基于 SPRT、MDP 与贝叶斯理论的电子产品生产决策研究

摘要

电子产品生产，作为现代工业核心，已演变为涵盖供应链、质控与成本优化的复杂体系。精密电子器件的次品率管理成为成本控制关键，要求工厂以最低成本高效识别次品，降低损耗。制定基于次品率、市场售价及成本因素的科学生产检测策略，平衡市场需求与成本控制，是亟待深入研究的课题。

本文将从次品率的计算引入，从两个零配件合成一件成品出发，到多个零配件多道工序，逐步给出检测和拆解的决策策略，最后考虑到次品率的动态性，优化之前所给出的决策策略。

对于问题一，通过引入**序贯概率比检验（SPRT）**方法，本文构建了基于单侧比例检验的假设检验模型，以动态调整抽样量并减少检测成本。在两种不同信度要求下（95% 和 90%），分别计算了最低需要抽样次数和次品数拒收门槛，并通过模拟分析验证了模型的有效性。最后根据模拟结果给出了抽样检测方案。

对于问题二，通过构建**马尔科夫决策过程（MDP）**模型，我们详细定义了状态空间模型、动作空间模型、状态转移矩阵以及代价矩阵，以此来求解最优的检验和拆解策略。该模型综合考虑了各个生产阶段的次品率、成本和收益，最终给出了不同情况下的检验和拆解策略，并且给出了相应的期望收益。

对于问题三，针对问题二中 MDP 模型在处理大量零件和工序时的局限性，本文提出了一种创新的改进方法。我们在原有 MDP 模型的基础上，借鉴**模拟退火算法**的思想，简化了状态转移概率函数的构建过程。同时将多个状态性质相似的状态整合为一个状态组，以此来简化相应矩阵的计算，并根据给定的动作策略得到模拟状态转移矩阵和代价矩阵，得到了给定情形下的最优策略和期望收益。

对于问题四，在处理次品率非固定值的问题时，本文提出了一种基于**贝叶斯理论**的解决方案。首先，利用后验概率公式来估计次品率的先验分布；随后，基于这一先验分布，我们利用**蒙特卡洛模拟**的思想模拟了所有可能的生产策略，并通过比较这些策略的期望收益，从中筛选出最优策略。

最后，我们对模型进行了优缺点分析。

关键字：SPRT　MDP　贝叶斯理论　蒙特卡洛模拟

一、 问题重述

1.1 问题背景

电子产品生产，作为现代工业的重要组成部分，已经发展成为了一个涉及供应链管理、质量控制和成本优化的复杂系统。作为精密化的电子器件，零配件的次品率也逐渐纳入了工厂成本控制范围。工厂需要在在最少的成本内检测出次品，从而减少不必要的损失。在实际的生产过程中，如何根据零配件的次品率、成品的市场售价以及各种成本因素，制定出最为科学合理的生产和检测策略，以确保既能满足市场需求，又能控制生产成本，这无疑是一个值得深入探讨和研究的问题。

1.2 需要解决的问题

我们通过分析相关数据，运用数学思想，建立数学模型来研究下列问题：

问题 1. 对于有某个标称值的某批零件，确立抽样检测方案，在不同的置信度下，确定是否拒收或者接收该批零配件。

问题 2. 在已知零配件和成品的次品率、购买单价、检测成本、装配成本、市场售价、调换损失和拆解费用的情况下，做出在不同阶段是否检测，是否拆解的决策，从而使得成本最小化。

问题 3. 在更复杂的生产环境中（存在多道工序和多个零配件时），确定零配件、半成品、成品的检测策略和不合格产品的拆解策略。

问题 4. 我们将完善问题 2 和问题 3 给出的决策模型，在零配件、半成品和成品的次品率是基于抽样检测得到的情况下，即存在一定的检测误差时，重新确定问题 2 和问题 3 中的检测和拆解策略。

结合以上内容，我们将通过数学建模和数据分析，为工厂提供利润最高的生产策略。

二、 问题分析

2.1 问题一分析

在标称值 10% 的情况下，为了在一定的置信度内确定拒收或者接收方案，采用使用单侧比例检验来判定次品率是否超过标称值。采用序贯概率比检验（SPRT）方法来得到最少的抽样次数。通过计算样本似然比并与预设阈值进行比较，以决定是否接收或拒收零配件。

2.2 问题二分析

问题二的核心在于优化生产过程，降低成本，提高产品质量。我们需要考虑各个阶段的次品率、成本和收益，并制定合理的决策方案。通过建立状态空间模型、动作空间模型、状态转移矩阵和奖励函数，我们可以利用 MDP 马尔科夫决策模型，通过价值迭代法求解最优的检验和拆解策略。

2.3 问题三分析

对于问题三，由于零件数量和工序的增加，再直接使用 MDP 模型列状态转移概率函数变得过于复杂，因此我们在原来 MDP 模型的基础上进行改进，同时整合了具有相同性质的状态得到了数量较少的状态组，然后借鉴模拟退火算法的思想，根据给定的动作策略模拟状态转移矩阵和代价矩阵，从而求解。

2.4 问题四分析

问题四中，次品率不再是固定值，因此需要先通过后验概率公式得到次品率的先验分布，再根据这个先验分布去模拟所有可能出现的策略，在所有的策略中，再进行取优，即可得到最优策略。

三、 模型的假设

- **售出时不合格产品能完全被调换**：用户购买不合格产品之后能被发现，即企业售出不合格产品后产生调换损失的概率是 100%。
- **检测时厂家能完全检测出不合格产品**：厂家进行检测时，当存在次品，能 100% 被检出。
- **抽样检测和零配件的分布完全随机**：抽样检测采取完全随机的方式抽样，因此抽样得到的零配件是符合预期分布的，另外，零配件的分布也应该是符合其分布的规律的。
- **检测结果的长期有效性**：在产品生产的过程中，企业会严格监控零件的加工过程，因此每个产品或半成品的组成情况在任何时间都是明确可知的，又因为在组装前质量合格的零件因为拆解过程不会造成零件损害，因此在拆解后之前检验的合格零配件无需进行二次检验。
- **拆解后检测**：为简化模型和减少不合格零件带来的进一步的潜在成本，所有的半成品、成品拆解后会检测所有未检测零件。

四、 符号说明

符号	说明
n	最少抽样次数
c	次品数拒收门槛
α	第一类错误概率
β	第二类错误概率
L	似然比
s_n	零配件/成品的不同状态
$a_1 \sim a_4$	对于不同状态采取的不同动作
$P(a)$	状态转移矩阵
$R(s,a)$	状态 s 采取动作 a 之后的奖励函数
$V(s)$	状态 s 的价值函数
E_n	某一策略下的期望代价
θ	零配件的次品率

五、 模型的建立与求解

5.1 问题一：抽样检测方案设计

5.1.1 单侧比例检验

我们使用假设检验的方法来处理问题中的置信度的问题。[1] 在假设检验中，置信度是指是指在重复进行同样的实验或样本抽取时，能够包含真实参数值的置信区间的比例。

在假设检验中，**零假设**通常表示没有效应或者没有差异的状态，**备择假设**表示研究者试图证明的效应或差异。由于问题给出的两种情况都只给了一个置信值，我们采用假设检验中的单侧比例检验来建模。

其中，零假设和备择假设的具体值如下：

- **零假设** H_0：次品率 $p \le p_0$（例如 $p_0 = 10\%$）。
- **备择假设** H_1：次品率 $p > p_0$（例如 $p_1 = 15\%$）。

在假设检验中，第一类错误（拒绝合格品的概率）用显著性水平 α 表示，第二类错误（接收不合格品的概率）用 β 表示。

5.1.2 序贯概率比检验（SPRT）模型

为达到该目标，本文采用了序贯概率比检验（Sequential Probability Ratio Test, SPRT）的方法设计抽样检测方案[2]，并基于实际次品率进行模拟分析。

1.SPRT 模型介绍

序贯概率比检验（Sequential Probability Ratio Test, SPRT）是由沃德（A. Wald）在 1947 年提出的一种统计假设检验方法。与传统的固定样本量假设检验不同，SPRT 的优势在于其灵活性，允许在抽样过程中根据累积的样本信息动态地做出决策，这意味着在许多情况下，SPRT 能够以更少的样本量达到相同的统计决策效果，从而提高检测效率并降低成本。

在 SPRT 中，检验过程如下：

- 在每次抽样后，计算样本的似然比 L。
- 将似然比 L 与两个预先设定的阈值 A 和 B 进行比较：

$$\begin{cases} \text{如果} L > A, & \text{则拒绝零假设} H_0, \text{接受备择假设} H_1. \\ \text{如果} L < B, & \text{则接受零假设} H_0, \text{拒绝备择假设} H_1. \\ \text{如果} B \leq L \leq A, & \text{则继续抽样，直到达到某一决策阈值.} \end{cases}$$

SPRT 的优点在于能够动态调整样本量，通常在实际应用中可以显著减少样本量，从而降低检测成本和时间。

2.SPRT 的阈值计算

SPRT 的核心是通过不断抽样，计算样本的似然比，然后将其与预先设定的两个阈值 A 和 B 进行比较。阈值的计算公式为：

$$A = \frac{1-\beta}{\alpha}, \quad B = \frac{\beta}{1-\alpha} \tag{1}$$

其中，阈值 A 是拒绝零假设的决策阈值，而阈值 B 是接受零假设的决策阈值。这两个阈值决定了在 SPRT 过程中何时停止抽样并做出最终决策。

另外，α 和 β 分别表示：

α：第一类错误概率，即 $\alpha = 0.05$ 对应于 95% 的信度。

β：第二类错误概率，通常取值为 0.1 或 0.05。

根据不同的方案，分别计算两个方案的阈值：

- **方案一**（拒收方案）：显著性水平 $\alpha_1 = 0.05$，第二类错误概率 $\beta_1 = 0.05$。
- **方案二**（接收方案）：显著性水平 $\alpha_2 = 0.1$，第二类错误概率 $\beta_2 = 0.1$。

3. 似然比计算

在每次抽样时，计算当前的似然比 L，该比率反映了在备择假设和零假设下，观察到的样本数据的相对可能性：

$$L = \left(\frac{p_1}{p_0}\right)^X \times \left(\frac{1-p_1}{1-p_0}\right)^{n-X} \tag{2}$$

其中，X 表示在已抽取的 n 个样本中发现的次品数量，n 是到目前为止的抽样次数。似然比 L 的大小将决定是继续抽样还是停止并做出决策。

- X 为次品数量；
- n 为抽样次数。

当 $L > A$ 时，拒绝零假设 H_0，认为次品率超过标称值；当 $L < B$ 时，接受零假设 H_0，认为次品率未超过标称值。

5.1.3 模型求解与可视化

模拟抽样过程

通过编写 Python 代码，模拟多次抽样过程，并记录每次抽样的次数和次品数量。根据 SPRT 的规则，判断是否接受或拒绝该批零配件，从而得到每种抽样方案最少的抽样次数 n 以及次品数拒收门槛 c。

结果可视化

将每次抽样的接受/拒绝零配件的临界点（即最少的抽样次数）以频数为纵坐标，得到每次模拟的抽样次数 n 以及次品数拒收门槛 c 的分布情况直方图（100000 次抽样过程模拟）。

对于问题中的两种情况，定义“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”为**情况 1**，定义“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”为**情况 2**。针对这两种情况分别作出需要的最少的抽样次数和次品数拒收门槛与其频数（图例中简称为抽样数和次品数）的直方图并且采用拟合曲线的方式拟合分布，如下图所示。

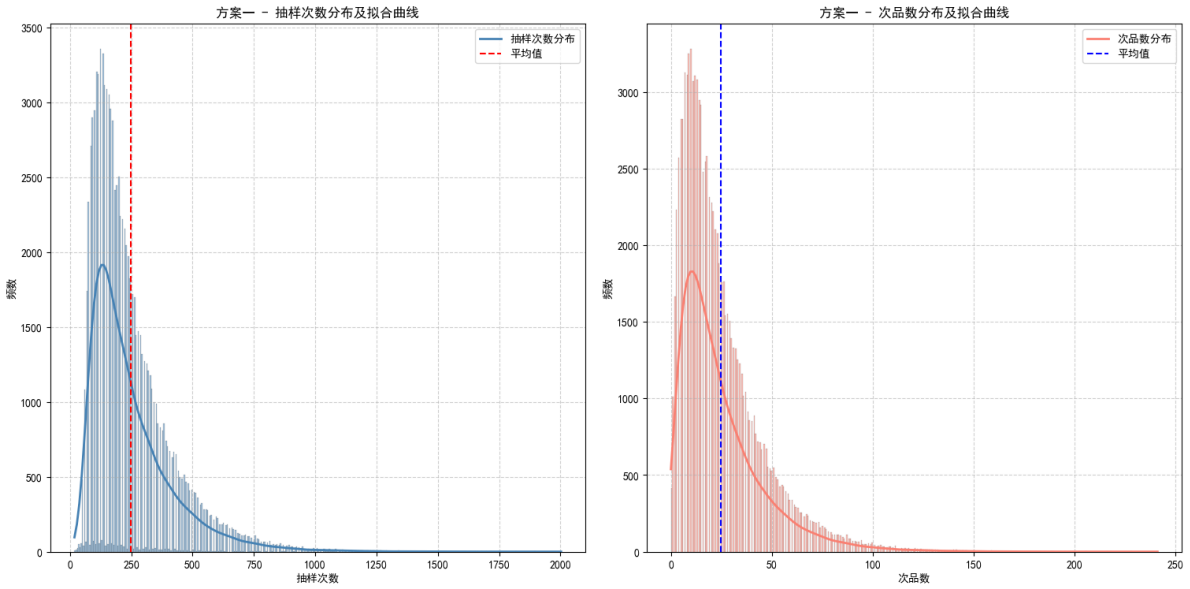

图 1　抽样数与次品数分布拟合曲线（方案一）

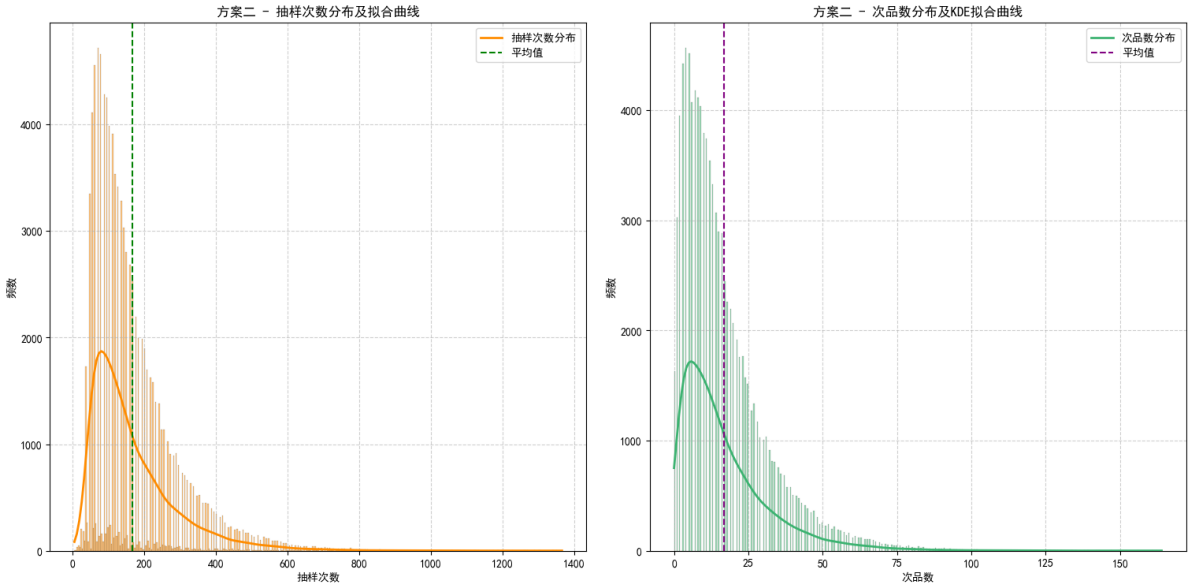

图 2　抽样数与次品数分布拟合曲线（方案二）

抽样数与次品数的关系

得到了抽样数的分布规律之后，为了更进一步地得到需要的最少的抽样次数与次品数拒收门槛之间的关系，使用三维散点图展示二者与频数之间的关系，并对频数取对数以更好地展示数据的分布。为了使得图形更加清晰，过滤掉频数小于等于 1 的数据点，仅保留频数较高的部分进行分析。得到的三维散点图如下图所示。

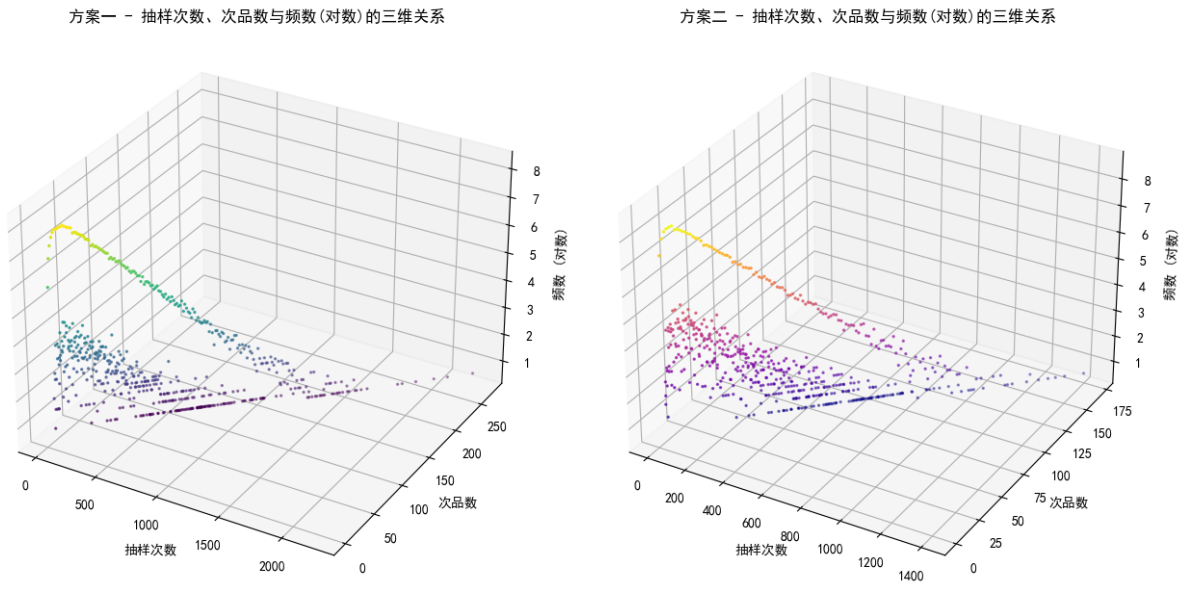

图 3　抽样数与次品数频数分布三维散点图（方案一和方案二）

5.1.4 求解结果与结论

分析上面的图可以发现，当样本分布随机且足够大时，需要的最少抽样次数频数呈现出类似于卡方分布的分布，这说明在大部分情况下，所需的抽样次数集中在一个较小的区间内，而极端情况下可能需要更多的抽样次数。

同时，由于卡方分布的特殊性，我们也可以选择各最低抽样次数的平均值的抽样次数作为题目的答案。这是因为平均出现的区域代表了一种平衡的选择，既避免了极端情况下的过多抽样，也避免了抽样次数过少而导致检验结果不稳定的情况；另外，平均值和频数最高的值接近且更多，平均值比平均最高值更多说明有更多更合理的情况被考虑，能在增加的更少的次数内包含更多的情况。此外，当多次足够多次数（100000 次）模拟时，平均值的数量趋于稳定。

两种情况下计算各最低抽样次数的平均值如下所示：

情况 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）：最低抽样次数平均值 248.04，取 **248 次**，对应的次品数拒收门槛 38.00，取 **38 件**，即：

最少随机抽样 248 次，当次品数大于 38 件时，则拒收该批零配件。

情况 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）：最低抽样次数平均值 167.96，取 **168 次**，对应的次品数拒收门槛 16.12，取 **16 件**，即：

最少随机抽样 168 次，当次品数小于等于 16 件时，则接受该批零配件。

5.2 问题二：零配件检测和成品拆解决策

5.2.1 马尔可夫决策过程（MDP）模型

问题二涉及了多个阶段的决策, 这些决策具有随机性和依赖性：每个零配件和成品的次品率决定了后续装配产品的质量，而企业的决策（如是否检测或拆解）会影响整体成本和收益。由于生产过程中每个阶段的决策都会影响未来的状态（如产品是否合格、是否需要拆解等），因此，我们采用具有” 马尔科夫性” 的 MDP 模型。

马尔科夫决策模型（Markov Decision Process, MDP）是一种用于数学化建模决策过程的框架，特别适用于在具有随机动态环境的系统中做出最优决策。在 MDP 中，系统状态的变化仅依赖于当前状态和所选择的动作，而与过去的状态和动作无关，这一特性称为马尔科夫性。这种特性使得 MDP 特别适用于解决具有不确定性和阶段性决策的复杂问题，尤其是在每个决策步骤都可能影响未来决策的情况下。此外，问题二的目标是通过优化检测和拆解决策，最大化收益并最小化成本损失，这符合 MDP 的目标函数（即最大化累积期望奖励）。[3]

MDP 通过定义状态空间、动作空间、状态转移概率、即时奖励函数以及一个（可能是折扣的）长期回报目标，为决策者提供了一个结构化的方式来平衡即时利益与未来潜在收益，从而找到最优策略。

在本题中，状态空间、动作空间、状态转移概率矩阵、奖励函数采取如下定义：

状态空间

状态空间 S 由若干个变量组成：

$$S = \{(s_0, s_1, s_2, s_3, ..., s_{18})\} \tag{3}$$

其中：

- D ($s_0 \sim s_4$) ：至少有一个零配件因检测不合格而丢弃
- O (s_5) ：初始状态，所有零配件未检测未组装
- P ($s_6 \sim s_9$) ：成品待检测
- T_1($s_{10} \sim s_{13}$)：成品检测不合格
- T_2($s_{14} \sim s_{17}$)：成品售卖后被退回
- S (s_{18}) ：成品售卖后未退回，即成功售卖

其中，有一个零配件因检测不合格而丢弃的情况下，剩余未丢弃的零配件可能是未检测，也可能是检测合格，共五个状态；考虑到零配件的检测情况，成品待检测、成品检测不合格、成品售卖后被退回的三种情况均由这四个状态构成：零配件均未检测，零配件全部检测合格，零配件（1 或 2）检测其一合格。

对 $s_0 \sim s_4$ 明确定义，用二元元组的两项表示零配件 1 和 2 的检测结果，则有：s_0 对应 (-1,-1)，s_1 对应 (-1,0)，s_2 对应 (0,-1)，s_3 对应 (-1,1)，s_4 对应 (1,-1)。

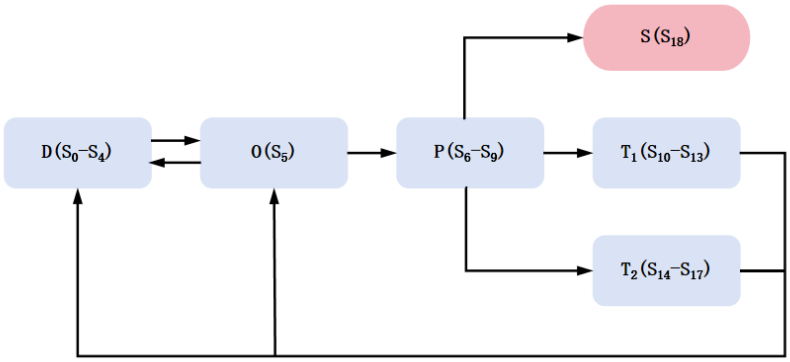

图 4　状态转移流程图

动作空间

动作空间 A 由四个二元变量组成：

$$A = \{(a_1, a_2, a_3, a_4) \mid a_1, a_2, a_3, a_4 \in \{0, 1\}\} \tag{4}$$

其中：

- a_1: 是否检测零配件 1（0：不检测，1：检测）
- a_2: 是否检测零配件 2（0：不检测，1：检测）
- a_3: 是否检测成品（0：不检测，1：检测）
- a_4: 对不合格成品的处理（0：丢弃，1：拆解）

状态依赖的动作空间

对于每个状态 $s_i \in \{S_5, S_6, \ldots, S_{14}\}$，其对应的动作集合 $A(s_i) \subseteq A$ 可以表示为：

$$A(S_5) = A(S_5) = \{a_1, a_2\}$$

$$A(S_6) = A(S_7) = A(S_8) = \{a_3\}$$

$$A(S_9) = A(S_{10}) = A(S_{11}) = A(S_{12}) = A(S_{13}) = A(S_{14}) = \{a_4\}$$

状态 S_5 到 S_{14} 的动作约束可以形式化为以下定义：

$$A(s_i) = \begin{cases} \{a_1, a_2\}, & \text{如果} s_i = S_5 \\ \{a_3\}, & \text{如果} s_i \in \{S_6, S_7, S_8\} \\ \{a_4\}, & \text{如果} s_i \in \{S_9, S_{10}, S_{11}, S_{12}, S_{13}, S_{14}\} \end{cases}$$

转移概率函数

转移概率函数 $P(s' \mid s, a)$ 表示从状态 s 采取动作 a 后转移到状态 s' 的概率。根据问题描述，我们可以定义：

$$P(s' \mid s, a) = \prod_{m,n,p} P(s'_m \mid s_n, a_p) \tag{5}$$

其中每个部分的概率由当前状态、次品率和动作来决定。在本文构建的模型中，状态主要有以下几种转移方式：

1. **缺失补偿**：当有零件因为不合格而丢弃时，对缺失的零件进行补充，从 $s_0 \sim s_4$ 转移到 s_5;
2. **零件检测（合格）并组装**：对零配件进行检测或者不检测，然后组装为成品,从 s_5 转移到 $s_6 \sim s_9$（成品检测）或 $s_{14} \sim s_{18}$（成品不检测，直接售卖）;
3. **零件检测（不合格）**：对不合格产品进行丢弃，从 s_5 转移到 $s_0 \sim s_4$;
4. **成品检测**：对成品进行检测，从 $s_6 \sim s_9$ 转移到 $s_{10} \sim s_{13}$（检测不合格），或者转移到 s_{18}(检测合格);
5. **不合格产品拆解**：对不合格产品进行拆解并检测，从 $s_{10} \sim s_{13}$ 或者 $s_{14} \sim s_{17}$ 转移到 $s_6 \sim s_9$ 或 $s_0 \sim s_4$;
6. **不合格产品丢弃**：对不合格产品直接丢弃，从 $s_{10} \sim s_{13}$ 或者 $s_{14} \sim s_{17}$ 转移到 s_5;

主要针对这几种转移方式构建状态转移矩阵，其他的转移方式转移函数取零。

由于列举全部的转移矩阵过于繁琐，这里以零件检测（不合格）的状态转移矩阵为例，其他转移矩阵的求解过程相似。定义零配件 1 的次品率为 α_1，零配件 2 的次品率为 α_2，是否检测的动作 a_1 和 a_2 之前已给出，对于某一零配件（以零配件 1 为例），定义其检测行为函数为 $\Phi_1(i)$，i 表示检测结果，-1 表示检测不合格，0 表示未检测，1 表示检测合格，则

$$\Phi_1(i) = \begin{cases} \alpha_1 a_1, & i = 1 \\ 1 - \alpha_1, & i = 0 \\ \alpha_1(1 - a_1), & i = -1 \end{cases} \tag{6}$$

$\Phi_2(i)$ 的定义同理，由之前 $s_0 \sim s_4$ 与两个零配件检测状态的二元元组的对应关系，可以得到 s_5 到 $s_0 \sim s_4$ 的状态转移可以定义为这样一个函数 $f(i,j)$：

$$f(i,j) = \Phi_1(i)\Phi_2(j) \tag{7}$$

由此，可以得到由 s_5 到 $s_0 \sim s_4$ 的状态转移矩阵。其他的状态转移函数同理。

状态转移概率矩阵

对于动作 $a \in A$，状态转移矩阵 $P(a)$ 定义为：

$$P(a) = \begin{pmatrix} P(s_1 \mid s_1, a) & P(s_2 \mid s_1, a) & P(s_3 \mid s_1, a) & P(s_4 \mid s_1, a) \\ P(s_1 \mid s_2, a) & P(s_2 \mid s_2, a) & P(s_3 \mid s_2, a) & P(s_4 \mid s_2, a) \\ P(s_1 \mid s_3, a) & P(s_2 \mid s_3, a) & P(s_3 \mid s_3, a) & P(s_4 \mid s_3, a) \\ P(s_1 \mid s_4, a) & P(s_2 \mid s_4, a) & P(s_3 \mid s_4, a) & P(s_4 \mid s_4, a) \end{pmatrix}$$

其中，$P(s_j \mid s_i, a)$ 表示从状态 s_i 采取动作 a 后转移到状态 s_j 的概率。由于只存在

部分状态到部分状态的转移，完整的状态转移矩阵可以表示为如下分块矩阵：

$$\begin{bmatrix} A_{lu}^{5\times10} & O^{5\times9} \\ A_{ud}^{1\times10} & O^{1\times9} \\ O^{4\times10} & C^{4\times9} \\ D^{8\times10} & O^{8\times9} \\ O^{1\times10} & O^{1\times9} \end{bmatrix}$$

具体含义如下所示，部分矩阵的推导过程省略，详见代码文件：

- A_{lu} 代表缺失补偿概率矩阵，用于描述次品零件组向未检测零件组的转移情况（在解决问题时，我们认为存在零件缺失时就需要重新购买）损坏，次品零件组转移的方向是固定的，即补充购买检测为次品的零件；
- A_{ud} 代表零件组装检测概率矩阵，代表未检测零件组向各种检测情况的转移概率分布，该矩阵取值受到零件检测策略的影响。
- C 代表成品检测和售卖的概率矩阵，代表未检测产品向成品检测后不合格及成品售卖后被退回（不合格）状态转移的概率，该矩阵取值受到产品检测策略的影响。
- D 代表不合格产品拆解的概率矩阵，代表拆解不合格产品向各种次品零件组和未检测零件组的转移概率，该矩阵取值受到拆解策略的影响。

奖励函数

奖励函数 $R(s,a)$ 表示在状态 s 采取动作 a 后获得的即时奖励，由以下几个部分组成：检测成本、装配成本、成品检测收益、成品不合格损失、不合格成品拆解收益：

$$\begin{aligned} R(s,a) = &-c_1a_1 - c_2a_2 - c_3a_3 - c_4I(s = S_5) \\ &+pI(s = S_{18}) - lI(s = S_{10} \cup S_{11} \cup S_{12} \cup S_{13}) \\ &-da_4I(s = S_{10} \cup S_{11} \cup S_{12} \cup S_{13} \cup S_{14} \cup S_{15} \cup S_{16}) \end{aligned} \tag{8}$$

其中：

- c_1, c_2, c_3: 分别为零配件 1、零配件 2 和成品的检测成本
- c_4: 装配成本
- p: 市场售价
- l: 调换损失
- d: 拆解费用
- $I(\cdot)$: 指示函数，当条件成立时为 1，否则为 0

5.2.2 模型求解与可视化

MDP 模型的目标是找到一个最优策略，使得在长期内的累计收益（或成本）最大化。为此，我们将使用**价值迭代**方法来求解最优策略。

价值迭代

价值迭代是一种动态规划的方法，主要用于求解 MDP 中的最优策略。它的基本思想是通过不断更新每个状态的“价值”，最终收敛到最优价值函数。在这之后，我们就可以根据最优价值函数推导出最优策略。

价值迭代通过以下递推关系来更新每个状态的价值：

$$V(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V(s') \right] \tag{9}$$

其中：

- $V(s)$ 是状态 s 的价值，表示在状态 s 采取最优策略时的期望收益。
- $R(s,a)$ 是在状态 s 采取动作 a 所获得的即时奖励。
- $P(s' \mid s,a)$ 是在状态 s 采取动作 a 后转移到状态 s' 的概率。
- γ 是折扣因子，表示未来收益的折现系数，通常 $0 \leq \gamma < 1$。
- A 是动作的集合，表示在状态 s 可以采取的所有可能动作。

价值迭代的求解步骤

我们首先为每个状态赋予一个初始价值，例如 $V_0(s) = 0$，即初始时所有状态的价值均为 0。折扣因子 γ 可以设定为一个接近 1 的值（例如 0.9），以确保未来的收益也得到适当考虑。

在每次迭代中，对于每个状态 s，我们需要计算所有可能动作 a 的价值，然后选择其中最大的值来更新该状态的价值。具体地，对于状态 s 和动作 a，其价值更新为：

$$V_{k+1}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V_k(s') \right] \tag{10}$$

计算过程中的每个部分解释如下：

- **即时奖励** $R(s,a)$：根据企业在当前状态下选择的动作 a 所带来的即时收益或成本。这个奖励函数已经在问题描述中给出，包含了检测成本、装配成本、市场售价、调换损失和拆解费用等。
- **转移概率** $P(s' \mid s,a)$：这是一个根据次品率和企业的决策（如检测与否）决定的概率。我们需要根据给定的次品率计算从一个状态 s 转移到另一个状态 s' 的概率。
- **折扣因子** γ：用于控制未来收益的重要性。

当价值函数 $V(s)$ 收敛后，我们可以根据每个状态 s 的最优价值选择最优动作 a。具体来说，对于每个状态 s，最优策略 $\pi^*(s)$ 是使得价值函数最大的动作：

$$\pi^*(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V(s') \right] \tag{11}$$

通过该策略提取步骤，我们可以得到在每个状态下最优的检测和拆解释策。如下图5所示，为价值迭代过程中某一过程的状态转移矩阵和价值矩阵。

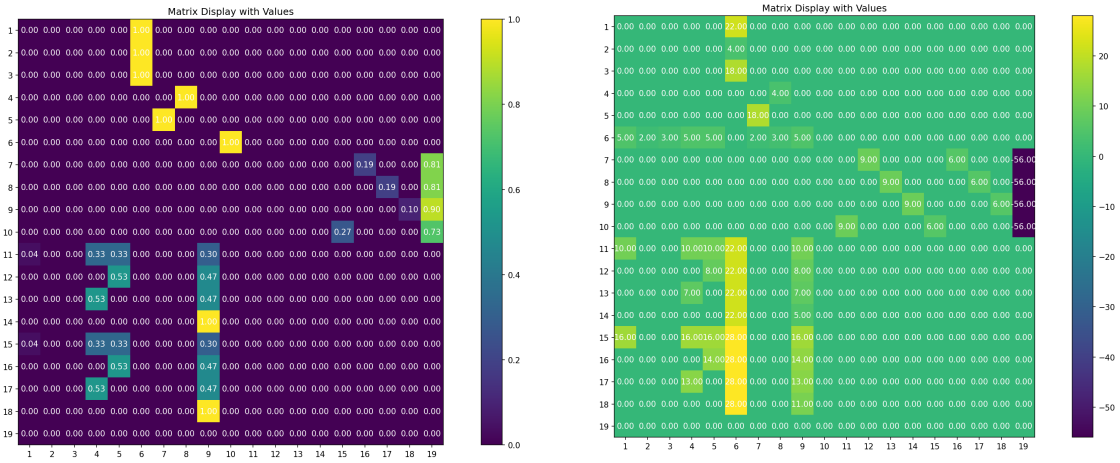

图 5　状态转移矩阵和价值矩阵举例

5.2.3 求解结果及其分析

通过求解上述模型，我们可以得到针对不同状态的最优决策。这些决策将指导企业在生产过程中的各个阶段如何进行检测、装配和处理不合格品，以最大化长期收益。

对于题中的每种情况，我们需要代入相应的参数值，运行值迭代算法，然后分析得到的最优策略，从而给出具体的决策方案。检测策略结果如表1所示。

表 1　题中不同情况下的检测策略

情况	零件一检测	零件二检测	是否丢弃	成品检测	期望利润
1	是	否	否	是	15.87
2	是	是	否	是	13.42
3	是	否	否	是	15.86
4	是	是	否	是	14.38
5	否	是	否	是	15.08
6	否	否	是	是	16.95

5.3 问题三：多零件多工序下的检测和拆解决策

5.3.1 问题的分析与简化

对于 m 道工序和 n 个零配件的生产决策，本质上是问题二的扩展与组合。但是在实际解决问题时，随着零件数量和工序的增加，状态空间的大小急剧膨胀。这就导致了像第二问一样写出状态转移矩阵和代价矩阵成为了一件及其困难的事情。因此，我们有必要对状态空间进行重新定义和压缩。

这里，我们在问题二所给出的状态转移矩阵的基础上，，在两道工序的基础上，重新审视每一个部件的状态，并详细给出其基本含义：

状态空间

状态空间 S 由若干个变量 s' 组成：

$$s' = \{(c_i, h_j, p)\} \tag{12}$$

其中：

- c_i: 表示零件 i 对应的状态，分别为：未检测，检测合格，检测不合格（需要购买补充），c_i 表示对应 i 个零件的状态集合
- h_j: 表示半成品 j 对应的状态，分别为：未制造，未检测，检测合格，检测不合格（需要购买补充），h_j 表示对应 i 个零件的状态集合
- p: 表示产品对应的状态，分别为：未制造，未检测，检测合格（成功售卖），检测不合格（丢弃，缺失），售卖退回，c_i 表示对应 i 个零件的状态集合

状态组

为了有效地压缩状态空间并减少计算复杂度，可以将具有相似性质的状态组合到一个状态组中。根据问题描述，我们可以将状态空间被划分为 8 个状态组。每个状态组可以通过对零配件、半成品和成品的状态进行约束条件来描述。

1. D：至少有一个零配件因检测不合格而丢弃
    - 准入状态：O，HT
    - 限制条件：至少有一个零件检测不合格。
2. O：初始状态
    - 准入状态：D，HT
    - 限制条件：存在零件待检测。
3. HP：半成品待检测
    - 准入状态：O
    - 限制条件：存在半成品待检测，且零件没有检测不合格的。
4. HT：半成品检测不合格

- 准入状态：HP，T1，T2
- 限制条件：存在某个半成品检测不合格，且零件没有检测不合格的。

5. P：成品待检测
   - 准入状态：HP
   - 限制条件：成品待检测，且没有半成品和零件检测不合格。
6. T1：成品检测不合格
   - 准入状态：P
   - 限制条件：成品检测不合格，没有半成品和零件检测不合格。
7. T2：成品售卖后被退回
   - 准入状态：P
   - 限制条件：成品售卖后被退回。
8. F：成品售卖后未退回（成功售卖）
   - 准入状态：P
   - 限制条件：成品售卖后未退回

状态之间的转换如下图所示：

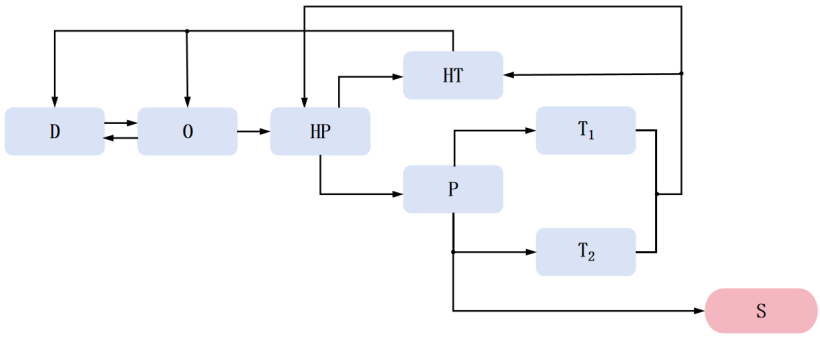

图 6　状态转移流程图

5.3.2 模型求解过程分析

针对上一节中提到的 8 种压缩状态，直接计算其相应的状态转移矩阵也比较复杂，但是在动作策略给定的情况下，通过多次模拟生产过程，得到模拟状态转移矩阵和模拟代价矩阵的复杂度显著下降。

显然，最快经过三次状态转移，我们可以结束状态转移过程，当状态转移次数超过 3 次时，我们能获得的最大收益会随转移次数的增加而减少。针对这种情况，我们可以借鉴**模拟退火算法**的思路，通过随机扰动进行求解[4]。

我们的具体策略如下：

1. **初始化策略**

   - **均匀初始化**：在算法初始阶段，我们假设每个操作的选择概率是均匀的。对于每个操作，选择该操作的初始概率设置为 0.5。初始策略可以表示为一个动作向量 $\mathbf{a} = (a_1, a_2, \ldots, a_l)$，其中 $a_i \in [0, 1]$ 表示第 i 个操作的选择概率。

2. **模拟生产过程**

   - **多次模拟**：我们进行多次模拟生产过程。在每隔 n_1 轮模拟时，记录当前得到的状态转移矩阵 **P** 和代价矩阵 **C**。通过状态转移矩阵 **P**，我们可以计算从初始状态出发，经过 k 次转移到达终态 F 的概率 P_F。结合代价矩阵 **C**，计算当前策略下的期望代价 E_0：

$$E_0 = \sum_{k=0}^{3} \sum_{s \in S} P(s \to F \mid k) \cdot C(s)$$

   其中 $P(s \to F \mid k)$ 表示经过 k 次转移从状态 s 到达终态 F 的概率，而 $C(s)$ 表示状态 s 的代价。我们限制状态转移次数不超过 3 次，因为更多的状态转移不会显著增加收益。

3. **扰动策略（模拟退火过程）**

   - **随机扰动**：在每轮迭代中，我们对动作向量 **a** 进行小幅度随机扰动，记为 $\mathbf{a}' = \mathbf{a} + \Delta\mathbf{a}$，其中 $\Delta\mathbf{a}$ 是一个在小范围内的随机扰动。扰动后的向量 $\mathbf{a}'$ 的每个元素依然满足 $a_i' \in [0, 1]$。
   - **退火规则**：在扰动后，我们计算新的期望代价 E_1，并根据模拟退火的接受准则来决定是否接受当前的扰动：
     - 若 $E_1 - E_0 > 0$（即代价增加），则以概率 $acc_1 = e^{-\frac{E_1 - E_0}{T}}$ 拒绝该扰动（其中 T 是当前的" 温度"），否则接受该扰动并减小扰动幅度。
     - 若 $E_1 - E_0 \leq 0$（即代价减少），则以概率 $acc_2 = 1$ 接受该扰动，并更新动作向量 **a**。
   - **温度衰减**：随着迭代次数的增加，温度 T 逐渐降低。通常，温度的衰减规则是：

$$T_{new} = \alpha \cdot T_{old}$$

   其中 α 是衰减系数，通常取 $0.8 \leq \alpha \leq 0.99$。

4. **策略收敛**

   - **停止条件**：当动作向量 **a** 收敛或达到最大迭代次数时，算法停止。此时动作向量 **a** 可以用于生成最终的决策方案。具体地，若 $a_i > 0.5$，则选择对应的操作；若 $a_i \leq 0.5$，则不选择该操作。

5.3.3 求解结果及其分析

对上述过程对原有的 MDP 进行改进，进行二次建模，编写代码求解，得到结果如下，此时对应的收益期望为 41.65:

表 2　多零件多工序下的检测策略

检测项目	是否检测	检测/拆解项目	是否检测/拆解
零件 1 检测	是	零件 2 检测	是
零件 3 检测	是	零件 4 检测	是
零件 5 检测	是	零件 6 检测	是
零件 7 检测	是	零件 8 检测	是
半成品 1 检测	否	半成品 1 拆解	是
半成品 2 检测	否	半成品 2 拆解	是
半成品 3 检测	否	半成品 3 拆解	是
成品检测	是	成品拆解	否

5.4 问题四：存在检测误差时的决策模型优化

5.4.1 MDP 模型的优化

后验概率

由于次品率是通过抽样检测方法得到的，我们需要使用贝叶斯公式来估算次品率的分布，并在此基础上计算对应的解题策略。

假设我们对零配件进行了抽样检测，得到了一些样本数据。我们可以使用这些数据来估算零配件的次品率。设零配件的次品率为 θ，抽样检测结果为 D，则根据贝叶斯公式[5]，我们有:

$$P(\theta) = \frac{P(\theta \mid D)P(D)}{P(D \mid \theta)} \tag{13}$$

其中:

- $P(\theta \mid D)$ 是后验概率，即在观察到数据 D 后，零配件次品率为 θ 的概率，即在问题二和问题三中观测到的次品率。
- $P(D \mid \theta)$ 是似然函数，即在零配件次品率为 θ 的条件下，观察到数据 D 的概率。
- $P(\theta)$ 是先验概率，即在观察到数据 D 之前，零配件次品率为 θ 的概率。

- $P(D)$ 是证据因子，是数据 D 的总概率，可以通过积分或求和得到。

假设次品率 θ 服从一个先验分布，比如 Beta 分布，然后根据抽样结果 $P(\theta \mid D)$ 更新这个分布。

更新后的 MDP 模型

在得到次品率的后验分布后，我们需要更新 MDP 模型中的转移概率函数和奖励函数。由于次品率的不确定性，我们的模型将是一个随机 MDP（SMDP）。

状态空间、动作空间和奖励函数保持不变。

转移概率函数需要根据后验分布来更新。由于状态转移概率函数的部分取值与次品率有关，需要根据次品率的后验概率分布来更新状态转移函数，例如，对于零件检测（不合格）的状态转移，我们需要计算在给定的次品率分布下，从状态 s_5 转移到 $s_0 \sim s_4$ 的概率。同时，状态转移概率矩阵的取值也将根据后验分布进行相应的调整。

5.4.2 模型的求解

蒙特卡洛模拟

由于模型现在是随机的，我们可以使用蒙特卡洛模拟来近似最优策略。

蒙特卡洛模拟（Monte Carlo Simulation）是一种基于随机抽样的数值方法，用于解决那些无法通过解析方法直接求解的问题[6]。它通过模拟随机过程来近似计算复杂系统的概率分布和期望值。蒙特卡洛模拟的核心思想是：如果一个系统或过程可以被描述为随机过程，那么可以通过对随机变量的多次抽样来估计系统的各种统计量，如均值、方差等。在本题中，由于样品率的分布是随机的，我们采用蒙特卡洛模拟的相关思想，随机抽取次品率进行求解。

主要步骤如下：

1. 从后验分布中抽取次品率的样本;
2. 对于每个样本，使用价值迭代或策略迭代来求解 MDP;
3. 计算所有样本的平均策略和价值

5.4.3 求解结果及其分析

通过蒙特卡洛模拟，我们可以得到在不同次品率样本下的最优策略和对应的期望利润。然后，我们可以分析这些结果，给出一个综合的最优决策方案。

对于问题二，表3是基于贝叶斯公式和蒙特卡洛模拟的求解结果：

其中，与原策略吻合概率的计算方法是在不同次品率样本下的最优策略中，原来的策略所占的比例。每种情况下最优策略的期望分布如图7所示：

表 3　存在检验误差的单个成品检测策略

情况	零件一检测	零件二检测	是否丢弃	成品检测	与原策略吻合概率	期望利润	期望利润标准差
1	是	否	否	是	54.2%	15.87	0.17
2	是	是	否	是	99.4%	13.41	0.21
3	是	否	否	是	53.5%	15.89	0.17
4	是	是	否	是	100.0%	14.26	0.19
5	否	是	否	是	98.7%	15.07	0.18
6	否	否	是	是	99.96%	16.95	0.16

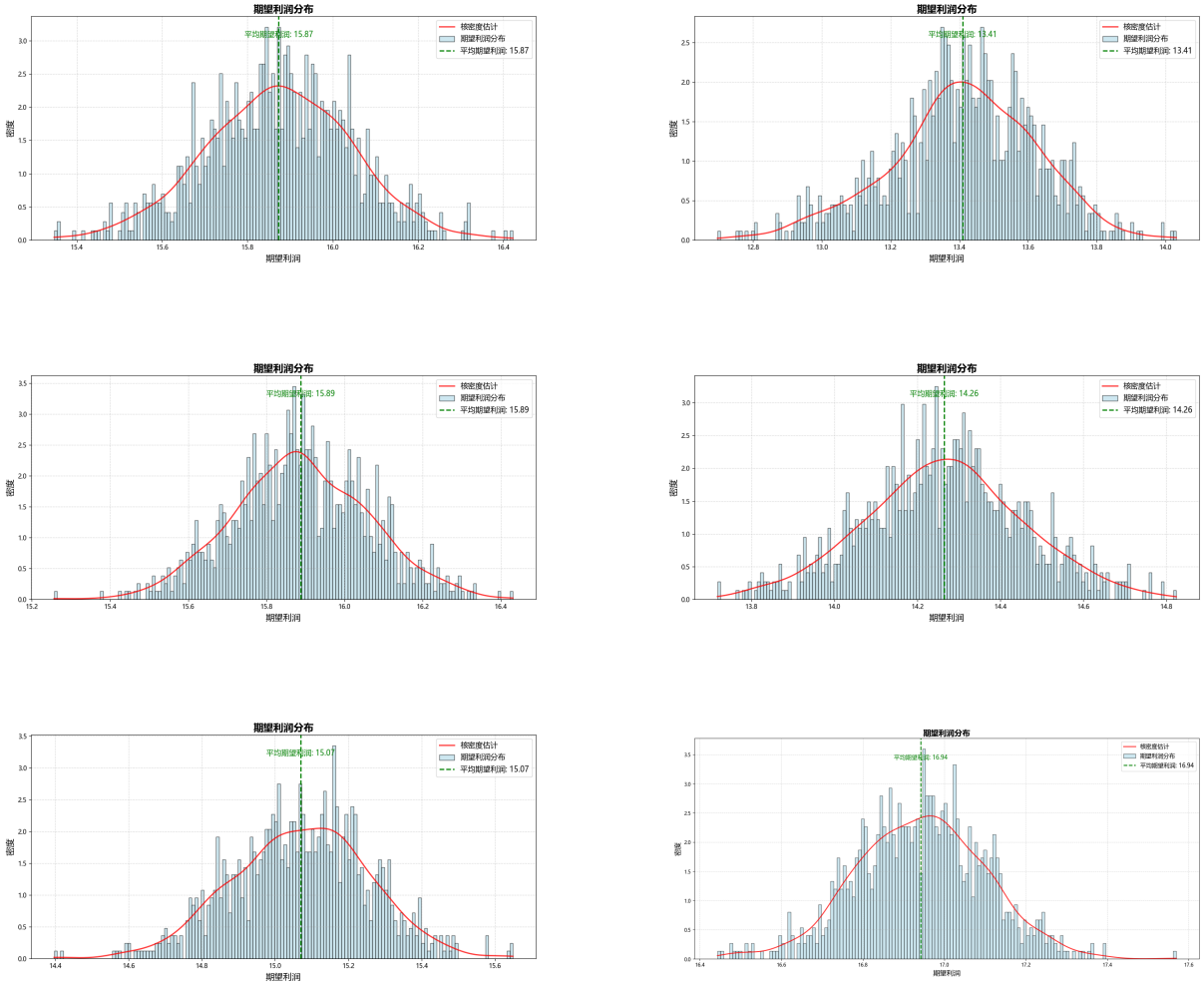

图 7　每种情况下的期望利润分布图

同样地，对于问题三，当次品率计算出的先验分布分布时，我们可以得到下图所示的期望利润分布图。

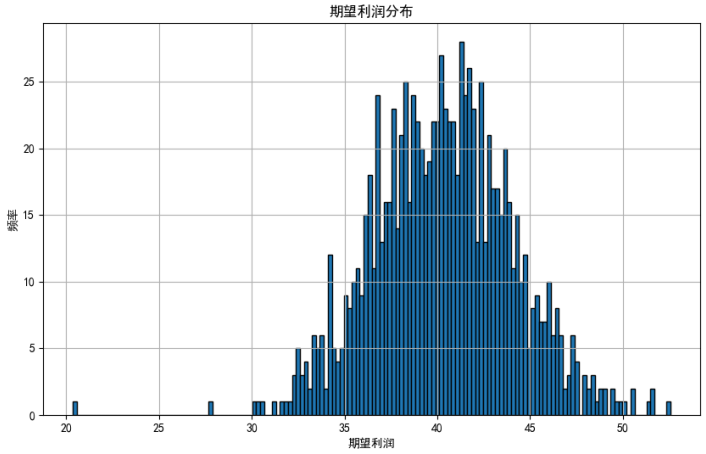

图 8　多零件多工序下的期望利润分布图

六、 模型的评价

6.1 模型的优点

1. 问题 1 的序贯概率比检验模型能够根据抽样结果动态调整抽样量，有效降低检测成本。通过模拟分析，验证了模型在不同信度下的有效性，并给出了具体的抽样检测方案。
2. 问题 2 和问题 3 的马尔可夫决策过程模型能够综合考虑生产过程中的多个阶段和影响因素，制定最优的检测和拆解决策，实现成本最小化和收益最大化。
3. 模型适用性广：该模型适用于单个零件、多个零件和多个工序的生产环境，具有较强的普适性。
4. 考虑检测误差：问题四中，模型考虑了检测误差对次品率的影响，并提出了基于后验概率分布的优化方案，提高了模型的鲁棒性。

6.2 模型的缺点

1. 随着零件数量和工序的增加，MDP 模型的状态空间和动作空间进一步膨胀，导致计算复杂度增加，原有的优化算法可能不适用；
2. 对供应链风险的考虑不足：模型主要关注生产环节的成本和收益，但并未考虑供应链风险的影响。在实际生产中，供应链风险会影响零件的供应情况和成本，进而影响生产决策。例如，如果供应链风险较高，即使成本较低，也可能选择增加检测以确保产品质量，以降低供应链风险。

参考文献

[1] 李奇明, 徐德义. 参数估计与假设检验: 原理、方法与误区 [J]. 大学教育,2018,(2):40-42.DOI:10.3969/j.issn.2095-3437.2018.02.013.

[2] 刘向宇. 序贯概率比检验的一个应用 [J]. 天津电大学报,2009,13,(2):56-58.DOI:10.3969/j.issn.1008-3006.2009.02.014.

[3] 陈键. 马尔科夫决策模型的大偏差及其相关问题 [D]. 清华大学,2021.10.27266/d.cnki.gqhau.2021.000106.

[4] 朱华君. 数学建模方法与实践. 高等教育出版社, 2015.

[5] 陈正方. 数学建模: 理论、方法与应用. 科学出版社, 2018.

[6] 朱陆陆. 蒙特卡洛方法及应用 [D]. 华中师范大学,2014.

附录 A　支撑材料目录

- **P1**
  - 1.1.py　　问题一模型的求解和柱状图的绘制
  - 1.2.py　　问题一模型的求解和三维散点图的绘制
- **P2**
  - State_transition_matrix.py　　问题二状态转移矩阵的生成
  - Cost_matrix.py　　问题二代价矩阵的生成
  - 2_main.py　　问题二模型的求解
- **P3**
  - 3_main.py　　问题三模型的求解
- **P4**
  - 4.1.py　　问题四单成品策略求解
  - 4.2.py　　问题四多零件多工序策略求解

附录 B　序贯概率比检验模型

1. 模型的求解和柱状图的绘制

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib
from scipy import stats

# 更改字体设置为 SimHei，支持中文显示
matplotlib.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
matplotlib.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

# 设置标称次品率和备择假设次品率
p0 = 0.1 # H0: 标称次品率
p1 = 0.15 # H1: 假设次品率较高

# 方案一：拒收方案
alpha1 = 0.05 # 第一类错误概率（拒绝合格品的概率）
beta1 = 0.05 # 第二类错误概率（接收不合格品的概率）

# 方案二：接收方案
alpha2 = 0.1 # 第一类错误概率（拒绝合格品的概率）
beta2 = 0.1 # 第二类错误概率（接收不合格品的概率）
```

```
# 计算SPRT的阈值（分别计算）
A1 = (1 - beta1) / alpha1 # 方案一的上限阈值
B1 = beta1 / (1 - alpha1) # 方案一的下限阈值

A2 = (1 - beta2) / alpha2 # 方案二的上限阈值
B2 = beta2 / (1 - alpha2) # 方案二的下限阈值

print(f"方案一 - 拒收：上限阈值 A1: {A1:.2f}, 下限阈值 B1: {B1:.2f}")
print(f"方案二 - 接收：上限阈值 A2: {A2:.2f}, 下限阈值 B2: {B2:.2f}")


# 模拟抽样过程
def sprt_simulation(A, B, p0, p1, actual_defect_rate):
    n_samples = 0 # 抽样次数
    X = 0 # 累积次品数

    while True:
        n_samples += 1
        is_defective = np.random.rand() < actual_defect_rate # 基于实际次品率p0模拟次品出现
        X += is_defective

        # 计算当前的似然比
        likelihood_ratio = (p1 / p0) ** X * ((1 - p1) / (1 - p0)) ** (n_samples - X)

        # 判断是否越过阈值
        if likelihood_ratio > A:
            decision = "拒绝 H0（次品率过高）"
            break
        elif likelihood_ratio < B:
            decision = "接受 H0（次品率正常）"
            break

    return n_samples, X


# 运行100000次模拟
n_simulations = 100000

# 方案一：拒收方案
samples_counts_1 = []
defects_counts_1 = []

for _ in range(n_simulations):
    n_samples, X = sprt_simulation(A1, B1, p0, p1, p0)
    samples_counts_1.append(n_samples)
    defects_counts_1.append(X)
```

```python
# 方案二：接收方案
samples_counts_2 = []
defects_counts_2 = []

for _ in range(n_simulations):
    n_samples, X = sprt_simulation(A2, B2, p0, p1, p0)
    samples_counts_2.append(n_samples)
    defects_counts_2.append(X)


# 计算平均值和频数最高的数
def calculate_statistics(data):
    mean_value = np.mean(data)
    mode_value = stats.mode(data, keepdims=False)[0] # 计算频数最高的数，并设置 keepdims=False
    return mean_value, mode_value


# 方案一统计结果
mean_samples_1, mode_samples_1 = calculate_statistics(samples_counts_1)
mean_defects_1, mode_defects_1 = calculate_statistics(defects_counts_1)

print(f"方案一 - 抽样次数的平均值：{mean_samples_1:.2f}，频数最高的数：{mode_samples_1}")
print(f"方案一 - 次品数的平均值：{mean_defects_1:.2f}，频数最高的数：{mode_defects_1}")

# 方案二统计结果
mean_samples_2, mode_samples_2 = calculate_statistics(samples_counts_2)
mean_defects_2, mode_defects_2 = calculate_statistics(defects_counts_2)

print(f"方案二 - 抽样次数的平均值：{mean_samples_2:.2f}，频数最高的数：{mode_samples_2}")
print(f"方案二 - 次品数的平均值：{mean_defects_2:.2f}，频数最高的数：{mode_defects_2}")


# 根据均值周围 ±1 的区间计算次品数的平均值
def calculate_defects_mean_within_range(samples_counts, defects_counts, mean_samples):
    lower_bound = mean_samples - 1
    upper_bound = mean_samples + 1

    # 筛选出在该区间内的次品数
    defects_in_range = [defects_counts[i] for i in range(len(samples_counts))
                        if lower_bound <= samples_counts[i] <= upper_bound]

    # 计算该区间内次品数的平均值
    mean_defects_in_range = np.mean(defects_in_range) if defects_in_range else float('nan')

    return mean_defects_in_range
```

```
# 计算方案一的次品数均值
mean_defects_in_range_1 = calculate_defects_mean_within_range(samples_counts_1,
    defects_counts_1, mean_samples_1)
print(f"方案一 - 抽样次数在均值±1区间内的次品数平均值: {mean_defects_in_range_1:.2f}")

# 计算方案二的次品数均值
mean_defects_in_range_2 = calculate_defects_mean_within_range(samples_counts_2,
    defects_counts_2, mean_samples_2)
print(f"方案二 - 抽样次数在均值±1区间内的次品数平均值: {mean_defects_in_range_2:.2f}")

# 增加 bins 的数量以缩小频数统计区间
bins_count = 400 # 增加 bins 的数量以获得更平滑的直方图

# 可视化方案一的抽样次数分布并使用KDE拟合曲线
plt.figure(figsize=(14, 8))

plt.subplot(1, 2, 1)
sns.histplot(samples_counts_1, bins=bins_count, kde=True, color='steelblue', stat='count',
    edgecolor='black',
           kde_kws={'bw_adjust': 1.5}, line_kws={'linewidth': 2}) # 调整 bw_adjust
               使KDE曲线更加平滑
plt.axvline(mean_samples_1, color='red', linestyle='dashed', linewidth=1.5)
plt.grid(True, linestyle='--', alpha=0.6)
plt.xlabel('抽样次数')
plt.ylabel('频数')
plt.title('方案一 - 抽样次数分布及拟合曲线')
plt.legend(['抽样次数分布', '平均值'])

plt.subplot(1, 2, 2)
sns.histplot(defects_counts_1, bins=bins_count, kde=True, color='salmon', stat='count',
    edgecolor='black',
           kde_kws={'bw_adjust': 1.5}, line_kws={'linewidth': 2}) # 调整 bw_adjust
               使KDE曲线更加平滑
plt.axvline(mean_defects_1, color='blue', linestyle='dashed', linewidth=1.5)
plt.grid(True, linestyle='--', alpha=0.6)
plt.xlabel('次品数')
plt.ylabel('频数')
plt.title('方案一 - 次品数分布及拟合曲线')
plt.legend(['次品数分布', '平均值'])

plt.tight_layout()
plt.show()

# 可视化方案二的抽样次数分布并使用KDE拟合曲线
plt.figure(figsize=(14, 8))

plt.subplot(1, 2, 1)
```

```
sns.histplot(samples_counts_2, bins=bins_count, kde=True, color='darkorange', stat='count',
    edgecolor='black',
           kde_kws={'bw_adjust': 1.5}, line_kws={'linewidth': 2}) # 调整 bw_adjust
               使KDE曲线更加平滑
plt.axvline(mean_samples_2, color='green', linestyle='dashed', linewidth=1.5)
plt.grid(True, linestyle='--', alpha=0.6)
plt.xlabel('抽样次数')
plt.ylabel('频数')
plt.title('方案二 - 抽样次数分布及拟合曲线')
plt.legend(['抽样次数分布', '平均值'])

plt.subplot(1, 2, 2)
sns.histplot(defects_counts_2, bins=bins_count, kde=True, color='mediumseagreen',
    stat='count', edgecolor='black',
           kde_kws={'bw_adjust': 1.5}, line_kws={'linewidth': 2}) # 调整 bw_adjust
               使KDE曲线更加平滑
plt.axvline(mean_defects_2, color='purple', linestyle='dashed', linewidth=1.5)
plt.grid(True, linestyle='--', alpha=0.6)
plt.xlabel('次品数')
plt.ylabel('频数')
plt.title('方案二 - 次品数分布及KDE拟合曲线')
plt.legend(['次品数分布', '平均值'])

plt.tight_layout()
plt.show()
```

2. 模型的求解和三维散点图的绘制

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # 导入3D绘图模块
import matplotlib

# 更改字体设置为 SimHei, 支持中文显示
matplotlib.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
matplotlib.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

# 设置标称次品率和备择假设次品率
p0 = 0.1 # H0: 标称次品率
p1 = 0.15 # H1: 假设次品率较高

# 方案一: 拒收方案
alpha1 = 0.05 # 第一类错误概率 (拒绝合格品的概率)
beta1 = 0.05 # 第二类错误概率 (接收不合格品的概率)

# 方案二: 接收方案
alpha2 = 0.1 # 第一类错误概率 (拒绝合格品的概率)
```

```
beta2 = 0.1  # 第二类错误概率（接收不合格品的概率）

# 计算SPRT的阈值（分别计算）
A1 = (1 - beta1) / alpha1 # 方案一的上限阈值
B1 = beta1 / (1 - alpha1) # 方案一的下限阈值

A2 = (1 - beta2) / alpha2 # 方案二的上限阈值
B2 = beta2 / (1 - alpha2) # 方案二的下限阈值

print(f"方案一 - 拒收：上限阈值 A1: {A1:.2f}, 下限阈值 B1: {B1:.2f}")
print(f"方案二 - 接收：上限阈值 A2: {A2:.2f}, 下限阈值 B2: {B2:.2f}")

# 模拟抽样过程
def sprt_simulation(A, B, p0, p1, actual_defect_rate):
    n_samples = 0 # 抽样次数
    X = 0 # 累积次品数

    while True:
        n_samples += 1
        is_defective = np.random.rand() < actual_defect_rate # 基于实际次品率p0模拟次品出现
        X += is_defective

        # 计算当前的似然比
        likelihood_ratio = (p1 / p0) ** X * ((1 - p1) / (1 - p0)) ** (n_samples - X)

        # 判断是否越过阈值
        if likelihood_ratio > A:
            decision = "拒绝 H0（次品率过高）"
            break
        elif likelihood_ratio < B:
            decision = "接受 H0（次品率正常）"
            break

    return n_samples, X

# 运行10000次模拟
n_simulations = 100000

# 方案一：拒收方案
samples_counts_1 = []
defects_counts_1 = []

for _ in range(n_simulations):
    n_samples, X = sprt_simulation(A1, B1, p0, p1, p0)
    samples_counts_1.append(n_samples)
    defects_counts_1.append(X)
```

```python
# 方案二：接收方案
samples_counts_2 = []
defects_counts_2 = []

for _ in range(n_simulations):
    n_samples, X = sprt_simulation(A2, B2, p0, p1, p0)
    samples_counts_2.append(n_samples)
    defects_counts_2.append(X)

# 生成频数（z轴）的数据
def calculate_frequency(samples_counts, defects_counts, bins):
    hist, xedges, yedges = np.histogram2d(samples_counts, defects_counts, bins=bins)
    x_pos, y_pos = np.meshgrid(xedges[:-1] + 0.5, yedges[:-1] + 0.5, indexing="ij")
    x_pos = x_pos.ravel()
    y_pos = y_pos.ravel()
    z_pos = hist.ravel()

    # 过滤频数小于1的点
    mask = z_pos >= 1
    x_pos = x_pos[mask]
    y_pos = y_pos[mask]
    z_pos = z_pos[mask]

    return x_pos, y_pos, z_pos

# 计算方案一的频数
bins = 1000 # 增加bins数量以缩小频数区间
x_pos_1, y_pos_1, z_pos_1 = calculate_frequency(samples_counts_1, defects_counts_1, bins)

# 计算方案二的频数
x_pos_2, y_pos_2, z_pos_2 = calculate_frequency(samples_counts_2, defects_counts_2, bins)

# 对z轴频数取log
z_pos_1_log = np.log1p(z_pos_1) # 取log1p避免对0取log
z_pos_2_log = np.log1p(z_pos_2)

# 绘制3D散点图

# 方案一：拒收方案的3D散点图
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x_pos_1, y_pos_1, z_pos_1_log, c=z_pos_1_log, cmap='viridis',
    marker='o',s=2)
ax.set_xlabel('抽样次数')
ax.set_ylabel('次品数')
ax.set_zlabel('频数（对数）')
ax.set_title('方案一 - 抽样次数、次品数与频数(对数)的三维关系')
```

```
fig.colorbar(scatter, ax=ax, shrink=0.5, aspect=5)
plt.show()

# 方案二：接收方案的3D散点图
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x_pos_2, y_pos_2, z_pos_2_log, c=z_pos_2_log, cmap='plasma',
    marker='o',s=2)
ax.set_xlabel('抽样次数')
ax.set_ylabel('次品数')
ax.set_zlabel('频数 (对数)')
ax.set_title('方案二 - 抽样次数、次品数与频数(对数)的三维关系')
fig.colorbar(scatter, ax=ax, shrink=0.5, aspect=5)
plt.show()
```

附录 C　单成品 MDP 模型

1. 状态转移矩阵的生成

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

def create_A_lu():
    A_lu = np.zeros((5, 10))
    A_lu[0:3, 5] = 1
    A_lu[3, 7] = 1
    A_lu[4, 6] = 1
    return A_lu

def create_A_ud(a1, a2, ap1, ap2):
    """
    创建A_ud矩阵，表示从原始零件组转移到各状态的概率。

    参数:
    a1 : 检测零件1的决策概率
    a2 : 检测零件2的决策概率
    ap1 : float, 零件1的次品率 (0 <= ap1 <= 1)
    ap2 : float, 零件2的次品率 (0 <= ap2 <= 1)

    返回:
    numpy.ndarray: 9x1的矩阵，表示9种状态的概率
    """
    A_ud = np.zeros((10, 1))

    # 状态 (-1,-1): 两个零件都检测且都不合格
    A_ud[0] = a1 * a2 * ap1 * ap2

    # 状态 (-1,0): 只检测零件1且不合格，不检测零件2
    A_ud[1] = a1 * (1 - a2) * ap1

    # 状态 (0,-1): 不检测零件1，只检测零件2且不合格
    A_ud[2] = (1 - a1) * a2 * ap2

    # 状态 (-1,1): 检测零件1不合格，检测零件2合格
    A_ud[3] = a1 * a2 * ap1 * (1 - ap2)

    # 状态 (1,-1): 检测零件1合格，检测零件2不合格
    A_ud[4] = a1 * a2 * (1 - ap1) * ap2

    #不能自迭代
    A_ud[5] = 0
```

```
    A_ud[6] = a1 * (1 - a2) * (1 - ap1)

    A_ud[7] = (1 - a1) * a2 * (1 - ap2)

    # 状态 (1,1): 两个零件都检测且都合格
    A_ud[8] = a1 * a2 * (1 - ap1) * (1 - ap2)

    # 状态 sp(0,0): 两个零件都不检测
    A_ud[9] = (1 - a1) * (1 - a2)

    return A_ud

def create_A_p():
    # 创建一个n维单位矩阵
    identity_matrix = np.eye(4)
    # 向上循环移位
    shifted_matrix = np.roll(identity_matrix, shift=-1, axis=0)
    return shifted_matrix

##########################################################
#C矩阵


def calculate_defect_probability(i, j, ap1, ap2, bp):
    """
    计算不同零件构成的生产出成品的次品概率
    i: 零件1的状态 (0: 不检测, 1: 检测且合格)
    j: 零件2的状态 (0: 不检测, 1: 检测且合格)
    ap1: 零件1的次品率
    ap2: 零件2的次品率
    bp: 零件均合格情况下的产品次品率
    """
    if i == 0 and j == 0:
        return 1 - (1 - ap1) * (1 - ap2) * (1 - bp)
    elif i == 1 and j == 0:
        return 1 - (1 - ap2) * (1 - bp)
    elif i == 0 and j == 1:
        return 1 - (1 - ap1) * (1 - bp)
    else: # i == 1 and j == 1
        return bp


def create_matrix_C(ap1, ap2, bp, b):
    """
    创建产品检测决策矩阵C, 并进行扩展操作
    ap1: 零件1的次品率
    ap2: 零件2的次品率
```

```
    bp: 零件均合格情况下的产品次品率

    b: 产品检测动作
    """

    probabilities = [
        calculate_defect_probability(0, 0, ap1, ap2, bp),
        calculate_defect_probability(1, 0, ap1, ap2, bp),
        calculate_defect_probability(0, 1, ap1, ap2, bp),
        calculate_defect_probability(1, 1, ap1, ap2, bp)
    ]

    C = np.diag(probabilities)

    # 1. 将C向右侧复制一遍，并乘以1-b，原来位置的C乘以b
    extended_C = np.hstack((b * C, (1-b) * C))

    # 2. 将probabilities转化为列向量pro，将1-pro拼接到上述矩阵右侧
    pro = np.array(probabilities).reshape(-1, 1)
    extended_C = np.hstack((extended_C, 1 - pro))

    # 3. 将这个新生成的4*9矩阵向上循环移位一位
    final_matrix = np.roll(extended_C, -1, axis=0)

    return final_matrix


##################################################
#寻错拆解矩阵

def create_matrix_D(ap1, ap2, bp ,ds):

    """
    计算不同零件构成的生产出成品的次品概率
    i: 零件1的状态 (0: 不检测, 1: 检测且合格)
    j: 零件2的状态 (0: 不检测, 1: 检测且合格)
    ap1: 零件1的次品率
    ap2: 零件2的次品率
    bp: 零件均合格情况下的产品次品率
    ds:丢弃动作,取1丢弃，取0不丢弃
    """

    # 创建分子矩阵和分母矩阵
    numerator = np.zeros((4, 4))
    denominator = np.zeros((4, 4))

    # 计算每种零件构成的产品次品概率
```

```
    p_00 = calculate_defect_probability(0, 0, ap1, ap2, bp)
    p_10 = calculate_defect_probability(1, 0, ap1, ap2, bp)
    p_01 = calculate_defect_probability(0, 1, ap1, ap2, bp)
    p_11 = calculate_defect_probability(1, 1, ap1, ap2, bp)

    # 填充分子矩阵
    # 行: (0,0), (1,0), (0,1), (1,1)
    # 列: (-1,-1), (-1,1), (1,-1), (1,1)

    # 对于 (0,0) 行
    numerator[0, 0] = ap1 * ap2
    numerator[0, 1] = ap1 * (1-ap2)
    numerator[0, 2] = (1-ap1) * ap2
    numerator[0, 3] = (1-ap1) * (1-ap2) * bp

    # 对于 (1,0) 行
    numerator[1, 0] = 0
    numerator[1, 1] = 0
    numerator[1, 2] = ap2
    numerator[1, 3] = (1-ap2) * bp

    # 对于 (0,1) 行
    numerator[2, 0] = 0
    numerator[2, 1] = ap1
    numerator[2, 2] = 0
    numerator[2, 3] = (1-ap1) * bp

    # 对于 (1,1) 行
    numerator[3, 0] = 0
    numerator[3, 1] = 0
    numerator[3, 2] = 0
    numerator[3, 3] = 1 * bp

    # 填充分母矩阵
    denominator[0, :] = p_00
    denominator[1, :] = p_10
    denominator[2, :] = p_01
    denominator[3, :] = p_11

    # 使用矩阵除法计算最终结果
    D = np.divide(numerator, denominator, where=denominator!=0)

    expanded_matrix = np.zeros((8, 10))

#    复制原矩阵到expanded_matrix的前四行和后四行
    expanded_matrix[:4, [0,3,4,8]] = D
    expanded_matrix[4:, [0,3,4,8]] = D
```

```
    expanded_matrix = expanded_matrix * (1 - ds)
    expanded_matrix[:, 5] = ds

    return expanded_matrix

##########################################################
##########矩阵拼接

def State_transition_martix(a1, a2, ap1, ap2, bp, b, ds):
    # 创建矩阵
    A_lu = create_A_lu()
    A_ud = create_A_ud(a1, a2, ap1, ap2)
    C = create_matrix_C(ap1, ap2, bp, b)
    D = create_matrix_D(ap1, ap2, bp, ds)

    # 创建所需的零矩阵
    zero_5x9 = np.zeros((5, 9))
    zero_1x9 = np.zeros((1, 9))
    zero_4x9 = np.zeros((4, 9))
    zero_8x9 = np.zeros((8, 9))
    zero_1x10 = np.zeros((1, 10))
    zero_4x10 = np.zeros((4, 10))

    # 左侧的矩阵拼接 (5x10, 1x10, 4x10, 8x10, 1x10)
    left_matrix = np.vstack((A_lu, A_ud.T, zero_4x10, D, zero_1x10))

    # 右侧的矩阵拼接 (5x9, 1x9, 4x9, 8x9, 1x9)
    right_matrix = np.vstack((zero_5x9, zero_1x9, C, zero_8x9, zero_1x9))

    # 拼接最终矩阵
    combined_matrix = np.hstack((left_matrix, right_matrix))

    return combined_matrix



#combined_matrix = create_combined_matrix(a1, a2, ap1, ap2, bp, b, ds)
#print(combined_matrix)



'''
# 使用示例
ap1 = 0.1 # 零件1的次品率
ap2 = 0.1 # 零件2的次品率
bp = 0.1 # 零件均合格情况下的产品次品率
ds = 0
```

```
D = create_matrix_D(ap1, ap2, bp,ds)
print(D)
'''


'''
# 测试函数
ap1, ap2, bp, b = 0.1, 0.1, 0.1, 1
result = create_matrix_C(ap1, ap2, bp, b)
print(result)
'''

'''
# 示例使用
ap1 = 0.1 # 零件1的次品率
ap2 = 0.15 # 零件2的次品率
bp = 0.05 # 零件均合格情况下的产品次品率

C = create_matrix_C(ap1, ap2, bp)
print("产品检测决策矩阵C:")
print(C)



# 使用示例
A_lu = create_A_lu()

a1, a2 = 1, 1
ap1, ap2 = 0.1, 0.2

A_ud = create_A_ud(a1, a2, ap1, ap2)

print(A_lu)
print("\nA_ud matrix:")
print(A_ud)
print(create_A_p())
'''
```

2. 代价矩阵的生成

```
import numpy as np
"""
   参数:
   c1 : A购买成本
   c2 : B购买成本
   b1 : A检测成本
```

```
    b2 : B检测成本

    bf : 产品检测成本

    cp:组装成本

"""

def create_CA_lu(c1,c2):
    CA_lu = np.zeros((5, 10))
    CA_lu[0, 5] = c1+c2
    CA_lu[1, 5] = c1
    CA_lu[2, 5] = c2

    CA_lu[3, 7] = c1
    CA_lu[4, 6] = c2

    return CA_lu

def create_CA_ud(b1,b2):
    b=b1+b2
    CA_ud = np.zeros((10, 1))

    # 状态 (-1,-1): 两个零件都检测且都不合格
    CA_ud[0] = b

    # 状态 (-1,0): 只检测零件1且不合格，不检测零件2
    CA_ud[1] = b1

    # 状态 (0,-1): 不检测零件1，只检测零件2且不合格
    CA_ud[2] = b2

    # 状态 (-1,1): 检测零件1不合格，检测零件2合格
    CA_ud[3] = b

    # 状态 (1,-1): 检测零件1合格，检测零件2不合格
    CA_ud[4] = b

    #不能自迭代
    CA_ud[5] = 0

    CA_ud[6] = b1

    CA_ud[7] = b2

    CA_ud[8] = b
```

```
    CA_ud[9] = 0

    return CA_ud

def create_CA_p(cp):
    # 创建一个n维单位矩阵
    identity_matrix = np.eye(4)
    # 向上循环移位
    shifted_matrix = np.roll(identity_matrix, shift=-1, axis=0)
    return shifted_matrix * cp

################################################

def creat_CC(cp, fc ,bf):
    """
    参数:

    bf : 产品检测成本
    cp:组装成本
    fc:产品售价收益(在代价矩阵中为负数)

    """
    CC = np.array([
        [0.0, cp+bf, 0.0, 0.0, 0.0, cp, 0.0, 0.0, fc],
        [0.0, 0.0, cp+bf, 0.0, 0.0, 0.0, cp, 0.0, fc],
        [0.0, 0.0, 0.0, cp+bf, 0.0, 0.0, 0.0, cp, fc],
        [cp+bf, 0.0, 0.0, 0.0, cp, 0.0, 0.0, 0.0, fc]
    ])
    return CC


def creat_CD(c1,c2,b1,b2,xl,cl):

    """
    参数:
    c1 : A购买成本
    c2 : B购买成本
    b1 : A检测成本
    b2 : B检测成本

    bf : 产品检测成本
    cp:组装成本
    fc:产品售价收益(在代价矩阵中为负数)

    xl: 调换损失
    cl: 拆解成本
```

```
    """
    a=c1+c2
    b = b1+b2
    CD = np.array([
    [cl+b, 0, 0, cl+b, cl+b, a, 0, 0, cl+b, 0],
    [0, 0, 0, 0, cl+b2, a, 0, 0, cl+b2, 0],
    [0, 0, 0, cl+b1, 0, a, 0, 0, cl+b1, 0],
    [0, 0, 0, 0, 0, a, 0, 0, cl, 0],

    [cl+b+xl, 0, 0, cl+b+xl, cl+b+xl, a+xl, 0, 0, cl+b+xl, 0],
    [0, 0, 0, 0, cl+b2+xl, a+xl, 0, 0, cl+b2+xl, 0],
    [0, 0, 0, cl+b1+xl, 0, a+xl, 0, 0, cl+b1+xl, 0],
    [0, 0, 0, 0, 0, a+xl, 0, 0, cl+xl, 0]
])
    return CD
# 使用示例


#####################################
####拼接
def Cost_matrix(c1, c2, b1, b2, bf, cp, fc, xl, cl):
    # 创建矩阵
    CA_lu = create_CA_lu(c1, c2) # 5x9
    CA_ud = create_CA_ud(b1, b2) # 9x1
    CC = creat_CC(cp, fc, bf) # 4x9
    CD = creat_CD(c1, c2, b1, b2, xl, cl) # 8x10

    # 创建所需的零矩阵
    zero_5x9 = np.zeros((5, 9))
    zero_1x9 = np.zeros((1, 9))
    zero_4x9 = np.zeros((4, 9))
    zero_8x9 = np.zeros((8, 9))
    zero_1x10 = np.zeros((1, 10))
    zero_4x10 = np.zeros((4, 10))

    # 左侧的矩阵拼接 (5x10, 1x10, 4x10, 8x10, 1x10)
    left_matrix = np.vstack((CA_lu, CA_ud.T, zero_4x10, CD, zero_1x10))

    # 右侧的矩阵拼接 (5x9, 1x9, 4x9, 8x9, 1x9)
    right_matrix = np.vstack((zero_5x9, zero_1x9, CC, zero_8x9, zero_1x9))

    # 拼接最终矩阵
    combined_matrix = np.hstack((left_matrix, right_matrix))

    return combined_matrix
```

```
a1 = 0          # 零件1的检测动作，1测，0不测
a2 = 0          # 零件2的检测动作
ds = 0           # 丢弃动作，1则丢弃
b = 0           # 产品检测动作

ap1 = 0.1        # 零件1的次品率
ap2 = 0.1        # 零件2的次品率
bp = 0.1        # 零件均合格情况下的产品次品率

c1 = 4          # A购买成本
c2 = 18          # B购买成本
b1 = 2           # A检测成本
b2 = 3           # B检测成本
bf = 3          # 产品检测成本
cp = 6          # 组装成本
fc = -56        # 产品售价收益
xl = 6          # 调换损失
cl = 5           # 拆解成本

print(Cost_matrix(c1, c2, b1, b2, bf, cp, fc, xl, cl))
```

3. 模型的求解

```
import numpy as np
import matplotlib.pyplot as plt
from State_transition_martix import State_transition_martix
from Cost_matrix import Cost_matrix

plt.rcParams['font.sans-serif'] = ['Microsoft YaHei']
plt.rcParams['axes.unicode_minus'] = False



def value_iteration(transition_matrix, reward_matrix, gamma=1, epsilon=1e-6):
    """
    使用价值迭代法计算最优状态值函数

    参数:
    transition_matrix: 状态转移矩阵
    reward_matrix:  奖励矩阵
    gamma:          折扣因子
    epsilon:        收敛阈值

    返回值:
    V:              状态值函数
    """
    num_states = transition_matrix.shape[0]
    V = np.zeros(num_states)
```

```
    iteration = 0
    while True:
        iteration += 1
        V_old = V.copy()

        for s in range(num_states):
            if s == 5:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                    range(4)]
                V[s] = max(Q)
            elif 6 <= s <= 9:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                    range(2)]
                V[s] = max(Q)
            elif 10 <= s <= 14:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                    range(2)]
                V[s] = max(Q)
            else:
                V[s] = np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V))

        if iteration % 10 == 0:
            print(f"迭代次数: {iteration}, 最大值函数变化: {np.max(np.abs(V - V_old))}")

        if np.max(np.abs(V - V_old)) < epsilon:
            break

    return V


def get_optimal_policy(V, transition_matrix, reward_matrix, gamma=0.99):
    """
    根据最优状态值函数计算最优策略

    参数:
    V:              状态值函数
    transition_matrix: 状态转移矩阵
    reward_matrix:  奖励矩阵
    gamma:          折扣因子

    返回值:
    optimal_policy: 最优策略
    """
    num_states = transition_matrix.shape[0]
    optimal_policy = np.zeros(num_states, dtype=int)
```

```
    for s in range(num_states):
        if s == 5:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(4)]
            optimal_policy[s] = np.argmax(Q)
        elif 6 <= s <= 9:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(2)]
            optimal_policy[s] = np.argmax(Q)
        elif 10 <= s <= 14:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(2)]
            optimal_policy[s] = np.argmax(Q)

    return optimal_policy


def simulate_profit(transition_matrix, reward_matrix, policy, initial_state=5,
    terminal_state=18,
                num_simulations=10000):
    """
    模拟策略的利润

    参数:
    transition_matrix: 状态转移矩阵
    reward_matrix:  奖励矩阵
    policy:         策略
    initial_state:  初始状态
    terminal_state: 终止状态
    num_simulations: 模拟次数

    返回值:
    profits:         利润列表
    """
    profits = []
    for _ in range(num_simulations):
        state = initial_state
        total_profit = 0
        while state != terminal_state:
            action = policy[state]
            next_state = np.random.choice(len(transition_matrix[state]),
                p=transition_matrix[state])
            profit = reward_matrix[state, next_state]
            total_profit += profit
            state = next_state
        profits.append(total_profit)
    return profits
```

```python
def main():
    ap1, ap2, bp = 0.05, 0.05, 0.05
    c1, c2, b1, b2, bf, cp, fc, xl, cl, y = 4, 18, 2, 3, 3, 6, -56, 10, 40, 3

    best_policy = None
    best_value = float('-inf')

    for a1 in [0, 1]:
        for a2 in [0, 1]:
            for ds in [0, 1]:
                for b in [0, 1]:
                    transition_matrix = State_transition_martix(a1, a2, ap1, ap2, bp, b, ds)
                    # 使用负的代价矩阵作为奖励矩阵
                    reward_matrix = -Cost_matrix(c1, c2, b1, b2, bf, cp, fc, xl, cl)

                    V = value_iteration(transition_matrix, reward_matrix)
                    policy = get_optimal_policy(V, transition_matrix, reward_matrix)

                    if V[5] > best_value:
                        best_value = V[5]
                        best_policy = (a1, a2, ds, b)
                        best_V = V
                        best_transition_matrix = transition_matrix
                        best_reward_matrix = reward_matrix

    print(f"最优策略: a1={best_policy[0]}, a2={best_policy[1]}, ds={best_policy[2]},
        b={best_policy[3]}")
    print(f"该策略下的期望利润: {(best_value / y):.2f}")

    optimal_policy = get_optimal_policy(best_V, best_transition_matrix, best_reward_matrix)
    profits = simulate_profit(best_transition_matrix, best_reward_matrix, optimal_policy)

    expected_profit = np.mean(profits)
    profit_std = np.std(profits)

    print(f"模拟结果 - 期望利润: {(expected_profit / y):.2f}")
    print(f"模拟结果 - 利润标准差: {(profit_std / y):.2f}")

    plt.figure(figsize=(12, 6))
    plt.hist(profits, bins=50, edgecolor='black')
    plt.title("利润分布")
    plt.xlabel("利润")
    plt.ylabel("频率")
    plt.axvline(expected_profit, color='r', linestyle='dashed', linewidth=2)
    plt.text(expected_profit, plt.ylim()[1], f'期望值: {(expected_profit / y):.2f}',
```

```
            horizontalalignment='center', verticalalignment='bottom')
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(12, 6))
    plt.bar(range(len(best_V)), best_V)
    plt.title("各状态的状态价值")
    plt.xlabel("状态")
    plt.ylabel("价值")
    plt.xticks(range(len(best_V)))

    for i, v in enumerate(best_V):
        plt.text(i, v, f'{v:.2f}', ha='center', va='bottom')

    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    main()
```

附录 D　多零件多工序的优化 MDP 模型

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict


class ProductionEnvironment:
    def __init__(self):
        # 零件状态：0-未检测，1-检测合格，2-检测不合格
        self.parts_state = [0] * 8

        # 半成品状态：0-未制造，1-未检测，2-检测合格，3-检测不合格
        self.half_products_state = [0] * 3

        # 产品状态：0-未制造，1-未检测，2-检测合格(成功售卖)，3-检测不合格(丢弃)，4-售卖退回
        self.product_state = 0

        # 成本和收益参数
        self.part_inspection_cost = [1] * 8
        self.half_product_inspection_cost = [4] * 3
        self.product_inspection_cost = 6
        self.assembly_cost = [8] * 3 # 三个半成品的组装成本
        self.disassembly_cost = [6] * 3 + [10] # 三个半成品和一个成品的拆解成本
        self.replacement_loss = 40
        self.market_price = 200

        # 缺陷率
        self.defect_rate = [0.1] * 8 + [0.1] * 3 + [0.1]

    def get_state(self):
        return self.parts_state + self.half_products_state + [self.product_state]

    def reset(self):
        self.parts_state = [0] * 8
        self.half_products_state = [0] * 3
        self.product_state = 0
        return self.get_state()

    def step(self, action):
        reward = 0
        done = False

        if action < 8: # 检测零件
            part = action
            if self.parts_state[part] == 0: # 该零件未检测
```

```
            reward -= self.part_inspection_cost[part]
            if np.random.random() < self.defect_rate[part]:
                self.parts_state[part] = 2 # 不合格
            else:
                self.parts_state[part] = 1 # 合格
        else:
            reward -= 1 # 惩罚无效动作

    elif action < 11: # 检测半成品
        half_product = action - 8
        if self.half_products_state[half_product] == 1:
            reward -= self.half_product_inspection_cost[half_product]
            if np.random.random() < self.defect_rate[8 + half_product]:
                self.half_products_state[half_product] = 3 # 不合格
            else:
                self.half_products_state[half_product] = 2 # 合格
        else:
            reward -= 1 # 惩罚无效动作

    elif action == 11: # 检测成品
        if self.product_state == 1:
            reward -= self.product_inspection_cost
            if np.random.random() < self.defect_rate[11]:
                self.product_state = 3 # 不合格
                reward -= self.replacement_loss
            else:
                self.product_state = 2 # 合格
                reward += self.market_price
                done = True
        else:
            reward -= 1 # 惩罚无效动作

    elif action < 15: # 拆解半成品
        half_product = action - 12
        if self.half_products_state[half_product] == 3: # 只拆解不合格的半成品
            self.half_products_state[half_product] = 0
            start_index = half_product * 2
            end_index = start_index + 2
            for i in range(start_index, end_index):
                self.parts_state[i] = 0 # 重置零件状态
            reward -= self.disassembly_cost[half_product]
        else:
            reward -= 1 # 惩罚无效动作

    elif action == 15: # 拆解成品
        if self.product_state == 3 or self.product_state == 4:
            self.product_state = 0
```

```
                self.half_products_state = [0] * 3
                reward -= self.disassembly_cost[3]
            else:
                reward -= 1 # 惩罚无效动作

        new_state = self.get_state()
        return new_state, reward, done, {}

def compress_state(state):
    parts_state = state[:8]
    half_products_state = state[8:10]
    product_state = state[10]

    # D: 至少有一个零配件因检测不合格而丢弃
    if 2 in parts_state:
        return 'D'

    # O: 初始状态，存在零件待检测
    if 0 in parts_state:
        return 'O'

    # HP: 半成品待检测
    if 1 in half_products_state and 2 not in parts_state:
        return 'HP'

    # HT: 半成品检测不合格
    if 3 in half_products_state and 2 not in parts_state:
        return 'HT'

    # P: 成品待检测
    if product_state == 1 and 3 not in half_products_state and 2 not in parts_state:
        return 'P'

    # T1: 成品检测不合格
    if product_state == 3 and 3 not in half_products_state and 2 not in parts_state:
        return 'T1'

    # T2: 成品售卖后被退回
    if product_state == 4:
        return 'T2'

    # F: 成品售卖后未退回（成功售卖）
    if product_state == 2:
        return 'F'

    # 如果没有匹配到任何状态，返回一个错误状态
    return 'ERROR'
```

```
def simulate(env, policy, num_episodes, max_steps):
    transition_matrix = defaultdict(lambda: defaultdict(int))
    value_matrix = defaultdict(lambda: defaultdict(float))

    for _ in range(num_episodes):
        state = env.reset()
        compressed_state = env.compress_state(state)
        total_reward = 0

        for _ in range(max_steps):
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            next_compressed_state = env.compress_state(next_state)

            transition_matrix[compressed_state][next_compressed_state] += 1
            value_matrix[compressed_state][next_compressed_state] += reward

            total_reward += reward
            state = next_state
            compressed_state = next_compressed_state

            if done:
                break

    # Normalize transition matrix and value matrix
    for state in transition_matrix:
        total = sum(transition_matrix[state].values())
        for next_state in transition_matrix[state]:
            transition_matrix[state][next_state] /= total
            value_matrix[state][next_state] /= total

    return dict(transition_matrix), dict(value_matrix)

def calculate_expected_cost(transition_matrix, value_matrix, max_steps=3):
    states = list(transition_matrix.keys())
    state_to_index = {state: i for i, state in enumerate(states)}
    n_states = len(states)

    # Initialize cost matrix
    cost_matrix = np.zeros((n_states, n_states))
    for i, state in enumerate(states):
        for j, next_state in enumerate(states):
            if next_state in transition_matrix[state]:
                cost_matrix[i, j] = -value_matrix[state][next_state] # Negative because we want
                    to minimize cost
```

```
    # Initialize expected cost vector
    expected_cost = np.zeros(n_states)
    if 'F' in state_to_index:
        expected_cost[state_to_index['F']] = 0 # Terminal state has zero cost

    # Dynamic programming to calculate expected cost
    for _ in range(max_steps):
        new_expected_cost = np.zeros(n_states)
        for i, state in enumerate(states):
            if state == 'F':
                continue
            costs = []
            for j, next_state in enumerate(states):
                if transition_matrix[state][next_state] > 0:
                    cost = cost_matrix[i, j] + expected_cost[j]
                    costs.append(cost)
            if costs:
                new_expected_cost[i] = np.sum([c * transition_matrix[state][states[j]] for j, c
                    in enumerate(costs)])
            else:
                new_expected_cost[i] = expected_cost[i]
        expected_cost = new_expected_cost

    return {state: expected_cost[i] for i, state in enumerate(states)}

# Add calculate_expected_cost function to the class
ProductionEnvironment.calculate_expected_cost = staticmethod(calculate_expected_cost)

# Add simulate function to the class
ProductionEnvironment.simulate = staticmethod(simulate)

# 将压缩状态函数添加到 ProductionEnvironment 类中
ProductionEnvironment.compress_state = staticmethod(compress_state)

class ProductionOptimizer:
    def __init__(self, env, learning_rate=0.08, n_iterations=400, n_simulations=200,
        max_steps=20):
        self.env = env
        self.n_actions = 16 # 8个零件检测 + 3个半成品检测 + 1个成品检测 + 3个半成品拆解 +
            1个成品拆解
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.n_simulations = n_simulations
        self.max_steps = max_steps
        self.action_probs = np.ones(self.n_actions) / self.n_actions # 初始化为均匀分布
        self.cost_history = []
```

```
    def get_action(self, state):
        return np.random.choice(self.n_actions, p=self.action_probs)

    def optimize(self):
        best_cost = float('inf')
        best_action_probs = None

        for iteration in range(self.n_iterations):
            # 进行模拟
            transition_matrix, value_matrix = self.env.simulate(self.env, self.get_action,
                self.n_simulations, self.max_steps)

            # 计算期望代价
            expected_costs = self.env.calculate_expected_cost(transition_matrix, value_matrix)
            current_cost = expected_costs.get('0', float('inf')) # 从初始状态开始的期望代价

            self.cost_history.append(current_cost) # 记录当前代价

            # 更新最佳结果
            if current_cost < best_cost:
                best_cost = current_cost
                best_action_probs = self.action_probs.copy()
                print(f"迭代 {iteration}")

            # 对动作概率进行扰动
            perturbation = np.random.normal(0, 0.1, self.n_actions)
            new_action_probs = self.action_probs + self.learning_rate * perturbation
            new_action_probs = np.clip(new_action_probs, 0, 1)
            new_action_probs /= new_action_probs.sum() # 归一化

            # 使用新的动作概率进行模拟
            self.action_probs = new_action_probs
            new_transition_matrix, new_value_matrix = self.env.simulate(self.env,
                self.get_action, self.n_simulations, self.max_steps)
            new_expected_costs = self.env.calculate_expected_cost(new_transition_matrix,
                new_value_matrix)
            new_cost = new_expected_costs.get('0', float('inf'))

            # 如果新的代价更低，则保留新的动作概率；否则，以一定概率保留旧的动作概率
            if new_cost < current_cost or np.random.random() < 0.1: # 10%
                的概率接受更差的结果，以跳出局部最优
                self.action_probs = new_action_probs
            else:
                self.action_probs = (self.action_probs + new_action_probs) / 2 # 取平均以平滑变化

            # 检查是否收敛
            if np.allclose(self.action_probs, best_action_probs, atol=1e-4):
```

```
            print(f"在 {iteration} 次迭代后收敛")
            break

        return best_action_probs, best_cost

    def get_final_policy(self, action_probs):
        return [1 if prob > 0.5 else 0 for prob in action_probs]

    def plot_cost_history(self):
        plt.figure(figsize=(10, 6))
        plt.plot(self.cost_history)
        plt.title('优化过程')
        plt.xlabel('迭代次数')
        plt.ylabel('期望代价')
        plt.grid(True)
        plt.show()

    def plot_final_policy(self, final_policy):
        actions = ['检测零件' + str(i+1) for i in range(8)] + \
                  ['检测半成品' + str(i+1) for i in range(3)] + \
                  ['检测成品'] + \
                  ['拆解半成品' + str(i+1) for i in range(3)] + \
                  ['拆解成品']

        plt.figure(figsize=(14, 6))
        plt.bar(actions, final_policy)
        plt.title('最终生产策略')
        plt.xlabel('动作')
        plt.ylabel('决策 (0: 不执行, 1: 执行)')
        plt.xticks(rotation=45, ha='right')
        plt.tight_layout()
        plt.show()

env = ProductionEnvironment()
optimizer = ProductionOptimizer(env)
best_action_probs, best_cost = optimizer.optimize()
min_val = np.min(best_action_probs)
max_val = np.max(best_action_probs)
best_action_probs = (best_action_probs - min_val) / (max_val - min_val)
final_policy = optimizer.get_final_policy(best_action_probs)

print("最佳动作概率:", best_action_probs)
print("最佳期望代价:", best_cost)
print("最终策略:")
actions = ['检测零件' + str(i+1) for i in range(8)] + \
          ['检测半成品' + str(i+1) for i in range(3)] + \
          ['检测成品'] + \
```

```
        ['拆解半成品' + str(i+1) for i in range(3)] + \
        ['拆解成品']
for action, decision in zip(actions, final_policy):
    print(f"{action}: {'执行' if decision == 1 else '不执行'}")
```

附录 E　不同次品率分布下的决策模型取优

1. 单成品 MDP 模型优化

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from State_transition_martix import State_transition_martix
from Cost_matrix import Cost_matrix

plt.rcParams['font.sans-serif'] = ['Microsoft YaHei']
plt.rcParams['axes.unicode_minus'] = False


def value_iteration(transition_matrix, reward_matrix, gamma=1, epsilon=1e-6):
    num_states = transition_matrix.shape[0]
    V = np.zeros(num_states)

    while True:
        V_old = V.copy()

        for s in range(num_states):
            if s == 5:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                     range(4)]
                V[s] = max(Q)
            elif 6 <= s <= 9:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                     range(2)]
                V[s] = max(Q)
            elif 10 <= s <= 14:
                Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                     range(2)]
                V[s] = max(Q)
            else:
                V[s] = np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V))

        if np.max(np.abs(V - V_old)) < epsilon:
            break

    return V


def get_optimal_policy(V, transition_matrix, reward_matrix, gamma=0.99):
    num_states = transition_matrix.shape[0]
    optimal_policy = np.zeros(num_states, dtype=int)
```

```
    for s in range(num_states):
        if s == 5:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(4)]
            optimal_policy[s] = np.argmax(Q)
        elif 6 <= s <= 9:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(2)]
            optimal_policy[s] = np.argmax(Q)
        elif 10 <= s <= 14:
            Q = [np.sum(transition_matrix[s, :] * (reward_matrix[s, :] + gamma * V)) for _ in
                range(2)]
            optimal_policy[s] = np.argmax(Q)

    return optimal_policy


def simulate_profit(transition_matrix, reward_matrix, policy, initial_state=5,
    terminal_state=18,
                num_simulations=10000):
    profits = []
    for _ in range(num_simulations):
        state = initial_state
        total_profit = 0
        while state != terminal_state:
            action = policy[state]
            next_state = np.random.choice(len(transition_matrix[state]),
                p=transition_matrix[state])
            profit = reward_matrix[state, next_state]
            total_profit += profit
            state = next_state
        profits.append(total_profit)
    return profits


def generate_sample_rates(true_rate, sample_size):
    return np.random.binomial(sample_size, true_rate) / sample_size


# 美化后的期望利润分布可视化函数
def visualize_profits(expected_profits):
    plt.figure(figsize=(12, 6))

    # 绘制核密度估计
    kernel = stats.gaussian_kde(expected_profits)
    x_range = np.linspace(min(expected_profits), max(expected_profits), 200)
    plt.plot(x_range, kernel(x_range), 'r-', linewidth=2, label='核密度估计', alpha=0.8)
```

```
    # 绘制期望利润的直方图
    plt.hist(expected_profits, bins=150, edgecolor='black', alpha=0.6, density=True,
        color='lightblue', label='期望利润分布')

    # 设置标题和标签
    plt.title("期望利润分布", fontsize=16, fontweight='bold')
    plt.xlabel("期望利润", fontsize=14)
    plt.ylabel("密度", fontsize=14)

    # 计算并绘制平均值和标准差
    mean_profit = np.mean(expected_profits)
    std_profit = np.std(expected_profits)

    plt.axvline(mean_profit, color='g', linestyle='dashed', linewidth=2, label=f'平均期望利润:
        {mean_profit:.2f}')
    plt.text(mean_profit, plt.ylim()[1] * 0.9, f'平均期望利润: {mean_profit:.2f}',
            horizontalalignment='center', verticalalignment='bottom', fontsize=12,
                color='green')

    # 添加网格线和图例
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.legend(loc='upper right', fontsize=12)

    # 调整布局并显示
    plt.tight_layout()
    plt.show()

    print(f"\n平均期望利润: {mean_profit:.2f}")
    print(f"期望利润标准差: {std_profit:.2f}")


def main():
    c1, c2, b1, b2, bf, cp, fc, xl, cl = 4, 18, 8, 1, 2, 6, -56, 10, 5

    true_ap1, true_ap2, true_bp = 0.1, 0.2, 0.1
    sample_size = 1000

    num_iterations = 1000
    expected_profits = []

    count_a1 = {0: 0, 1: 0}
    count_a2 = {0: 0, 1: 0}
    count_ds = {0: 0, 1: 0}
    count_b = {0: 0, 1: 0}

    for _ in range(num_iterations):
```

```
    ap1 = generate_sample_rates(true_ap1, sample_size)
    ap2 = generate_sample_rates(true_ap2, sample_size)
    bp = generate_sample_rates(true_bp, sample_size)

    best_policy = None
    best_value = float('-inf')

    for a1 in [0, 1]:
        for a2 in [0, 1]:
            for ds in [0, 1]:
                for b in [0, 1]:
                    transition_matrix = State_transition_martix(a1, a2, ap1, ap2, bp, b, ds)
                    reward_matrix = -Cost_matrix(c1, c2, b1, b2, bf, cp, fc, xl, cl)

                    V = value_iteration(transition_matrix, reward_matrix)
                    policy = get_optimal_policy(V, transition_matrix, reward_matrix)

                    if V[5] > best_value:
                        best_value = V[5]
                        best_policy = (a1, a2, ds, b)
                        best_V = V
                        best_transition_matrix = transition_matrix
                        best_reward_matrix = reward_matrix

    count_a1[best_policy[0]] += 1
    count_a2[best_policy[1]] += 1
    count_ds[best_policy[2]] += 1
    count_b[best_policy[3]] += 1

    optimal_policy = get_optimal_policy(best_V, best_transition_matrix, best_reward_matrix)
    profits = simulate_profit(best_transition_matrix, best_reward_matrix, optimal_policy)
    expected_profit = np.mean(profits)
    expected_profits.append(expected_profit / 3) # 除以3

print("\n最优策略统计结果:")
print("                                    ")
print(" 参数   取值0      取值1     主导策略 ")
print("                                    ")
print(f" a1   {count_a1[0]:^9}  {count_a1[1]:^9}  {'0' if count_a1[0] > count_a1[1] else
    '1':^9} ")
print(f" a2   {count_a2[0]:^9}  {count_a2[1]:^9}  {'0' if count_a2[0] > count_a2[1] else
    '1':^9} ")
print(f" ds   {count_ds[0]:^9}  {count_ds[1]:^9}  {'0' if count_ds[0] > count_ds[1] else
    '1':^9} ")
print(f" b    {count_b[0]:^9}  {count_b[1]:^9}  {'0' if count_b[0] > count_b[1] else
    '1':^9} ")
print("                                    ")
```

```
    visualize_profits(expected_profits)


if __name__ == "__main__":
    main()
```

2. 多零件多工序 MDP 优化

```
import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt
from matplotlib import rcParams

# 设置字体为支持中文的字体，比如 SimHei（黑体）
rcParams['font.sans-serif'] = ['SimHei'] # 设置字体为 SimHei（黑体）
rcParams['axes.unicode_minus'] = False # 解决负号 '-' 显示为方块的问题

class ProductionEnvironment:
    def __init__(self):
        # 零件状态：0-未检测，1-检测合格，2-检测不合格
        self.parts_state = [0] * 8

        # 半成品状态：0-未制造，1-未检测，2-检测合格，3-检测不合格
        self.half_products_state = [0] * 3

        # 产品状态：0-未制造，1-未检测，2-检测合格(成功售卖)，3-检测不合格(丢弃)，4-售卖退回
        self.product_state = 0

        # 成本和收益参数
        self.part_inspection_cost = [1] * 8
        self.half_product_inspection_cost = [4] * 3
        self.product_inspection_cost = 6
        self.assembly_cost = [8] * 3 # 三个半成品的组装成本
        self.disassembly_cost = [6] * 3 + [10] # 三个半成品和一个成品的拆解成本
        self.replacement_loss = 40
        self.market_price = 200

        # 动态缺陷率初始化（Beta 分布参数）
        # 初始次品率为 10%，Beta(1, 9) 表示我们有1个不合格样本和9个合格样本的先验知识。
        self.defect_rate_params = [[1, 9]] * 8 + [[1, 9]] * 3 + [[1, 9]]

    def get_defect_rate(self, idx):
        # 根据 Beta 分布计算当前的次品率
        alpha, beta = self.defect_rate_params[idx]
        return alpha / (alpha + beta)
```

```
def update_defect_rate(self, idx, is_defective):
    # 根据检测结果更新 Beta 分布的参数
    alpha, beta = self.defect_rate_params[idx]
    if is_defective:
        alpha += 1 # 如果不合格，增加 alpha
    else:
        beta += 1 # 如果合格，增加 beta
    self.defect_rate_params[idx] = [alpha, beta]

def get_state(self):
    return self.parts_state + self.half_products_state + [self.product_state]

def reset(self):
    self.parts_state = [0] * 8
    self.half_products_state = [0] * 3
    self.product_state = 0
    return self.get_state()

def step(self, action):
    reward = 0
    done = False

    if action < 8: # 检测零件
        part = action
        if self.parts_state[part] == 0: # 该零件未检测
            reward -= self.part_inspection_cost[part]
            defect_rate = self.get_defect_rate(part)
            if np.random.random() < defect_rate:
                self.parts_state[part] = 2 # 不合格
                self.update_defect_rate(part, is_defective=True)
            else:
                self.parts_state[part] = 1 # 合格
                self.update_defect_rate(part, is_defective=False)
        else:
            reward -= 20 # 惩罚无效动作

    elif action < 11: # 检测半成品
        half_product = action - 8
        if self.half_products_state[half_product] == 1:
            reward -= self.half_product_inspection_cost[half_product]
            defect_rate = self.get_defect_rate(8 + half_product)
            if np.random.random() < defect_rate:
                self.half_products_state[half_product] = 3 # 不合格
                self.update_defect_rate(8 + half_product, is_defective=True)
            else:
                self.half_products_state[half_product] = 2 # 合格
                self.update_defect_rate(8 + half_product, is_defective=False)
```

```
            else:
                reward -= 20 # 惩罚无效动作

        elif action == 11: # 检测成品
            if self.product_state == 1:
                reward -= self.product_inspection_cost
                defect_rate = self.get_defect_rate(11)
                if np.random.random() < defect_rate:
                    self.product_state = 3 # 不合格
                    self.update_defect_rate(11, is_defective=True)
                    reward -= self.replacement_loss
                else:
                    self.product_state = 2 # 合格
                    self.update_defect_rate(11, is_defective=False)
                    reward += self.market_price
                    done = True
            else:
                reward -= 20 # 惩罚无效动作

        elif action < 15: # 拆解半成品
            half_product = action - 12
            if self.half_products_state[half_product] == 3: # 只拆解不合格的半成品
                self.half_products_state[half_product] = 0
                start_index = half_product * 2
                end_index = start_index + 2
                for i in range(start_index, end_index):
                    self.parts_state[i] = 0 # 重置零件状态
                reward -= self.disassembly_cost[half_product]
            else:
                reward -= 20 # 惩罚无效动作

        elif action == 15: # 拆解成品
            if self.product_state == 3 or self.product_state == 4:
                self.product_state = 0
                self.half_products_state = [0] * 3
                reward -= self.disassembly_cost[3]
            else:
                reward -= 20 # 惩罚无效动作

        new_state = self.get_state()
        return new_state, reward, done, {}


def compress_state(state):
    parts_state = state[:8]
    half_products_state = state[8:10]
    product_state = state[10]
```

```
    # D: 至少有一个零配件因检测不合格而丢弃
    if 2 in parts_state:
        return 'D'

    # O: 初始状态，存在零件待检测
    if 0 in parts_state:
        return 'O'

    # HP: 半成品待检测
    if 1 in half_products_state and 2 not in parts_state:
        return 'HP'

    # HT: 半成品检测不合格
    if 3 in half_products_state and 2 not in parts_state:
        return 'HT'

    # P: 成品待检测
    if product_state == 1 and 3 not in half_products_state and 2 not in parts_state:
        return 'P'

    # T1: 成品检测不合格
    if product_state == 3 and 3 not in half_products_state and 2 not in parts_state:
        return 'T1'

    # T2: 成品售卖后被退回
    if product_state == 4:
        return 'T2'

    # F: 成品售卖后未退回（成功售卖）
    if product_state == 2:
        return 'F'

    # 如果没有匹配到任何状态，返回一个错误状态
    return 'ERROR'


def simulate(env, policy, num_episodes, max_steps):
    transition_matrix = defaultdict(lambda: defaultdict(int))
    value_matrix = defaultdict(lambda: defaultdict(float))

    for _ in range(num_episodes):
        state = env.reset()
        compressed_state = env.compress_state(state)
        total_reward = 0

        for _ in range(max_steps):
```

```
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            next_compressed_state = env.compress_state(next_state)

            transition_matrix[compressed_state][next_compressed_state] += 1
            value_matrix[compressed_state][next_compressed_state] += reward

            total_reward += reward
            state = next_state
            compressed_state = next_compressed_state

            if done:
                break

    # Normalize transition matrix and value matrix
    for state in transition_matrix:
        total = sum(transition_matrix[state].values())
        for next_state in transition_matrix[state]:
            transition_matrix[state][next_state] /= total
            value_matrix[state][next_state] /= total

    return dict(transition_matrix), dict(value_matrix)


def calculate_expected_cost(transition_matrix, value_matrix, max_steps=3):
    states = list(transition_matrix.keys())
    state_to_index = {state: i for i, state in enumerate(states)}
    n_states = len(states)

    # Initialize cost matrix
    cost_matrix = np.zeros((n_states, n_states))
    for i, state in enumerate(states):
        for j, next_state in enumerate(states):
            if next_state in transition_matrix[state]:
                cost_matrix[i, j] = -value_matrix[state][next_state] # Negative because we want
                    to minimize cost

    # Initialize expected cost vector
    expected_cost = np.zeros(n_states)
    if 'F' in state_to_index:
        expected_cost[state_to_index['F']] = 0 # Terminal state has zero cost

    # Dynamic programming to calculate expected cost
    for _ in range(max_steps):
        new_expected_cost = np.zeros(n_states)
        for i, state in enumerate(states):
            if state == 'F':
```

```
                continue
            costs = []
            for j, next_state in enumerate(states):
                if transition_matrix[state][next_state] > 0:
                    cost = cost_matrix[i, j] + expected_cost[j]
                    costs.append(cost)
            if costs:
                new_expected_cost[i] = np.sum([c * transition_matrix[state][states[j]] for j, c
                    in enumerate(costs)])
            else:
                new_expected_cost[i] = expected_cost[i]
        expected_cost = new_expected_cost

    return {state: expected_cost[i] for i, state in enumerate(states)}

beta=1.5

# Add calculate_expected_cost function to the class
ProductionEnvironment.calculate_expected_cost = staticmethod(calculate_expected_cost)

# Add simulate function to the class
ProductionEnvironment.simulate = staticmethod(simulate)

# 将压缩状态函数添加到 ProductionEnvironment 类中
ProductionEnvironment.compress_state = staticmethod(compress_state)


class ProductionOptimizer:
    def __init__(self, env, learning_rate=0.1, n_iterations=100, n_simulations=50,
        max_steps=15):
        self.env = env
        self.n_actions = 16 # 8个零件检测 + 3个半成品检测 + 1个成品检测 + 3个半成品拆解 +
            1个成品拆解
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.n_simulations = n_simulations
        self.max_steps = max_steps
        self.action_probs = np.ones(self.n_actions) / self.n_actions # 初始化为均匀分布
        self.cost_history = []

    def get_action(self, state):
        return np.random.choice(self.n_actions, p=self.action_probs)

    def optimize(self):
        best_cost = float('inf')
        best_action_probs = None
```

```
    for iteration in range(self.n_iterations):
        # 进行模拟
        transition_matrix, value_matrix = self.env.simulate(self.env, self.get_action,
            self.n_simulations,
                                                    self.max_steps)

        # 计算期望代价
        expected_costs = self.env.calculate_expected_cost(transition_matrix, value_matrix)
        current_cost = expected_costs.get('0', float('inf')) # 从初始状态开始的期望代价

        self.cost_history.append(current_cost) # 记录当前代价

        # 更新最佳结果
        if current_cost < best_cost:
            best_cost = current_cost
            best_action_probs = self.action_probs.copy()
            print(f"迭代 {iteration}")

        # 对动作概率进行扰动
        perturbation = np.random.normal(0, 0.1, self.n_actions)
        new_action_probs = self.action_probs + self.learning_rate * perturbation
        new_action_probs = np.clip(new_action_probs, 0, 1)
        new_action_probs /= new_action_probs.sum() # 归一化

        # 使用新的动作概率进行模拟
        self.action_probs = new_action_probs
        new_transition_matrix, new_value_matrix = self.env.simulate(self.env,
            self.get_action, self.n_simulations,
                                                        self.max_steps)
        new_expected_costs = self.env.calculate_expected_cost(new_transition_matrix,
            new_value_matrix)
        new_cost = new_expected_costs.get('0', float('inf'))

        # 如果新的代价更低，则保留新的动作概率；否则，以一定概率保留旧的动作概率
        if new_cost < current_cost or np.random.random() < 0.1: # 10%
            的概率接受更差的结果，以跳出局部最优
            self.action_probs = new_action_probs
        else:
            self.action_probs = (self.action_probs + new_action_probs) / 2 # 取平均以平滑变化

        # 检查是否收敛
        if np.allclose(self.action_probs, best_action_probs, atol=1e-4):
            print(f"在 {iteration} 次迭代后收敛")
            break

    return best_action_probs, best_cost
```

```
    def get_final_policy(self, action_probs):
        return [1 if prob > 0.5 else 0 for prob in action_probs]

    def plot_cost_history(self):
        plt.figure(figsize=(10, 6))
        plt.plot(self.cost_history)
        plt.title('优化过程')
        plt.xlabel('迭代次数')
        plt.ylabel('期望代价')
        plt.grid(True)
        plt.show()

    def plot_final_policy(self, final_policy):
        actions = ['检测零件' + str(i + 1) for i in range(8)] + \
                  ['检测半成品' + str(i + 1) for i in range(3)] + \
                  ['检测成品'] + \
                  ['拆解半成品' + str(i + 1) for i in range(3)] + \
                  ['拆解成品']

        plt.figure(figsize=(14, 6))
        plt.bar(actions, final_policy)
        plt.title('最终生产策略')
        plt.xlabel('动作')
        plt.ylabel('决策 (0: 不执行, 1: 执行)')
        plt.xticks(rotation=45, ha='right')
        plt.tight_layout()
        plt.show()


# 运行模拟并计算期望利润
profits = []
num_simulations = 2

for _ in range(num_simulations):
    env = ProductionEnvironment()
    optimizer = ProductionOptimizer(env)
    best_action_probs, best_cost = optimizer.optimize()
    profits.append(optimizer.cost_history[-1]) # 获取最终的期望利润

profits = [profit * beta for profit in profits]

# 绘制期望利润的分布图
plt.figure(figsize=(10, 6))
plt.hist(profits, bins=150, edgecolor='black')
plt.title('期望利润分布 ')
plt.xlabel('期望利润')
plt.ylabel('频率')
```

```
plt.grid(True)
plt.show()
```