect_halfproducts * N * np.array(a41_43))
    # 成品的合格率 (成品由三个半成品装配而成)
    product_quality_rate = quality_halfproduct1 * quality_halfproduct2 * quality_halfproduct3 * (1 - a19
* (1 - detect_final))
    # 半成品装配成本
    halfproduct_assembly_cost = N * c6
    # 成品装配成本
    assembly_cost = N * c4
    # 成品检测成本
    final_testing_cost = detect_final * N * a39
    # 不合格品的调换成本
    replacement_cost = (1 - detect_final) * N * (1 - product_quality_rate) * c2
    # 不合格成品的拆解成本和回收收益
    scrap_cost = float(disassemble) * N * (1 - product_quality_rate) * c3
    scrap_revenue = float(disassemble) * N * (1 - product_quality_rate) * 0.6 * c1
    # 总成本计算
    total_cost = np.sum(part_purchase_costs) + np.sum(part_testing_costs) + \
                 halfproduct_assembly_cost + halfproduct_testing_costs + \
                 assembly_cost + final_testing_cost + replacement_cost + \
                 scrap_cost - scrap_revenue
    return total_cost
# 动态规划递推公式
def dp_optimization(s, N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6,
detect_parts,detect_halfproducts, detect_final, disassemble):
    # s 是当前工序的状态
    if s == "final":
        # 最终工序状态，返回成品成本
        return production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6,
detect_parts, detect_halfproducts, detect_final, disassemble)
    # 计算当前工序的所有可能决策的最小值
    min_cost = float('inf')
    for a_i in [0, 1]:  # 决策变量是否检测/拆解
        current_cost = production_cost_stage(s, a_i, N, a11_18, a21_28, a31_38, detect_parts,
detect_halfproducts, detect_final)
        next_stage_cost = dp_optimization("final", N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1,
c2, c3, c4, c6, detect_parts, detect_halfproducts, detect_final, disassemble)
        total_cost = current_cost + next_stage_cost
        min_cost = min(min_cost, total_cost)
    return min_cost
```

```
# 计算每个组合的最优值
def optimize_dynamic_programming(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, combination):
    detect_parts, detect_halfproducts, detect_final, disassemble = combination
    # 初始状态设置（例如从第一个阶段开始）
    initial_state = "start"
    # 使用动态规划计算最优成本
    min_cost = dp_optimization(initial_state, N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, detect_parts, detect_halfproducts, detect_final, disassemble)
    return min_cost
# 生成所有组合
def generate_combinations():
    detect_parts_combinations = list(itertools.product([0, 1], repeat=8))  # 零配件 1-8 是否检测的组合
    detect_halfproducts_combinations = list(itertools.product([0, 1], repeat=3))  # 半成品 1-3 是否检测的组合
    detect_final_combinations = list(itertools.product([0, 1], repeat=1))  # 成品是否检测的组合
    disassemble_combinations = list(itertools.product([0, 1], repeat=1))  # 不合格成品是否拆解的组合
all_combinations=list(itertools.product(detect_parts_combinations,detect_halfproducts_combinations,detect_final_combinations, disassemble_combinations))
    return all_combinations
# 保存结果到 Excel
def save_results_to_excel(results, filename):
    df = pd.DataFrame(results, columns=["Detect_Parts", "Detect_HalfProducts", "Detect_Final", "Disassemble", "Min_Cost"])
    df.to_excel(filename, index=False)
# 参数设置
N = 1000  # 成品数量
a11_18 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # 零件 1 至 8 的次品率
a19 = 0.1  # 成品次品率
a21_28 = [2, 8, 12, 2, 8, 12, 8, 12]  # 零件 1 至 8 的购买单价
a31_38 = [1, 1, 2, 1, 1, 2, 1, 2]  # 零件 1 至 8 的检测成本
a39 = 6  # 成品的检测成本
a41_43 = [4, 4, 4]  # 半成品的检测成本
c1 = 200  # 成品市场售价
c2 = 40  # 不合格产品调换损失
c3 = 10  # 成品拆解费用
c4 = 8  # 成品装配成本
c6 = 8  # 半成品装配成本
# 生成所有组合
combinations = generate_combinations()
```

```
# 计算每个组合的最优成本
results = []
for idx, combination in enumerate(combinations):
    min_cost = optimize_dynamic_programming(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, combination)
    results.append([combination[0],combination[1], combination[2], combination[3], min_cost])
# 保存结果到 Excel
save_results_to_excel(results, r"C:\Users\86191\Desktop\问题三\result-DP.xlsx")


-----------------------------------------------------------------------------------------------------------------------
3-2：求解强化学习下的最优解
import numpy as np
import pandas as pd
import itertools
import random
# 定义生产成本计算函数
def production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, detect_parts,detect_halfproducts, detect_final, disassemble):
    # 将输入的布尔型变量转换为数值
    detect_parts = np.array(detect_parts)
    detect_halfproducts = np.array(detect_halfproducts)
    detect_final = detect_final[0]  # 解包 detect_final 元组，使其成为单个布尔值
    disassemble = disassemble[0]    # 解包 disassemble 元组，使其成为单个布尔值
    # 零件 1 至 8 的采购和检测
    part_quality_rates = np.zeros(8)
    part_purchase_costs = np.zeros(8)
    part_testing_costs = np.zeros(8)
    # 计算每个零件的采购和检测成本
    for i in range(8):
        part_quality_rates[i] = 1 - a11_18[i] * (1 - detect_parts[i])  # 零件合格率
        part_quantity = N / part_quality_rates[i]  # 需要采购的零件数量
        part_purchase_costs[i] = part_quantity * a21_28[i]  # 零件采购成本
        part_testing_costs[i] = detect_parts[i] * N * a31_38[i]  # 零件检测成本
    # 半成品 1、2、3 的合格率
    quality_halfproduct1 = np.prod(part_quality_rates[:3])  # 半成品 1 由零件 1、2、3 组成
    quality_halfproduct2 = np.prod(part_quality_rates[3:6])  # 半成品 2 由零件 4、5、6 组成
    quality_halfproduct3 = np.prod(part_quality_rates[6:8])  # 半成品 3 由零件 7、8 组成
    # 半成品检测成本
    halfproduct_testing_costs = np.sum(detect_halfproducts * N * np.array(a41_43))
    # 成品的合格率 (成品由三个半成品装配而成)
    product_quality_rate = quality_halfproduct1 * quality_halfproduct2 * quality_halfproduct3 * (1 - a19 * (1 - detect_final))
```

```
    # 半成品装配成本
    halfproduct_assembly_cost = N * c6
    # 成品装配成本
    assembly_cost = N * c4
    # 成品检测成本
    final_testing_cost = detect_final * N * a39
    # 不合格品的调换成本
    replacement_cost = (1 - detect_final) * N * (1 - product_quality_rate) * c2
    # 不合格成品的拆解成本和回收收益
    scrap_cost = float(disassemble) * N * (1 - product_quality_rate) * c3
    scrap_revenue = float(disassemble) * N * (1 - product_quality_rate) * 0.6 * c1
    # 总成本计算
    total_cost = np.sum(part_purchase_costs) + np.sum(part_testing_costs) + \
                 halfproduct_assembly_cost + halfproduct_testing_costs + \
                 assembly_cost + final_testing_cost + replacement_cost + \
                 scrap_cost - scrap_revenue
    return total_cost
# Q-learning 的设置
class QLearning:
    def __init__(self, state_space_size, action_space_size, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.q_table = np.zeros((state_space_size, action_space_size))  # 初始化 Q 表
        self.alpha = alpha  # 学习率
        self.gamma = gamma  # 折扣因子
        self.epsilon = epsilon  # 探索率
    # 选择动作：ε-greedy 策略
    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice([0, 1])  # 随机选择动作 (是否检测/拆解)
        else:
            return np.argmax(self.q_table[state])  # 选择最大 Q 值对应的动作
    # Q 表更新
    def update_q_table(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state])  # 选择下一个状态的最佳动作
        td_target = reward + self.gamma * self.q_table[next_state, best_next_action]
        td_error = td_target - self.q_table[state, action]
        self.q_table[state, action] += self.alpha * td_error  # 更新 Q 值
# 用 Q-learning 来优化生产成本
def optimize_q_learning(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, combination,
episodes=1000):
    detect_parts, detect_halfproducts, detect_final, disassemble = combination
    # 定义状态和动作空间
```

```
    state_space_size = 2  # 代表是否进行检测等决策的状态
    action_space_size = 2  # 动作：检测(1)或不检测(0)
    # 初始化 Q-learning
    ql = QLearning(state_space_size, action_space_size)
    # 学习过程
    for episode in range(episodes):
        state = 0  # 假设初始状态
        done = False
        while not done:
            # 选择动作
            action = ql.choose_action(state)
            # 根据动作计算奖励（负的成本作为奖励）
            reward = -production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4,
c6,detect_parts, detect_halfproducts, detect_final, disassemble)
            # 转移到下一个状态
            next_state = 1 if action == 1 else 0  # 假设简单的状态转移
            # 更新 Q 表
            ql.update_q_table(state, action, reward, next_state)
            state = next_state
            if state == 1:  # 终止条件，可以根据具体逻辑定义
                done = True
    # 在学习完成后，返回 Q 表中最大的 Q 值作为最优成本
    best_action = np.argmax(ql.q_table[0])  # 从初始状态选择最佳动作
    best_cost = -ql.q_table[0, best_action]  # 返回负的 Q 值即成本
    return best_cost
# 生成所有组合
def generate_combinations():
    detect_parts_combinations = list(itertools.product([0, 1], repeat=8))  # 零配件 1-8 是否检测的组
合
    detect_halfproducts_combinations = list(itertools.product([0, 1], repeat=3))  # 半成品 1-3 是否检
测的组合
    detect_final_combinations = list(itertools.product([0, 1], repeat=1))  # 成品是否检测的组合
    disassemble_combinations = list(itertools.product([0, 1], repeat=1))  # 不合格成品是否拆解的组
合
all_combinations=list(itertools.product(detect_parts_combinations,detect_halfproducts_combinations,de
tect_final_combinations, disassemble_combinations))
    return all_combinations
# 保存结果到 Excel
def save_results_to_excel(results, filename):
    df = pd.DataFrame(results, columns=["Detect_Parts", "Detect_HalfProducts", "Detect_Final",
"Disassemble", "Min_Cost"])
    df.to_excel(filename, index=False)
```

```
# 参数设置
N = 1000  # 成品数量
a11_18 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # 零件 1 至 8 的次品率
a19 = 0.1  # 成品次品率
a21_28 = [2, 8, 12, 2, 8, 12, 8, 12]  # 零件 1 至 8 的购买单价
a31_38 = [1, 1, 2, 1, 1, 2, 1, 2]  # 零件 1 至 8 的检测成本
a39 = 6  # 成品的检测成本
a41_43 = [4, 4, 4]  # 半成品的检测成本
c1 = 200  # 成品市场售价
c2 = 40  # 不合格产品调换损失
c3 = 10  # 成品拆解费用
c4 = 8  # 成品装配成本
c6 = 8  # 半成品装配成本
# 生成所有组合
combinations = generate_combinations()
# 计算每个组合的最优成本
results = []
for idx, combination in enumerate(combinations):
    min_cost = optimize_q_learning(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6,
combination)
    results.append([combination[0],combination[1], combination[2], combination[3], min_cost])
# 保存结果到 Excel
save_results_to_excel(results, r"C:\Users\86191\Desktop\问题三\result-QL.xlsx")
```

3-3：求解贪心算法下的最优解

```
import random
import numpy as np
import itertools
import pandas as pd
# 定义生产成本计算函数
def production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6,
detect_parts,detect_halfproducts, detect_final, disassemble):
    # 将输入的列表和布尔型变量转换为 NumPy 数组以支持逐元素运算
    detect_parts = np.array(detect_parts)
    detect_halfproducts = np.array(detect_halfproducts)
    detect_final = detect_final[0]  # 解包 detect_final 元组，使其成为单个布尔值
    disassemble = disassemble[0]     # 解包 disassemble 元组，使其成为单个布尔值
    # 零件 1 至 8 的采购和检测
    part_quality_rates = np.zeros(8)
    part_purchase_costs = np.zeros(8)
    part_testing_costs = np.zeros(8)
    # 计算每个零件的采购和检测成本
```

```
    for i in range(8):
        part_quality_rates[i] = 1 - a11_18[i] * (1 - detect_parts[i])  # 零件合格率
        part_quantity = N / part_quality_rates[i]  # 需要采购的零件数量
        part_purchase_costs[i] = part_quantity * a21_28[i]  # 零件采购成本
        part_testing_costs[i] = detect_parts[i] * N * a31_38[i]  # 零件检测成本
    # 半成品 1、2、3 的合格率
    quality_halfproduct1 = np.prod(part_quality_rates[:3])  # 半成品 1 由零件 1、2、3 组成
    quality_halfproduct2 = np.prod(part_quality_rates[3:6])  # 半成品 2 由零件 4、5、6 组成
    quality_halfproduct3 = np.prod(part_quality_rates[6:8])  # 半成品 3 由零件 7、8 组成
    # 半成品检测成本
    halfproduct_testing_costs = np.sum(detect_halfproducts * N * np.array(a41_43))
    # 成品的合格率 (成品由三个半成品装配而成)
    product_quality_rate = quality_halfproduct1 * quality_halfproduct2 * quality_halfproduct3 * (1 - a19
* (1 - detect_final))
    # 半成品装配成本
    halfproduct_assembly_cost = N * c6
    # 成品装配成本
    assembly_cost = N * c4
    # 成品检测成本
    final_testing_cost = detect_final * N * a39
    # 不合格品的调换成本
    replacement_cost = (1 - detect_final) * N * (1 - product_quality_rate) * c2
    # 不合格成品的拆解成本和回收收益
    scrap_cost = float(disassemble) * N * (1 - product_quality_rate) * c3
    scrap_revenue = float(disassemble) * N * (1 - product_quality_rate) * 0.6 * c1
    # 总成本计算
    total_cost = np.sum(part_purchase_costs) + np.sum(part_testing_costs) + \
                 halfproduct_assembly_cost + halfproduct_testing_costs + \
                 assembly_cost + final_testing_cost + replacement_cost + \
                 scrap_cost - scrap_revenue
    return total_cost
# 贪婪算法进行优化
def greedy_optimization(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6, initial_solution,
max_iters=100):
    # 初始化组合和其成本
    current_solution = initial_solution
    current_cost = production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4,
c6,current_solution[0], current_solution[1], current_solution[2], current_solution[3])
    for _ in range(max_iters):
        found_better = False
        # 遍历所有可能的邻居
        for neighbor in generate_neighbors(current_solution):
```

```
                neighbor_cost = production_cost_3(N, a11_18, a19, a21_28, a31_38, a41_43, a39, c1, c2, c3, c4, c6,neighbor[0], neighbor[1], neighbor[2], neighbor[3])
                # 如果找到更优的邻居，则更新当前解
                if neighbor_cost < current_cost:
                    current_solution = neighbor
                    current_cost = neighbor_cost
                    found_better = True
                    break  # 一旦找到更优解，立即退出内层循环，进行下一轮搜索
            if not found_better:
                break  # 如果没有找到更优的解，则停止搜索
    return current_solution, current_cost
# 生成所有可能的组合
def generate_combinations():
    detect_parts_combinations = list(itertools.product([0, 1], repeat=8))  # 零配件 1-8 是否检测的组合
    detect_halfproducts_combinations = list(itertools.product([0, 1], repeat=3))  # 半成品 1-3 是否检测的组合
    detect_final_combinations = list(itertools.product([0, 1], repeat=1))  # 成品是否检测的组合
    disassemble_combinations = list(itertools.product([0, 1], repeat=1))  # 不合格成品是否拆解的组合

    all_combinations=list(itertools.product(detect_parts_combinations, detect_halfproducts_combinations,detect_final_combinations, disassemble_combinations))
    return all_combinations
# 生成某个组合的邻居
def generate_neighbors(solution):
    detect_parts, detect_halfproducts, detect_final, disassemble = solution
    neighbors = []
    # 翻转每个零件的检测状态
    for i in range(len(detect_parts)):
        neighbor_parts = list(detect_parts)
        neighbor_parts[i] = 1 - neighbor_parts[i]  # 翻转检测状态
        neighbors.append((neighbor_parts, detect_halfproducts, detect_final, disassemble))
    # 翻转每个半成品的检测状态
    for i in range(len(detect_halfproducts)):
        neighbor_halfproducts = list(detect_halfproducts)
        neighbor_halfproducts[i] = 1 - neighbor_halfproducts[i]
        neighbors.append((detect_parts, neighbor_halfproducts, detect_final, disassemble))
    # 翻转成品检测状态
    neighbor_final = list(detect_final)
    neighbor_final[0] = 1 - neighbor_final[0]
    neighbors.append((detect_parts, detect_halfproducts, neighbor_final, disassemble))
```

```
        # 翻转拆解状态
        neighbor_disassemble = list(disassemble)
        neighbor_disassemble[0] = 1 - neighbor_disassemble[0]
        neighbors.append((detect_parts, detect_halfproducts, detect_final, neighbor_disassemble))
        return neighbors
# 保存结果到 Excel
def save_results_to_excel(results, filename):
        df = pd.DataFrame(results, columns=["Detect_Parts", "Detect_HalfProducts", "Detect_Final",
"Disassemble", "Min_Cost"])
        df.to_excel(filename, index=False)
# 参数设置
N = 1000  # 成品数量
a11_18 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  # 零件 1 至 8 的次品率
a19 = 0.1  # 成品次品率
a21_28 = [2, 8, 12, 2, 8, 12, 8, 12]  # 零件 1 至 8 的购买单价
a31_38 = [1, 1, 2, 1, 1, 2, 1, 2]  # 零件 1 至 8 的检测成本
a39 = 6  # 成品的检测成本
a41_43 = [4, 4, 4]  # 半成品的检测成本
c1 = 200  # 成品市场售价
c2 = 40  # 不合格产品调换损失
c3 = 10  # 成品拆解费用
c4 = 8  # 成品装配成本
c6 = 8  # 半成品装配成本
# 生成所有组合
combinations = generate_combinations()
# 保存所有组合的最优解
results = []
# 运行贪婪优化算法
for combination in combinations:
        best_combination, min_cost = greedy_optimization(N, a11_18, a19, a21_28, a31_38, a41_43, a39,
c1, c2, c3, c4, c6, combination)
results.append([best_combination[0],best_combination[1],best_combination[2],best_combination[3],
min_cost])
# 保存结果到 Excel
save_results_to_excel(results, r"C:\Users\86191\Desktop\问题三\result-Greedy.xlsx")
```

4.问题四

4-1：估计问题二的各类次品率的值

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import beta
```

```
from matplotlib import rcParams
# 设置字体及显示风格
rcParams['font.sans-serif'] = ['SimHei']
rcParams['axes.unicode_minus'] = False
# 定义基于不完全 Beta 函数的置信区间计算方法
def beta_binomial_confidence_interval(k, n, confidence_level):
    """
    使用不完全 Beta 函数计算置信区间
    """
    alpha = 1  # 先验参数
    beta_prior = 1  # 先验参数
    lower_bound = beta.ppf((1 - confidence_level) / 2, k + alpha, n - k + beta_prior)
    upper_bound = beta.ppf(1 - (1 - confidence_level) / 2, k + alpha, n - k + beta_prior)
    return lower_bound, upper_bound
# 蒙特卡洛模拟方法
def monte_carlo_simulation(k, n, n_simulations):
    """
    通过蒙特卡洛模拟基于不完全 Beta 分布进行抽样，更新次品率
    """
    # 在不完全 Beta 分布上进行 n_simulations 次模拟
    samples = beta.rvs(k + 1, n - k + 1, size=n_simulations)
    return samples  # 返回模拟得到的次品率样本
# 情况设置，给定每个场景的次品率、购买单价、检测成本等参数
scenarios = {
    '情况 1': {'零配件 1': {'次品率': 0.1, '购买单价': 4, '检测成本': 2},
               '零配件 2': {'次品率': 0.1, '购买单价': 18, '检测成本': 3},
               '成品': {'次品率': 0.1, '装配成本': 6, '检测成本': 3}},
    '情况 2': {'零配件 1': {'次品率': 0.2, '购买单价': 4, '检测成本': 2},
               '零配件 2': {'次品率': 0.2, '购买单价': 18, '检测成本': 3},
               '成品': {'次品率': 0.2, '装配成本': 6, '检测成本': 3}},
    '情况 3': {'零配件 1': {'次品率': 0.1, '购买单价': 4, '检测成本': 2},
               '零配件 2': {'次品率': 0.1, '购买单价': 18, '检测成本': 3},
               '成品': {'次品率': 0.1, '装配成本': 6, '检测成本': 3}},
    '情况 4': {'零配件 1': {'次品率': 0.2, '购买单价': 4, '检测成本': 1},
               '零配件 2': {'次品率': 0.2, '购买单价': 18, '检测成本': 1},
               '成品': {'次品率': 0.2, '装配成本': 6, '检测成本': 2}},
    '情况 5': {'零配件 1': {'次品率': 0.1, '购买单价': 4, '检测成本': 8},
               '零配件 2': {'次品率': 0.2, '购买单价': 18, '检测成本': 1},
               '成品': {'次品率': 0.1, '装配成本': 6, '检测成本': 2}},
    '情况 6': {'零配件 1': {'次品率': 0.05, '购买单价': 4, '检测成本': 2},
               '零配件 2': {'次品率': 0.05, '购买单价': 18, '检测成本': 3},
               '成品': {'次品率': 0.05, '装配成本': 6, '检测成本': 3}}
```

```
}
# 抽样检测数据，所有零配件和成品的样本量 n = 3501
sampling_data = {
    '情况 1': {'零配件 1': {'k': 350, 'n': 3501}, '零配件 2': {'k': 350, 'n': 3501}, '成品': {'k': 350, 'n': 3501}},
    '情况 2': {'零配件 1': {'k': 700, 'n': 3501}, '零配件 2': {'k': 700, 'n': 3501}, '成品': {'k': 700, 'n': 3501}},
    '情况 3': {'零配件 1': {'k': 350, 'n': 3501}, '零配件 2': {'k': 350, 'n': 3501}, '成品': {'k': 350, 'n': 3501}},
    '情况 4': {'零配件 1': {'k': 700, 'n': 3501}, '零配件 2': {'k': 700, 'n': 3501}, '成品': {'k': 700, 'n': 3501}},
    '情况 5': {'零配件 1': {'k': 350, 'n': 3501}, '零配件 2': {'k': 700, 'n': 3501}, '成品': {'k': 350, 'n': 3501}},
    '情况 6': {'零配件 1': {'k': 175, 'n': 3501}, '零配件 2': {'k': 175, 'n': 3501}, '成品': {'k': 175, 'n': 3501}},
}
# 初始化更新结果的列表
updated_results = []
# 针对每个情况，进行抽样检测和蒙特卡洛模拟更新次品率
for scenario_name, scenario_data in scenarios.items():
    result = {'情况': scenario_name}
    for item_name, item_data in scenario_data.items():
        k = sampling_data[scenario_name][item_name]['k']
        n = sampling_data[scenario_name][item_name]['n']
        # 直接基于不完全 Beta 分布进行蒙特卡洛模拟，模拟样本量为问题一中给定的 n 值
        simulated_theta_samples = monte_carlo_simulation(k, n, n_simulations=n)
        # 计算模拟得到的均值作为更新后的次品率
        updated_theta = np.mean(simulated_theta_samples)
        # 保存更新后的次品率
        result[f'{item_name}更新后的次品率'] = updated_theta
    # 添加结果到列表
    updated_results.append(result)
# 将结果转为 DataFrame 并展示
df = pd.DataFrame(updated_results)
print(df)
# 保存为 Excel 文件
df.to_excel('问题 2_蒙特卡洛更新次品率.xlsx', index=False)
# 可视化结果
ax = df.plot(x='情况', kind='bar', stacked=False, figsize=(10, 6), color=['red', 'orange', 'yellow'])
# 修改横坐标为横向显示
plt.xticks(rotation=0)
plt.ylabel('更新次品率')
plt.show()
```

4-2：估计问题三的各类次品率的值

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from scipy.stats import beta
from matplotlib import rcParams
# 设置字体及显示风格
rcParams['font.sans-serif'] = ['SimHei']
rcParams['axes.unicode_minus'] = False
# 定义基于不完全 Beta 函数的抽样检验方法
def beta_binomial_confidence_interval(k, n, confidence_level):
    """
    使用不完全 Beta 函数计算置信区间
    """
    alpha = 1  # 先验参数
    beta_prior = 1  # 先验参数
    lower_bound = beta.ppf((1 - confidence_level) / 2, k + alpha, n - k + beta_prior)
    upper_bound = beta.ppf(1 - (1 - confidence_level) / 2, k + alpha, n - k + beta_prior)
    return lower_bound, upper_bound
# 蒙特卡洛模拟方法
def monte_carlo_simulation(k, n, n_simulations):
    """
    通过蒙特卡洛模拟基于不完全 Beta 分布进行抽样，更新次品率
    """
    # 在不完全 Beta 分布上进行 n_simulations 次模拟
    samples = beta.rvs(k + 1, n - k + 1, size=n_simulations)
    return samples  # 返回模拟得到的次品率样本
# 问题三中涉及的零配件、半成品和成品数据
components = {
    '零配件 1': {'次品率': 0.1, '购买单价': 2, '检测成本': 1},
    '零配件 2': {'次品率': 0.1, '购买单价': 8, '检测成本': 1},
    '零配件 3': {'次品率': 0.1, '购买单价': 12, '检测成本': 2},
    '零配件 4': {'次品率': 0.1, '购买单价': 2, '检测成本': 1},
    '零配件 5': {'次品率': 0.1, '购买单价': 8, '检测成本': 1},
    '零配件 6': {'次品率': 0.1, '购买单价': 12, '检测成本': 2},
    '零配件 7': {'次品率': 0.1, '购买单价': 8, '检测成本': 1},
    '零配件 8': {'次品率': 0.1, '购买单价': 12, '检测成本': 2},
}
semi_products = {
    '半成品 1': {'次品率': 0.1, '装配成本': 8, '检测成本': 4, '拆解费用': 6},
    '半成品 2': {'次品率': 0.1, '装配成本': 8, '检测成本': 4, '拆解费用': 6},
    '半成品 3': {'次品率': 0.1, '装配成本': 8, '检测成本': 4, '拆解费用': 6},
}
final_product = {
    '成品': {'次品率': 0.1, '装配成本': 8, '检测成本': 6, '拆解费用': 10, '售价': 200, '调换损失': 40}
}
```

```
# 抽样检测数据
sampling_data = {
    '零配件 1': {'k': 5, 'n': 50},
    '零配件 2': {'k': 10, 'n': 50},
    '零配件 3': {'k': 8, 'n': 50},
    '零配件 4': {'k': 3, 'n': 50},
    '零配件 5': {'k': 7, 'n': 50},
    '零配件 6': {'k': 2, 'n': 50},
    '零配件 7': {'k': 6, 'n': 50},
    '零配件 8': {'k': 4, 'n': 50},
    '半成品 1': {'k': 6, 'n': 50},
    '半成品 2': {'k': 9, 'n': 50},
    '半成品 3': {'k': 5, 'n': 50},
    '成品': {'k': 7, 'n': 50},
}
# 初始化更新结果的列表
updated_results = []
# 遍历零配件、半成品和成品，计算更新次品率
for item_name, data in {**components, **semi_products, '成品': final_product}.items():
    k = sampling_data[item_name]['k']
    n = sampling_data[item_name]['n']
    # 使用蒙特卡洛模拟基于不完全 Beta 分布进行次品率估计
    simulated_theta_samples = monte_carlo_simulation(k, n, n_simulations=n)
    # 计算模拟得到的均值作为更新后的次品率
    updated_theta = np.mean(simulated_theta_samples)
    # 保存更新结果
    updated_results.append({
        '项目': item_name,
        '更新次品率': updated_theta
    })
# 将结果转为 DataFrame 并展示
df = pd.DataFrame(updated_results)
print(df)
# 保存为 Excel 文件
df.to_excel('问题 3 零配件半成品成品更新次品率.xlsx', index=False)
# 可视化结果，移除“项目”标签并调整横坐标标签
ax = df.plot(x='项目', kind='bar', stacked=False, figsize=(10, 6), color='orange')
# 设置横坐标标签水平显示，并移除横坐标标题
plt.xticks(rotation=0)
plt.xlabel('')  # 移除横坐标的 "项目" 标签
plt.ylabel('更新次品率')
plt.show()
```