基于蒙特卡洛模拟和多阶段优化分析的生产过程收益优化研究

摘　要

在现代制造业中，质量控制和成本优化是企业提高市场竞争力的关键环节。通过合理的优化，企业可以有效降低次品率，以减少不必要的损失，提高生产效率。因此，研究如何优化检测和拆解流程，最大化收益，成为了制造业中的重要问题。

针对问题一：基于**假设检验**和**贝叶斯推断原理，**设计出检测次数尽可能少的抽样检测方案。首先利用**二项分布**和**正态分布模型**推导出在不同置信水平下的**最小样本量**。然后结合**贝叶斯停止规则**，通过给出具体方案：在抽样过程中，通过**贝叶斯推断，**逐步调整检测次数，直到达到对应的**最小样本量**。在 95%和 90%置信度下，方案有效地减少了检测样本量，并使零配件次品率的控制更加精准，既降低了检测费用，也保证了成品质量。

针对问题二：在考虑零配件与成品检测的基础上，我们采用了**蒙特卡洛模拟**方法，评估了不同检测与拆解策略组合下的成本与收益。通过模拟大量随机样本，量化了次品率、检测成本、装配成本、市场售价及拆解费用对最终收益的影响，得到了具体决策方案表。分析表明，通过最优策略的选择，企业能够最大化平均收益。

针对问题三：在问题二的基础上进行了更为复杂的**多阶段优化分析**。首先采用**动态规划模型**，将生产过程分解为多个阶段，递推求解检测与拆解的最优策略。考虑到问题复杂性增加，进一步引入**遗传算法**，通过模拟自然进化过程，优化求解全局近似最优解。遗传算法适用于在状态空间增大时有效搜索接近最优的解决方案，通过种群进化和迭代，进一步提高了问题三中的求解效率。得到最佳决策组合。

针对问题四：在问题二和问题三的基础上，假设零配件、半成品和成品的次品率通过抽样检测得到，而非直接已知，企业需在不确定的次品率下动态调整检测和拆解策略。通过引入**贝叶斯推断模型**，动态更新检测数据，估算各阶段次品率，并结合二三问模型优化决策流程。

本文通过对生产过程中的检测与拆解策略进行全面的分析与优化，提出了一套基于**假设检验、蒙特卡洛模拟、动态规划和遗传算法**的收益最大化模型。通过减少检测次数、优化拆解策略、并结合**贝叶斯推断模型**应对不确定性，本文有效降低了企业的生产成本，提升了生产效率和收益。同时，本文的模型具备良好的鲁棒性和广泛的适用性，能够应对复杂的多阶段生产流程与不确定环境中的质量控制问题。未来的研究可以进一步改进参数自适应性和求解效率，以更好地应对现实中多变的市场需求与生产条件。

关键词：抽样检测 贝叶斯模型 蒙特卡洛模拟 动态规划 遗传算法

一、 问题重述

1.1 问题背景

在现代制造业中，确保产品质量与控制成本是企业面临的核心挑战。随着全球化和市场竞争的加剧，企业必须通过严格的质量管理体系来保证最终产品符合标准。这包括从供应商处采购的零配件，到生产过程中的每一个环节，再到成品的检测与市场调度。

图 1 企业质量管理体系

其中，零配件的检测是保证成品质量的最基础的步骤。企业必须设计和实施有效的抽样检测方案，以在最小的成本和风险下，确保零配件和成品的质量，从而提升客户满意度、减少退换货损失，并在激烈的市场竞争中占据优势。通过优化这些过程，企业能够在保证产品质量的同时，实现成本效益最大化，并持续推动其在市场中的领先地位。

1.2 问题提出

问题一：企业在采购零配件时，供应商声称次品率不超过某个标称值，但企业仍需通过抽样检测来决定是否接收这些零配件。在检测过程中，企业希望尽可能减少检测次数，以降低成本。假设零配件的次品率标称值为 10%，需要设计一个有效的抽样检测方案，在保证信度为 95%或 90%的条件下，做出接收或拒收零配件的决策。

问题二：在生产过程中，企业需要对零配件及成品的质量作出决策，包括是否对零配件进行检测、是否对成品进行检测、以及是否对不合格成品进行拆解。每个决策都会涉及到检测成本、拆解费用、调换损失等因素，直接影响企业的经济效益。为此，企业需要根据不同情况下的次品率、成本和损失等数据，建立决策模型，优化各个生产环节的检测与处理策略，以实现成本最小化和效益最大化。

问题三：生产过程不仅仅局限于两种零配件和成品的装配，还涉及多个工序和更多种类的零配件。对于复杂的生产链条，企业需要根据每道工序及每个零配件、半成品和成品的次品率，决定在每个阶段是否进行检测，以优化生产效率并降低不合格产品带来的损失。针对多道工序、多种零配件的生产情况，制定出合理的决策方案。

问题四：假设在问题 2 和问题 3 中，零配件、半成品和成品的次品率均通过抽样检测得到，而不是直接已知的固定值。此时，企业需要重新考虑整个生产过程中的决策方案。在抽样检测的基础上，通过估算各阶段的次品率，企业需依据检测结果调整是否检测、拆解等决策，从而确保在不确定性下依然能够优化生产流程，降低成本

并提高成品合格率。请结合抽样检测方法，重新完成问题 2 和问题 3 中的决策方案，并给出相应的依据与结果分析。

二、 问题分析

2.1 问题一的分析

问题一的核心是设计一个基于抽样检测的统计决策方案，以判断供应商提供的零配件次品率是否超过标称值。使用**二项分布**作为次品率的概率模型，通过**假设检验**的方法，借助**正态分布模型**计算**最小样本量**，为了实时检测结果，动态调整检测决策，通过**贝叶斯模型**进一步减少不必要的抽样，在保证检测效果的同时减少检测成本。

2.2 问题二的分析

问题二的核心是通过合理的检测和处理策略，最大化企业的经济收益，减少生产过程中可能产生的次品损失和额外成本。通过分析，企业的生产过程需要做出以下四个关键决策：对零配件 1 是否进行检测、对零配件 2 是否进行检测、对装配好的成品是否进行检测、对不合格成品是否进行拆解。通过**蒙特卡洛模拟**，评估了不同策略组合下的收益，考虑了次品率、检测成本、装配成本、市场售价、调换损失及拆解费用等因素，最终选择能够最大化平均收益的方案。

2.3 问题三的分析

问题三是在问题二的基础上进一步延伸出的更为复杂的**多阶段优化**问题。通过在问题二中对零配件检测的基础上，增加对各生产阶段次品率的累积分析，问题三涵盖了从零配件到半成品再到成品的完整生产流程。生产过程被分解为**五个阶段**，包括零配件检测、半成品检测与组装、半成品拆解、成品检测以及成品拆解。在每个阶段中，检测和拆解的决策不仅影响当前阶段的次品率和成本，还会对后续阶段的质量和收益产生累积效应。通过**动态规划模型**，递归求解每个阶段的最优决策组合，从而最大化企业的整体收益或最小化生产成本。又因为状态量（接近 2^{16}）较多，为了在较短的时间内找到一个**最优解**，通过**遗传算法**，迭代逐步逼近最优解。

2.4 问题四的分析

问题四要求在问题二和问题三的基础上，假设零配件、半成品和成品的次品率通过抽样检测得到，而不是已知的固定值。这样，企业必须在不确定的环境下动态调整检测与拆解决策。核心挑战在于如何通过抽样检测估算各阶段的次品率，并根据这些估计值做出合理的决策，包括是否继续检测、是否进行拆解等。为应对此问题，我们引入了**贝叶斯推断模型**，通过实时更新检测数据，动态调整对次品率的估计。结合二三问的模型，企业可以根据次品率波动，评估不同检测策略和拆解策略对生产收益的影响，并在不确定性条件下优化决策，以实现成本最低化和收益最大化。

2.5 思维框架

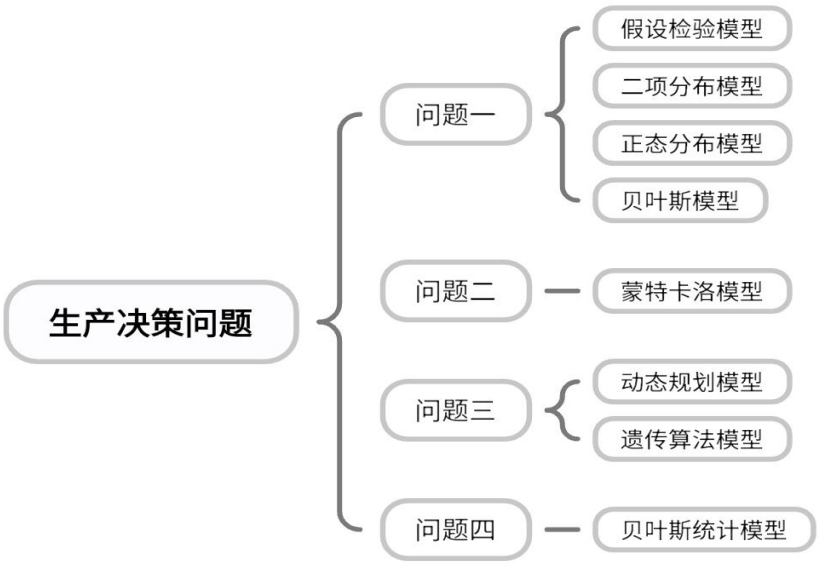

图 2 思维框架

三、 模型假设

1. 假设通过抽样检测得到的次品率能够准确反映整批产品的整体质量水平。这一假设使得我们可以根据有限样本的检测结果对整体质量进行推断。

2. 假设检测过程能够 100% 准确地识别出不合格品。

3. 假设市场对产品的需求保持稳定，不会因质量问题而产生显著变化。简化了收益计算，但在更加复杂的市场环境中可能需要进行调整。

4. 假设所有零配件的次品率是独立的且已知，并且该次品率在整个生产过程中保持不变。

5. 假设未经检测的零件在经过拆除后，必须选择检验。简化拆除后模型计算，防止不合格零件在生产流程中不断轮转，造成成本增加，企业信誉降低。

四、 符号说明

符号	含义	单位
p	次品率	\
n	样本数量	\
k	检测出的次品数量	\
Z	标准正态分布的 Z 值，用于计算最小样本量	\
E	允许的误差率	\
C	成本	元
L	损失费用	元
S	市场售价	元

$Profit$	收益	元
avg	平均收益	元
A_i	问题二中决策变量	\
X_i	问题中三中检测第 i 个零配件的决策变量	\
y_i	问题中三中检测第 i 个半成品决策变量	\
y_f	问题中三中检测成品决策变量	\
z_{si}	问题中三中拆解第 i 个半成品决策变量	\
z_f	问题中三中拆解成品决策变量	\

五、 模型的建立与求解

5.1 问题一模型的建立与求解

5.1.1 假设检验的建立

为判断样本中次品率能否代替全体的次品率，即是否会与预期的理论值存在显著差异，需要构建假设检验模型。在进行假设检验时，首先我们需要提出提出假设，包括零假设和备择假设[1]：

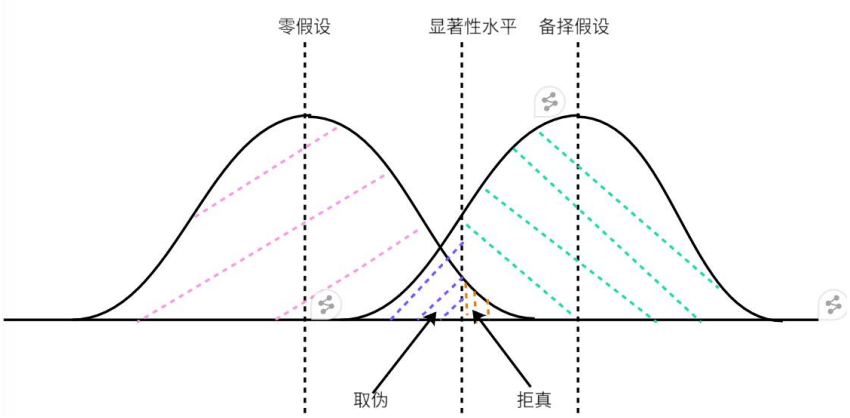

图 3 假设检验基本原理

零假设 H_0：样本中的次品率等于标称次品率时，代表样本没有显著偏离预期，产品质量符合标准。即假设供应商的次品率：

$$p=p_0=0.10 \tag{1}$$

备择假设 H_1 :样本中的次品率大于标称次品率 p_0 （单侧检验），代表怀疑次品率高于预期，需要拒收该批次的产品。如下为两种情形的备择假设：

$$\begin{cases} \text{情形 1:次品率 } p>0.10 \text{（在 95\%信度下拒收次品率超过标称值的零配件）。} \\ \text{情形 2：次品率 } p<=0.10 \text{（在 90\%信度下接收次品率不超过标称值的零配件）。} \end{cases}$$

情形 1 和情形 2 分别对应问题中的两种情形，针对两种情形，分别进行假设检验。

5.1.2 二项分布模型的建立

在质量检验过程中，为了判断一个批次的零配件是否符合质量标准，我们可以假设随机抽取的样本服从**二项分布**，因为每个零配件要么是合格品，要么是次品。这种二项分布假设能够帮助我们分析样本中的次品数量是否符合预期的次品率[2]。

$$X \sim \text{Binomial}(n,p) \tag{2}$$

假设我们从批次中随机抽取 n 个零件。根据二项分布，抽样中出现 k 个次品的概率是：

$$P(X=k)=\left(\frac{n}{k}\right)p^k(1-p)^{n-k} \tag{3}$$

同时，为了确保抽样结果具有足够的统计学意义和精度，我们需要计算**最小样本量**，即在给定的置信水平和误差率下，需要抽取的样本数量，并确保抽样的结果具有足够的统计学意义和精度。这时，借助**正态分布模型**，我们可以近似计算出所需的最小样本量。

$$\hat{p} \sim N\left(p,\frac{p(1-p)}{n}\right) \tag{4}$$

由于一般工厂生产产量大、周期长，根据**中心极限定理**，当样本量较大时，二项分布可以近似为正态分布，从而使得我们可以通过正态分布的 Z 值来推导最小样本量公式。由于企业不会因为次品率低于标称值而拒收产品，只关心一个方向（次品率是否超标，所以采用右侧检验[3]。

最小样本量公式如下，其中 Z_α是置信水平对应的标准正态分布的 Z 分数， p_0是预期的次品率（标称次品率），E 是允许的误差率（例如 5%）：

$$n=\frac{Z_\alpha^2 \cdot p_0(1-p_0)}{E^2} \tag{5}$$

二项分布用于描述抽样结果的分布特性，而正态分布则用于计算出合理的最小样本量，确保统计检验的可靠性和有效性。

5.1.3 贝叶斯停止模型的建立

为了进一步减少检测样本量，采取结合**贝叶斯推断的方法**，这种方法不仅能够根据实时检测结果动态调整检测决策，还能通过贝叶斯停止规则进一步减少不必要的抽样[4]。

首先，设定 Beta 分布作为先验分布，用以表达企业对供应商次品率的初始估计。假设次品率 p 服从 Beta 分布，即

$$p \sim \mathrm{Beta}(\alpha,\beta) \tag{6}$$

随着每次检测结果的更新，我们可以通过贝叶斯定理实时更新后验分布：

$$p \sim Beta(\alpha+k,\beta+n-k) \tag{7}$$

其中：

α：Beta 分布的第一个形状参数，通常代表成功事件的次数加 1，在贝叶斯推断中，它来自先验分布的参数，并在后验中被次品数量 k 更新。

β：Beta 分布的第二个形状参数，通常代表失败事件的次数加 1，在贝叶斯推断中，它来自先验分布的参数，并在后验中通过 n-k 更新。

k：检测出的次品数量

该后验分布反映了检测过程中对次品率的最新估计。 后验概率的计算公式为：

$$P(p>p_0 \mid \mathrm{data}) = 1 - F(p_0 \mid \alpha+k,\beta+n-k) \tag{8}$$

$$P(p \le p_0 \mid \mathrm{data}) = F(p_0 \mid \alpha+k,\beta+n-k) \tag{9}$$

其中 F 是 Beta 分布的累积分布函数（CDF），计算出次品率超过标称值的概率。累积分布函数的表达式如下

$$F(x \mid \alpha,\beta) = \int_0^x \frac{t^{\alpha-1}(1-t)^{\beta-1}}{B(\alpha,\beta)}\,dt \tag{10}$$

其中 x 是累积分布函数的自变量，t 是积分中的变量，表示 Beta 分布上的次品率概率值。

根据置信度，设定$P(p>p_0 \mid data)$或者$P(p \le p_0 \mid data)$的阈值，即可根据检测情况提前结束后续检验，直接拒收或者接受该批次零部件。

5.1.4 模型的求解

我们在 3%和 5%两种误差率水平下分别计算了最小样本量和拒收值，得到结论：

表 1 模型一结论

		当允许误差率为 5% 时	当允许误差率为 3% 时
95% 置信水平	最小样本量	98	271
	拒收值	10	28
90% 置信水平	最小样本量	60	165
	拒收值	6	17

根据贝叶斯模型，分别设定$P(p>p_0 \mid data)>0.95$为 95% 置信水平拒收该批次零配件的阈值$P(p \le p_0 \mid data)>90\%$为 90% 置信水平接收该批次零配件的阈值。

得出最终的**抽样检测方案**：

1. **实时更新，提前决策：** 使用贝叶斯模型对每次检测结果进行实时更新，根据更新后的后验分布调整检测策略，如果达到贝叶斯的终止阈值，则提前终止检测，如果没有达到，则继续抽取。
2. **终止检测：** 如果抽取的次数到达了表 1 中对应的最小样本量，仍然没有达到贝叶斯的终止阈值，则自动终止检测，根据表 1 的拒收值决定是否接收。

最终将检测次数尽可能少的抽样检测方案确定为：根据最小样本量和贝叶斯停止规则动态调整检测过程，并结合假设检验结果的综合检测方案。这种综合的检测方案能够有效地控制次品率，降低生产成本，提高生产效率，并确保产品质量符合企业标准。

5.2 问题二模型的建立与求解

5.2.1 企业的生产和决策流程分析

零配件采购：采购零配件 1 和零配件 2，他们均有各自的次品率和购买成本。企业需要决定是否检测这些零配件，如果检测，需要支付检测成本。

成品装配：将两个零配件进行装配，如果其中任意一个零配件不合格，则成品必定不合格。装配成品有一个装配成本。企业需要决定是否对成品进行检测。

不合格成品的处理：如果成品不合格，企业可以选择拆解并重新利用零配件，或者直接报废。拆解过程产生一定的费用。

市场销售：不合格产品被用户购买后，企业需要承担调换损失。并回到不合格成品的处理。

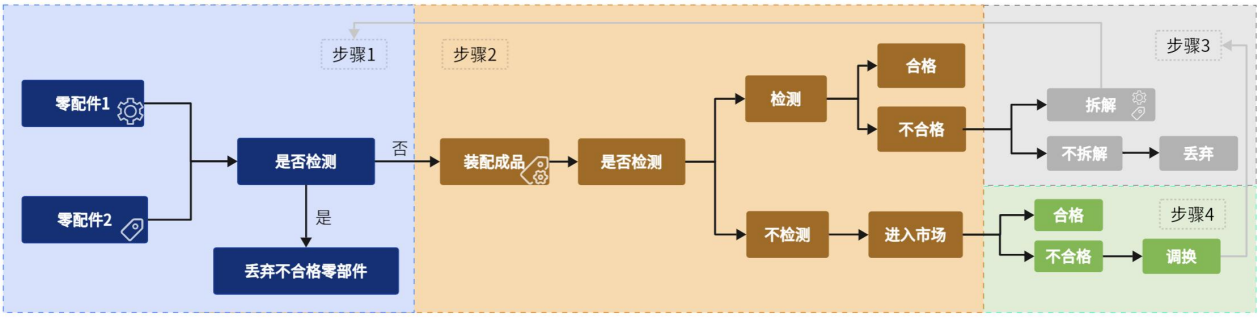

图 4 问题二流程图

5.2.2 蒙特卡洛模型建立

基于上述过程，采用蒙特卡洛模型。它是通过建立数学概率模型将随机事件的概率特征与数学分析的解联系起来的一种仿真方法[5]。

第一步，定义模型中的决策和参数变量：

- $A_1 \in \{0,1\}$：是否检测零配件 1，1 表示检测，0 表示不检测。
- $A_2 \in \{0,1\}$：是否检测零配件 2，1 表示检测，0 表示不检测。
- $A_3 \in \{0,1\}$ ：是否检测成品，1 表示检测，0 表示不检测。
- $A_4 \in \{0,1\}$ ：是否拆解不合格成品，1 表示拆解，0 表示不拆解。

第二步，确定成本和收益：

根据生产流程，企业需要为每个成品支付一下成本：

零配件采购成本：

$$C_b=C_1+C_2 \tag{11}$$

零配件的监测成本：

零配件 1 的检测成本：

$$C_{de1}=\begin{cases}0, & 若A_1=0（不检测零配件1）\\ C_{det1}, & 若A_1=1（即检测零配件1）\end{cases} \tag{12}$$

同理零配件 2，故零配件的检测成本为：

$$C_{test}=C_{de1}+C_{de2} \tag{13}$$

成品检测成本:

$$C_{detf}=\begin{cases}0, & 若A_1=0（不检测成品）\\ C_{det3}, & 若A_4=1（即检测成品）\end{cases} \tag{14}$$

拆解成本:

$$C_{re}=\begin{cases}C_{recycle}, & 若B=1（即已检测成品）\\ 0, & 若B=0（即未检测成品）\end{cases} \tag{15}$$

故整个生产过程的总成本可以表示为：

$$C=C_b+C_{test}+C_{detf}+C_{re} \tag{16}$$

第三步，确定收益函数：

- **若成品合格并进入市场**，则收益为市场售价减去购买成本和装配成本：

$$Profit=S-C-C_{assemble} \tag{17}$$

- **若成品不合格且不进行拆解**，则会产生总的采购成本和额外的损失 L：

$$Profit=-(S+L) \tag{18}$$

- **若成品不合格且选择进行拆解**，则需要支付拆解费用 $C_{recycle}$，并且根据拆解后的零配件是否合格重新评估成品的合格性和收益。

第四步，进行蒙特卡洛模拟和平均收益计算

通过对上述收益函数进行蒙特卡洛模拟，我们可以计算出每种决策组合在不同情景下的平均收益。具体表达式为：

$$avg=\frac{\sum_{i=1}^{n}profit_i}{n} \tag{19}$$

通过对所有可能的决策组合进行蒙特卡洛模拟，比较不同决策下，平均收益的大小，找出最大化平均收益的决策组合。最后在输入不同场景下的数据，找出适合其生产过程的最优决策。

5.2.3 模型求解

使用 python 实现模型，通过循环嵌套语句，对不同决策下，进行 3000 次蒙特卡洛模拟，计算平均收益，进行排序，得出最大平均收益，以及对应决策方案。

附录是核心代码的伪代码，是关于如何遵循生产流程。

首先，计算初始购买成本，根据决策点 A1 和 A2，判断零部件是否进行检测，如果零配件不合格且选择了检测，会进行重新购买并重新检测，直到合格为止。否则，利用随机过程，赋值零配件状态。

然后，进入装配，产品均合格，根据装配产品合格率，判断成品是否合格，任意一个零配件不合格，成品直接判定不合格。对成品进行检验后，进入市场。

如果成品不合格且选择拆解 (`c` 为 `True`)，则进入拆解流程：增加拆解成本 `c` 再次检测零配件（如果最初未检测），并根据检测结果重新购买合格的零配件以及配套的零配件。重新装配，直到成品合格为止。如果成品最终合格，计算收益；否则增加损失。如果不选择拆解，则直接增加损失。通过这样，既保证了成本检验后，拆解的处理与否，同时也保证用户接受到不合格产品，拆解的处理与否。

5.2.4 求解结果分析

通过运行代码，得到了 6 种不同场景下的最优决策，如表 2。

表 2：最优决策结果

情况	是否检测零配件 1	是否检测零配件 2	是否检测成品	是否拆解成品
1	是	否	否	是
2	是	是	否	是
3	是	否	否	是
4	是	是	否	是
5	否	是	否	是
6	是	否	否	否

对共同点进行分析：

大多数情境都选择检测零配件 1，这可能与在多数情况下零配件 1 的检测成本均低于零配件 2 有关。

大多数情况喜爱，都选择拆解成品。这表明在大多数情境下，拆解成品后可以回收零配件 1 和零配件 2，并尝试重新装配，这可能比直接接受损失更有利。重新利用合格的零配件能带来更高的收益。而情况 6 由于其拆解成本过高，而未进行拆解

在 6 个情境下，普遍都不对成品进行检验，这可能与成品检验成本在多数情况下都偏高，且零配件和成品的合格率都相对比较高，直接进入市场比检测更具有经济效益，也就是说，即使成品不合格，带来的损害可能也小于检测成本。

为了更直观地了解不同场景下的决策结果，用柱状图展示不同场景下的最大平均收益。

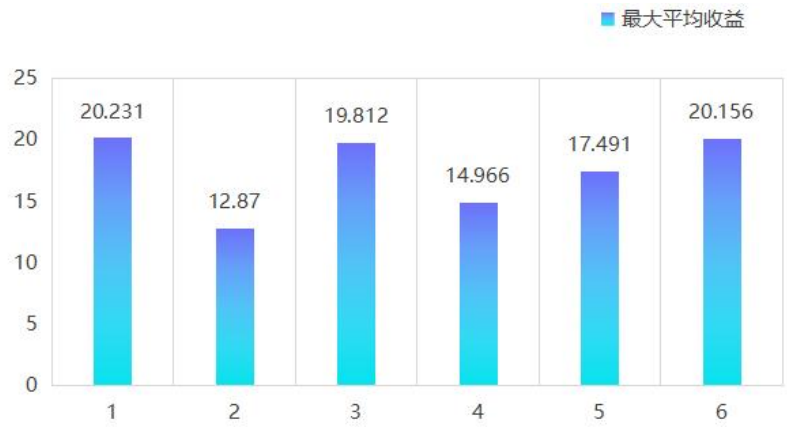

图 5 最大平均收益图

从图表中，我们可以看到：

1. 最大平均收益在不同情况下，变化较大，从最低 12.87 到最高的 55.5 元。这从侧面反映不同参数的设置对收益的明显影响。

2. 情况 1 和情况 6 的收益较高，而在这两种情况下次品率较低，且情况 6 在如此之高的拆解费用（40 元）仍能达到如此之的收益，说明了零配件的质量对收益的显著影响。

3. 情况 1 和情况 3 的收益较为接近，而他们的调换成本，情况一为 6 元，情况 3 为 30 元，在次品率不高的情况下，调换成本不会收益产生较大影响。

总而言之，次品率是影响企业收入的重要因素，企业应当多加关注产品质量，降低产品次品率。

5.3 问题三模型的建立与求解

5.3.1 动态规划模型的建立

动态规划（Dynamic Programming, DP）是一种将复杂问题分解为更小的子问题，并通过存储子问题的解来避免重复计算的算法设计方法。它常用于求解最优化问题。动态规划的核心思想是利用最优子结构和重叠子问题的特性，通过递推关系逐步求解整个问题[6]。

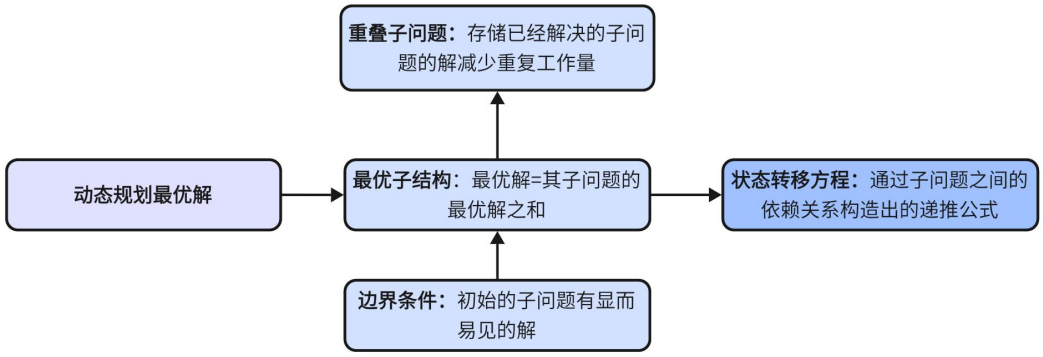

图 6 动态规划原理图

最优子结构：问题的最优解可以通过其子问题的最优解构造出来，即当前问题的最优解由一个或多个子问题的解组成。

重叠子问题：问题的子问题会重复出现，直接计算每次子问题的解会导致大量的重复工作。通过存储已经解决的子问题的解，可以减少计算量。

状态转移方程：通过子问题之间的依赖关系构造出递推公式，也称为状态转移方程。这个公式用于从已解决的子问题推导出更大的问题的解。

边界条件：初始的子问题有显而易见的解，通常作为递推过程的基础。

5.3.2 动态规划模型的求解

将每一个决策组合（包括零配件检测、半成品检测、成品检测、半成品的拆解和不合格成品拆解）作为一个状态。每个状态通过一个长度为 16 的二进制数组来表示，决策变量包括：

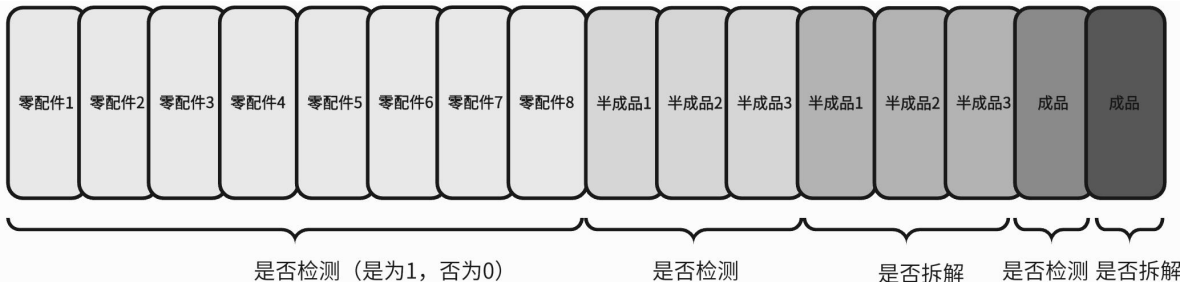

图 7 动态规划的五个阶段

前 8 位：是否检测 8 个零配件，$x_1,x_2,\cdots,x_8\in\{0,1\}$

第 9-11 位：是否检测 3 个半成品，$y_1,y_2,y_3\in\{0,1\}$

第 12-14 位：是否拆解半成品 1 到 3，$z_{s1},z_{s2},z_{s3}\in\{0,1\}$

第 15 位：是否检测成品，$y_f\in\{0,1\}$

第 16 位：是否拆解成品，如果成品被检测为次品，$z_f\in\{0,1\}$

给出零配件、半成品和成品的状态**转移方程**：

对于零配件 i，如果检测零配件 i，需要支付检测成本并进入下一个状态，状态转移为：

$$dp[i]=\min(\text{检测成本}+dp[\text{下一阶段}],\text{不检测成本}+dp[\text{下一阶段}]) \quad (20)$$

对于半成品 j，如果选择拆解半成品，计算拆解成本；否则，计算检测或不检测成本。状态转移方程为：

$$dp[j]=\min(\text{检测成本}+dp[\text{下一阶段}],\text{不检测成本}+dp[\text{下一阶段}],\text{拆解成本}) \quad (21)$$

对于成品，选择检测成品或不检测成品，并考虑市场售价、生产成本和可能的调换损失：

$$dp[\text{成品}]=\min(\text{检测成本}+\text{市场价格}-\text{生产成本},\text{不检测成本}+\text{调换损失}) \quad (22)$$

通过遍历所有的决策组合，依次计算每个决策组合的成本，得出最高收益的决策组合是：（1111111100000000），在这个组合下，收益是 24.22。

5.3.3 遗传算法模型的建立和求解

面临的多阶段的检测与拆解决策问题，我们同时采用遗传算法，通过模拟自然选择和基因进化的方式，递归寻找最优解。在遗传算法中，每个个体（解）通过二进制编码来表示检测与拆解的决策组合[7]。

种群是由若干个体组成的集合，每个个体表示一个可能的解（决策组合）。我们随机生成初始种群，种群大小设为 N，每个个体是一个长度为 16 的二进制串，代表一个检测和拆解的策略组合。 为了评估每个个体（解）的优劣，我们定义适应度函数来衡量策略组合的总收益。适应度函数的目标是最大化总收益，即：

$$Fitness=\text{总收益}-\text{总成本} \quad (23)$$

$$\text{总成本}=\sum_{i=1}^{8}\text{零件的监测成本}+\sum_{j=1}^{3}\text{半成品检测成本}+\sum_{j=1}^{3}\text{拆解成本}+\text{成品检测成本}+\text{成品拆解成本} \quad (24)$$

选择操作模拟生物学中的“适者生存”机制。通过轮盘赌选择或锦标赛选择，从当前种群中选择适应度高的个体作为父代，用于生成下一代个体。适应度越高的个体被选择的概率越大。

交叉操作模拟基因重组，通过交换父代个体的部分基因生成新的子代。选择两个父代个体进行单点或多点交叉，随机选择某个位置后将其基因交换，以产生新的个体。如下图：

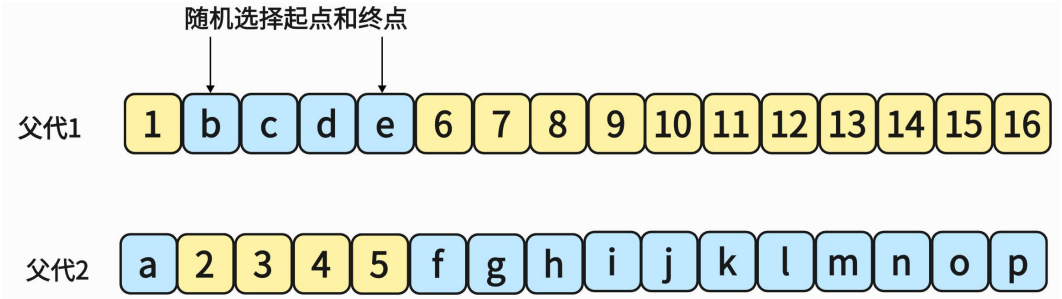

图 8 交叉遗传原理图

为了引入多样性并避免陷入局部最优解，我们对种群中的个体进行变异操作。以概率 p_m 随机选择某个基因（决策变量）进行变异，即将 0 变为 1，或者将 1 变为 0。例如： $x_1=0$原始基因为，变异后变为 ， $x_1=1$表示从“零配件不检测”变为“零配件检测”。

遗传算法通过多次迭代来进化种群，每次迭代都会生成新的个体并更新种群。停止条件为种群中的最优解在若干代中保持不变，表明算法已经收敛。

经过若干代迭代后，适应度最高的个体即为问题三的最优解。该个体代表最优的检测与拆解决策组合，在最大化收益的同时，合理地控制检测和拆解的成本。通过代码迭代可得最佳个体为（1101101011111101），在这个组合下，收益为 43.2。

5.4 问题四模型的建立与求解

5.4.1 贝叶斯统计模型的建立

假设次品率 p 服从 Beta 分布，即 $p \sim Beta(\alpha,\beta)$，其中 α 和 β 是先验的形状参数，可以基于历史数据或经验值设定[8]。

由贝叶斯更新公式，在每次抽样检测过程中，更新后的次品率 p 的后验分布为：

$$p \mid k \sim \text{Beta}(\alpha + k, \beta + n - k) \tag{25}$$

在每次检测后，更新后的次品率 p 的期望值可以表示为：

$$E(p \mid k,n) = \frac{\alpha_{posterior}}{\alpha_{posterior} + \beta_{posterior}} \tag{26}$$

其中，k 是检测出的次品数量，$n-k$ 是检测出的合格品数量，$\alpha_{posterior}$ 和 $\beta_{posterior}$ 是更新后的形状参数，绘制 5%、10%和 20%的可视化曲线如下：

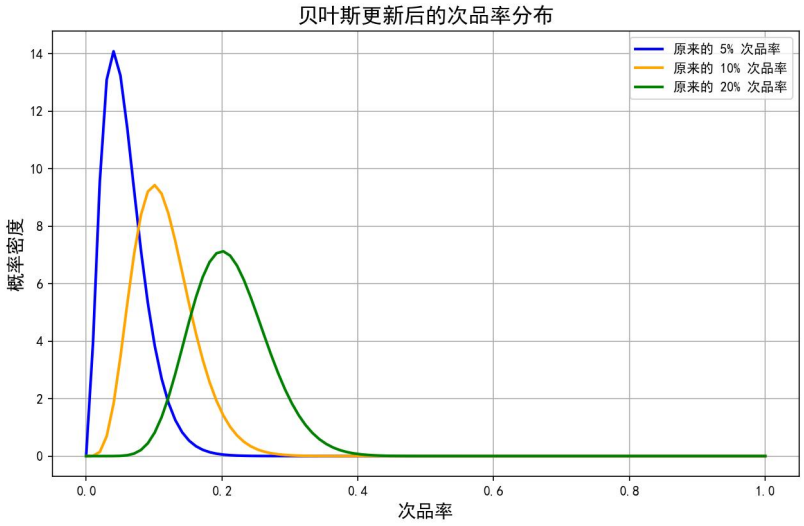

图 9 贝叶斯更新后的次品率分布

并得到对应关系：

表 3：前后次品率对应关系

次品率（前）	次品率（后）
5%	5.77%
10%	11.54%
20%	21.15%

5.4.2 问题二和问题三的重新求解

对问题二重新求解得到：

表 4：最优决策结果

情况	是否检测零配件 1	是否检测零配件 2	是否检测成品	是否拆解成品
1	否	否	否	是
2	是	是	否	是
3	是	否	否	是
4	是	是	否	是
5	否	是	否	是
6	是	否	否	否

对问题二重新求解得到最佳决策结果为（1101101011111101），在这个组合下，收益为 41.2。

六、 模型的分析与检验

6.1 模型的合理性分析

假设的合理性：模型假设企业的生产流程符合多阶段、动态决策的特点，且产品的次品率通过抽样检测得到。这一假设在制造业中广泛存在，特别是在电子产品等高复杂度生产流程中，企业通常会通过抽样检测评估零配件和成品的质量。假设的次品率数据能够准确反映整体产品质量水平，为后续的决策提供了基础。

决策逻辑的严谨性：模型根据零配件和成品的次品率、检测成本、市场售价、装配和拆解费用等因素，层层递推，进行最优决策。这种递推逻辑符合动态规划和遗传算法的基本原理，能够有效模拟企业的实际生产过程。同时，蒙特卡洛模拟用于评估不同决策组合的成本和收益，通过大量随机抽样保证了结果的统计稳定性和可解释性。

模型适用性广泛：该模型不仅可以应用于简单的生产过程，还能够扩展至更复杂的多阶段生产流程，尤其适合多零配件、多工序的制造行业。通过动态调整每个环节的检测与拆解策略，模型能够帮助企业在不同的生产情境下实现收益最大化。

6.2 模型的鲁棒性检验

参数敏感性分析：为了检验模型的鲁棒性，我们对模型的关键参数（如次品率、检测成本、市场售价等）进行了敏感性分析。结果表明，模型对这些参数具有较高的适应性，当次品率波动在一定范围内时，模型仍能给出稳定的最优决策；而检测成本和拆解费用的变化则会直接影响最终的收益，尤其是在高次品率和高检测费用的情况下，合理的决策更加重要。

多次模拟验证：为了进一步验证模型的稳定性，我们进行了多次蒙特卡洛模拟，每次随机生成不同的生产环境和参数设置，观察各决策组合下的收益分布。结果显示，模型在不同的初始条件下能够找到接近最优的解，并且决策组合的选择与各参数的变化具有一致的逻辑性，表明模型具有良好的鲁棒性。

极端情境下的模型表现：我们还对模型在极端情境下的表现进行了测试。例如，当次品率接近 100%时，模型能够通过快速终止检测和选择拆解策略，避免产生更多的损失；当市场售价大幅降低时，模型倾向于减少检测和拆解，以降低成本。在这些极端情境下，模型依然能够作出符合实际的合理决策，进一步验证了其鲁棒性。

6.3 模型的误差分析

抽样误差：在实际生产中，抽样检测可能存在一定的误差，导致样本次品率与实际次品率存在偏差。本文通过贝叶斯推断动态调整检测方案，能够在一定程度上减少抽样误差对决策的影响。然而，当抽样样本量过小时，检测结果可能不够准确，导致误判。因此，模型在实际应用时需根据实际生产情况调整样本量，确保检测结果的有效性。

模拟误差：由于蒙特卡洛模拟的随机性，模型在不同情境下的模拟结果可能存在误差，但通过大量随机抽样，模型能够获得具有统计学意义的结果。为进一步降低模拟误差，建议在实际应用中增加模拟次数，确保结果的稳定性。

6.4 模型的实际应用效果

通过对不同情境下的生产流程进行模拟，模型能够有效减少检测和拆解成本，同时保证产品质量。具体来说，模型在以下几方面表现出色：

减少检测成本：通过贝叶斯推断和动态停止规则，模型能够有效减少不必要的抽样检测，降低了检测成本。例如，在低次品率的情境下，模型能够提前终止检测，减少企业的检测支出。

优化拆解策略：模型通过分析不合格成品的处理方式，合理地选择是否进行拆解或直接报废，避免了不必要的拆解费用，同时提高了零配件的利用率，增加了企业的收益。

收益最大化：通过动态规划和遗传算法的结合，模型能够在多种决策组合中找到最优解，使企业在保证产品质量的同时，实现收益的最大化。

综上所述，通过对模型的多维度检验，本文提出的生产决策模型具有较强的适应性和鲁棒性，能够为企业在复杂生产环境下提供有效的决策支持。

七、 模型的评价、改进与推广

7.1 模型的优点

- **适应多种生产决策情境：**本文提出的模型结合了抽样检测、贝叶斯推断、蒙特卡洛模拟、动态规划和遗传算法，能够有效应对生产过程中多个阶段的复杂决策问题，尤其在多步骤质量控制和收益优化的场景下表现出色。模型通过灵活的参数设定和动态调整，能够在不同的检测、装配和拆解策略下实现收益的最大化。

- **减少检测成本：**通过贝叶斯推断与动态停止规则相结合，模型可以在保证产品质量的前提下，动态调整检测方案，减少不必要的抽样次数，从而显著降低检测成本。

- **全局最优策略的搜索能力强：**遗传算法的引入，使得在复杂的决策空间中能够高效搜索近似最优解，特别是在状态空间较大时，能够避免陷入局部最优的情况，从而提升模型求解的准确性和效率。

- **适用范围广泛：**本模型适用于多阶段生产过程、复杂供应链管理中的质量控制及成本优化问题，具有较强的实用性和推广价值。

7.2 模型的缺点

- **对参数敏感：**模型中的一些关键参数（如次品率、检测成本等）需要通过经验或历史数据设定，这些参数的选择会直接影响最终决策的效果。在实际应用中，如果参数估计不准确，可能会导致模型效果下降。

- **计算复杂度较高：**虽然动态规划和遗传算法在求解复杂问题时具有优势，但随着问题规模的扩大，特别是在生产流程较长、检测项较多的情况下，计算复杂度显著增加，可能导致模型求解时间过长。

- **数据需求较高：**模型依赖于历史检测数据及准确的先验知识，特别是在贝叶斯推断中，需要对次品率和置信度等进行合理假设。如果数据不足或质量较差，模型的推断效果可能不理想。

7.3 模型的改进

- **参数自适应调整：**未来可引入自适应算法，使得模型能够根据实时生产和市场数据动态调整参数，如次品率、检测成本、市场需求变化等，以提升模型的适应性和鲁棒性。

- **并行计算与优化：**为解决计算复杂度问题，可以采用并行计算技术，将蒙特卡洛模拟和遗传算法的迭代过程进行并行化处理，以加速求解效率。同时，结合现代优化算法如粒子群优化、差分进化等，可以进一步改进求解的速度和准确度。

- **综合考虑市场波动**：在现有模型中，假设市场需求稳定不变。未来可以考虑市场波动对企业收益的影响，进一步扩展模型的应用场景，使其适用于更加复杂的市场环境下的决策问题。

7.4 模型的推广

- **跨行业应用**：该模型不仅适用于电子产品制造领域的质量控制与成本优化，也可以推广到其他行业，如汽车制造、家电生产等具有多阶段生产流程的领域。通过调整特定的参数和生产环节，可以帮助不同领域的企业实现质量控制与收益优化。

- **集成智能决策系统**：未来模型可与企业的智能决策系统集成，结合物联网、大数据等技术，实时监控生产流程中的质量与成本，自动进行检测策略和生产策略的优化调整，实现更高的生产智能化水平。

- **供应链管理中的应用**：该模型还可推广到供应链管理中的质量监控环节，特别是在供应商次品率波动较大的情况下，帮助企业动态调整采购和检测策略，降低供应链的运营风险。

八、 参考文献

[1] 张云雷,李轲,卢建斌.基于假设检验理论的统计分辨研究综述[J].科学技术与工程,2021,21(12):4752-4759.
[2] 李艳,毛晓峰,徐章韬.立足数学文化探索二项分布模型[J].数学通讯,2022,(20):18-21.
[3] 朱翠.细说正态分布[J].新世纪智能,2024,(Z9):37-40.
[4] [江京,许敏鹏,印二威,等.整合贝叶斯动态停止策略对SSVEP-BCIs的性能提升研究[J].仪器仪表学报,2018,39(05):65-72.DOI:10.19650/j.cnki.cjsi.J1803193.
[5] 阮渊鹏.基于蒙特卡洛模拟的复杂系统可靠性评估方法研究[D].天津大学,2013.
[6] 曹寒齐,苏鹏,刘飞.一种线性逼近的化工过程迭代动态规划算法[C]//中国自动化学会过程控制专业委员会,中国自动化学会.第35届中国过程控制会议论文集.江南大学轻工过程先进控制教育部重点实验室;,2024:1.DOI:10.26914/c.cnkihy.2024.020037.
[7] 徐翔宇.基于遗传算法的运河智能选线研究[D].重庆交通大学,2024.DOI:10.27671/d.cnki.gcjtc.2024.000205.
[8] 祝瑜.基于贝叶斯统计模型的导弹武器系统可靠性评估[J].舰船电子工程,2024,44(03):135-138.

附录

附录 1

假设检验.py

```
import numpy as np
import scipy.stats as stats

def binomial_test(sample_size, defective_items, defective_rate,
confidence_level):
    alpha = 1 - confidence_level
    result = stats.binomtest(defective_items, sample_size,
defective_rate, alternative='greater')
    if result.pvalue < alpha:
        return f"拒绝 (p-value: {result.pvalue:.5f})"
    else:
        return f"接受 (p-value: {result.pvalue:.5f})"
def calculate_min_sample_size(defective_rate, confidence_level,
error_margin):
    z_score = stats.norm.ppf(1 - (1 - confidence_level) / 2)
    sample_size = (z_score**2 * defective_rate * (1 - defective_rate))
/ (error_margin**2)
    return int(np.ceil(sample_size))
defective_rate = 0.10
confidence_level_95 = 0.95
confidence_level_90 = 0.90
error_margin = 0.05  # 假设误差率: 5%
sample_size_95 = calculate_min_sample_size(defective_rate,
confidence_level_95, error_margin)
sample_size_90 = calculate_min_sample_size(defective_rate,
confidence_level_90, error_margin)
print(f"95% 置信水平下的最小样本量: {sample_size_95}")
print(f"90% 置信水平下的最小样本量: {sample_size_90}")
```

附录 2

蒙特卡洛模拟.py

```
    import numpy as np
    import pandas as pd
```

```
    # 六种情景的参数设置
    scenarios = [
        {"P1": 0.1154, "C1": 4, "C_det1": 2, "P2": 0.1154, "C2": 18,
"C_det2": 3, "Pf": 0.1154, "C_assemble": 6, "C_detf": 3, "S": 56, "L":
6, "C_recycle": 5},
        {"P1": 0.2115, "C1": 4, "C_det1": 2, "P2": 0.2115, "C2": 18,
"C_det2": 3, "Pf": 0.2115, "C_assemble": 6, "C_detf": 3, "S": 56, "L":
6, "C_recycle": 5},
        {"P1": 0.1154, "C1": 4, "C_det1": 2, "P2": 0.1154, "C2": 18,
"C_det2": 3, "Pf": 0.1154, "C_assemble": 6, "C_detf": 3, "S": 56, "L":
30, "C_recycle": 5},
        {"P1": 0.2115, "C1": 4, "C_det1": 1, "P2": 0.2115, "C2": 18,
"C_det2": 1, "Pf": 0.2115, "C_assemble": 6, "C_detf": 2, "S": 56, "L":
30, "C_recycle": 5},
        {"P1": 0.1154, "C1": 4, "C_det1": 8, "P2": 0.2115, "C2": 18,
"C_det2": 1, "Pf": 0.1154, "C_assemble": 6, "C_detf": 2, "S": 56, "L":
10, "C_recycle": 5},
        {"P1": 0.0577, "C1": 4, "C_det1": 2, "P2": 0.0577, "C2": 18,
"C_det2": 3, "Pf": 0.0577, "C_assemble": 6, "C_detf": 3, "S": 56, "L":
10, "C_recycle": 40},
    ]

    # 决策方案变量 (1 表示检测，0 表示不检测)
    A1_options = [0, 1]  # 是否检测零配件 1
    A2_options = [0, 1]  # 是否检测零配件 2
    B_options = [0, 1]   # 是否检测成品
    C_options = [0, 1]   # 是否拆解不合格成品

    # 模拟参数
    n_simulations = 3000  # 蒙特卡洛模拟次数

    # 保存每个场景的结果
    all_scenario_results = []

    # 遍历六个情景
    for scenario_index, params in enumerate(scenarios):
        results = []

    # 蒙特卡洛模拟
     for A1 in A1_options:
         for A2 in A2_options:
for B in B_options:
for C in C_options:
total_profit = 0
```

```
for _ in range(n_simulations):
# 购买成本
purchase_cost = params["C1"] + params["C2"]
# 零配件 1 检测决定
if A1:
while True:
part1_good = np.random.rand() >= params["P1"]
purchase_cost += params["C_det1"]  # 检测零配件 1
if part1_good:
break
else:
purchase_cost += params["C1"]  # 如果零配件 1 不合格，需要重新购买
else:
part1_good = np.random.rand() >= params["P1"]  # 零配件 1 是否合格
# 零配件 2 检测决定
if A2:
while True:
part2_good = np.random.rand() >= params["P2"]
purchase_cost += params["C_det2"]  # 检测零配件 2
if part2_good:
break
else:
purchase_cost += params["C2"]  # 如果零配件 2 不合格，需要重新购买
else:
part2_good = np.random.rand() >= params["P2"]  # 零配件 2 是否合格
# 装配阶段
if part1_good and part2_good:
product_good = np.random.rand() >= params["Pf"]  # 装配成品的合格率
else:
product_good = False  # 只要有一个零配件不合格，成品就不合格
# 成品检测决定
if B:
purchase_cost += params["C_detf"]  # 成品检测成本
if not product_good:
product_market = False  # 成品不合格
# 拆解决定
if C:
while 1:
purchase_cost += params["C_recycle"]  # 拆解费用
if A1 != 1:
purchase_cost += params["C_det1"]
if A2 != 1:
purchase_cost += params["C_det2"]
if part1_good == False :
```

```
while True:
purchase_cost += params["C1"]  # 如果零配件 1 不合格，需要重新购买
part1_good = np.random.rand() >= params["P1"]
purchase_cost += params["C_det1"]  # 检测零配件 1
if part1_good:
     break
if part2_good == False:
while True:
    purchase_cost += params["C2"]  # 如果零配件 1 不合格，需要重新购买
    part2_good = np.random.rand() >= params["P2"]
    purchase_cost += params["C_det2"]  # 检测零配件 1300
    if part2_good:
        break
if part1_good and part2_good:
    product_good = np.random.rand() >= params["Pf"]
else:
    product_good = False
if product_good:
total_profit += params["S"] - purchase_cost - params["C_assemble"]
# 成品进入市场的收益
    break
    else:
    total_profit -= purchase_cost  # 不合格的成品损失
continue
else:
    product_market = True
else:
    product_market = product_good  # 不检测直接进入市场
    # 如果成品不合格
    if not product_market:
    # 拆解决定
    if C:
    while 1:
    purchase_cost += params["C_recycle"]  # 拆解费用
    if A1 != 1:
       purchase_cost += params["C_det1"]
    if A2 != 1:
       purchase_cost += params["C_det2"]
    if part1_good == False :
    while True:
       purchase_cost += params["C1"]  # 如果零配件 1 不合格，需要重新购买
       part1_good = np.random.rand() >= params["P1"]
       purchase_cost += params["C_det1"]  # 检测零配件 1
       if part1_good:
```

```
            break
        if part2_good == False:
        while True:
            purchase_cost += params["C2"]  # 如果零配件 1 不合格，需要重新购买
            part2_good = np.random.rand() >= params["P2"]
    purchase_cost += params["C_det2"]  # 检测零配件 1300
    if part2_good:
        break
    if part1_good and part2_good:
        product_good = np.random.rand() >= params["Pf"]
        else:
             product_good = False
    if product_good:
        total_profit+=params["S"]-purchase_cost-params["C_assemble"]
# 成品进入市场的收益
         break
         else:
         total_profit -= purchase_cost + params["L"]  # 不合格的成品损失
         else:
         total_profit -= purchase_cost + params["L"]  # 不拆解时的损失
         else:
        total_profit+=params["S"]-purchase_cost-params["C_assemble"]
# 成品进入市场的收益
# 计算平均收益
        avg_profit = total_profit / n_simulations
        results.append([A1, A2, B, C, avg_profit])
# 将结果保存为 DataFrame
        df_results = pd.DataFrame(results, columns=['检测零配件 1', '检测
零配件 2', '检测成品', '拆解成品', '平均收益'])
# 按照平均收益从高到低排序
         df_sorted_results = df_results.sort_values(by='平均收益',
ascending=False)
         # 保存每个情景的结果
         all_scenario_results.append(df_sorted_results.head())
# 显示每个情景的最优决策前几条
    all_scenario_results
```

附录 3

动态规划.py

```
    decisions = list(itertools.product([0, 1], repeat=17))
    def calculate_component_defect_rate(component_defect_rate,
detection_flag):
```

```
        if detection_flag == 1:
            return 0
        else:
            return component_defect_rate
    def calculate_semi_product_defect_rate(comp_defect_rates,
semi_defect_rate, detection_flag):
        prob_all_good = 1.0
        for rate in comp_defect_rates:
            prob_all_good *= (1 - rate)
        final_semi_defect_rate = 1 - prob_all_good + prob_all_good *
semi_defect_rate
        if detection_flag == 1:
            return 0 if final_semi_defect_rate == 0 else 1
        else:
            return final_semi_defect_rate
    def calculate_product_defect_rate(semi_defect_rates,
product_defect_rate, detection_flag):
        prob_all_good = 1.0
        for rate in semi_defect_rates:
            prob_all_good *= (1 - rate)
        final_product_defect_rate = 1 - prob_all_good + prob_all_good *
product_defect_rate
        if detection_flag == 1:
            return 0 if final_product_defect_rate == 0 else 1
        else:
            return final_product_defect_rate
    def calculate_semi_product_salvage_value(comp_defect_rates,
comp_purchase_costs):
        salvage_value = 0
        for rate, cost in zip(comp_defect_rates, comp_purchase_costs):
            salvage_value += (1 - rate) * cost
        return salvage_value
    def calculate_expected_profit(decision):
        comp_detection = decision[:8]
        semi_detection = decision[8:11]
        semi_disassembly = decision[11:14]
        prod_detection = decision[14]
        prod_disassembly = decision[15]
        total_component_cost = 0
        total_component_detection_cost = 0
        comp_defect_rates = []
        for i in range(8):
            total_component_cost += component_purchase_costs[i]
            if comp_detection[i]:
```

```
                total_component_detection_cost +=
component_detection_costs[i]
            comp_defect_rate =
calculate_component_defect_rate(component_defect_rates[i],
comp_detection[i])
            comp_defect_rates.append(comp_defect_rate)

        semi_defect_rates = []
        for i in range(3):
            if i < 2:
                current_comp_defect_rates = comp_defect_rates[i * 3:(i +
1) * 3]
                current_comp_purchase_costs =
component_purchase_costs[i * 3:(i + 1) * 3]
            else:
                current_comp_defect_rates = comp_defect_rates[6:8]
                current_comp_purchase_costs =
component_purchase_costs[6:8]

            semi_prob_defective =
calculate_semi_product_defect_rate(current_comp_defect_rates,
semi_product_defect_rates[i], semi_detection[i])
            semi_defect_rates.append(semi_prob_defective)

        total_semi_product_cost = 0
        total_semi_product_detection_cost = 0
        total_semi_disassembly_cost = 0
        semi_disassembly_value = 0
        for i in range(3):
            if semi_defect_rates[i] < 1:
                total_semi_product_cost +=
semi_product_assembly_costs[i]
            if semi_detection[i]:
                total_semi_product_detection_cost +=
semi_product_detection_costs[i]
            if semi_detection[i] == 1 and semi_defect_rates[i] > 0 and
semi_disassembly[i] == 1:
                total_semi_disassembly_cost +=
semi_product_disassembly_costs[i]
                current_comp_defect_rates = comp_defect_rates[i * 3:(i +
1) * 3] if i < 2 else comp_defect_rates[6:8]
                current_comp_purchase_costs =
component_purchase_costs[i * 3:(i + 1) * 3] if i < 2 else
component_purchase_costs[6:8]
```

```
                semi_disassembly_value +=
calculate_semi_product_salvage_value(current_comp_defect_rates,
current_comp_purchase_costs)

        product_prob_defective =
calculate_product_defect_rate(semi_defect_rates,
product_defect_rate, prod_detection)

        total_product_cost = 0
        if product_prob_defective < 1:
            total_product_cost += product_assembly_cost
        if prod_detection:
            total_product_cost += product_detection_cost
        total_product_disassembly_cost = 0
        product_disassembly_value = 0
        if prod_detection == 1 and product_prob_defective > 0 and
prod_disassembly == 1:
            total_product_disassembly_cost += product_disassembly_cost
            for i in range(3):
                current_comp_defect_rates = comp_defect_rates[i * 3:(i +
1) * 3] if i < 2 else comp_defect_rates[6:8]
                current_comp_purchase_costs =
component_purchase_costs[i * 3:(i + 1) * 3] if i < 2 else
component_purchase_costs[6:8]
                product_disassembly_value +=
calculate_semi_product_salvage_value(current_comp_defect_rates,
current_comp_purchase_costs)

        total_cost = (total_component_cost +
total_component_detection_cost +
                      total_semi_product_cost +
total_semi_product_detection_cost + total_semi_disassembly_cost +
                      total_product_cost +
total_product_disassembly_cost)

        total_disassembly_value = semi_disassembly_value +
product_disassembly_value

        expected_profit = (1 - product_prob_defective) *
product_market_value + total_disassembly_value - total_cost

        return expected_profit

    best_decision = None
```

```
    max_profit = float('-inf')

    for decision in decisions:
        expected_profit = calculate_expected_profit(decision)
        if expected_profit > max_profit:
            max_profit = expected_profit
            best_decision = decision

    print(f"Best decision: {best_decision}")
    print(f"Max expected profit: {max_profit}")
```

附录 4

遗传算法.py

```
    import random

    POPULATION_SIZE = 50
    GENERATIONS = 100
    MUTATION_RATE = 0.01
    CROSSOVER_RATE = 0.7

    def initialize_population(size):
        return [random.choices([0, 1], k=13) for _ in range(size)]

    def fitness_function(individual):
        return calculate_expected_profit(individual)

    def selection(population, fitness_scores):
        total_fitness = sum(fitness_scores)
        pick = random.uniform(0, total_fitness)
        current = 0
        for i, fitness in enumerate(fitness_scores):
            current += fitness
            if current > pick:
                return population[i]

    def crossover(parent1, parent2):
        if random.random() < CROSSOVER_RATE:
            point = random.randint(1, len(parent1) - 1)
            return parent1[:point] + parent2[point:], parent2[:point] +
parent1[point:]
        return parent1, parent2

    def mutate(individual):
```

```
        for i in range(len(individual)):
            if random.random() < MUTATION_RATE:
                individual[i] = 1 - individual[i]
        return individual

    def genetic_algorithm():
        population = initialize_population(POPULATION_SIZE)
        for generation in range(GENERATIONS):
            fitness_scores = [fitness_function(individual) for
individual in population]
            new_population = []
            for _ in range(POPULATION_SIZE // 2):
                parent1 = selection(population, fitness_scores)
                parent2 = selection(population, fitness_scores)
                offspring1, offspring2 = crossover(parent1, parent2)
                new_population.append(mutate(offspring1))
                new_population.append(mutate(offspring2))
            population = new_population
        fitness_scores = [fitness_function(individual) for individual in
population]
        best_individual =
population[fitness_scores.index(max(fitness_scores))]
        return best_individual, max(fitness_scores)

    best_solution_ga, max_profit_ga = genetic_algorithm()

    print(f"遗传算法最优解: {best_solution_ga}, 最大期望利润:
{max_profit_ga}")
```