

基于遍历模拟和遗传算法的生产决策模型

摘要

随着社会与工业的进步，更加高质量的产品往往可以吸引更多顾客的兴趣，但这类产品通常伴随着过长的加工合成流程，每个零件、模块的不同决策都可能造成不必要的成本。因此，本文针对多道工序加工生产过程的决策问题，根据不同情况选择递归模拟和遗传算法进行求解，使企业能够实现质量控制和成本优化之间的平衡。

针对问题一，问题一的关键在于如何通过有效的抽样方案来实现质量控制，在最小化样本量的同时，兼顾成本和零配件质量。本文利用Cochran公式和单边检验的方法求解，最终得出：当总体样本数量足够大时，在信度为95%时，最小样本数趋向于定值**98**；在信度为90%时，最小样本数趋向于定值**60**。

针对问题二，本文以100件利润最大化为目标进行策略选择。对于4个决策变量进行0或1两种取值，得到16种决策方案组合。通过遍历算法，利用动态规划思想，对各阶段进行细致模拟，找到在不同情况下使利润最大化的决策方案。若拆解则还需考虑递归策略，以计算此部分带来的额外收益。最后利用敏感性分析证实了该决策方案的可靠程度，为决策结果提供了依据。

针对问题三，问题三利用了细致分析问题二得到的普适规律，简化了问题三多道程序加工产品的决策流程，最后将决策变量控制在5个，共计32种情况。本问仍采用遍历模拟的策略进行求解，同时针对问题三新提出的半成品拆解问题，通过建立全流程决策模型进行全流程的决策把关，从整体上控制全流程的最大利润，其中最优决策方案为：检测零件、检测半成品、不检测成品、不拆解半成品、拆解成品，对应平均的单价赚取利润为**41.425**元。

针对问题四，问题四的关键则是在通过使用问题一建立的最小样本量模型对问题二和三的数据进行抽样检测，得到修正后的次品率，以此为基础进行全流程的决策。本文提出了多工序产品遗传算法决策模型，通过将问题二和三中的决策变量作为遗传因子进行迭代，同时进行交叉变异等优化算法，对遗传得到的最佳个体进行把关，最后得到遗传适应度最高的最佳个体作为最终的决策方案。

关键词：抽样检测；遍历模拟；递归策略；遗传算法；动态规划

一、问题重述

1.1 问题背景

现有某企业需要分别购买两种不同的零配件来生产某种畅销的电子产品，并将两个零配件装配成成品。其中装配规则为：只要一个零配件不合格，成品一定不合格；两个零配件均合格，但成品仍可能不合格。在处理不合格成品方面，企业可选择报废，也可选择拆解，不损坏零配件，但需要支付拆解费用。因此，企业需要在零配件的采购、成品的装配质量以及不合格成品的处理上进行权衡，以尽量降低成本和提升生产效率。在这种情况下，请建立数学模型，为企业解决下面的问题。

1.2 问题提出

问题一：供应商保证零配件次品率不超某标称值，请为企业设计最少次数的抽样检测方案以决定是否接收这批零配件，检测费由企业承担。现还需分别考虑以下两种信度要求，给出具体检测方案及结果：

在95%的置信度下，若检测到次品率超过标称值10%，则拒收零配件；

在90%的置信度下，若检测到次品率未超过标称值10%，则接收零配件。

问题二：在两种零配件和成品次品率已知的条件下，为企业生产过程的各个阶段作出决策，具体决策点包括：

(1) 零配件检测阶段：决定对零配件 1 和/或零配件 2 是否进行检测，未检测的直接装配，检测不合格则丢弃。

(2) 成品检验阶段：决定装配后是否检测成品，不检测直接上市，检测则仅合格品上市。

(3) 不合格成品的处理阶段：检测出的不合格成品是否拆解重用零配件，不拆解则丢弃，拆解后重复零配件和成品检测流程。

(4) 不合格成品的退换阶段：无条件退换不合格品，并承担调换损失，退回品重复不合格成品处理流程。

结合上述决策点，制定具体的决策方案，并针对表 1 中的情形，给出具体的决策方案、依据以及相应的指标结果。

问题三：针对一个包含 m 道工序和 n 个零配件的生产过程，在已知每个零配件、半成品和成品的次品率的情况下，需要重复问题二，制定出一个生产过程的决策方案。并结合题中所给的零配件组装情况和具体数值，重复类似问题二的分析，给出生产过程的决策方案。具体目标是优化生产流程，以最小化生产成本同时保证产品质量。决策

方案应基于次品率、成本效益分析等因素，并明确决策的依据及相应指标。

问题四：假设问题二和问题三中提到的零配件、半成品和成品的次品率是通过抽样检测方法获得的。基于这一假设，重新完成对问题二和问题三。此外，还需要考虑抽样检测的不确定性，并在决策方案中体现灵活性和适应性，以应对次品率波动可能带来的风险。重新完成的分析应确保决策方案更加符合实际情况，并能够有效控制生产成本和产品质量。

二、问题分析

2.1 问题一的分析

该题要求在特定条件下，为企业设计检测次数尽可能少的抽样检测方案，其本质是通过合理的抽样检测方案，评估供应商提供的零配件的质量，以此决定是否接收该批零配件。核心在于如何利用统计方法来判断次品率，并在特定的置信水平下做出接收或拒收的决定，同时尽量减少检测次数，以降低企业的成本。本文基于不同的信度要求来给出具体的抽样次数，模型的策略是通过抽样检测评估零配件质量。首先根据假设检验理论，分别设定两种假设。接着，参考 [1] 中的方法，利用传统的Cochran公式 [2] 计算在给定置信水平下所需的样本量。考虑到样本量的有限性，采用有限总体修正公式进行必要的调整。再根据题目要求，为95%和90%两种置信水平下设定了最小样本量的计算方式。在95%的置信水平下，若检测结果显示次品率超标，企业将拒绝接收零配件；反之，在90%置信水平下，若结果表明次品率在可接受范围内，则接收。通过这种方法，企业能够在不同情况下制定出科学的检测策略。最后，为了便于分析不同总体数量对最小样本量的影响，将在不同总体数量下的所需最小样本量的关系可视化。

2.2 问题二的分析

该题要求根据已知条件，在四个决策点上制定合理的决策方案，并在此基础上，为表1的情形给出具体的决策方案和依据以及相应的指标结果。该题的核心在于为企业生产过程的各个阶段制定最优的决策方案，以最小化次品率带来的成本，包括检测成本、拆解成本等。这涉及到对零配件和成品的质量检测策略、不合格品的处理策略以及用户购买后的调换策略。由于决策变量较少，本文采用遍历算法，对共计 $2^4 = 16$ 种决策组合进行细致情形模拟，并选取总利润最大的决策组合作为最优解。首先，利用题目中已知参数和四个决策变量构建利润函数，再遍历16种决策组合，计算出每种组合下的总利润。将每种组合的总利润进行比较，选择总利润最大的组合作为最优决策。

方案，即可得到成本最小化的决策方案。最后将最优决策方案及其对应的总利润输出到表格中，并给出每种情况下的具体决策。这些结果经过可视化验证，可为企业制定有效的质量控制策略提供有力的支持。

2.3 问题三的分析

在问题三中，考虑了一个涉及多个工序和零配件的生产过程。具体而言，该问题关注的是如何在一个包含 m 道工序和 n 个零配件的环境中，根据已知的次品率、购买和装配成本等数据，制定合理的生产决策方案。本题要求重复问题二，给出生产过程的决策方案。并针对2道工序、8个零配件的情况，结合题目已知信息，给出具体的决策方案，以及决策的依据及相应指标。对问题二的求解结果进行分析，本文将发现规律应用到本问题中，缩小决策变量的选择范围，简化算法流程，在第二问的基础上，仍然采用遍历算法，对共计 $2^5 = 32$ 种决策组合进行模拟，并选取总利润最大的决策组合作为最优解。最后利用图表进行可视化，比较不同策略的优劣，并将总利润最大的情况作为最终选择的策略，为企业提供更加科学的生产决策方案。

2.4 问题四的分析

在问题四中，题干假设零配件、半成品和成品的次品率是通过抽样检测方法获得的。基于这一假设，重新完成对问题二和问题三的分析。此外，还需要考虑抽样检测的不确定性，并在决策方案中体现灵活性和适应性，以应对次品率波动可能带来的风险。由于不确定性增加，对次品率的修正等过程变得更加复杂，单纯采用遍历模拟每种情况的做法不太现实，不仅计算复杂度增加，而且无法根据某种指标以保证模型的准确性与合理性。因此，本题采用遗传算法这种启发式算法来对所有可能的决策情况进行随机选择并不断迭代更新，从而不断得到逐渐逼近最优解的结果。采用遗传算法来解决能够有效提高模型的求解效率，同时能够适应多种带不确定性的情况，相比于遍历算法，遗传算法更加灵活，具有更高的普适性。

三、模型假设

考虑到现实中的工业生产流程，以及模型构建的合理性，本文做出以下假设：

1. 假设所有零配件、半成品和成品的检测过程相互独立，检测结果不会受到其他零配件或工序的影响。
2. 假设检测设备和方法能够准确判定零配件、半成品和成品的合格与否，不存在误判或漏判的情况。

3. 假设零配件1和零配件2的次品率是相互独立的，即一个零配件的次品率不受另一个零配件次品率的影响。
4. 假设零配件1和零配件2的数量足以满足成品生产的需求，即不存在因零配件短缺而导致的生产中断。
5. 假设市场需求稳定，即成品销量不受次品率或质量检测策略的影响。这允许本文将重点放在质量控制和成本优化上，而不是市场需求预测。
6. 假设拆解不合格成品并将其重新装配成合格成品的效率与初次装配相同。在实际情况中，拆解和重装可能需要更多的时间和资源。

四、符号说明

符号	表示含义	单位
p	总体数量	个
p_0	次品率	%
$conf_level$	置信水平	/
$error_margin$	容忍误差	/
m	最小样本量	个
$profit_t$	总利润	元
$revenue_t$	总收入	元
$cost_t$	总成本	元
$profit_a$	拆解成品后带来的收益	元
opt_1	是否检测零件1	/
opt_2	是否检测零件2	/
opt_p	零件的检测策略	/
opt_m	半成品的检测策略	/
opt_f	成品的检测策略	/
opt_{rm}	半成品是否拆解	/
opt_{rf}	成品是否拆解	/
$p_{assembly}$	零件组装后的良品率	%
n_1	零件1的数量	个
n_2	零件2的数量	个
$n_{product}$	成品的数量	个

符号	表示含义	单位
$real_n$	实际成品数量	个
p_{hidden}	隐藏的次品率	%
$cost_{parts}$	零配件总检测成本	元
$cost_{ap}$	零配件总装配成本	元
$cost_{product}$	成品总检测成本	元
$cost_{rework}$	总拆解成本	元
$cost_{replace}$	调换损失	元
s	成品的售价	元/个
$earn$	单次售卖企业利润（不考虑拆解后再组装售卖的 那部分利润）	元
n_{mi}	理想状态下无次品的半成品数量	个
n_{fi}	理想状态下无次品的成品数量	个
p_{part}	零件的次品率	%
p_{mid}	半成品的次品率	%
n_{mid}	半成品数量	个
$cost_{c1}$	零件1的检测成本	元
$cost_{c2}$	零件2的检测成本	元
$cost_{c3}$	零件3的检测成本	元
$cost_{assembly}$	装配成本	元
$cost_{mc}$	半成品检测成本	元
$cost_c$	成品检测成本	元
$cost_{mrw}$	半成品拆解费用	元
$cost_{frw}$	成品拆解费用	元
n_{mp}	前次企业生产的半成品数量	个
n_{fp}	前次企业生产的成品数量	个
$cost_{amc}$	半成品总检测成本	元
p_0	次品率	%
$price$	零配件购买单价	元/个
$inspect$	是否检测零件（0为不检测， 1为检测）	/
$cost_{inspect}$	检测费用	元
$cost_f$	单个成品的成本	/
$loss_{unp}$	未检测零件的损失	元

符号	表示含义	单位
$cost_{assembly}$	装配成本	元
$cost_{amc}$	半成品总检测成本	元
$cost_{replace}$	调换或退货损失	元
$cost_{rework}$	拆解费用	元
C_i	每种情况下的总成本	元
R	拆解半成品后的总收入	元
$cost_{pt}$	零配件总成本	元
$revenue_{rmt}$	拆解半成品后的总收益	元
$cost_{rft}$	成品拆解后的总成本	元
$size_p$	种群数量	/
num_{gen}	迭代次数	/
$rate_{mutation}$	变异率	/
$size_{tournament}$	锦标赛大小	/
$size_{elite}$	精英个体数量	/

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 模型的建立

本题可视为假设检验与抽样方案设计的问题，问题的核心在于如何通过抽样检测来评估零配件的质量以决定是否从供应商接收一批零配件。这涉及到统计学的假设检验理论，需运用概率论与数理统计知识。[8]中提供了抽样检验的相关理论，[9]中提供了假设的建立方法，对本文具有指导性意义。本文将根据题目条件，在两种特定的置信水平下，设计出合理的抽样检测方案，以判断供应商提供的零配件次品率是否满足要求。为使设计检测次数尽可能少，本文采用Cochran公式和单边检验的方法来计算最小样本量，[2]中对Cochran公式进行了详细的介绍，包括由来以及具体的统计分析过程。同时，本文在此基础上加入修正值，以考虑总样本量过少的情况。

1) 针对95%信度的情形：

为了计算出给定总体数量下的最小样本量，使得在95%置信水平下能够判断是否拒收。首先，计算出标准正态分布中对应于给定置信水平的Z值。由于问题中未明确

提及检验为双侧还是单侧，但通常在质量控制中，关心较多的是次品率是否超过了某个标准，因此本文使用单侧检验。其次，本文参阅 [6] 中的公式推导，应用 Cochran 公式计算出初步的样本量。为更加准确地反映出从有限总体中抽样的情况，在总体数量有限的情形下，采用有限总体修正公式来调整样本量。最后将最小样本量取整以确保满足统计推断的需求。这些样本量能够帮助企业在不同情况下选择合适的检测次数，使得决策更加可靠。计算最小样本量的式子如下：

$$m = \frac{Z^2 \cdot p_0 \cdot (1 - p_0)}{E^2} \quad (1)$$

其中 p_0 为次品率的标称值，默认为 10%， E 为容忍误差，默认设置为 5%。

约束条件如下：

$$\text{conf_level} = 0.95 \quad (2)$$

其中 conf_level 为置信度。

由于总体零配件数量未知，因此需要考虑小样本的情况，对 Cochran 公式进行修正。修正公式如下：

$$m = \frac{m}{1 + \frac{m-1}{p}} \quad (3)$$

考虑到在不同总体数量下所需最小样本量的关系，且修正公式对总体数量少的情况影响较大，因此本文将总体数量分为大、小样本两种情形来分布探索最小样本量的变化，并在不同总体数量下，对所需最小样本量的关系进行可视化处理。

2) 针对90%信度的情形：

模型的建立过程和上述的情形基本一致，只需将置信度改为 90%。在这种情况下，重复上述步骤，同样将总体数量分为大、小样本两种情形来分布探索最小样本量的变化，并在不同总体数量下对所需最小样本量的关系进行可视化。

5.1.2 模型的求解

根据模型的建立过程，求解步骤如下：

step1：相关变量值的确定

使用正态分布的分位数函数，根据置信水平计算出单侧检验的 Z 值，为后面的 Cochran 公式计算做准备。

由于企业声称一批零件的标称值 $\leq 10\%$ ，因此在对概率 p 进行选择时，考虑到标称值的具体值并加以保守估计，本文将其设置为 10%，即 $p_0 = 0.1$ 。此时能够保证

在95%的置信水平下，最小样本量选择的合理性。

step2: Cochran 公式计算样本量

利用Cochran 公式估计在二项分布下给定置信水平和容忍误差时所需的最小样本量。[5]中指出Cochran方法的普适性，通过使用 Cochran 公式1，企业可以有效地计算出在不同的置信水平情况下所需的样本量，从而最大限度降低采购风险。在实施抽样检测后，企业可依据检测结果决定是否接收零配件，确保产品质量符合标准。

step3：有限总体修正

如果总体数量有限，则需要对样本量进行修正，以考虑从有限总体中抽样时的样本代表性。如果总体数量非常大或被认为是无限的，则不进行修正。

step4：返回取整结果

为确保样本量足够大，对最小样本量向上取整，保证样本量不会小于所需的最小值，避免可能的统计分析问题。

step5：关系可视化

为了清晰地看到在不同总体数量下所需的最小样本量是如何变化的，最终对在不同总体数量下的所需最小样本量的关系可视化。

当信度为95%时，对总体数量和样本数进行整合，分别针对小样本和大样本计算出最小样本量，并绘制了它们随总体数量变化的图表，如图 1所示。

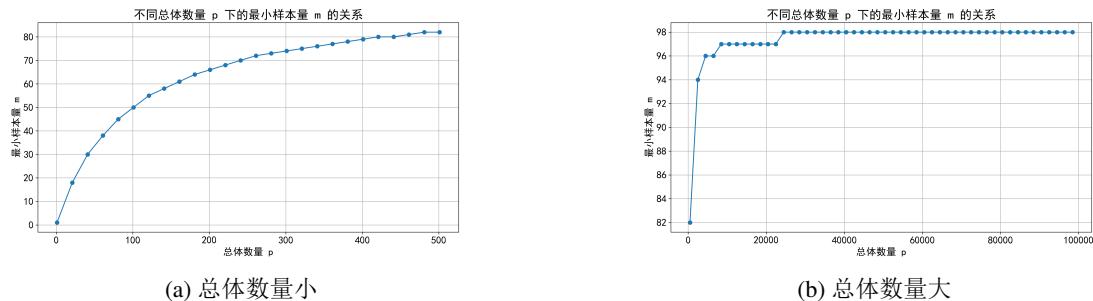


图 1: 最小样本量随总体数量变化图

从1a中可知，当总体数量较小时，随着总体数量的增加，所需的最小样本量也会相应增长，在总体数量比较小时，增长比较明显。而随着总体数量的增加，增长速率逐渐减小，最终最小样本量趋于一个相对较低的水平。从1b中可知，当总体样本数量足够大时，最小样本数趋向于定值 98。此时该结果即为直接使用 Cochran 公式计算出的最小样本量。

当信度为90%时，对总体数量和样本数进行整合，分别针对小样本和大样本计算出最小样本量，并绘制了它们随总体数量变化的图表，如图 2所示。

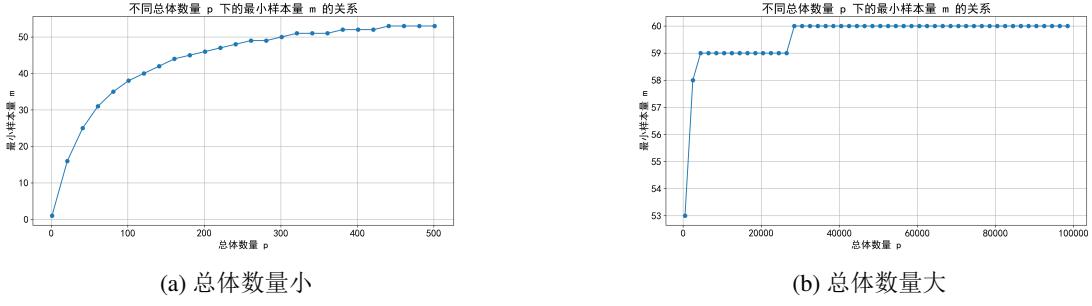


图 2: 最小样本量随总体数量变化图

从2a中可知，其最小样本量的增长趋势大致于95%置信度下的情况相同。从2b中可见，当总体样本数量足够大时，最小样本数趋向于定值 60，即为直接使用 Cochran 公式计算出的最小样本量值。

step6：结果分析

根据上述计算以及图表展示可知，在两种不同的信度下，总体数量较大时所需的小样本量均相对稳定，不会随着总体数量的增加而显著变化。这也说明在总体数量较大时，企业可以通过抽样检测来评估零配件的质量，以决定是否接收零配件，从而降低成本，提高效率。而如果考虑到总体数量有限的情况，需要对样本量进行修正，由图1a和图2a可知，修正效果符合实际情况，可确保样本量的有效性。

5.2 问题二模型的建立与求解

5.2.1 模型的建立

针对问题二的决策问题，结合 [10]中的最大利润法来取得最优解，本文以利润最大化为目标进行策略选择。由于题目已给出四个决策阶段分别对应的决策要点，经过排列组合计算，可以得到16种可能的决策组合。因此，通过遍历每种情况和每种决策组合，便可获得每种情况中利润最大化的最佳决策方案。本文将对四个决策阶段的策略选择进行建模，即是否检测两种零配件，是否检测成品，以及是否拆解不合格成品，模型的核心在于找到最佳的检测与处理的决策组合，使得利润最大化，为企业带来最大的收益。

1) 目标函数的确定

模型的目的是为企业制定利润最大的决策方案，目标函数可以表示为：

$$\max profit_t = revenue_t - cost_t + profit_a \quad (4)$$

其中 $profit_t$ 为总利润, $revenue_t$ 为总收入, 即对所有不含次品的成品进行销售所获

得的金额， $cost_t$ 为总成本， $profit_a$ 为拆解成品后，重复步骤1和步骤2等过程所带来的收益。

2) 决策变量及约束条件

决策变量为：

opt_1, opt_2 : 是否检测零配件1和零配件2;

opt_f : 是否检测成品；

opt_{rf} : 是否对不合格成品进行返工。

以上各个决策变量的取值为 0 或 1，表示是否进行相应的操作。

约束条件为：

$opt_1, opt_2, opt_f, opt_{rf} \in \{0, 1\}$ 。

所有的决策都要考虑企业的成本支出，不能超过企业的资金预算。

3) 变量求解

由于遍历算法需要细致考虑到不同阶段下，不同决策所对应的情况，因此需考虑的条件较多，计算公式较为繁杂，各项变量的具体计算表达式如下：

成品的次品率由零件的次品率决定，通过计算 opt_1 和 opt_2 策略下次品率的最大值，来确定组装成品的良品率：

$$p_{assembly} = 1 - \max(opt_1 \cdot p_1, opt_2 \cdot p_2) \quad (5)$$

根据木桶效应，当两个零件一一配对时，最终成品数量取决于次品率较高的零件，即取最大的次品率。

根据良品率 $p_{assembly}$ 计算出的成品数量为：

$$n_{product} = n_1 \cdot p_{assembly} \quad (6)$$

理想状态下，完全没有次品的成品个数分为两种情况：若 opt_1 和 opt_2 均为 1，则选择最大的次品率；否则，使用各零件的次品率相乘：

$$n_{fi} = \begin{cases} n_1 \cdot (1 - \max(p_1, p_2)) \cdot (1 - p_3) & \text{if } opt_1 = 1 \text{ and } opt_2 = 1 \\ n_1 \cdot (1 - p_1) \cdot (1 - p_2) \cdot (1 - p_3) & \text{otherwise} \end{cases} \quad (7)$$

根据是否进行成品检测 (opt_f) 来决定实际投入市场的成品数量:

$$real_n = \begin{cases} n_{fi} & \text{if } opt_f = 1 \\ n_{product} & \text{otherwise} \end{cases} \quad (8)$$

如果不检测成品 ($opt_f = 0$), 则实际生产出的成品可能有隐藏的次品, 计算隐藏的次品率如下:

$$p_{hidden} = 1 - \frac{n_{fi}}{real_n} \quad (9)$$

零配件总检测成本 $cost_{parts}$:

$$cost_{parts} = opt_1 \cdot cost_{c1} \cdot n_1 + opt_2 \cdot cost_{c2} \cdot n_2 \quad (10)$$

其中 opt_1 和 opt_2 分别为零件1和零件2是否检测的决策变量(1表示检测, 0表示不检测), $cost_{c1}$ 和 $cost_{c2}$ 分别为零件1和零件2的检测成本, n_1 和 n_2 分别为零件1和零件2的数量。

零配件总装配成本 $cost_{ap}$:

$$cost_{ap} = cost_{assembly} \cdot n_f \quad (11)$$

其中 $cost_{assembly}$ 为装配成本, n_f 为成品数量。

成品总检测成本 $cost_{product}$:

$$cost_{product} = opt_f \cdot cost_c \cdot n_f \quad (12)$$

其中 opt_f 为成品是否检测的决策变量, $cost_c$ 为成品检测成本, n_f 为成品数量。

总拆解成本 $cost_{rework}$:

拆解成本是指将不合格的成品拆解成零配件, 以便重新装配或进行其他处理所需支付的费用:

$$cost_{rework} = opt_{rework} \cdot cost_{rework} \cdot p_f \cdot n_f \quad (13)$$

其中 opt_{rework} 为是否进行对零配件返工的决策变量 (1表示拆解, 0表示不拆解), $cost_{rework}$ 为单个不合格成品的拆解费用, n_f 为成品数量。

调换损失 $cost_{replace}$:

调换损失是指在生产过程中, 当发现成品不合格时, 如果选择不进行返工, 即不

解构并重新装配，而是直接将这些次品进行调换，这一过程中所产生的成本：

$$cost_{replace} = (1 - opt_{rework}) \cdot cost_{replace} \cdot p_f \cdot n_f \quad (14)$$

其中 $1 - opt_{rework}$ 为对零配件返工的决策变量，确保只有在不选择返工 ($opt_{rework} = 0$) 的情况下，才会计算次品成品的调换损失， $cost_{replace}$ 为成品调换损失，包括物流成本和企业信誉损失等， n_f 为成品数量。

总成本是以上几项成本的总和：

$$cost_t = cost_{parts} + cost_{assembly} + cost_{product} + cost_{rework} + cost_{replace} \quad (15)$$

其中 $cost_{parts}$ 为零配件总检测成本， $cost_{assembly}$ 为零配件总装配成本， $cost_{product}$ 为成品总检测成本， $cost_{rework}$ 为总拆解成本， $cost_{replace}$ 为调换损失。

总销售额 $revenue_t$ ：

$$revenue_t = n_{fi} \cdot s \quad (16)$$

其中 n_{fi} 为理想成品数量，表示装配后，实际生成不包含次品的成品数量， s 为每个成品的售价。当进行销售时，只有不含次品的成品才能被售出，而实际为次品的成品会被客户退回，或者被企业检测出来进行处理，不会流入市场。

最后，企业的利润等于总收入减去总成本，再加上成品销售的利润：

$$earn = n_{fi} \cdot (s - s_1 - s_2) - cost_t \quad (17)$$

3) 递归过程

由题干可知，当不合格品退回时，需根据决策选择是否拆解。若进行拆解，则需要对拆解后的零配件重复进行步骤1和步骤2的检测，再次计算利润。因此，本文采用递归的方法，对拆解后的零配件再次进行检测，计算出这部分递归调用函数带来的额外利润，最终汇总统计得到总利润。

a. 递归终止条件

考虑到用递归解决问题时，首先需要确定终止条件，以免进入无限递归，导致程序崩溃。当企业未拆卸成品，全部成品售出无法带来收益时，再次将次品进行拆解递归必不可能带来利润。当拆卸后得到的零件数量和上一次相同时，说明上一次组装的零件无法卖出导致这一次得到的零件数量没有减少，说明次品率过高，重构后卖不出去，因此没有再次拆解的必要。所以在本题中，递归终止条件为：当本轮利润 < 0 ，或者零件数相较上次未减少时，递归结束。

b. 次品率修正

在递归过程中，次品率会随着拆解次品的增多而增加，因此需要先对次品率进行修正，得到新的次品率后再进行递归调用，以反映拆解次品后的实际情况。具体次品率修正公式如下：

$$p_{1,new} = \begin{cases} \frac{p_1}{1-(1-p_1)(1-p_2)(1-p_3)}, & \text{if } opt_1 = 0 \text{ and } opt_2 = 0 \\ 0, & \text{if } opt_1 = 1 \text{ and } opt_2 = 0 \\ \frac{p_1}{1-(1-p_1)(1-p_3)}, & \text{if } opt_1 = 0 \text{ and } opt_2 = 1 \\ 0, & \text{if } opt_1 = 1 \text{ and } opt_2 = 1 \end{cases} \quad (18)$$

$$p_{2,new} = \begin{cases} \frac{p_2}{1-(1-p_1)(1-p_2)(1-p_3)}, & \text{if } opt_1 = 0 \text{ and } opt_2 = 0 \\ \frac{p_2}{1-(1-p_2)(1-p_3)}, & \text{if } opt_1 = 1 \text{ and } opt_2 = 0 \\ 0, & \text{if } opt_1 = 0 \text{ and } opt_2 = 1 \\ 0, & \text{if } opt_1 = 1 \text{ and } opt_2 = 1 \end{cases} \quad (19)$$

$$p_{3,new} = \begin{cases} \frac{p_3}{1-(1-p_1)(1-p_2)(1-p_3)}, & \text{if } opt_1 = 0 \text{ and } opt_2 = 0 \\ \frac{p_3}{1-(1-p_2)(1-p_3)}, & \text{if } opt_1 = 1 \text{ and } opt_2 = 0 \\ \frac{p_3}{1-(1-p_1)(1-p_3)}, & \text{if } opt_1 = 0 \text{ and } opt_2 = 1 \\ 1, & \text{if } opt_1 = 1 \text{ and } opt_2 = 1 \end{cases} \quad (20)$$

5.2.2 模型的求解

在模型中，目的是通过不同的检测和返工策略来获取最大总利润以最小化成本。再利用给定的零件次品率、成本及决策参数，即可找到在不同决策组合下的最优解决方案。考虑 [7] 中对决策变量的分析，本文对于 4 个决策变量，令每个变量有 0 或 1 两种取值，因此总共有 $2^4 = 16$ 种组合。通过遍历所有决策方案组合，即可找到在不同生产情形下使得企业总利润最大化的最优解。具体求解步骤如下：

step1：初始化

首先定义一些参数包括决策变量、成本、零件的次品率等来确定成本函数，总成本等于装配成本、成品检测成本、拆解费用、成品的调换损失的总和。

step2：遍历所有决策组合

通过构建好的利润函数，针对题干表 1 中的不同情形，遍历所有可能决策组合，这些组合包括四个二进制选项（0 或 1），分别对应于是否检测零件 1、零件 2、成品和是否进行返工。例如，(0, 1, 0, 1) 表示不检测零件 1，检测零件 2，不检测成品，并进行返工。

为使结果更加结构化，将6种情况、每种共16个决策组合和计算出的利润附加到 results 列表中，便于获取每种情况的最佳决策（见附录1）。随着决策组合的遍历，模型可以全面评估在不同情况下实施的每个决策的经济效益，从而为优化决策提供依据。

step3：结果分析

遍历计算得到结果后，为了便于分析最佳决策方案中的6种情况，将不同策略下的利润变化和各种情况下的最佳决策利润进行可视化绘制，如图3、图4所示：

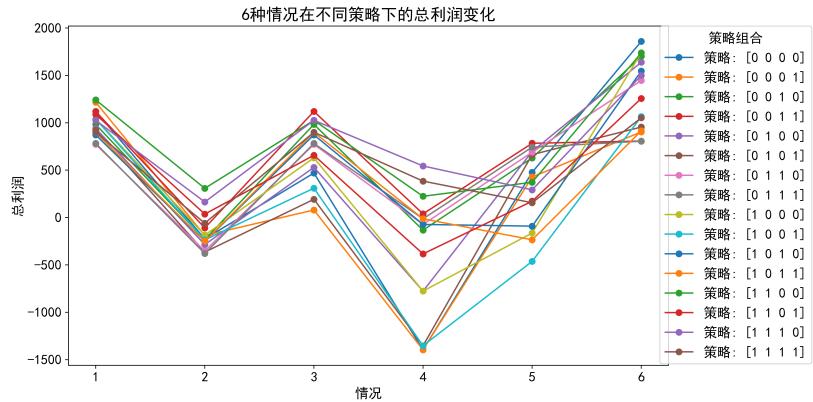


图 3: 6种情况在不同策略下的利润变化图

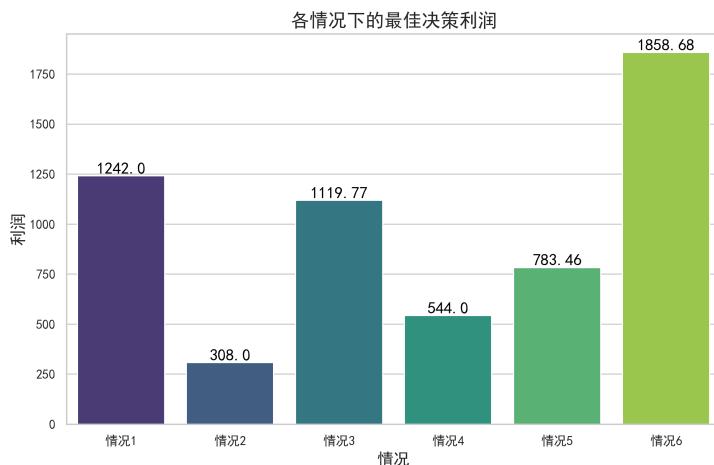


图 4: 各种情况下的最佳决策利润图

由上两图可知，不同的决策组合在不同情况下呈现的趋势大致相同。结合题干中表1各情况的数据进行分析可知，其中第二种由于次品率均较高，检测成本也较高，因此最佳决策利润相比其他5种情况明显较低。第六种情况由于其次品率均较低，其最佳决策对应产生的利润最大。而第四种情况不仅次品率较高，调换损失的费用也明显高于其他情况，因此若选择调换策略，企业所要付出的成本明显高于其他情况，所以在图表中出现较大的负值，可见其亏损较大。通过可视化不同策略下的利润变化图，进一步验证了模型的可靠性，并直观地展示了不同策略对利润的影响，为企业在实际生

产中制定有效的质量控制策略提供了有力的支持。

由于我们需要对6种情况下的最优策略进行单独分析，因此将这部分数据整理出来，得到最佳决策方案如下表2所示：

表 2: 每个情况的最佳方案

情况	零件1检测	零件2检测	成品检测	不合格成品拆解	利润
情况1	检测	检测	不检测	不拆解	1242.00
情况2	检测	检测	不检测	不拆解	308.00
情况3	不检测	不检测	检测	拆解	1119.77
情况4	检测	检测	不检测	不拆解	544.00
情况5	不检测	不检测	检测	拆解	783.46
情况6	不检测	不检测	不检测	不拆解	1858.68

由表2可知，对于不同情况，最佳决策方案在利润上具有显著差异。多数情况零件1和零件2的检测策略相同，这是因为木桶效应，在单独检测只能保证一个零件是否为次品，但是与混有次品的另一个零件进行组装时，仍会在配对时选择到次品，经过装配后次品率进一步提升，而且还浪费成本。在情况6中，由于零件和成品的次品率都较低，因此最优策略均不检测，且拆解费用比其他的任何一种情况都高出8倍，拆解成本过大，因此最优策略也是选择不拆解。通过观察表2的数据，有力验证了模型求解的有效性。对于不同的决策情况，企业应根据实际情况选择合适的检测和返工策略，以最大化利润。

step4：敏感性分析

为了验证决策结果的可靠程度，本文通过计算并收集了在不同零配件（零配件1和零配件2）的不同次品率（5%, 10%, 15%, 20%）、不同（检测成本（0, 2, 5, 10元/件）、成品次品率（5%, 10%, 15%, 20%）、成品检测成本（0, 2, 5, 10元/件）和调换损失（0, 5, 10, 15, 20元/件）等条件下的最佳决策方案及相关信息。并对齐一一进行敏感性分析，做出相应的图表（文件材料路径见附录1）。此处以对调换损失的调整为例，经过整理得到的结果如下图5所示：

从图5中可以看出，当调换损失逐渐增大时，最优决策折线均会相应降低，这是因为调换损失的增加会导致企业的成本增加，从而降低企业的利润。

再对零件1的次品率进行参数调整，得到的结果如图6所示：

从图6中可以看出，当零件1的次品率逐渐增大时，最优决策折线均会相应降低，这是因为零件1的次品率增大会导致企业的成本增加，从而降低企业的利润。

附件中关于敏感性分析的图片也呈现出类似的效果，经过分析总结可得，不同的参数组合对最佳决策方案的影响是显著的。在不同的次品率、检测成本、调换损失等条件下，最佳决策方案会有所不同。例如，当零配件的次品率较低时，不拆解成品的利

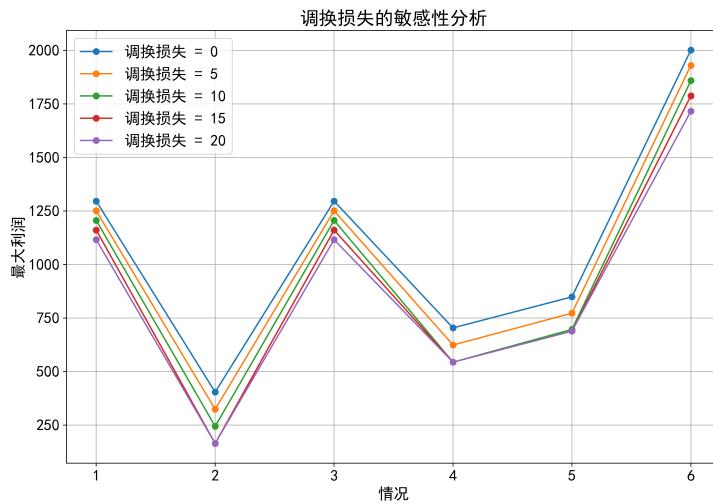


图 5: 调换损失参数调整结果

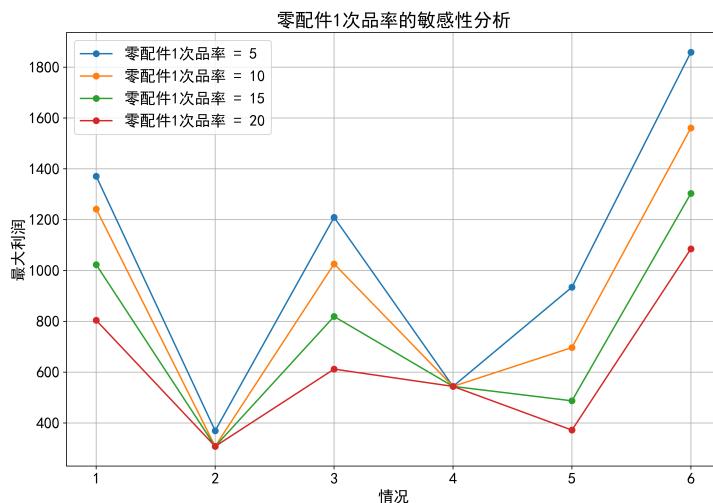


图 6: 零配件1次品率参数调整结果

润最大，而当零配件的次品率较高时，检测成本较低的情况下，对两种零件都进行检测所获的利润最大。因此，企业在实际生产中应根据不同的生产情况和参数条件，选择合适的检测和返工策略，以最大化利润。

5.3 问题三模型的建立与求解

5.3.1 模型的建立

1) 模型设定

由第二问的解法分析可知，无论是否拆解不合格成品，其最优策略中的零件1和零件2的决策方案大多数情况均相同，唯一例外的是情况5，此时是因为零件1的次品率明

显低于零件2，且检测成本还比零件2高出8倍，因此该种较为极端的情况选择不检测零件1，检测零件2。此外，在本题中，零件的次品率均相同，且检测成本差距不大。因此，在本题为了简化模型，我们将3种零件的决策合并为一种，即只考虑是否检测所有零件，不再细分单独检测，且这种策略同样适用于半成品的检测策略。最终汇总后，确定了5个阶段的决策，即零件是否检测，半成品是否检测，成品是否检测，半成品是否拆解，成品是否拆解，经排列组合后共有32种决策。通过遍历每种情况和每种决策组合，结合第二问的求解思路进行模拟计算，便可获得每种情况中利润最大化的最佳决策方案。

2) 模拟各阶段流程

本题的流程于第二问类似，但中间多了道半成品的工序，相应地多出半成品的检测、装配、拆解的流程情况。从整体上看，对各阶段进行细致梳理，可得到以下流程图：

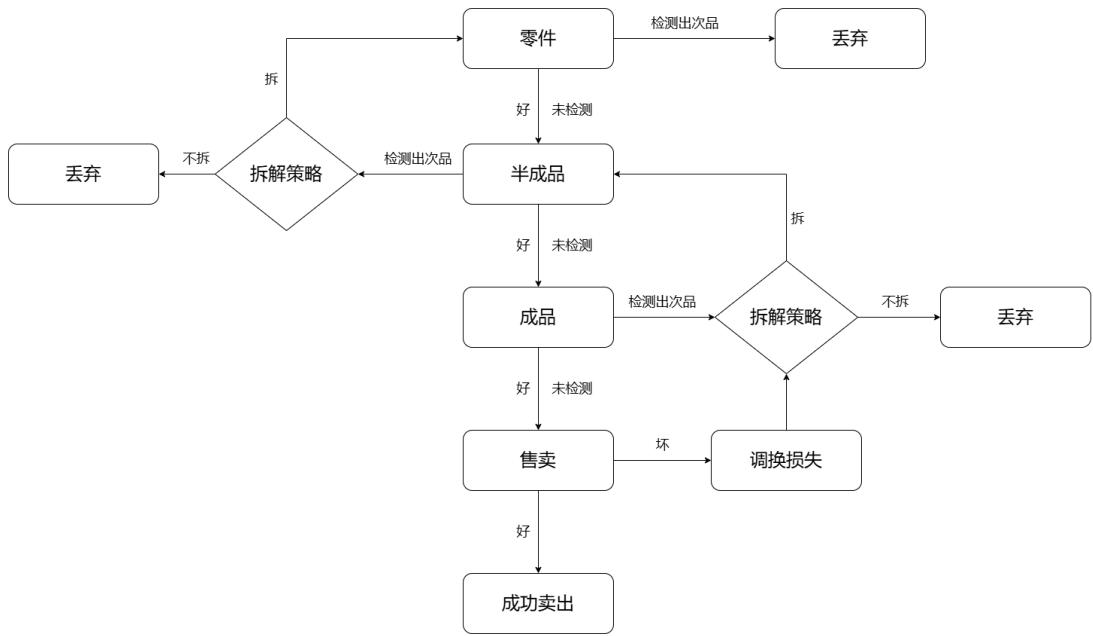


图 7: 问题三流程图

3) 阶段过程更新

由于增加了半成品的工序，需要对第二问的模型进行相应的更新。具体变更的公式如下：

五个决策变量的定义为： $opt_p, opt_m, opt_f, opt_{rm}, opt_{rf} \in \{0, 1\}$ ，其中， opt_p 、 opt_m 、 opt_f 分别为零件、半成品、成品的检测策略， opt_{rm} 、 opt_{rf} 分别为半成品是否拆解和成品是否拆解的策略。

理想状态下，完全没有次品的半成品数量 n_{mi} 依据是否对零件进行检测决定，公式表达如下：

$$n_{mi} = \begin{cases} n \cdot (1 - p_{part}) \cdot 0.9 & \text{if } opt_p = 1 \text{ (零件都检测)} \\ n \cdot (1 - p_{part})^3 \cdot 0.9 & \text{otherwise} \end{cases} \quad (21)$$

由于装配过程的次品率均为10%，因此良品率固定为90%，即0.9。同理，完全没有次品的成品数量 n_{fi} 依据是否对零件进行检测决定，公式表达如下：

$$n_{fi} = \begin{cases} n_{mid} \cdot (1 - p_{mid}) \cdot 0.9 & \text{if } opt_m = 1 \text{ (半成品都检测)} \\ n_{mid} \cdot (1 - p_{mid})^3 \cdot 0.9 & \text{otherwise} \end{cases} \quad (22)$$

在本小问的递归迭代策略中，由于半成品和成品各自都有拆解策略，因此会有多种递归的可能：半成品拆为零件进行递归，成品拆为半成品，由半成品开始进行递归，或者是成品不断拆解为零件，从零件开始进行递归。由于次品率计算较为复杂，本题在此处进行了简化，修正后的零件次品率以及半成品次品率的计算公式如式23、24所示：

$$p_{part} = \frac{p_{part}}{1 - (1 - p_{part})^3} \quad \text{if } opt_{rm} = 1 \text{ and } opt_{rf} = 1 \quad (23)$$

$$p_{mid} = \frac{p_{mid}}{1 - (1 - p_{mid})^3} \quad \text{if } opt_{rm} = 0 \text{ and } opt_{rf} = 1 \quad (24)$$

其他公式形式均与第二问相同，利用新的阶段流程图应用公式计算，不断迭代模拟，即可找到在不同生产情形下使得企业总利润最大化的最优解决方案。

5.3.2 模型的求解

本问模型的求解流程与第二问类似，但在计算过程中需要考虑成品与半成品的拆解情况，选择不同的递归情形。具体求解步骤如下：

step1：初始化

首先可定义一些参数，如决策变量、成本、零件的次品率等。根据题干中提供的信息，具体参数如下：

- 零件1、零件2的检测成本： $cost_{c1} = cost_{C2} = 1$
- 零件3的检测成本： $cost_{c3} = 2$
- 装配成本： $cost_{assembly} = 8$
- 成品调换损失： $cost_{replace} = 40$

- 半成品检测成本: $cost_{mc} = 4$
- 成品检测成本: $cost_c = 6$
- 半成品拆解费用: $cost_{mrw} = 6$
- 成品拆解费用: $cost_{frw} = 10$
- 刚开始零件的次品率: $p_{part} = 0.1$
- 刚开始半成品的次品率: $p_{mid} = 0.1$
- 记录前一次企业生产的半成品数量: $n_{mp} = 100$
- 记录前一次企业生产的成品数量: $n_{fp} = 100$

step2: 算法求解过程

从整体上看, 本题求解过程与第二问相似, 通过遍历所有决策组合, 计算每种组合下的利润, 找到利润最大值对应的策略即为最优解。

step3: 递归策略

由于成品和半成品各自都有拆解策略, 因此需要定义不同递归函数来提供调用, 进行功能区分。第一个递归函数用于计算从零件到成品所能获得的利润, 第二个递归函数用于计算从半成品到成品所能获得的利润。不同策略调用不同递归函数的情形如下表 3 所示:

表 3: 不同策略下递归方法的选择

半成品拆解策略	成品拆解策略	递归方法
1	1	从零件到成品
0	1	从半成品到成品
1	0	从零件到成品
0	0	不拆解, 不递归

通过递归调用, 计算出这部分递归调用函数带来的额外利润, 最终汇总统计得到总利润。

step4: 结果分析

通过对32种策略进行遍历模拟, 得到每种策略下的利润值, 结果如图8所示:

可以看出, 当策略编号为25, 即(1 1 0 0 1)的策略最好, 此时100套零件获得的利润为4000元。

此外, 在图中的前8种策略中, 可以明显看出获得的利润均为负值。经过分析可知, 这8种策略都是不对零件和半成品进行检测的策略。然而随着生产的进行, 次品率会逐渐增加, 导致拆解的成本增加, 并且由于生成的成品次品率过高, 导致有效产品数过低, 若成品流入市场, 由极大概率需要进行调换, 需要承担更多的调换损失, 从而使得利润变为负值。因此, 对零件和半成品进行检测是非常重要的, 可以有效降低次品率, 提高企业的利润。

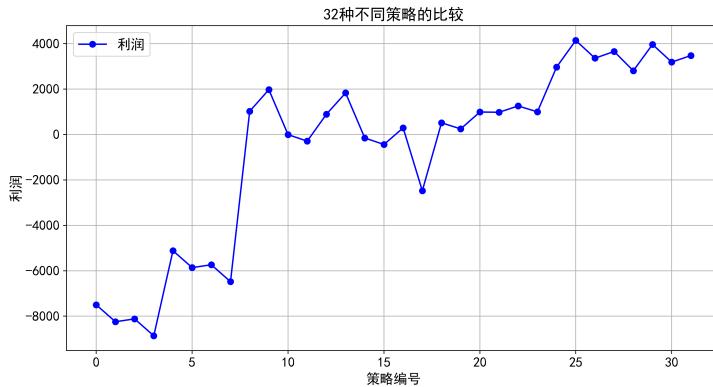


图 8: 32种策略下的利润变化图

5.4 问题四模型的建立与求解

5.4.1 模型的建立

在本问题中，由于次品率不再是已知条件，而是根据第一问的检测结果得到的，因此需要对次品率进行修正。

修正需要先根据1和3式计算出修正后的样本量，再根据式25计算出修正后的次品率：

$$rate_{adjusted} = rate_{original} + Z \cdot \sqrt{\frac{rate_{original} \cdot (1 - rate_{original})}{sample_size}} \quad (25)$$

其中，修正后的次品率为置信区间的上界。将修正后的次品率代入第二、三问的模型中，再进行后续步骤的求解。

1) 遗传算法优化求解模型

由于次品率不再是固定的值，而是根据抽样检测确定，因此带有明显误差，用传统遍历模拟递归的方式不太符合实际，需要考虑的决策变量以及不确定性因素增多，且算法复杂度过高。因此，本文采用遗传算法来求解问题四，以解决问题四中的最优决策问题。

遗传算法（Genetic Algorithm, GA）是一种基于自然选择和遗传机制的优化算法。[3]、[4]等人对遗传算法的发展历史，概念等做了详细的介绍。它模仿了生物进化过程，利用选择、交叉和变异等操作在解决复杂优化问题上具有显著效果。遗传算法最早由约翰·霍兰德（John Holland）在20世纪70年代提出，并被广泛应用于工程、科学、经济等领域。

a. 遗传算法的基本概念

- **个体与种群:** 个体代表一个可能的解, 多个个体组成一个种群, 表示一组可能的解。
- **基因与染色体:** 基因为个体的特征参数, 染色体由基因构成, 表示一个完整的解。
- **适应度函数:** 用于评估个体优劣, 适应度值越高表示个体越优。

b. 遗传算法的基本步骤

遗传算法流程如图9所示:

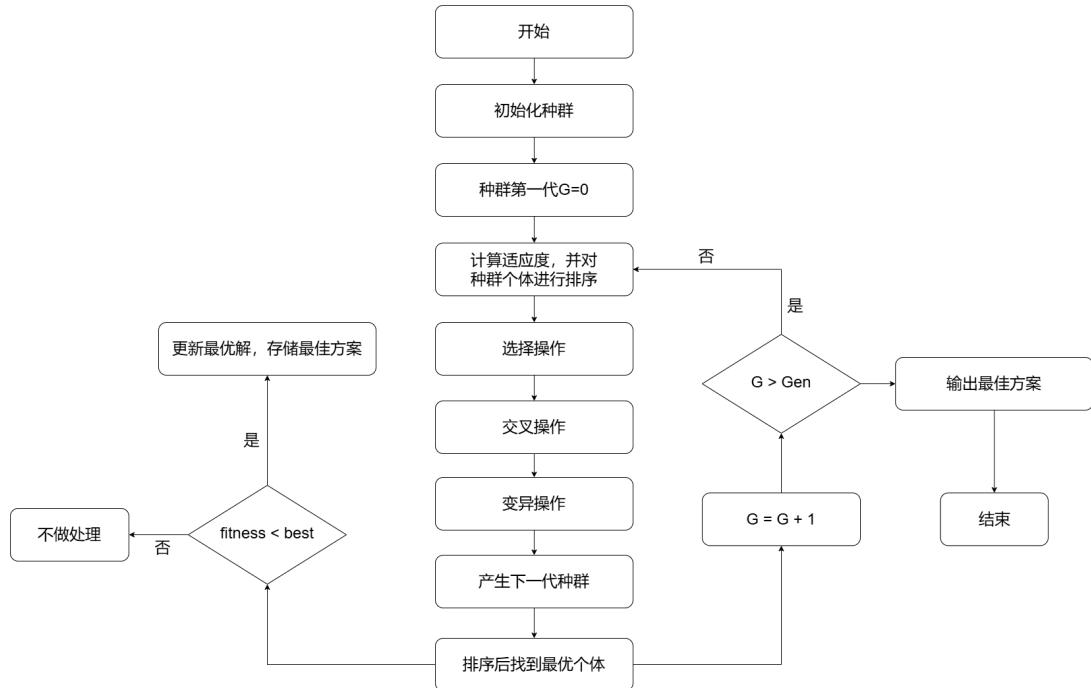


图 9: 问题四遗传算法流程图

各步骤具体介绍如下:

1. **初始化:** 随机生成初始种群。
2. **适应度评估:** 计算种群中每个个体的适应度值。
3. **选择:** 根据适应度选择较优个体。
4. **交叉与变异:** 通过交叉操作产生新个体, 并进行随机变异。
5. **生成新种群:** 更新种群, 继续评估。
6. **终止条件与输出:** 当满足终止条件时, 输出最优解。

c. 算法伪代码

遗传算法因其简单有效, 被广泛应用于各种优化问题中, 成为求解复杂问题的重要工具之一。因此, 本模型利用遗传算法优化了零配件和成品在不同情况下一系列检

Algorithm 1 遗传算法

```
初始化种群  
评估适应度  
while 终止条件未满足 do  
    选择、交叉、变异操作生成新种群  
    评估新种群  
end while  
输出最优解
```

测与拆解的决策。模型基于模拟的生产数据，通过对每种情况的不同决策模式的评估，旨在最大化收入与最小化成本，从而为决策者提供可靠的参考依据。

首先，通过初始化种群以随机生成若干个体（决策方案），并使用适应度函数评估每个个体的经济效益。适应度函数综合考虑了各情况的检测成本、拆解费用和市场售价，从而计算出每个决策方案的总收入与总成本。接着，通过锦标赛选择操作选取表现较好的个体，并通过单点交叉与变异操作生成新的个体，从而引入遗传变异以探索更优解。此外，模型还引入了局部优化步骤，对最后找到的最佳个体进行细微调整，以确保获得更高的适应度值。遗传算法的应用成功找到了最佳的决策方案，最终通过可视化展示了不同情况的总成本与总收入，使企业的决策者能够清晰地理解其决策的经济影响。

2) 针对问题二的遗传算法求解模型

此问题是一个组合优化问题，涉及到在不同情况下，对零配件和成品进行检测及不合格产品处理的决策优化，目标是最大化总收入与总成本之差（即利润）。对于问题二，遗传算法可以应用于寻找最优的生产过程决策方案。这些决策包括是否检测零配件、是否检测成品、是否拆解不合格成品等。具体步骤如下：

定义决策变量：模型的决策变量通过一个二进制编码的个体表示，该个体编码具体包括零件1、零件2、成品是否检测以及是否拆解四个决策共4位，每位由0或1组成。

确定成本与收益计算公式：

对于每一种情况的总成本和总收益，模型的计算公式如下：

$$C_i = \sum_{j=1}^2 (x_{i,j} \cdot (p_{i,j} \cdot s_{i,j} + cost_{c,i,j})) + x_{i,4} \cdot cost_{rework,i} \quad (26)$$

其中 C_i 为总成本，如果 $x_{i,4} = 1$ ，意味着拆解成品，则总成本 $C_i = \text{拆解费用} + \text{所有生产线的成本之和}$ 。如果 $x_{i,4} \neq 1$ ，意味着不拆解成品，则总成本 $C_i = \text{所有生产线的成本之和}$ 。

约束条件：决策变量值必须是0或1，每种情况下的决策是独立的。

3) 针对问题三的遗传算法求解模型

本模型采用遗传算法优化零配件、半成品和成品的检测与拆解决策，以最小化成本并最大化收益。通过对8种零配件与3种半成品的次品率、购买单价、装配成本以及检测成本等数据的分析，构建一个基于适应度评估的策略。具体而言，首先随机生成一组决策方案。随后，通过适应度函数计算每个方案的总收益与总成本，依据适应度值选择优秀个体进行繁殖。在交叉和变异操作的影响下，模型不断迭代，更新种群，逐步向最优解靠近。最终，通过适应度评估，找到表现最佳的决策方案。模型的建立过程如下：

1. 零配件检测决策：

对每个零配件，计算检测和未检测的成本，公式如下：

$$cost_{pt} = \sum_{i=1}^8 (p_{defect_i} \cdot price_i \cdot inspect_i + cost_{inspect_i} \cdot inspect_i) \quad (27)$$

其中， $cost_{parts}$ 为所有零配件的总检测成本， i 为当前零配件的索引（从 1 到 8）， $cost_{inspect_i}$ 为第 i 个零配件的检测费用， $inspect_i$ 表示是否检测第 i 个零配件的决策变量（1 表示检测，0 表示不检测）， p_{defect_i} 为第 i 个零配件的次品率，即生产过程中不合格品的比例， $price_i$ 为第 i 个零配件的购买价格。

如果不检测，需增加未检测零件的损失：

$$loss_{unp} = \sum_{i=1}^8 (1 - inspect_i) \cdot cost_{inspect_i} \quad (28)$$

其中， $loss_{unp}$ 为未检测零件所造成的损失， $1 - inspect_i$ 表示第 i 个零配件是否未检测（若为 1，表示未检测）， $cost_{inspect_i}$ 为第 i 个零配件的检测费用。

2. 半成品检测决策：

计算半成品的总成本与收益，公式如下：

$$cost_{amc} = \sum_{j=1}^3 ((p_{defect_j} \cdot cost_{assembly_j} + cost_{inspect_j}) \cdot inspect_j) \quad (29)$$

其中， $cost_{amc}$ 表示所有半成品的总检测成本， j 表示当前半成品的索引（从 1 到 3）， p_{defect_j} 为第 j 个半成品的次品率， $cost_{assembly_j}$ 表示第 j 个半成品的装配成本， $cost_{inspect_j}$ 表示第 j 个半成品的检测费用， $inspect_j$ 表示是否检测第 j 个半成品的决策变量（1 表示检测，0 表示不检测）。

$$revenue_{rmt} = R - cost_{replace} \quad (30)$$

其中, $revenue_{rmt}$ 表示拆解半成品后的总收益, R 表示拆解半成品所产生的总收入, $cost_{replace}$ 为由于调换或退货造成的损失费用。

3. 成品检测与拆解:

成品的适应度计算包括检测与拆解的信息:

$$cost_f = p_0 \cdot cost_{assembly} \cdot inspect \quad (31)$$

加总成品拆解带来的额外收益和成本:

$$cost_{rft} = cost_{rework} + \sum (\text{相关费用或收益}) \quad (32)$$

其中, $cost_{rft}$ 表示成品拆解后的总成本, $cost_{rework}$ 为拆解成品所需的费用, Σ 表示与拆解相关的其他费用或收益。

5.4.2 模型的求解

1) 利用遗传算法求解问题二

step1: 次品率修正

由于次品率是根据第一问的检测结果得到的, 因此需要对次品率进行修正。对修正前后的次品率进行比较, 并进行以下图表可视化。

对概率进行修正后, 得到的零件1次品率变化情况如图 10 所示。

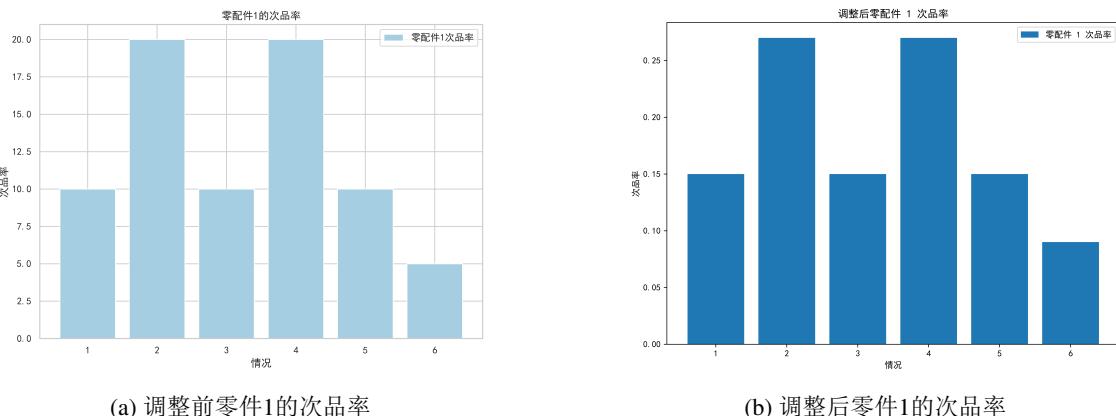


图 10: 零件1的次品率变化图

对概率进行修正后, 得到的零件2次品率变化情况如图 11 所示。

对概率进行修正后, 得到的成品次品率变化情况如图 12 所示。

由上述三个图表可知: 修正后的次品率均 \geq 修正前的次品率。

step2: 初始化

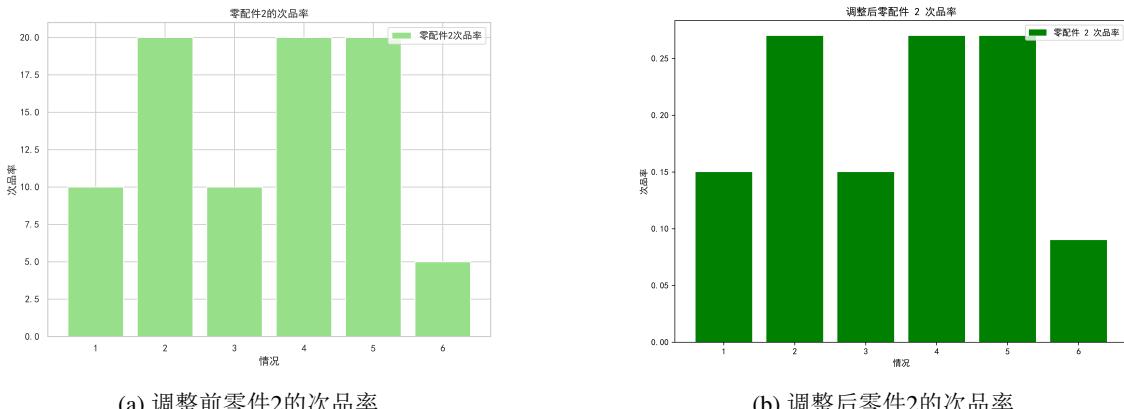


图 11: 零件2的次品率变化图

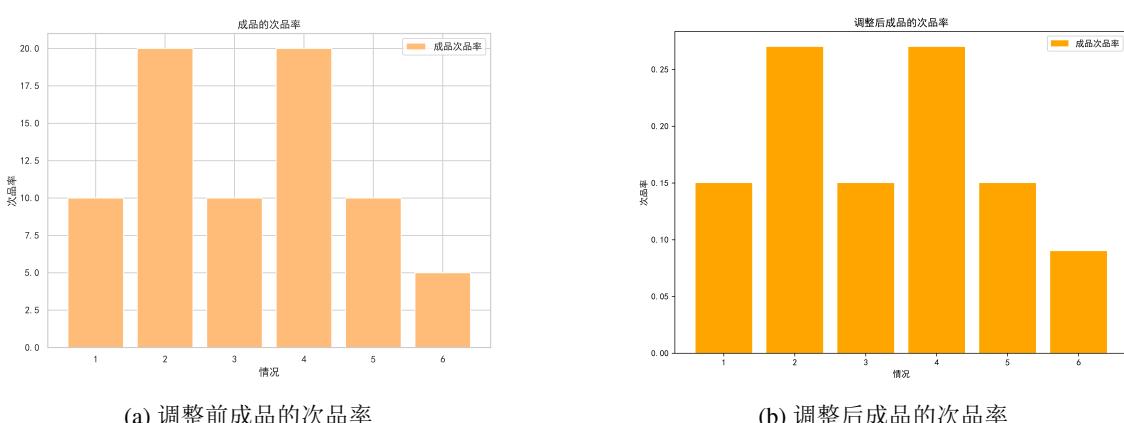


图 12: 成品的次品率变化图

生成一个随机初始种群，种群中的每个个体都代表一个可能的解决方案，即决策变量的组合。

step3：适应度计算

对于种群中的每个个体，即每个解决方案，计算其适应度，即总收入与总成本之差。

step4：选择操作

使用锦标赛选择法从当前种群中选择个体以生成交配池。锦标赛选择法通过随机选择5个个体，然后选择这些个体中适应度最高的一个进入交配池。

step5：交叉操作

在交配池中随机选择两个个体进行交叉，即基因重组，以生成新的子代个体。这里使用单点交叉。使用锦标赛选择法从当前种群中选择个体以生成交配池。锦标赛选择法通过随机选择5个个体，然后选择这些个体中适应度最高的一个进入交配池。

step6：变异操作

以一定的概率，对子代个体的某些基因进行变异即，翻转0和1，以增加种群的多

样性。

step7：精英策略

将当前种群中适应度最高的几个个体（本例中为5个）直接复制到下一代种群中，以确保优秀基因不会丢失。

step8：局部搜索

对遗传算法找到的最优解进行局部搜索，以进一步提高解的质量。这里使用简单的邻域搜索方法，即尝试翻转最优解中的每个基因，并保留使适应度提高的变异。终止条件：

step9：终止条件

当达到预定的迭代次数为100次时，算法终止，并输出最优解和最优适应度。并将结果可视化如13，将最优解应用到每种情况上，计算并可视化每种情况下的总成本和总收入。从13中可以明显看出，适应度值呈现出一个总体上升的趋势。这表明随着迭代的进行，算法或进化过程中的个体在不断优化，其适应度在逐渐提高。在适应度上升的过程中，存在一定的波动。这种波动可能是由于算法在搜索过程中探索了不同的解决方案，并进行了比较和筛选。然而，从整体趋势来看，这些波动并未改变上升的总方向。此外，在达到第50代后，适应度值趋于稳定，这可能意味着算法已经收敛到了一个相对优秀的解。

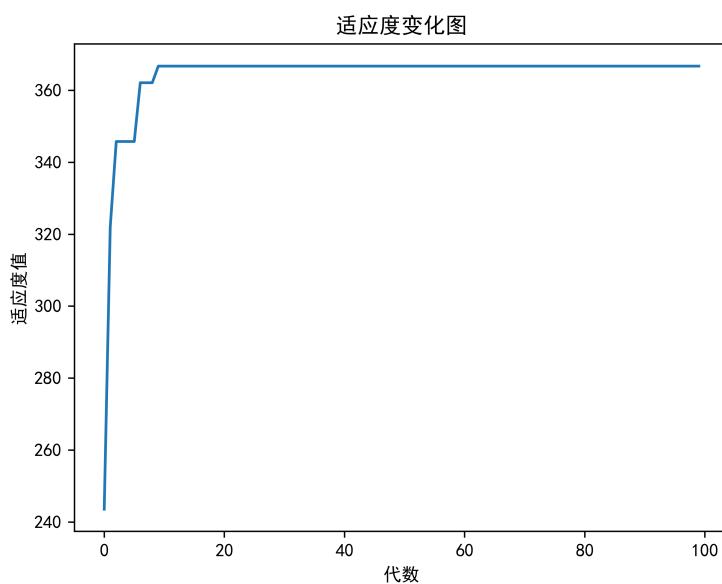


图 13: 适应度变化变化曲线图

1) 利用遗传算法求解问题三

step1：参数设置

根据题意可设置：市场售价为200，调换损失为40，种群数量为 $size_p = 100$ ，迭代次数为 $num_{gen} = 200$ ，变异率为 $rate_{mutation} = 0.15$ ，锦标赛大小为 $size_{tournament} = 5$ ，精英个体数量为 $size_{elite} = 10$ 。

step2：初始化种群

使用二进制编码初始化种群，代表每个个体（决策方案）是否对某个零配件、半成品或成品进行检测或拆解。

step3：适应度计算

利用适应度函数计算每个个体的收益与成本，根据检测与拆解的决策来计算总成本、总收益和未检测零配件的损失。通过对零配件成本、半成品成本和成品成本的计算进行适应度的评估，具体步骤如下：对于每个零配件，如果进行检测，则成本为次品率 \times 购买单价 + 检测成本；如果不检测，则将未检测的检测成本加入未检测损失。考虑同样的计算方式，计算装配成本和检测成本，并根据是否进行检测来调整成本和收益。对拆解产生的零配件进行减损。最后计算成品的检测与拆解费用，并相应计算可能的收益。

step4：选择操作

选择操作是从当前种群中选择出适应度较高的个体。这里采用锦标赛选择的方法。

step5：交叉与变异操作

交叉操作是遗传算法中用于产生新个体的重要机制，它将两个父个体结合产生子个体。从种群中随机选择若干个个体。返回这些个体作为选择结果。变异操作则用于增加遗传多样性，避免算法陷入局部最优。通过生成一个随机掩码，确定哪些基因需要变异，然后反转这些基因（0变为1，1变为0）。

step6：输出最佳决策方案

根据最佳个体的基因输出零配件、半成品和成品的检测与拆解决策。并绘制适应度随代数变化的图表如14所示，以便观察进化过程。从图中可知，从图表的最左侧开始，适应度呈现出明显的上升趋势，这表明在算法的初期阶段，解决方案的质量在逐步提升，即算法在逐步逼近更优的解。在整个过程中，适应度并非一直单调递增，而是存在波动。这种波动可能反映了算法在搜索空间中的探索与利用过程，即在发现新解的同时，也在评估和调整这些解的质量。在第100代时，适应度达到了一个显著的峰值2000点。这表明在这一代，遗传算法找到了一个非常优质的解。随后，适应度有所回落，但在第175代时再次达到一个高峰1900点，尽管这个峰值略低于第100代的峰值。这表明算法在继续搜索的过程中，虽然暂时偏离了最优解，但最终还是能够找到接近最优的解。从第175代之后，适应度逐渐稳定下来，并在第200代时保持在大约1800点。

的水平。这表明算法可能已经收敛到一个相对稳定的解集，或者在该算法的设定下，无法进一步显著提高解的质量。这一过程反映出遗传算法在搜索空间中的动态行为以及其对决策问题求解的适应性。

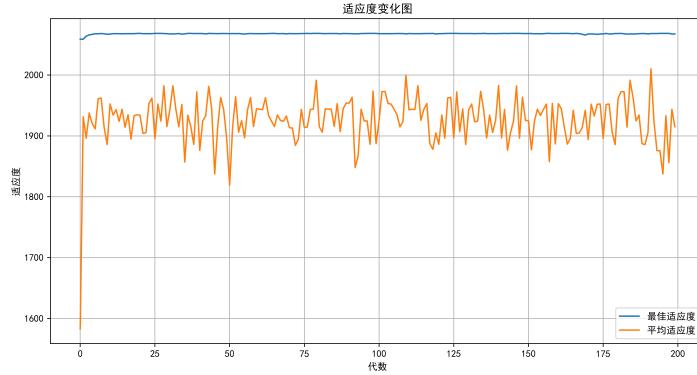


图 14: 适应度变化变化曲线图

六、模型的评价与推广

6.1 模型的优点

1. 问题一的模型通过统计学的假设检验理论和Cochran公式以及有限总体修正公式，得到最小的抽样检测样本量，为企业提供了量化的、系统化的抽样检测方案，可以最小化抽样检测的费用，帮助企业控制成本。
2. 问题二的模型通过遍历所有可能的决策组合，利用动态规划思想分阶段计算每种组合下的利润，使得决策过程细致，精确度更高。同时通过遍历算法和敏感性分析的细致观察，可以得到流程之间的规律和联系，从而简化了多流程时的决策问题。
3. 问题三的模型面对多道程序加工问题，建立全流程的决策模型，对于现实存在的许多多道工序加工产品可以做到决策的把控，最小化成本，最大化利润。

6.2 模型的缺点

1. 问题二的模型通过遍历法求解，只适用于决策变量少的情况。在实际中，若决策变量增多或情况更为复杂，遍历法可能变得不切实际，计算量将急剧增加。
2. 采用遍历策略对各个阶段进行细致模拟，需要考虑的变量条件过多，计算量较大，且容易受到噪声的影响，导致模型的稳定性较差。
3. 问题四采用的遗传算法存在的一定的随机性，每次迭代生成的过程中人为控制，导致结果存在一定的误差。

6.3 模型的推广

1. 对于更复杂的决策问题，问题二的模型可以集成其他优化算法，如遗传算法、粒子群算法等，来求解最优解，以提高求解效率和准确性。
2. 在实际生活中，可以进一步考虑更多的因素，如生产线的效率、设备的利用率等，以建立更为全面的生产决策模型。

参考文献

- [1] W. G. Cochran. The comparison of percentages in matched samples. *Biometrika*, 37(3/4):256–266, 1950.
- [2] William G Cochran. Sampling techniques. *John Wiley& Sons*, 1977.
- [3] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm-a literature review. In *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pages 380–384. IEEE, 2019.
- [4] Tom V Mathew. Genetic algorithm. *Report submitted at IIT Bombay*, 53, 2012.
- [5] Chanuan Uakarn, Kajohnsak Chaokromthong, Nittaya Sintao, Kasem Bundit University, and Stamford International University Rajamangala University of Technology Suvarnabhumi. Sample size estimation using yamane and cochrane and krejcie and morgan and green formulas and cohen statistical power analysis by g*power and comparisons. *APHEIT INTERNATIONAL JOURNAL*, 10(2):76–88, 2021.
- [6] 同济大学数学系. 概率论与数理统计. 人民邮电出版社, 2017.
- [7] 惠高峰. 数学模型中的0、1变量的使用的案例. 中小企业管理与科技(下旬刊), (11):269–270, 2014.
- [8] 杨乃军. 计量抽样检验应用研究. Master’s thesis, 南京理工大学, 2010.
- [9] 王军虎. 统计检验中假设的设置方法. 统计与决策, 38(21):57–59, 2022.
- [10] 黎楠. 新能源汽车最优生产决策研究. Master’s thesis, 重庆大学, 2020.

附录

附录1：支撑材料文件目录

附录2：代码

附录1：支撑材料文件目录

文件列表名	文件名
code （4个问题的求解代码）	Q1.py Q2.py Q2_敏感性分析.py Q3.py Q4_GA重算问题二.py Q4_GA重算问题三.py Q4_问题二次品率计算.py Q4_问题三次品率计算.py
data （利用的表格数据）	Q2_6种情况在不同策略下的总利润.xlsx Q2_每个情况的最佳方案.xlsx Q3_所有策略对应的利润.xlsx

文件列表名	文件名
image (生成的图片)	
敏感性分析	

附录2：代码

问题一代码

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

# 设置字体为SimHei以支持中文显示
# 设置字体大小
plt.rcParams.update({'font.size': 16})    # 修改整体字体大小，可以根据需要调整
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False   # 解决负号显示问题

def calculate_min_sample_size(p, p0=0.1, confidence_level=0.95,
                               error_margin=0.05):
    # 计算给定总体数量 p 下的最小样本量 m，使得在95%置信水平下能够判断是否拒收。

    # 参数：
    # p (int): 总体数量
    # p0 (float): 次品率的估计（标称值，默认为10%）
    # confidence_level (float): 置信水平（默认95%）
    # error_margin (float): 容忍误差（默认5%）

    # 返回：
    # int: 最小样本量 m

    Z = norm.ppf(confidence_level)    # 单侧检验，正态分布查表得到Z值
    # 使用 Cochran 公式计算样本量
    m = (Z ** 2 * p0 * (1 - p0)) / (error_margin ** 2)

    # 如果总体数量有限，使用有限总体修正公式
    if p < np.inf:
        m_adjusted = m / (1 + (m - 1) / p)
    else:
        m_adjusted = m

    return int(np.ceil(m_adjusted))    # 向上取整以确保足够的样本量

# 1.1 小样本
# 不同总体数量下的样本量计算
p_values = np.arange(1, 502, 20)    # 小样本
m_values = [calculate_min_sample_size(p) for p in p_values]

# 绘制样本量随总体数量变化的图表
plt.figure(figsize=(12, 6))
plt.plot(p_values, m_values, marker='o')
plt.title("不同总体数量下的最小样本量的关系pm")
plt.xlabel("总体数量 p")
plt.ylabel("最小样本量 m")
plt.grid(True)
# plt.show()
plt.savefig("../image/1_1.png", dpi=300)
```

```

# 1.2 大样本
# 不同总体数量下的样本量计算
p\_values = np.arange(500, 100001, 2000) # 大样本: 总体数量从50到10000
m\_values = [calculate\_min\_sample\_size(p) for p in p\_values]

# 绘制样本量随总体数量变化的图表
plt.figure(figsize=(12, 6))
plt.plot(p\_values, m\_values, marker='o')
plt.title('不同总体数量 p 下的最小样本量 m 的关系')
plt.xlabel('总体数量 p')
plt.ylabel('最小样本量 m')
plt.grid(True)
# plt.show()
plt.savefig('../image/1\_2.png', dpi=300)

# 2.1 小样本
# 不同总体数量下的样本量计算
p\_values = np.arange(1, 502, 20) # 小样本
m\_values = [calculate\_min\_sample\_size(p, confidence\_level=0.9) for
    p in p\_values]

# 绘制样本量随总体数量变化的图表
plt.figure(figsize=(12, 6))
plt.plot(p\_values, m\_values, marker='o')
plt.title('不同总体数量 p 下的最小样本量 m 的关系')
plt.xlabel('总体数量 p')
plt.ylabel('最小样本量 m')
plt.grid(True)
# plt.show()
plt.savefig('../image/1\_3.png', dpi=300)

# 2.2 大样本
# 不同总体数量下的样本量计算
p\_values = np.arange(500, 100001, 2000) # 大样本: 总体数量从50到10000
m\_values = [calculate\_min\_sample\_size(p, confidence\_level=0.9) for
    p in p\_values]

# 绘制样本量随总体数量变化的图表
plt.figure(figsize=(12, 6))
plt.plot(p\_values, m\_values, marker='o')
plt.title('不同总体数量 p 下的最小样本量 m 的关系')
plt.xlabel('总体数量 p')
plt.ylabel('最小样本量 m')
plt.grid(True)
# plt.show()
plt.savefig('../image/1\_4.png', dpi=300)

# 综上所述:
# (1) 小样本修正, 随总体数增加, 样本数增加; 大样本就直接用Cochran公式, 算出来最大是98
# (2) 小样本修正, 随总体数增加, 样本数增加; 大样本就直接用Cochran公式, 算出来最大是60

```

问题二主体代码

```

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

```

```

# 设置字体大小
plt.rcParams.update({'font.size': 14}) # 修改整体字体大小，可以根据需要调整
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
import itertools

# 输入表格数据
data = {
    "情况": [1, 2, 3, 4, 5, 6],
    "零配件1次品率": [10, 20, 10, 20, 10, 5],
    "零配件1购买单价": [4, 4, 4, 4, 4, 4],
    "零配件1检测成本": [2, 2, 2, 1, 8, 2],
    "零配件2次品率": [10, 20, 10, 20, 20, 5],
    "零配件2购买单价": [18, 18, 18, 18, 18, 18],
    "零配件2检测成本": [3, 3, 3, 1, 1, 3],
    "成品次品率": [10, 20, 10, 20, 10, 5],
    "成品装配成本": [6, 6, 6, 6, 6, 6],
    "成品检测成本": [3, 3, 3, 2, 2, 3],
    "市场售价": [56, 56, 56, 56, 56, 56],
    "调换损失": [6, 6, 30, 30, 10, 10],
    "拆解费用": [0, 0, 0, 0, 0, 40]
}

df = pd.DataFrame(data)

# 记录前一次企业生产的成品数量
previous_n = 100

# 计算利润函数
def count_profit(opt1, opt2, opt_check, opt_rework, p1, p2, p3,
                 p_check_cost1, p_check_cost2, assembly_cost,
                 p_check_cost, replace_cost, rework_cost, n1, n2):
    # 递归退出条件
    if n1 < 1 or n2 < 1:
        return 0 # 没赚钱也没亏钱
    # 考虑策略1、2
    p_assembly = 1 - max(opt1 * p1, opt2 * p2)
    # 生产的成品数量
    product_n = n1 * p_assembly
    # 理想状态下，完全没有次品的成品个数
    ideal_n = 0
    if opt1 == 1 and opt2 == 1:
        ideal_n = n1 * (1 - max(p1, p2)) * (1 - p3)
    else:
        ideal_n = n1 * (1 - p1) * (1 - p2) * (1 - p3)
    # 根据策略3，得到实际成品数量
    real_n = ideal_n if opt_check == 1 else product_n
    # 如果不选择检测，仍有次品，需要计算次品率
    if real_n != 0:
        p_hidden = 1 - ideal_n / real_n
    else:
        p_hidden = 0 # 或者其他适当的处理方式
    # 零件检测成本
    cost_parts = opt1 * p_check_cost1 * n1 + opt2 * p_check_cost2 * n2
    # 装配成本
    cost_assembly = assembly_cost * product_n

```

```

# 成品检测成本
cost_product = opt_check * p_check_cost * product_n
# 拆解和调换成本
cost_rework = opt_rework * rework_cost * (product_n - ideal_n)
cost_replace = replace_cost * p_hidden * real_n
# 如果选择拆解, 那还要重新得到零件数去递归
reject_n = opt_rework * (product_n - ideal_n)
# 总成本
total_cost = cost_parts + cost_assembly + cost_product +
    cost_rework + cost_replace
# 总售额: 理想的成品数量
total_revenue = ideal_n * 56
# 当前赚的利润
is_earned = ideal_n * 34 - total_cost

global previous_n
print(f'当前赚的利润为: {is_earned}, 上一次拆解的零件数: {previous_n}, 这一次
      拆解的零件数: {reject_n}')
# 如果拆解的零件数比上一次少, 说明才有拆解的必要, 否则拆解后再组装还是次品, 卖不出去
if is_earned > 0 and previous_n - 1 > reject_n:
    previous_n = reject_n
    additional_profit = f_profit(opt1, opt2, opt_check, opt_rework,
        p1, p2, p3, p_check_cost1, p_check_cost2,
        assembly_cost, p_check_cost,
        replace_cost, rework_cost,
        reject_n, reject_n)
else:
    additional_profit = 0
# 总利润
total_profit = total_revenue - total_cost + additional_profit
print(f'共{ideal_n}件, 卖了}{total_revenue}元, 重加
      工}{additional_profit}元, 成本为}{total_cost}元')
print(f'成品回溯, 总利润共{total_profit}元')
return total_profit

# 递归, 要先做次品率的概率修正
def f_profit(opt1, opt2, opt_check, opt_rework, p1, p2, p3,
    p_check_cost1, p_check_cost2, assembly_cost, p_check_cost,
    replace_cost, rework_cost, n1, n2):
    # print('before', p1, p2, p3)
    if opt1 == 0 and opt2 == 0:
        p1_new = p1 / (1 - (1 - p1) * (1 - p2) * (1 - p3))
        p2_new = p2 / (1 - (1 - p1) * (1 - p2) * (1 - p3))
        p3_new = p3 / (1 - (1 - p1) * (1 - p2) * (1 - p3))
    elif opt1 == 1 and opt2 == 0:
        p1_new = 0
        p2_new = p2 / (1 - (1 - p2) * (1 - p3))
        p3_new = p3 / (1 - (1 - p2) * (1 - p3))
    elif opt1 == 0 and opt2 == 1:
        p1_new = p1 / (1 - (1 - p1) * (1 - p3))
        p2_new = 0
        p3_new = p3 / (1 - (1 - p1) * (1 - p3))
    else:
        p1_new = 0
        p2_new = 0
        p3_new = 1
    return count_profit(opt1, opt2, opt_check, opt_rework, p1_new,
        p2_new, p3_new, p_check_cost1, p_check_cost2,

```

```

        assembly_cost, p_check_cost, replace_cost,
        rework_cost, n1, n2)

# 定义所有可能的决策组合（16种策略）
decisions = list(itertools.product([0, 1], repeat=4))
# 初始化存储结果的列表
results = []
# 遍历每种情况和每种决策组合
for index, row in df.iterrows():
    for decision in decisions:
        # 记录前一次企业生产的成品数量
        previous_n = 100
        # 拿到4个决策参数
        opt1, opt2, opt_check, opt_rework = decision
        # 从 row 中提取参数
        p1 = row['零配件1次品率'] / 100
        p2 = row['零配件2次品率'] / 100
        p3 = row['成品次品率'] / 100
        assembly_cost = row['成品装配成本']
        p_check_cost = row['成品检测成本']
        replace_cost = row['调换损失']
        rework_cost = row['拆解费用']
        p_check_cost1 = row['零配件1检测成本']
        p_check_cost2 = row['零配件2检测成本']
        n1 = 100 # 零件1数量，这里可以是一个固定值或从其他地方读取
        n2 = 100 # 零件2数量
        # 计算该种决策下的利润
        profit = count_profit(opt1, opt2, opt_check, opt_rework, p1, p2,
                               p3, p_check_cost1, p_check_cost2,
                               assembly_cost, p_check_cost, replace_cost
                               , rework_cost, n1, n2)
        results.append({
            "情况": row["情况"],
            "零件检测1": decision[0],
            "零件检测2": decision[1],
            "成品检测": decision[2],
            "不合格成品拆解": decision[3],
            "利润": profit - (4 + 18) * 100
        })
# 转换为DataFrame输出结果
results_df = pd.DataFrame(results)
# 输出结果
results_df.to_excel('../data/种情况在不同策略下的总利
润Q2_6.xlsx', index=False)

# 按利润最大化排序，获取每种情况的最佳决策
best_decisions = results_df.loc[results_df.groupby("情况")["利
润"].idxmax()]

print("\n最佳决策方案: ", best_decisions)

# 按利润最大化排序，获取每种情况的最佳决策
best_decisions = results_df.loc[results_df.groupby("情况")["利
润"].idxmax()]

# 获取所有的决策组合
unique_decisions = results_df[['零件检测1', '零件检测2', '成品检测', '不合格成
品拆解']].drop_duplicates()

```

```

# 16条策略折线图，横坐标6种情况，纵坐标总利润
# 为每种策略绘制折线图
plt.figure(figsize=(12, 6))

for index, decision in unique_decisions.iterrows():
    # 获取当前策略的所有行
    strategy_rows = results_df[
        (results_df['零件检测1'] == decision['零件检测1']) &
        (results_df['零件检测2'] == decision['零件检测2']) &
        (results_df['成品检测'] == decision['成品检测']) &
        (results_df['不合格成品拆解'] == decision['不合格成品拆解'])
    ]

    # 绘制当前策略的折线图
    plt.plot(strategy_rows['情况'], strategy_rows['利润'], marker='o', label=f"策略: {decision.values}")

# 可视化图表
# 表1 6种情况在不同策略下的总利润变化
plt.title('6种情况在不同策略下的总利润变化')
plt.xlabel('情况')
plt.ylabel('总利润')
plt.legend(loc='upper right', bbox_to_anchor=(1.25, 1.02), title="策略组合")
plt.tight_layout()
plt.savefig('../image/6种情况在不同策略下的总利润变化/6.png', dpi=500)

# 复制数据并更新列名为中文
best_decision_df = best_decisions.copy()
best_decision_df.columns = ['情况', '零件检测1', '零件检测2', '成品检测', '不合格成品拆解', '利润']

# 替换检测和拆解的值
best_decision_df['零件检测1'] = best_decision_df['零件检测1'].map({1: '检测', 0: '不检测'})
best_decision_df['零件检测2'] = best_decision_df['零件检测2'].map({1: '检测', 0: '不检测'})
best_decision_df['成品检测'] = best_decision_df['成品检测'].map({1: '检测', 0: '不检测'})
best_decision_df['不合格成品拆解'] = best_decision_df['不合格成品拆解'].map({1: '拆解', 0: '不拆解'})

# 保留利润小数点后两位
best_decision_df['利润'] = best_decision_df['利润'].apply(lambda x: f"{x:.2f}")

# 将情况前缀更新为"情况1-6"
best_decision_df['情况'] = best_decision_df['情况'].apply(lambda x: f'情况{x}')

# 输出到 Excel 文件
best_decision_df.to_excel('../data/每个情况的最佳方案Q2_.xlsx', index=False)

# 绘制表格
fig, ax = plt.subplots(figsize=(12, 4))

# 移除坐标轴
ax.xaxis.set_visible(False)

```

```

ax.yaxis.set_visible(False)
ax.set_frame_on(False)

# 创建表格
table = plt.table(cellText=best_decision_df.values,
                   colLabels=best_decision_df.columns,
                   cellLoc='center',
                   loc='center',
                   bbox=[0, 0, 1, 1])

# 设置表格样式
table.auto_set_font_size(False)
table.set_fontsize(12)
table.scale(1.2, 1.2) # 调整表格大小

plt.title('每个情况的最佳方案', fontsize=16)
plt.tight_layout()
# plt.savefig('每个情况的最佳方案.png', dpi=500)
plt.show()

# 创建数据
data = {
    '情况': ['情况1', '情况2', '情况3', '情况4', '情况5', '情况6'],
    '利润': [1242.00, 308.00, 1119.77, 544.00, 783.46, 1858.68]
}

# 转换为 DataFrame
df = pd.DataFrame(data)

# 设置seaborn主题
sns.set(style="whitegrid")

# 设置字体大小
plt.rcParams.update({'font.size': 14}) # 修改整体字体大小，可以根据需要调整
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

# 绘制柱状图
plt.figure(figsize=(10, 6))
barplot = sns.barplot(x='情况', y='利润', data=df, palette="viridis")

# 添加数据标签
for index, row in df.iterrows():
    barplot.text(index, row['利润'] + 10, round(row['利润'], 2), color='black', ha="center")

# 设置标题和标签
plt.title('各情况下的最佳决策利润', fontsize=16)
plt.xlabel('情况', fontsize=14)
plt.ylabel('利润', fontsize=14)

# 显示图表
# plt.show()
plt.savefig('../image/2_col.png', dpi=300)

```

问题二敏感性分析部分代码

```
# 其中用到的计算利润的函数在上面的代码中已经定义，这里不再重复定义

# 创建“敏感性分析”目录
os.makedirs("敏感性分析 ../", exist_ok=True)

# 敏感性分析结果展示
def sensitivity_analysis(parameter, values):
    plt.figure(figsize=(12, 8))
    tables = []
    for value in values:
        temp_df = df.copy()
        temp_df[parameter] = value
        temp_results = compute_profit_for_decisions(temp_df, decisions)
        best_temp_decisions = temp_results.loc[temp_results.groupby("情况")["利润"].idxmax()]

        # 以参数值为x轴，最大利润为y轴绘制
        plt.plot(best_temp_decisions["情况"], best_temp_decisions["利润"], marker='o', linestyle='-', label=f'{parameter} = {value}')

        # 准备表格数据
        best_decision_table = best_temp_decisions.copy()
        best_decision_table[parameter] = f'{value}'
        best_decision_table["零件检测1"] = best_decision_table["零件检测1"].apply(
            lambda x: '检测' if x == 1 else '不检测')
        best_decision_table["零件检测2"] = best_decision_table["零件检测2"].apply(
            lambda x: '检测' if x == 1 else '不检测')
        best_decision_table["成品检测"] = best_decision_table["成品检测"].apply(
            lambda x: '检测' if x == 1 else '不检测')
        best_decision_table["不合格成品拆解"] = best_decision_table["不合格成品拆解"].apply(
            lambda x: '拆解' if x == 1 else '不拆解')

        # 添加表格到列表
        tables.append(best_decision_table)

    plt.xlabel('情况')
    plt.ylabel('最大利润')
    plt.title(f'{parameter}的敏感性分析')
    plt.legend()
    plt.grid(True)

    # 保存敏感性分析图像
    plt.savefig(f'敏感性分析 ../{parameter}的敏感性分析图像.png', dpi=500)
    plt.close()

    # 保存表格为PNG图片
    for i, table in enumerate(tables):
        fig, ax = plt.subplots(figsize=(12, 4))
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        ax.set_frame_on(False)
```

```

# 创建表格
table_fig = plt.table(cellText=table.values,
                      colLabels=table.columns,
                      cellLoc='center',
                      loc='center',
                      bbox=[0, 0, 1])

# 设置表格样式
table_fig.auto_set_font_size(False)
table_fig.set_fontsize(12)
table_fig.scale(1.2, 1.2) # 调整表格大小

plt.title(f'{parameter}={values[i]下的最佳方案}', fontsize=16)
plt.tight_layout()

# 保存表格图像
plt.savefig(f'敏感性分析在.../{parameter}={values[i]下的最佳方
案}.png', dpi=500)
plt.close()

# 进行敏感性分析
parameters = [
    ('零配件次品率1', [5, 10, 15, 20]),
    ('零配件次品率2', [5, 10, 15, 20]),
    ('成品次品率', [5, 10, 15, 20]),
    ('零配件检测成本1', [0, 2, 5, 10]),
    ('零配件检测成本2', [0, 2, 5, 10]),
    ('成品检测成本', [0, 2, 5, 10]),
    ('调换损失', [0, 5, 10, 15, 20])
]

for param, vals in parameters:
    sensitivity_analysis(param, vals)

```

问题三代码

```

import pandas as pd
import itertools
import matplotlib.pyplot as plt

# 设置字体大小
plt.rcParams.update({'font.size': 16}) # 修改整体字体大小，可以根据需要调整
# 防止中文乱码
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

# 构造DataFrame
data = {
    '零配件': [1, 2, 3, 4, 5, 6, 7, 8],
    '次品率': ['10%', '10%', '10%', '10%', '10%', '10%', '10%', '10%'],
    '购买单价': [2, 8, 12, 2, 8, 12, 8, 12],
    '检测成本': [1, 1, 2, 1, 1, 2, 1, 2],
}

df = pd.DataFrame(data)

```

```

# 利润最大化总利润 = 总收入 - 总成本

# 题目给的，都一样的常量
p_check_cost1 = p_check_cost2 = 1    # 零件1、零件2的检测成本
p_check_cost3 = 2    # 零件3的检测成本
assembly_cost = 8    # 装配成本
replace_cost = 40   # 成品调换损失
mid_check_cost = 4    # 半成品检测成本
p_check_cost = 6    # 成品检测成本
mid_rework_cost = 6    # 半成品拆解费用
p_rework_cost = 10   # 成品拆解费用
p_part = 0.1    # 刚开始零件的次品率
p_mid = 0.1    # 刚开始半成品的次品率
previous_mid_n = 100   # 记录前一次企业生产的半成品数量
previous_f_n = 100   # 记录前一次企业生产的成品数量

# 零件 - 成品全流程函数
def count_profit(opt1, opt2, opt3, opt4, opt5, n, p_part, p_mid):
    # 递归退出条件
    if n < 1:
        return 0    # 没赚钱也没亏钱

    # ----- 零件 - 半成品 -----
    # 企业实际生产的半成品数量
    p_to_mid = 1 - opt1 * p_part
    product_mid_n = p_to_mid * n
    # 理想状态下，完全没有次品的半成品个数
    if opt1 == 1:
        ideal_mid_n = n * (1 - p_part) * 0.9    # 组装
    else:
        ideal_mid_n = n * (1 - p_part) ** 3 * 0.9    # 3种零件 + 组装

    # 根据策略opt2，得到实际半成品数量
    real_mid_n = ideal_mid_n if opt2 == 1 else product_mid_n

    # 如果不选择检测，仍有次品，需要计算次品率
    p_mid_hidden = 1 - ideal_mid_n / real_mid_n

    # 零件检测成本
    cost_parts = 6 * n * opt1 * p_check_cost1 + 2 * n * opt1 *
                  p_check_cost3
    # 装配成半成品的成本
    cost_mid_assembly = assembly_cost * 3 * product_mid_n
    # 半成品检测成本
    cost_mid_check = opt2 * mid_check_cost * 3 * product_mid_n
    # 半成品拆解成本
    cost_mid_rework = opt4 * mid_rework_cost * 3 * (product_mid_n -
                                                       ideal_mid_n)

    # 根据opt4，选择是否对半成品进行拆解，如果拆解要进递归
    reject_mid_n = opt4 * (product_mid_n - ideal_mid_n)

    additional_profit1 = 0

    global previous_mid_n    # 上一次拆解后的半成品数

```

```

# 如果拆解的零件数比上一次少，说明才有拆解的必要，因为这样说明上一次拆解过后能至少卖
# 出去一个，否则拆解后再组装全都还是次品，卖不出去
if previous_mid_n - 1 > reject_mid_n and reject_mid_n > 1:
    previous_mid_n = reject_mid_n
    print('reject_mid_n: ', reject_mid_n)
    p_part = p_part / (1 - (1 - p_part) ** 3)
    # 递归，拿到剩下reject_mid_n套配件数，组装-生产-销售全流程，产生的利润
    additional_profit1 = count_profit(opt1, opt2, opt3, opt4, opt5,
                                      reject_mid_n, p_part, p_mid)
else:
    additional_profit1 = 0

# ----- 零件 - 成品 -----
# 企业实际生产的成品数量（根据是否检测半成品来考虑）
p_to_f = 1 - opt2 * p_mid
product_f_n = p_to_f * product_mid_n
# 理想状态下，完全没有次品的成品个数
ideal_f_n = 0
if opt2 == 1:
    ideal_f_n = product_mid_n * (1 - p_mid) * 0.9 # 3个半成品能正确组
    装配对
else:
    ideal_f_n = product_mid_n * (1 - p_mid) ** 3 * 0.9 # 3个半成品无
    法正确组装配对
# 根据策略opt3，得到实际成品数量
real_f_n = ideal_f_n if opt3 == 1 else product_f_n
# 如果不选择检测，仍有次品，需要计算次品率
p_f_hidden = 1 - ideal_f_n / real_f_n
# 装配为成品的成本
cost_f_assembly = assembly_cost * product_f_n
# 成品检测成本
cost_f_product = opt3 * p_check_cost * product_f_n
# 成品拆解成本
cost_f_rework = opt5 * p_rework_cost * (product_f_n - ideal_f_n)
# 调换成本
cost_replace = replace_cost * p_f_hidden * real_f_n
# 如果选择对成品进行拆解，那还要重新得到半成品数去递归
reject_f_n = opt5 * (product_f_n - ideal_f_n)
# 半成品 - 成品
global previous_f_n
print(f'上一次半成品的数量: {previous_f_n}, 这一次半成品的数量:
{reject_f_n}')
if previous_f_n - 1 > reject_f_n or reject_f_n > 1: # 得先确定有拆解的
    必要，不然拆完卖不出去浪费钱
    previous_f_n = reject_f_n
    if opt4 == 1 and opt5 == 1:
        # 都拆回零件，那么就计算最终拆成的零件总数，并作次品率p_mid修正，然后去递归
        p_part = p_part / (1 - (1 - p_part) ** 3)
        print('opt4=1, opt5=1')
        additional_profit2 = count_profit(opt1, opt2, opt3, opt4,
                                          opt5, reject_f_n, p_part, p_mid)
    elif opt4 == 0 and opt5 == 1:
        # 半成品不拆回零件，那么就计算最终拆成的零件总数，并作次品率修正，然后去递
        归
        p_mid = p_mid / (1 - (1 - p_mid) ** 3)
        print('opt4=0, opt5=1')
        additional_profit2 = count_middle_to_final_profit(opt2,
                                                       opt3, reject_f_n, p_mid)
    else:
        # 成品不进行拆解，那不考虑了

```

```

        additional_profit2 = 0
    else:
        additional_profit2 = 0
# 总成本
total_cost = cost_parts + cost_mid_assembly + cost_mid_check +
    cost_mid_rework + cost_f_assembly + cost_f_product +
    cost_f_rework + cost_replace
# 总收入
total_revenue = ideal_f_n * 200
# 总利润
total_profit = total_revenue - total_cost + additional_profit1 +
    additional_profit2
print(
    f'共{ideal_f_n}件, 卖了}{total_revenue}元, 半成品重加
        工}{additional_profit1}元, 成品重加工}{additional_profit2}元, 成本
        为}{total_cost}元')
print(f'成品回溯, 总利润共{total_profit}元')

return total_profit

# 半成品 - 成品带来的利润
def count_middle_to_final_profit(opt2, opt3, mid_n, p_mid):
    # 说明:
    # 已知半成品的数量mid_n, 次品率p_mid, (不拆解为零件) 求他们能带来的利润收益
    # 参数:
    # opt1-3: 不在本函数考虑范围之内
    # opt4: 半成品的检测策略
    # opt5: 成品的检测策略
    # opt6=0: 半成品是否拆解不拆解
    # opt7=0: 成品是否拆解拆解
    # mid_n: 理想成品个数
    # p_mid: 半成品的次品率
    # 中间量:
    # product_half1、product_half2、product_half3: 半成品的实际生产数量
    # p_half1_hidden、p_half1_hidden、p_half1_hidden: 如果不对半成品进行检测, 其中隐藏的
        次品率 (检测的话就是次品率为0)
    # 结果:
    # total_profit: 总利润
    # 递归退出条件
    if mid_n < 1:
        return 0 # 没赚钱也没亏钱

    # 企业生产的成品数
    p_to_f = 1 - opt2 * p_mid
    product_n = p_to_f * mid_n
    # 理想状态下, 完全没有次品的成品个数
    if opt2 == 1:
        ideal_f_n = mid_n * (1 - p_mid) * 0.9
    else:
        ideal_f_n = mid_n * (1 - p_mid) ** 3 * 0.9
    # 根据策略opt3, 得到实际成品数量
    real_f_n = ideal_f_n if opt3 == 1 else product_n
    # 如果不选择检测, 仍有次品, 需要计算次品率
    p_f_hidden = 1 - ideal_f_n / real_f_n
    # 半成品检测成本
    cost_mid_check = opt2 * mid_check_cost * 3 * mid_n
    # 装配为成品的成本
    cost_f_assembly = assembly_cost * product_n
    # 成品检测成本

```

```

cost_f_product = opt3 * p_check_cost * product_n
# 成品拆解成本
cost_f_rework = opt5 * p_rework_cost * (product_n - ideal_f_n)
# 调换成本
cost_replace = replace_cost * p_f_hidden * real_f_n

# 此时必对成品进行拆解，重新得到半成品数去继续递归该“半成品 - 成品” 函数，去继续计算收益
reject_f_n = product_n - ideal_f_n
global previous_f_n
if previous_f_n - 1 > reject_f_n and reject_f_n > 1:
    # 半成品不拆回零件，那么就计算最终拆成的零件总数，并作次品率修正，然后去递归
    previous_f_n = reject_f_n
    p_mid = p_mid / (1 - (1 - p_mid) ** 3)
    additional_profit2 = count_middle_to_final_profit(opt2, opt3,
        reject_f_n, p_mid)
else:
    # 成品不进行拆解，那不考虑了
    additional_profit2 = 0
# 总成本
total_cost = cost_mid_check + cost_f_assembly + cost_f_product +
    cost_f_rework + cost_replace
# 总收入
total_revenue = ideal_f_n * 200
# 总利润
total_profit = total_revenue - total_cost + additional_profit2
# 回溯打印
print(
    f'半成品成品回溯：共-{ideal_f_n}件，卖了}{total_revenue}元，成品重加工}{additional_profit2}元，成本为}{total_cost}元')
print(f'半成品成品回溯，总利润共-{total_profit}元')
return total_profit

```

```

# 定义所有可能的决策组合，后面遍历32种情况
decisions = list(itertools.product([0, 1], repeat=5))
# 初始化存储结果的列表
results = []
# 遍历每种情况和每种决策组合
for decision in decisions:
    # 拿到5个决策参数
    opt1, opt2, opt3, opt4, opt5 = decision
    # 从 row 中提取参数
    p_check_cost1 = p_check_cost2 = 1 # 零件1、零件2的检测成本
    p_check_cost3 = 2 # 零件3的检测成本
    assembly_cost = 8 # 装配成本
    replace_cost = 40 # 成品调换损失
    mid_check_cost = 4 # 半成品检测成本
    p_check_cost = 6 # 成品检测成本
    mid_rework_cost = 6 # 半成品拆解费用
    p_rework_cost = 10 # 成品拆解费用
    n = 100 # 零件数量（1-8各100个，理想情况组成100个成品）
    p_part = 0.1 # 刚开始零件的次品率
    p_mid = 0.1 # 刚开始半成品的次品率
    previous_mid_n = 100 # 记录前一次企业生产的半成品数量
    previous_f_n = 100 # 记录前一次企业生产的成品数量
    # 计算该种决策下的利润

```

```

    profit = count_profit(opt1, opt2, opt3, opt4, opt5, n, p_part,
                           p_mid)
    results.append({
        "零件检测": decision[0],
        "半成品检测": decision[1],
        "成品检测": decision[2],
        "不合格半成品拆解": decision[3],
        "不合格成品拆解": decision[4],
        "利润": profit - 64 * 100
    })
}

# 转换为DataFrame输出结果
results_df = pd.DataFrame(results)
results_df.to_excel('../data/所有策略对应的利润Q3_.xlsx', index=False)
# 可视化比较结果
# 从 Excel 文件中读取数据:
df = pd.read_excel('../data/所有策略对应的利润Q3_.xlsx')
# 绘制图表
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['利
润'], marker='o', linestyle='--', color='b', label='利润')
# 设置标题和标签
plt.title('种不同策略的比较32', fontsize=16)
plt.xlabel('策略编号', fontsize=14)
plt.ylabel('利润', fontsize=14)
# 添加网格
plt.grid(True)
# 添加图例
plt.legend()
# 显示图表
# plt.show()
plt.savefig('../image/3_res.png', dpi=500)

```

问题四修正问题二次品率代码

```

import numpy as np
import pandas as pd
from scipy.stats import norm
# 问题二次品率处理
# 计算 Cochran 公式中的样本量
def calculate_min_sample_size(p_total, p0=0.1, confidence_level=0.95,
                               error_margin=0.05):
    Z = norm.ppf(confidence_level) # 单侧检验，查表得到Z值
    # 使用 Cochran 公式计算样本量
    m = (Z ** 2 * p0 * (1 - p0)) / (error_margin ** 2)
    # 如果总体数量有限，使用有限总体修正公式
    m_adjusted = m / (1 + (m - 1) / p_total)
    return int(np.ceil(m_adjusted)) # 向上取整以确保足够的样本量
# 计算修正后的次品率
def adjust_defect_rate(original_rate, p_total, p0=0.1, confidence_level
=0.95, error_margin=0.05):
    # 计算样本量
    sample_size = calculate_min_sample_size(p_total, p0,
                                             confidence_level, error_margin)
    # 计算 Cochran 公式中的置信区间边界
    Z = norm.ppf(confidence_level)

```

```

margin_of_error = Z * np.sqrt((original_rate * (1 - original_rate)) / sample_size)
# 修正次品率为原始次品率加上边界
adjusted_rate = original_rate + margin_of_error
# 确保修正后的次品率不超过 1
adjusted_rate = min(adjusted_rate, 1)
return adjusted_rate
# 表格数据
data = {
    '情况': [1, 2, 3, 4, 5, 6],
    '零配件 1 次品率': [0.1, 0.2, 0.1, 0.2, 0.1, 0.05],
    '零配件 2 次品率': [0.1, 0.2, 0.1, 0.2, 0.2, 0.05],
    '成品次品率': [0.1, 0.2, 0.1, 0.2, 0.1, 0.05]
}
df = pd.DataFrame(data)
# 设定总体数量
p_total = 10000

# 计算修正后的次品率
def calculate_adjusted_defect_rate(df, p_total):
    adjusted_rates = []
    for _, row in df.iterrows():
        # 计算修正后的次品率
        adjusted_rate_1 = adjust_defect_rate(row['零配件 1 次品率'], p_total)
        adjusted_rate_2 = adjust_defect_rate(row['零配件 2 次品率'], p_total)
        adjusted_rate_finished = adjust_defect_rate(row['成品次品率'], p_total)

        adjusted_rates.append({
            '情况': row['情况'],
            '零配件 1 次品率': round(adjusted_rate_1, 4), # 保留四位小数
            '零配件 2 次品率': round(adjusted_rate_2, 4), # 保留四位小数
            '成品次品率': round(adjusted_rate_finished, 4) # 保留四位小数
        })
    return pd.DataFrame(adjusted_rates)

# 计算调整后的次品率
adjusted_df = calculate_adjusted_defect_rate(df, p_total)
print(adjusted_df)

# 问题二新数据展示
new_data = {
    '情况': [1, 2, 3, 4, 5, 6],
    '零配件 1 次品率': [0.15, 0.27, 0.15, 0.27, 0.15, 0.09],
    '零配件 2 次品率': [0.15, 0.27, 0.15, 0.27, 0.27, 0.09],
    '零配件 1 购买单价': [4, 4, 4, 4, 4, 4],
    '零配件 1 检测成本': [2, 2, 2, 1, 8, 2],
    '零配件 2 购买单价': [18, 18, 18, 18, 18, 18],
    '零配件 2 检测成本': [3, 3, 3, 1, 1, 3],
    '成品次品率': [0.15, 0.27, 0.15, 0.27, 0.27, 0.09],
    '装配成本': [6, 6, 6, 6, 6, 6],
    '市场售价': [56, 56, 56, 56, 56, 56],
    '调换损失': [6, 6, 30, 30, 10, 10],
    '拆解费用': [5, 5, 5, 5, 5, 40]
}

```

}

问题四修正问题三次品率代码

```
import numpy as np
import pandas as pd
from scipy.stats import norm
# 计算 Cochran 公式中的样本量
def calculate_min_sample_size(p_total, p0=0.1, confidence_level=0.95,
error_margin=0.05):
    Z = norm.ppf(confidence_level) # 单侧检验，查表得到Z值
    # 使用 Cochran 公式计算样本量
    m = (Z ** 2 * p0 * (1 - p0)) / (error_margin ** 2)
    # 如果总体数量有限，使用有限总体修正公式
    m_adjusted = m / (1 + (m - 1) / p_total)
    return int(np.ceil(m_adjusted)) # 向上取整以确保足够的样本量
# 计算修正后的次品率
def adjust_defect_rate(original_rate, p_total, p0=0.1, confidence_level
=0.95, error_margin=0.05):
    # 计算样本量
    sample_size = calculate_min_sample_size(p_total, p0,
        confidence_level, error_margin)
    # 计算 Cochran 公式中的置信区间边界
    Z = norm.ppf(confidence_level)
    margin_of_error = Z * np.sqrt((original_rate * (1 - original_rate))
        / sample_size)
    # 修正次品率为原始次品率加上边界
    adjusted_rate = original_rate + margin_of_error
    # 确保修正后的次品率不超过 1
    adjusted_rate = min(adjusted_rate, 1)
    return adjusted_rate
# 问题三的新数据
data_problem_3 = {
    '零配件': [1, 2, 3, 4, 5, 6, 7, 8],
    '次品率': [0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10, 0.10],
    '购买单价': [2, 8, 12, 2, 8, 12, 8, 12],
    '检测成本': [1, 1, 2, 1, 1, 2, 1, 2],
    '半成品': [1, 2, 3, , , , , 成品, , ]
}
df_problem_3 = pd.DataFrame(data_problem_3)
# 计算每个零配件的修正次品率
def calculate_adjusted_defect_rates(df, p_total):
    adjusted_rates = []
    for _, row in df.iterrows():
        adjusted_rate = adjust_defect_rate(row['次品率'], p_total)
        adjusted_rates.append({
            '零配件': row['零配件'],
            '修正次品率': round(adjusted_rate, 4), # 保留四位小数
            '购买单价': row['购买单价'],
            '检测成本': row['检测成本']
        })
    return pd.DataFrame(adjusted_rates)
# 设定总体数量
p_total = 10000
```

```

# 计算修正后的次品率
adjusted_df_problem_3 = calculate_adjusted_defect_rates(df_problem_3,
p_total)
print(adjusted_df_problem_3)
# 计算生产过程中的决策方案
def calculate_decision_scheme(df):
    # 假设的市场售价和调换损失
    market_price = 200
    exchange_loss = 40

    # 决策依据
    decision_results = []
    for _, row in df.iterrows():
        purchase_price = row['购买单价']
        detection_cost = row['检测成本']
        adjusted_defect_rate = row['修正次品率']

        # 计算装配成本
        assembly_cost = 8 # 假设的装配成本

        # 计算总成本
        total_cost = purchase_price + detection_cost + assembly_cost

        # 计算每件产品的净利润
        profit = market_price - total_cost

        # 决策
        decision = '检测' if adjusted_defect_rate > 0.1 else '不检测'
        decision_results.append({
            '零配件': row['零配件'],
            '修正次品率': round(adjusted_defect_rate, 4)
        })

    return pd.DataFrame(decision_results)
# 计算生产过程中的决策方案
decision_df = calculate_decision_scheme(adjusted_df_problem_3)
print(decision_df)

```

问题四遗传算法求解问题二代码

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
# 数据定义
data = {
    '情况': [1, 2, 3, 4, 5, 6],
    '零配件 1 次品率': [0.15, 0.27, 0.15, 0.27, 0.15, 0.09],
    '零配件 2 次品率': [0.15, 0.27, 0.15, 0.27, 0.27, 0.09],
    '零配件 1 购买单价': [4, 4, 4, 4, 4, 4],
    '零配件 1 检测成本': [2, 2, 2, 1, 8, 2],
    '零配件 2 购买单价': [18, 18, 18, 18, 18, 18],
    '零配件 2 检测成本': [3, 3, 3, 1, 1, 3],
    '成品次品率': [0.15, 0.27, 0.15, 0.27, 0.27, 0.09],
    '装配成本': [6, 6, 6, 6, 6, 6],
}

```

```

        , '市场售价': [56, 56, 56, 56, 56, 56],
        , '调换损失': [6, 6, 30, 30, 10, 10],
        , '拆解费用': [5, 5, 5, 5, 5, 40]
    }

# 遗传算法参数
population_size = 50
num_generations = 100
mutation_rate = 0.1
tournament_size = 5
elite_size = 5
# 初始化种群
def initialize_population(pop_size, num_genes):
    return np.random.randint(2, size=(pop_size, num_genes))
# 计算适应度
def calculate_fitness(individual, data):
    total_cost = 0
    total_revenue = 0
    for i in range(len(data['情况'])):
        part1_inspect = individual[i * 4] # 零配件 1 检测
        part2_inspect = individual[i * 4 + 1] # 零配件 2 检测
        product_inspect = individual[i * 4 + 2] # 成品检测
        discard_defective = individual[i * 4 + 3] # 不合格成品拆解

        if discard_defective == 1:
            total_cost += data[, '拆解费用'][i]
            if part1_inspect == 1:
                total_cost += data[, '零配件 1 次品率'][i] * data[, '零配件 1 购
                    买单价'][i] + data[, '零配件 1 检测成本'][i]
                total_revenue += data[, '市场售价'][i] - data[, '调换损失'][i]

            if part2_inspect == 1:
                total_cost += data[, '零配件 2 次品率'][i] * data[, '零配件 2 购
                    买单价'][i] + data[, '零配件 2 检测成本'][i]
                total_revenue += data[, '市场售价'][i] - data[, '调换损失'][i]
        else: # 不拆解的情况
            if part1_inspect == 1:
                total_cost += data[, '零配件 1 次品率'][i] * data[, '零配件 1 购
                    买单价'][i] + data[, '零配件 1 检测成本'][i]
            if part2_inspect == 1:
                total_cost += data[, '零配件 2 次品率'][i] * data[, '零配件 2 购
                    买单价'][i] + data[, '零配件 2 检测成本'][i]
            if product_inspect == 1:
                total_cost += data[, '成品次品率'][i] * data[, '装配成本'][i]
                total_revenue += data[, '市场售价'][i] - data[, '调换损失'][i]
    return total_revenue - total_cost

# 选择操作 - 锦标赛选择
def select(population, fitness):
    selected_indices = np.argsort(fitness)[-tournament_size:]
    return population[selected_indices]
# 交叉操作 - 单点交叉
def crossover(parent1, parent2):
    point = np.random.randint(1, len(parent1))
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2
# 变异操作
def mutate(individual, mutation_rate):
    mutation_mask = np.random.rand(len(individual)) < mutation_rate

```

```

        individual[mutation_mask] = 1 - individual[mutation_mask]
    return individual
# 局部搜索 - 使用邻域搜索
def local_search(individual):
    best = individual.copy()
    best_fitness = calculate_fitness(best, data)
    for i in range(len(best)):
        new_individual = best.copy()
        new_individual[i] = 1 - new_individual[i]
        new_fitness = calculate_fitness(new_individual, data)
        if new_fitness > best_fitness:
            best, best_fitness = new_individual, new_fitness
    return best
# 遗传算法主程序
def genetic_algorithm(data):
    num_genes = len(data['情况']) * 4
    population = initialize_population(population_size, num_genes)
    fitness_history = []
    for generation in range(num_generations):
        fitness = np.array([calculate_fitness(ind, data) for ind in
                           population])
        selected = select(population, fitness)
        new_population = []
        while len(new_population) < population_size - elite_size:
            parent1, parent2 = selected[np.random.choice(len(selected),
                                                          2, replace=False)]
            child1, child2 = crossover(parent1, parent2)
            new_population.append(mutate(child1, mutation_rate))
            new_population.append(mutate(child2, mutation_rate))
        elite_indices = np.argsort(fitness)[-elite_size:]
        elite = population[elite_indices]
        new_population.extend(elite)
        population = np.array(new_population)
        # 记录适应度历史
        fitness_history.append(np.max(fitness))
        # 增加多样性 - 随机替换一些个体
        num_replacements = int(population_size * 0.1)
        population[:num_replacements] = initialize_population(
            num_replacements, num_genes)
    final_fitness = np.array([calculate_fitness(ind, data) for ind in
                             population])
    best_index = np.argmax(final_fitness)
    best_individual = population[best_index]
    # 对最终结果进行局部优化
    best_individual = local_search(best_individual)
    best_fitness = calculate_fitness(best_individual, data)
    return best_individual, best_fitness, fitness_history

# 运行遗传算法并输出结果
best_decision, best_score, fitness_history = genetic_algorithm(data)
# 打印结果
print("最佳决策方案 (编码): ")
results = []
profits = []
for i, situation in enumerate(data['情况']):
    decision = best_decision[i * 4:(i + 1) * 4]
    # 计算每种情况的利润
    total_cost = 0

```

```

total_revenue = 0
if decision[3] == 1: # 不合格成品拆解
    total_cost += data[, 拆解费用][i]
    if decision[0] == 1:
        total_cost += data[, 零配件 1 次品率][i] * data[, 零配件 1 购买单
            价][i] + data[, 零配件 1 检测成本][i]
        total_revenue += data[, 市场售价][i] - data[, 调换损失][i]
    if decision[1] == 1:
        total_cost += data[, 零配件 2 次品率][i] * data[, 零配件 2 购买单
            价][i] + data[, 零配件 2 检测成本][i]
        total_revenue += data[, 市场售价][i] - data[, 调换损失][i]
else: # 不拆解的情况
    if decision[0] == 1:
        total_cost += data[, 零配件 1 次品率][i] * data[, 零配件 1 购买单
            价][i] + data[, 零配件 1 检测成本][i]
    if decision[1] == 1:
        total_cost += data[, 零配件 2 次品率][i] * data[, 零配件 2 购买单
            价][i] + data[, 零配件 2 检测成本][i]
    if decision[2] == 1:
        total_cost += data[, 成品次品率][i] * data[, 装配成本][i]
        total_revenue += data[, 市场售价][i] - data[, 调换损失][i]
profit = total_revenue - total_cost
profits.append(profit)
results.append([
    situation,
    decision[0], # 零配件 1 检测
    decision[1], # 零配件 2 检测
    decision[2], # 成品检测
    decision[3], # 拆解
])
# 转换为 DataFrame 并打印
df = pd.DataFrame(results, columns=[, 情况, , 零配件 1 检测, , 零配件 2 检
    测, , 成品检测, , 拆解])
print("最佳适应度值: ", best_score)
# 绘制适应度变化图
plt.plot(range(len(fitness_history)), fitness_history)
plt.xlabel('代数')
plt.ylabel('适应度值')
plt.title('适应度变化图')
plt.savefig('适应度变化曲线.png', dpi=500)
plt.show()

```

问题四遗传算法求解问题三代码

```

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
# 数据定义
data = {
    '零配件': [
        {'次品率': 0.15, '购买单价': 2, '检测成本': 1},
        {'次品率': 0.15, '购买单价': 8, '检测成本': 1},
        {'次品率': 0.15, '购买单价': 12, '检测成本': 2},
        {'次品率': 0.15, '购买单价': 2, '检测成本': 1},
        {'次品率': 0.15, '购买单价': 8, '检测成本': 1},
        {'次品率': 0.15, '购买单价': 12, '检测成本': 2}
    ]
}

```

```

    7: {'次品率': 0.15, '购买单价': 8, '检测成本': 1},
    8: {'次品率': 0.15, '购买单价': 12, '检测成本': 2},
},
'半成品': {
    1: {'次品率': 0.15, '装配成本': 8, '检测成本': 4, '拆解成零件': {1: 2, 2: 1}},
    2: {'次品率': 0.15, '装配成本': 8, '检测成本': 4, '拆解成零件': {3: 1, 4: 1}},
    3: {'次品率': 0.15, '装配成本': 8, '检测成本': 4, '拆解成零件': {5: 2, 6: 1}},
},
'成品': {
    '次品率': 0.15,
    '装配成本': 8,
    '拆解费用': 10,
    '拆解成半成品': {1: 2, 2: 1}
},
'市场售价': 200,
'调换损失': 40
}

# 遗传算法参数
population_size = 100
num_generations = 200
mutation_rate = 0.15
tournament_size = 5
elite_size = 10
# 初始化种群
def initialize_population(pop_size, num_genes):
    return np.random.randint(2, size=(pop_size, num_genes))
# 计算适应度
def calculate_fitness(individual, data):
    total_cost, total_revenue, undetected_parts_loss = 0, 0, 0

    # 零配件成本计算
    for i in range(8):
        inspect = individual[i]
        if i + 1 in data['零配件']:
            cost = (data['零配件'][i + 1]['次品率'] * data['零配件'][i + 1]['购买单价'] +
                    data['零配件'][i + 1]['检测成本']) * inspect
            total_cost += cost
            if inspect == 0:
                undetected_parts_loss += data['零配件'][i + 1]['检测成本']

    # 半成品成本计算
    for j in range(3):
        inspect = sum(individual[8 + 3 * j: 11 + 3 * j]) # 计算半成品的相关零件是否检测
        if j + 1 in data['半成品']:
            half_product = data['半成品'][j + 1]
            cost = (half_product['次品率'] * half_product['装配成本'] +
                    half_product['检测成本']) * inspect
            total_cost += cost

    # 拆解零件减少成本
    for part_id, amount in half_product['拆解成零件'].items():
        if part_id in data['零配件']:
            part_cost = data['零配件'][part_id]['次品率'] * data['零配件'][part_id]['购买单价'] * amount
            total_cost -= part_cost

```

```

        total_revenue += data[‘市场售价’] - data[‘调换损失’]
        if individual[part_id - 1] == 0: # 未检测零配件的拆解损
            失
            undetected_parts_loss += data[‘零配
件’][part_id][‘检测成本’]
# 成品成本计算
product_inspect = individual[17]
cost = (data[‘成品’][‘次品率’] * data[‘成品’][‘装配成
本’]) * product_inspect
total_cost += cost
total_revenue += data[‘市场售价’] - data[‘调换损失’]
# 成品拆解
if individual[18] == 1:
    total_cost += data[‘成品’][‘拆解费用’]
    for semi_id, amount in data[‘成品’][‘拆解成半成品’].items():
        if semi_id in data[‘半成品’]:
            semi_cost = (data[‘半成品’][semi_id][‘次品率’] * data[‘半成
品’][semi_id][‘装配成本’]) * amount
            total_cost -= semi_cost
            total_revenue += data[‘市场售价’] - data[‘调换损失’]

        for part_id, part_amount in data[‘半成品’][semi_id][‘拆解
成零件’].items():
            if part_id in data[‘零配件’]:
                part_cost = (data[‘零配件’][part_id][‘次品
率’] * data[‘零配件’][part_id][‘购买单
价’]) * part_amount
                total_cost -= part_cost
                total_revenue += data[‘市场售价’] - data[‘调换损
失’]
# 总成本考虑未检测零配件的损失
total_cost += undetected_parts_loss
return total_revenue - total_cost

# 选择操作
def select(population, fitness, tournament_size):
    if tournament_size > len(population):
        tournament_size = len(population)
    selected_indices = np.random.choice(len(population),
                                         tournament_size, replace=False)
    selected_fitness = fitness[selected_indices]
    best_index = selected_indices[np.argmax(selected_fitness)]
    return population[best_index]

# 交叉操作
def crossover(parent1, parent2):
    point = np.random.randint(1, len(parent1))
    return np.concatenate((parent1[:point], parent2[point:])), np.
        concatenate((parent2[:point], parent1[point:]))

# 变异操作
def mutate(individual, mutation_rate):
    mutation_mask = np.random.rand(len(individual)) < mutation_rate
    individual[mutation_mask] = 1 - individual[mutation_mask]
    return individual

# 遗传算法主函数
def genetic_algorithm(population_size, num_genes, num_generations,
                      mutation_rate, elite_size, tournament_size, data):
    population = initialize_population(population_size, num_genes)
    best_fitness_over_time = []
    avg_fitness_over_time = []
    for generation in range(num_generations):

```

```

fitness = np.array([calculate_fitness(ind, data) for ind in
    population])
best_fitness_over_time.append(np.max(fitness))
avg_fitness_over_time.append(np.mean(fitness))
elite = np.array([select(population, fitness, tournament_size)
    for _ in range(elite_size)])
new_population = elite.copy()
while len(new_population) < population_size:
    parent1 = select(population, fitness, tournament_size)
    parent2 = select(population, fitness, tournament_size)
    child1, child2 = crossover(parent1, parent2)
    child1 = mutate(child1, mutation_rate)
    child2 = mutate(child2, mutation_rate)
    new_population = np.vstack((new_population, child1, child2))
population = new_population[:population_size]
best_fitness = np.max(fitness)
best_individual = population[np.argmax(fitness)]
return best_individual, best_fitness, best_fitness_over_time,
avg_fitness_over_time

# 运行遗传算法
num_genes = 19
best_individual, best_fitness, best_fitness_over_time,
avg_fitness_over_time = genetic_algorithm(
    population_size, num_genes, num_generations, mutation_rate,
    elite_size, tournament_size, data
)
# 输出结果
print("最佳个体: ", best_individual)
print("最佳适应度: ", best_fitness)

# 翻译为决策表格
def translate_to_decision(best_individual, data):
    decisions = {
        '零配件': {i + 1: '检测' if best_individual[i] == 1 else '不检测',
                  for i in range(8)},
        '半成品': {j + 1: '检测' if sum(best_individual[8 + 3 * j: 11 + 3
            * j]) > 0 else '不检测',
                  for j in range(3)},
        '成品检测': '检测' if best_individual[17] == 1 else '不检测',
        '成品拆解': '拆解' if best_individual[18] == 1 else '不拆解',
    }
    # 计算利润
    profit = calculate_fitness(best_individual, data)
    decisions['利润'] = profit
    return decisions
decisions = translate_to_decision(best_individual, data)
# 保存决策到 Excel
df = pd.DataFrame(decisions)
df.to_excel('best_individual_decision.xlsx', index=False)
print("最佳个体决策已保存到 'best_individual_decision.xlsx'。")
# 绘制适应度变化图
plt.figure(figsize=(12, 6))
plt.plot(best_fitness_over_time, label='最佳适应度')
plt.plot(avg_fitness_over_time, label='平均适应度')
plt.xlabel('代数')
plt.ylabel('适应度')

```

```
plt.title('适应度变化图')
plt.legend()
plt.grid(True)
plt.savefig('问题三适应度变化曲线.png', dpi=500)
plt.show()
```