基于蒙特卡洛树搜索算法求解多阶段生产过程中的马尔可夫决策模型

摘要

随着电子产品的日益普及，电子产品需求量也随之增加, 企业从零配件采购到成品销售全过程的决策影响整个企业的经济效益。本文针对企业生产流程中各阶段的决策问题，基于抽样检测和多阶段目标规划思想，以企业生产总成本为目标建立了多阶段马尔可夫决策动态模型，并使用蒙特卡洛搜索算法对模型进行求解，得到生产过程中企业各阶段决策收益最大方案。

针对问题一，本文建立基于**抽样检测**的**假设检验模型**，在检测样本足够大的前提下，用正态分布来近似二项分布，初步得到在95％的信度下拒收次品率超过标称值(10％)的批次，所需的样本大小约为138；在90％的信度下接受次品率不超过标称值的批次，所需的样本大小约为98。再使用**序贯分析法**对假设检验模型优化，优化之后所需的样本量分别降低为94和67，降低了检测成本。

针对问题二，据题表1将生产过程划分为**多个阶段**，建立**多阶段成本-利润决策模型**，涵括零配件购买及检测、产品装配、成品检测和不合格品处理等多个环节，采用**分支定界法求**解组合优化问题。结果表明，各情形最优决策路径都为不检测。其中，情形六的收益最大为25.00元，通过**可视化结果**得出企业总体盈利，但有一定幅度波动(18.00－25.00元)，同时发现较高的次品率通常与较高的成本和较低的收入相关。

针对问题三，据题表2和图1，面对包含多道工序和多个零配件的复杂生产系统，本文建立了**多阶段马尔可夫决策模型**(*MDP*)，并使用**蒙特卡洛树搜索方法**(*MCTS*)来高效求解具有高度不确定性和复杂动态环境问题。求解结果表明，对于所有零配件，半成品及成品采取不检测，此决策方案的预期节点平均收益在150以上。而后通过迭代所有情形得到理论最大收益为164.8元，进一步验证决策的正确性。

针对问题四，引入抽样检测次品率的不确定因素，进一步优化模型建立**贝叶斯自适应马尔可夫决策过程**（*BAMDP*），基于第一问**序贯分析法**计算出三种半成品的次品率分别为21.92％，24.99％，15.94％，成品的次品率为50.77％。经**蒙特卡洛树搜索自适应算法**求出最优决策路径依然为不进行任何检测，同时得到最大收益为104.08元。

最后,我们对提出的模型进行全面的评价：本文的模型贴合实际,能合理解决提出的问题,具有实用性强,算法效率高等特点，在处理复杂生产系统的决策优化问题上展现了强大的能力。

关键词：抽样检测 分支定界法 蒙特卡洛树搜索方法 贝叶斯自适应马尔可夫决策过程

目录

一、 问题重述

1.1 问题背景

在电子产品制造过程中，企业需采购某些关键零部件并进行产品组装，为了在确保成品质量的同时控制成本，企业可通过抽样检测来监控零部件质量。然而，抽样检测、成品拆解以及返工操作均会带来一定的成本负担。因此，如何在生产流程中综合优化检测策略、返工处理与成本结构，成为企业面临的核心挑战。

这些核心问题构成了一个多变量、随机性强且涉及多重约束的复杂系统，如果通过概率模型、最优化算法等数学工具加以系统求解，可以帮助企业制定出具有实际可操作性的生产质量控制与成本优化方案，为企业的质量控制和成本管理提供理论依据和决策支持。

1.2 问题重述

问题一：在次品率标称值为10％提下，企业需要设计一个抽样检测方案，确保在95％水平下次品率超标时拒绝批次，在90％水平下次品率合格时接受批次，以平衡质量风险与检测成本。

问题二：在已知两种零配件和成品次品率的前提下，企业需要在不同阶段针对零配件和成品做出一系列决策。企业需要在零配件是否进行检测，装配好的成品是否进行检测，不合格的成品是否进行拆解处理等方面做出决策，综合评估每种决策的可行性和效益，最终提出最优策略并给出决策的依据与相应的指标结果。

问题三：在实际生产过程中，企业可能面临多个生产工序以及多个零配件的管理与组装问题。给定m道生产工序、n个零配件及其次品率、相关检测成本和处理成本，企业需要确定每个阶段的生产决策。针对每个零配件，决定是否进行采购、检测和组装；针对半成品和成品，决定是否进行检测；针对检测出的不合格品，决定是进行返工处理还是直接报废。总体上，企业需要依据表2中给出的数据，分析不同决策对生产成本、产品质量的影响，提出具体的生产决策方案，并给出相关的决策依据和指标分析。

问题四：如果实际中零配件、半成品和成品的次品率是通过抽样检测得出的。基于抽样检测结果，需要对整个生产流程的各个环节进行重新评估和优化，重新对问题二和问题三的决策进行优化，包括采购、组装以及不合格品处理的决策。最终提出基于抽样检测数据的改进方案和决策依据。

二、 问题分析

2.1 问题一的分析

问题一中要求确立合理的零配件检测方案。可以归结为统计学中的假设检验问题，已知该批次零配件次品率标称值为10％抽取的样本量n与标称值q_0之间满足$n \cdot p_0 \geq 5$和$n \cdot (1-p_0) \geq 5$时，使用正态分布来近似二项分布，在题目给定的信度下，采用双边检验，分别对95％和90％样本量进行计算，得出样本量 n和接受/拒绝的次品数量上限 ，最终得出抽样检测方案。

2.2 问题二的分析

问题二中要求对生产流程中的各个阶段作出决策。已知两种零配件和成品次品率，企业需要对零配件和已装配的成品是否进行检测以及不合格的产品是否进行拆解进行决策，根据表1所给出的企业在生产中遇到的六种情况，对于每一种情形建立多阶段逐

步优化检测、装配和处理决策，最终以最低总成本为目标函数，据此建立多阶段成本-利润决策模型，利用分支界定法求得六种情况下企业的指标结果。

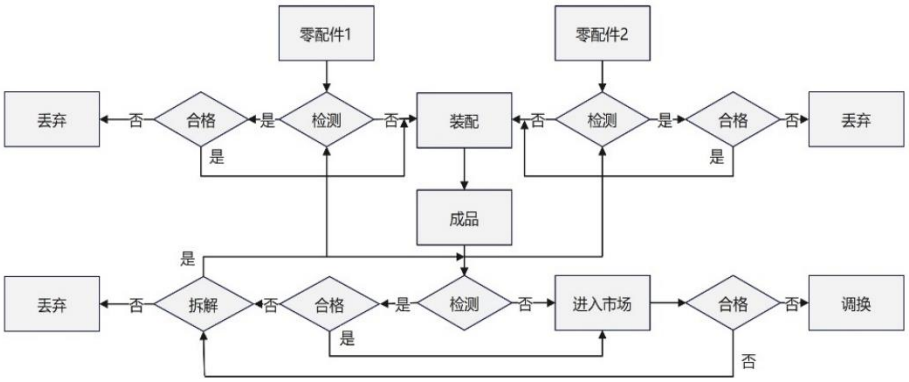

图 1　企业生产流程

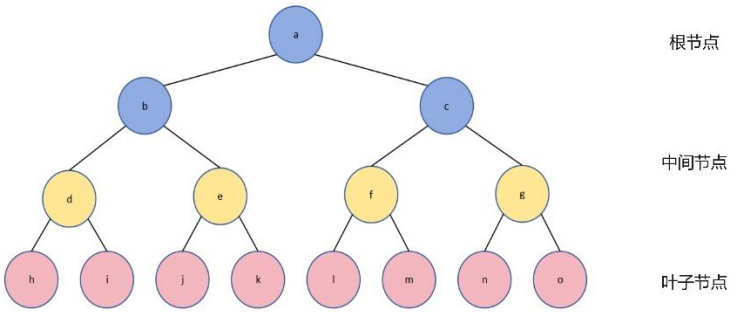

图 2　分支定界法示意图

2.3 问题三的分析

问题三中要求在多阶段的生产过程中，对多个零配件，半成品以及成品进行决策处理。为了应对复杂生产系统中的决策挑战，我们引入了马尔可夫决策过程（MDP）来建模。该模型旨在通过状态空间、行动空间、状态转移概率和奖励函数的设计，为企业提供可靠的决策参考。

首先，模型的状态空间涵盖了生产系统中的所有关键元素，包括零配件、半成品以及成品的不同决策。这些决策代表了生产线中各环节的不同阶段，以确保系统对整个生产过程的全面覆盖。

其次，行动空间是指在每一个特定的状态下，企业可以采取的可行决策。每个决策的选择会对系统未来的状态产生影响，具体体现在状态转移概率上。通过准确地设定状态转移概率，模型能够模拟决策实施后的可能结果。

最后，奖励函数的设计则直接反映了每个决策的经济效益。通过评估不同决策对利润、成本和次品率的影响，奖励函数帮助企业识别最优决策路径。特别是在涉及到次品率时，次品不仅影响直接成本，还会波及企业的长期收入。因此，次品率在模型中需要得到高度关注。

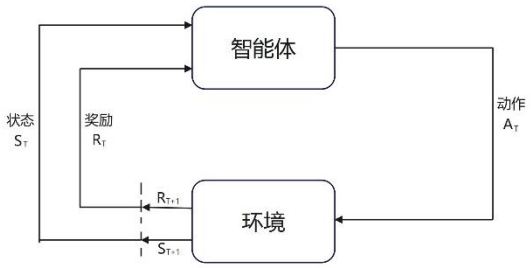

图 3 马尔可夫决策示意图

2.4 问题四的分析

问题四中，我们需要在考虑抽样检测零件次品率不确定的前提下，进一步完善和扩展之前的决策模型。问题的核心在于如何在不确定的质量信息下做出最优生产决策。鉴于该生产过程中决策的不确定性，采用结合马尔可夫决策过程与贝叶斯推断的混合模型以适应系统的动态特性以及应对抽样检测引发的次品率不确定性。

在构建模型时，关键因素包括：如何将抽样检测结果整合到决策中；如何基于不同置信水平更新产品质量的信念；以及在不确定性条件下制定最佳的生产与检测策略。

三、 模型假设与符号说明

3.1 模型基本假设

- 假设所有零件的品质相同；
- 假设合格的产品生产后无滞销现象；
- 假设生产产品的各个环节互不影响；
- 假设市场对产品质量的反应是可预测的；
- 假设装配和拆解过程中不会对零件造成损害。

3.2 符号说明

符号	物理意义
$c_{p1}^{c}, c_{p2}^{c}, ..., c_{pi}^{c}$	零配件的单价
$c_{d1}^{c}, c_{d2}^{c}, ..., c_{di}^{c}$	零配件的检测成本
c_{d}^{f}	成本检测成本
c_{a}^{f}	成品拆解成本
c_{f}^{f}	装配成本
c_{s}	调换成本
$p_{1}^{c}, p_{2}^{c}, ..., p_{i}^{c}$	零配件的次品率
p^{f}	成品的次品率
s^{f}	成品售价
$n_{1}^{c}, n_{2}^{c}, ..., n_{i}^{c}$	零配件的数量
n^{f}	成品的数量
$P(s'\|s,a)$	状态转移概率
$R(s,a)$	奖励函数
γ	折扣因子

四、 模型建立与求解

4.1 问题一模型建立与求解

4.1.1 抽样检测假设检验模型建立

假设检验（Hypothesis Testing）是一种统计学方法，旨在通过样本数据对总体做出推断，从而判断假设是否成立。它广泛应用于质量检测、市场调查、实验分析等领域。

(1)确定假设

原假设：H_0：零配件的次品率$p \leqslant p_0$，即零配件次品率未超过标称值；

备择假设：H_1：零配件的次品率$p > p_0$，即零配件次品率超过标称值。

此处p_0为供应商声称零配件次品率的标称值，p为零配件真实的次品率，通过抽样来估计p，并由此确定是否接受这批零配件。

(2)抽样方案的确立

抽样检测可以看作是二项分布问题，假设从一批零配件中抽取n个样品，其中次品数量X服从二项分布：

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

其中p为真实的次品率，样本量为n。

根据大数定理，当p较大时，二项分布可以近似为正态分布：

$$\hat{p} \sim N\left(p, \frac{p(1-p)}{n}\right) \tag{2}$$

其中$\hat{p}$为样本次品率。

(3)确定样本量n

我们希望通过尽可能少的样本数来实现给定的信度。根据正态分布的性质在给定显著性水平α的情况下，临界值$z_{\alpha/2}$可以从标准正态表中查得。

对于拒收标准（95%的信度），我们使用95%置信区间，即：

$$z_{0.05/2} \approx 1.96$$

对于接收标准（90%的信度），我们使用90%置信区间，即：

$$z_{0.10/2} \approx 1.645$$

对于二项分布，次品率的置信区间可以表示为：

$$\hat{p} \pm z_{\alpha/2} \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{3}$$

由此可以计算需要的样本量。

(4)拒收和接受规则

i.拒收规则:如果在抽样过程中观测到的次品率α超10%的置信区间上限，即

$$\hat{p} > 10\% + z_{0.05/2} \cdot \sqrt{\frac{0.1 \times 0.9}{n}} \tag{4}$$

则在95%的信度下拒收该批次零配件。

ii.接收规则:如果在抽样过程中观测到的次品率α不超过10%的置信区间下限，即

$$\hat{p} \leq 10\% - z_{0.10/2} \cdot \sqrt{\frac{0.1 \times 0.9}{n}} \tag{5}$$

则在90%的信度下接收该批次零配件。

4.1.2 正态近似求解假设检验模型

(1)样本量n计算：根据正态近似和二项分布，样本量n可以通过以下公式计算：

$$n = \frac{Z_{\alpha/2}^2 \cdot p_0(1-p_0)}{E^2} \leftarrow \tag{6}$$

其中：

- $z_{\alpha/2}$为标准正态分布在置信水平α下的临界值；
- p_0为零件次品率的标称值；
- E是允许的误差；

假定允许接受的误差幅度为5%，由此可以计算出：

确定拒收的样本量n：我们希望在95%的信度下拒收零配件，以5%的误差容忍度来估计次品率，那么$z_{0.05/2}=1.96$。根据公式：

$$n = \left(\frac{z_{\alpha/2}}{E}\right)^2 \cdot \hat{p}(1-\hat{p}) \tag{7}$$

取次品率的估计值为10%：

$$n = \left(\frac{1.96}{0.05}\right)^2 \cdot 0.1 \cdot 0.9 \approx 138 \tag{8}$$

因此，需要抽样大约138零配件来检测是否在95%的信度下可以拒收该批次。

确定接受的样本量n：类似的，对于90%的信度和5%的误差容忍度，有$z_{0.10/2}=1.645$。根据公式得：

$$n = \left(\frac{1.645}{0.05}\right)^2 \cdot 0.1 \cdot 0.9 \approx 98 \tag{9}$$

因此，需要抽样大约98个零配件来判断是否在90%的信度下可以接收该批次。

(2)最大次品数量的确定：

- 在95%置信水平下，可以容忍的次品率标称值为10%。

$$c_{95\%} = \lceil n_{95\%} \cdot p_0 \rceil = \lceil 138 \cdot 0.1 \rceil = 13.8$$

所以，在95%信度下，最大允许次品数量$c_{95\%}=13.8$，向下取整为13。

- 在90%置信水平下，可以容忍的次品标称值为10%。

$$c_{90\%} = \lceil n_{90\%} \cdot p_0 \rceil = \lceil 98 \cdot 0.1 \rceil = 9.8$$

所以，在90%信度下，最大允许次品数量$c_{90\%}=9.8$，向下取整为9。

(3)结果分析：

- **拒收(95%)信度：**抽取138个样本，若不合格数超过13个，则拒收该批次。
- **接受(90%)信度：**抽取98个样本，若不合格数超过 9 个，则接受该批次。

4.1.3 序贯分析法优化假设检验模型

序贯分析法（*SequentialAnalysis*）通过在样本的每一步进行统计检验来尽早做出决策，进而减少平均样本数量（*ASN*）。*Wald* 的序贯概率比检验（*SPRT*）是这种方法的经典之一。相较于传统的固定样本量方法，*SPRT* 能在保证相同的显著性水平和检验功效下，通常用更少的样本数完成检验。序贯分析法的一般步骤为：

(1)假设：

- 零假设：H_0：次品率$p \le p_0$(例如标称值$p_0=0.1$)。
- 备择假设：H_1：次品率$p > p_1$(一个比p_0大的值，比如$p_1=0.12$)

(2)序贯检验准则：每次抽取一个样本后，计算样本的似然比：

$$\Lambda_n = \prod_{i=1}^{n} \frac{P(X_i|H_1)}{P(X_i|H_0)} \tag{10}$$

其中：

• X_i 是第 i 个样本的结果，$P(X_i|H_1)$ 和 $P(X_i|H_0)$ 分别表示假设 H_1 和 H_0 下观测值的概率。

(3)确立决策边界：

• 当 $\Lambda_n \geq A$ 时，拒绝 H_0，接受 H_1。

• 当 $\Lambda_n \leq B$ 时，接受 H_0，拒绝 H_1。

• 当 $B < \Lambda_n < A$ 时，继续抽样。

上下限 A 和 B 由显著性水平 α 和检验功效 β 确定：

$$A = \frac{1-\beta}{\alpha}, \quad B = \frac{\beta}{1-\alpha} \tag{11}$$

(4)结果分析：

根据计算，得出优化后所需抽取的样本数量有所减少，进一步增大企业生产效益。

4.1.4 可视化结果分析

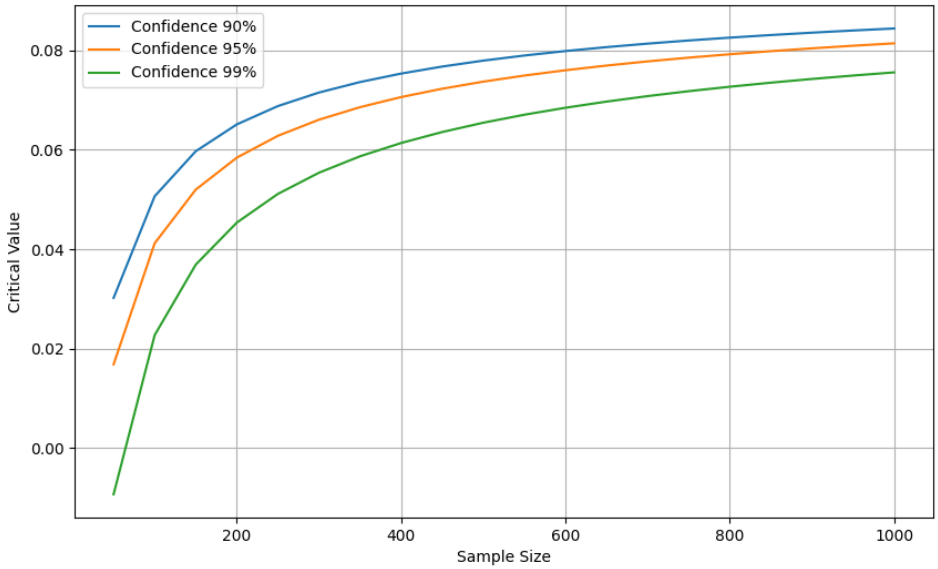

图 4 零配件接受和拒收能力分析

由图分析可知：

(1) 图中显示了关键值随样本大小增加而逐渐减少。这反映了一个核心统计原理：当样本量增加时，估计的精度提高，导致标准误差减小。因此，置信区间变窄，需要的关键值也随之降低。

(2) 图中包含三条曲线，分别代表90%，95%，和99%的置信水平。可以观察到，随着置信水平的提高，关键值也相应增加。这是因为更高的置信水平需要更宽的置信区间以包含真实参数，因此，需要更大的关键值来确保该置信水平。

(3) 在样本大小较小的情况下，关键值的下降非常显著。这表明在小样本情况下增加少量样本可以显著提高统计推断的精确性。随着样本大小继续增加，关键值的下降速

度减缓，尤其是当样本大小超过一定数值时，关键值的进一步下降较为平缓。这表明在较大的样本大小后，进一步增加样本量带来的精度提高有限。

4.2 问题二模型建立与求解

4.2.1 多阶段成本-利润决策模型建立

根据问题二的分析过程，要求对题表1中的情形给出具体决策方案，并给出决策依据及相应的指标结果。对于生产中零配件购买成本，零配件检测，成本检测，成品装配以及不合格产品拆解，这些因素共同构成了决策的约束条件和优化目标。最终目标要求找到一种最优的决策组合，使得企业能够在保证产品质量的前提下收益最大化。

决策变量

定义$i=1,2,\cdots n$，假设所有零配件数量相同，即$n_1^c=n_2^c=...=n_i{}^c$，$x_1,x_2,...,x_i$表示是否对零配件进行检测。y表示是否对成品进行检测，z表示是否对不合格产品进行拆解。

目标函数

1)零配件购买成本：

$$C_p^c=\sum_{i=1}^{i}n_i^c c_{pi}^c \tag{12}$$

2)零配件检测成本：

$$C_d^c=\sum_{i=1}^{i}n_i^c x_i c_{di}^c \tag{13}$$

3)成品检测成本：

$$C_d^f=c_d^f y((n^c-xp^c n^c)\times(1-p^f)) \tag{14}$$

4)成品拆解成本：

$$C_a^f=c_a^f z((n^c-xp^c n^c)\times p^f) \tag{15}$$

5)成品装配成本：

$$C_s^f=c_s^f\times((n^c-xp^c n^c)\times(1-p^f)) \tag{16}$$

6)调换成本：

$$C_s=c_s y((n^c-xp^c n^c)\times p^f) \tag{17}$$

7)利润：

$$S=s^f\times((n^c-xp^c n^c)\times(1-p^f)) \tag{18}$$

8)成品数量为：

$$n^f=(n^c-xp^c n^c)\times(1-p^f) \tag{19}$$

其中：

- $n^c-xp^c n^c$是检测合格后的零配件数量；
- $(1-p^f)$是成品的合格率。

整合所有成本项和利润，目标是最小化总成本：

$$\min Z=n^c\left(\sum_{i=1}^{i}c_{pi}^c+\sum_{i=1}^{i}x_i c_{di}^c\right)+((n^c-x_i p^c n^c)\times(1-p^f))(c_d^f y+c_a^f zp^f+c_s yp^f-s^f+c_s^f) \tag{20}$$

约束条件

- x_i,y,z为0或1，确保决策的唯一性和可执行性；
- $x_i\in\{0,1\}$表示是否检测零配件；

• $y,z\in\{0,1\}$ 表示是否检测成品和是否进行拆解。

模型总述

$$\min Z=n^c\left(\sum_{i=1}^{i}c_{pi}^c+\sum_{i=1}^{i}x_i c_{di}^c\right)+((n^c-x_i p^c n^c)\times(1-p^f))(c_d^f y+c_a^f z p^f+c_s y p^f-s^f+c_s^f) \tag{21}$$

$$s.t.\begin{cases}x_i\in\{0,1\}\ \ i=1,2,\cdots,n\\ y,z\in\{0,1\}\end{cases}$$

4.2.2 多阶段成本-利润模型分支定界法求解

分支定界法（Branch and Bound, B&B）是一种求解组合优化问题的通用算法，尤其适用于离散优化问题。分支定界法通过系统地探索解空间，并在搜索过程中使用剪枝策略来减少不必要的计算。该算法的核心思想是将问题划分为更小的子问题，然后通过界限来排除无效的子问题。算法的主要流程如下：

Step 1: 初始化

设定初始问题的解空间，并计算一个初始界限。根据题目，界限可设以理论效益最低为下限，理论收益最高为上限。

Step 2: 分支

将当前问题划分成多个子问题，即将总收益最大划分为每个环节所需的成本最少。每个子问题对应解空间树中的一个节点，代表解空间的一个划分。

对每个新生成的子问题，计算其界限（最大可能收益或最小成本）。如果该界限不如当前最优解，或不满足问题的约束条件，则可以剪枝，即放弃继续搜索该子问题。
如果子问题对应的界限比当前已知的最优解更优，则保留该子问题，并进一步分支。

Step 4: 选择下一个分支节点

选择未处理的子问题中的一个节点，继续重复分支和定界的步骤。常用策略包括深度优先（DFS）、广度优先（BFS）以及最优界限优先的选择策略。

Step 5: 终止条件

当没有更多的子问题需要处理时，算法终止，返回当前的最优解。如果所有可能的子问题都被处理或剪枝，那么当前最优解即为全局最优解。

最终解得结果如下表：

	最优决策路径	最大收益
情况一	$[0,0,0,0]$	23.00
情况二	$[0,0,0,0]$	18.00
情况三	$[0,0,0,0]$	23.00
情况四	$[0,0,0,0]$	18.00
情况五	$[0,0,0,0]$	23.00
情况六	$[0,0,0,0]$	25.00

表 1　各情形最优决策路径及最大收益

4.2.3 可视化结果分析

对于两种零配件，每种零配件可以选择检测或不检测，对于零配件的决策一共有 $2^2=4$ 种。对于每个成品，可以选择检测或不检测，同时对于不合格的成品可以选择拆卸或不拆卸，因此，对于成品的决策也有 4 种，因此总策略数一共有 16 种，将所有决策结果可视化，并求解出每种情形下的最优决策路径。

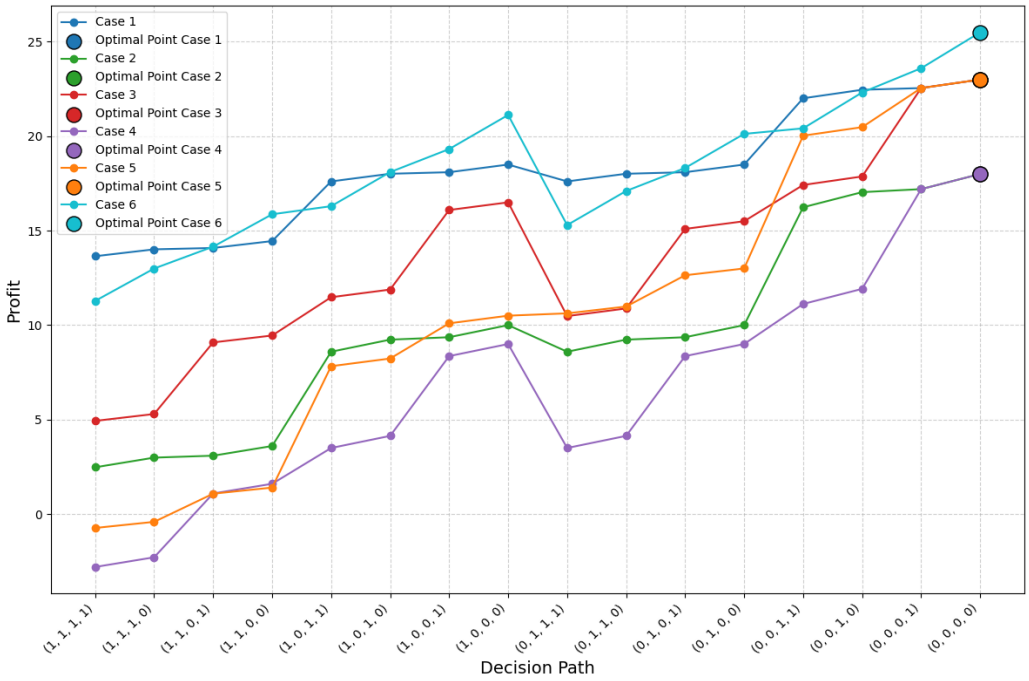

图 5　不同场景下的决策结果可视图

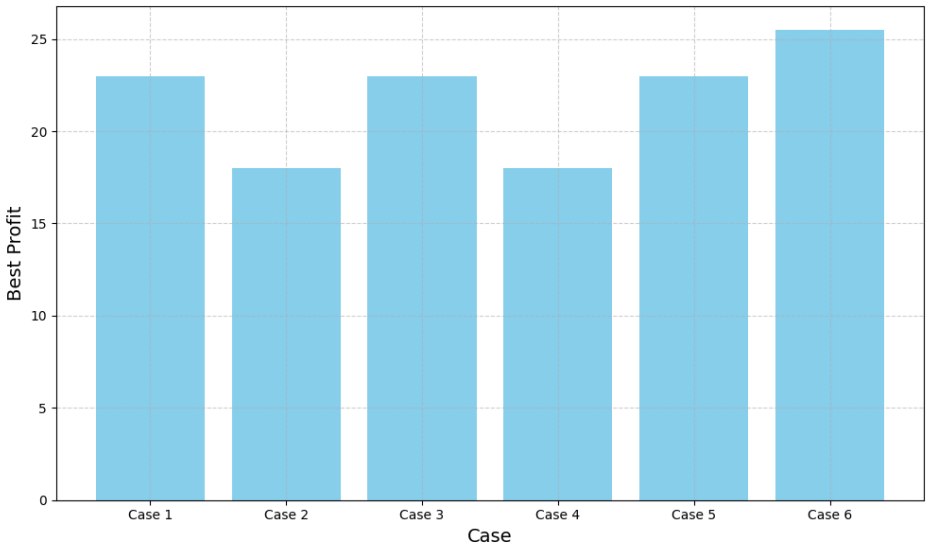

图 6　最优决策路径柱状图

由分支定界法并将结果可视化，得出了六种情况下的最优决策路径。

根据图4，题表2给出的所有情形中，在企业生产中零配件一和零配件二的检测，成品检测以及不合格品是否拆解的四个阶段决策过程中，当所有环节都不进行检测时，即决策变量为$(0,0,0,0)$时，各个情况的总利润都达到最大值，每种情况的收益值由18.00元到25.00元不等，其中情况六的收益最高。

由此得出，收入的变动通常比成本的变动大，这表明企业的利润主要受到收入变化的影响。多数情况下，企业的总收入明显高于总成本，保证了各种条件下的盈利性，尽管盈利水平存在显著的差异。次品率是成本和收入的一个重要决定因素，较低的次品率通常与较低的成本和较高的收入相关，较高的次品率通常与较高的成本和较低的收入相关。

4.3 问题三模型建立与求解

4.3.1 多阶段马尔可夫决策模型建立

为了建立马尔可夫决策过程（*MDP*）模型并找到最优策略π^*，需要明确*MDP*的五个关键元素：状态空间、行动空间、状态转移概率、奖励函数和策略。结合之前的分析，以下是完整的*MDP*模型构建及求解最优策略的步骤：

(1)*MDP*模型的定义：

MDP 模型通常用一个四元组(S,A,P,R)表示，其中：

•状态空间S：包含生产过程中所有可能的状态，定义为一个 12 维向量，分别对应 8 个零配件、3 个半成品和 1 个成品的状态。

• 行动空间A：包含在每个状态下可以采取的所有可能行动，包括检测、不检测、装配、销售和拆解。

•状态转移概率$P(s'|s,a)$：描述了在状态s下采取行动a后，系统转移到状态s'的概率。

• 奖励函数$R(s,a)$：定义为在状态s下采取行动a所获得的收益或成本，目标是最大化长期回报。

(2)参数定义：

i.状态空间S：

每个状态S表示为：

$$s=(s_1,s_2,...,s_8,s_9,s_{10},s_{11},s_{12}) \tag{22}$$

其中：

• s_1到s_8：分别表示8个零配件的状态，包括数量、质量（合格或次品）、检测状态（已检测或未检测）。

• s_9到s_{11}：分别表示3个半成品的状态，包含当前工序进度和质量状态。

• s_{12}：表示成品的状态，包括数量、质量状态、是否准备进入市场等。

ii.行动空间A：

行动空间包括以下操作：

• 检测：Inspect_i对指定的零配件、半成品或成品进行检测。

•不检测：DoNotInspect_i不对指定的零配件、半成品进行检测，直接进行下一步。

• 装配：$\text{Assemble}_{i,j}$将指定的零配件组合成半成品或将半成品组合成成品。

• 销售：Sell_{12}销售成品。

• 拆解：Disassemble_{12}拆解不合格成品，回收零配件以便重新使用。

iii.状态转移概率$P(s'|s,a)$：

状态转移概率的定义需要考虑每个行动对状态的影响，以及各个环节的不确定性因素，如次品率、检测准确率等：

• 检测行动的转移概率：

$$P(s'|s,\text{Inspect}_i)=p_{\text{合格}}\times\text{检测准确率} \tag{23}$$

或者

$$(1-p_{\text{合格}})\times\text{误检率} \tag{24}$$

具体取决于检测成果的正确性。

• 装配行动的转移概率：

$$P(s'|s,\text{Assemble}_{i,j}) = p_{\text{装配成功}} \tag{25}$$

表示成功装配成半成品或成品。

- 销售行动的转移概率：

$$P(s'|s,\text{Sell}_{12}) = p_{\text{合格}} \times p_{\text{成功销售}} \tag{26}$$

包括合格概率和市场接受概率。

- 拆解行动的转移概率：

$$P(s'|s,\text{Disassemble}_{12}) = p_{\text{拆解成功}} \tag{27}$$

指拆解成功回收零配件的概率。

iv.奖励函数$R(s,a)$：

根据问题二提供的成本和收益模型，奖励函数综合考虑生产过程中的各种成本和收益：$R(s,a) =$ 销售收益 $-$ (零配件购买成本 $+$ 检测成本 $+$ 装配成本 $+$ 拆解费用 $+$ 调换损失)

即：

$$R(s,a) = s^f \cdot n^f - \left(n^c \left(\sum_{i=1}^{8} c_{pi}^c + \sum_{i=1}^{8} x_i \cdot c_{di}^c \right) + n^f (c_d^f \cdot y + c_a^f \cdot z + c_s \cdot y + c_s^f) \right) \tag{28}$$

(3)求解最优策略π^*：

Step 1：初始化

初始化所有状态的值函数$V(s) = 0$。

Step 2：策略迭代或值迭代

- 值迭代：使用贝尔曼方程更新值函数：

$$V(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s') \right) \tag{29}$$

其中，γ是折扣因子，用于平衡当前和未来收益。

- 策略迭代：交替进行策略评估和策略改进：
  - 策略评估：计算在当前策略下的值函数。
  - 策略改进：更新策略，使其在每个状态下选择能够最大化长期回报的行动。

Step 3：收敛

重复值迭代或策略迭代步骤，直到值函数$V(s)$收敛，确定最优值函数$V^*(s)$。

Step 4：确定最优策略π^*

根据最优值函数，确定每个状态下的最优行动：

$$\pi^*(s) = \arg\max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right) \tag{30}$$

4.3.2 蒙特卡洛树搜索算法求解

由于问题三中的决策具有高度不确定性和复杂动态环境，我们可以采用蒙特卡洛搜索树（*MCTS*）算法来解决马尔可夫决策过程。算法的主要步骤包括：

(1)初始化(*Init*)：首先，初始化搜索树，树的根节点代表当前的状态。

(2) 选择(*Selection*)：从根节点开始，选择最优的子节点继续进行探索，通常使用*UCT*(*UpperConfidenceboundsappliedtoTrees*)公式来平衡探索与利用。

(3) 扩展(*Expansion*)：一旦到达非叶子节点，如果这个节点还没有完全展开（即还有未探索的行动），创建一个或多个新的子节点，对应于可能的行动。

(4) 模拟(*Simulation*)：从新扩展的节点开始，使用随机策略进行模拟，直到达到预定的深度或游戏结束。

(5) 回溯(*Backpropagation*)：模拟结束后，将模拟的结果（如奖励或胜负）回传到路径上的所有节点，并更新这些节点的统计数据。

(6) 重复(Re*peat*)：重复选择、扩展、模拟和回溯步骤，直到达到计算时间或资源限制。

(7) 最终决策：基于树的根节点的统计信息，选择最佳行动。这通常是选择访问次数最多或平均回报最高的行动。

4.3.3 迭代法验证模型

使用256×256的邻接矩阵表示决策组合指数如下，反映了所有决策组合下的总成本并记录有效的利润值。

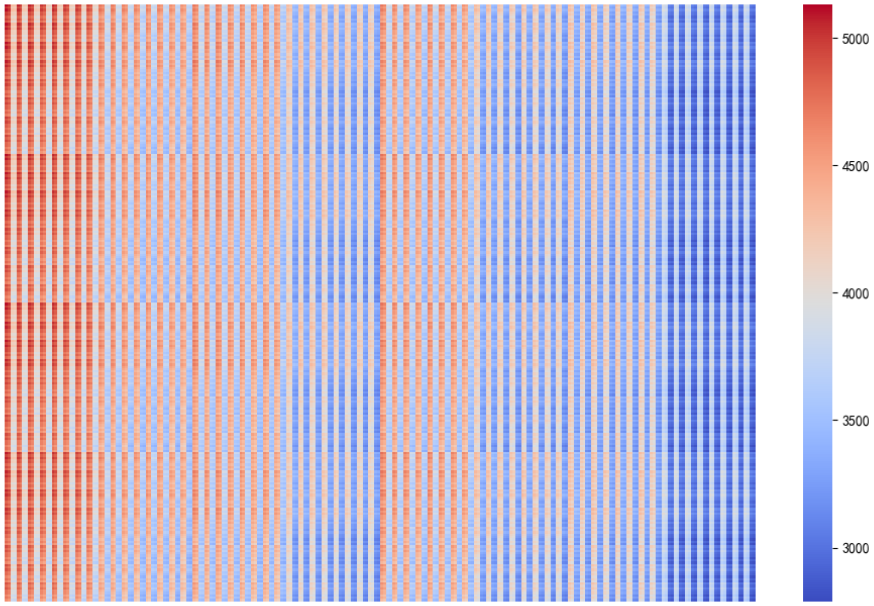

图 7 决策组合指数邻接矩阵热力图

同时，采用迭代法验证方案或发现问题的边界条件。通过迭代所有非解决方案的情况来验证某个解决方案的正确性或稳健性。下图所示为迭代法求得决策路径及企业最大收益可视化散点图，找到最大收益为164.8元，此时的找到的最优决策路径：为[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]，即对于所有的零配件，最优策略都是"装配"；对于半成品和成品，最优策略是"销售"。

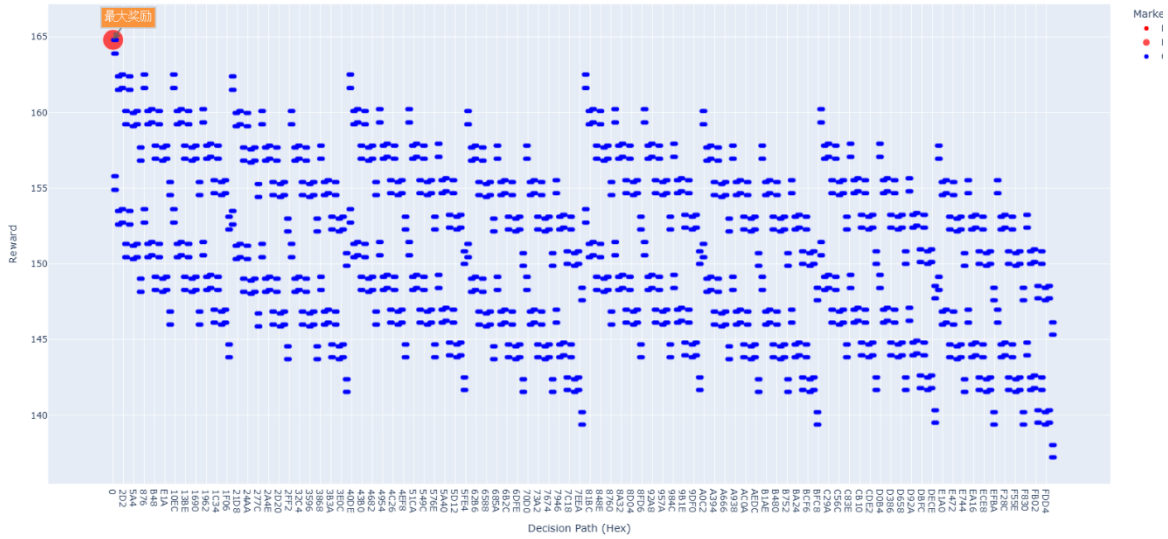

图 8　迭代法散点图

4.3.4 可视化结果分析

根据所建立的模型，使用蒙特卡洛搜索树算法分别将决策空间和节点收益可视化，得到如下图

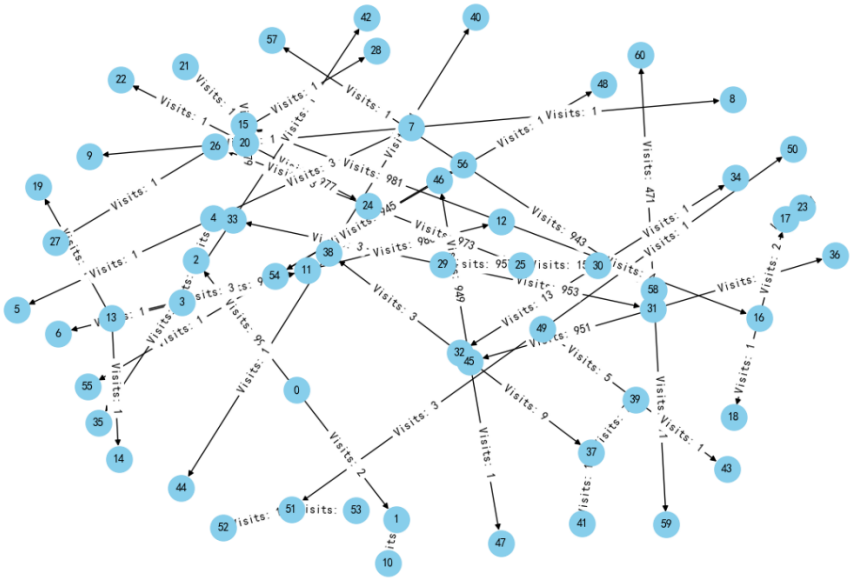

图 9　决策空间概率图

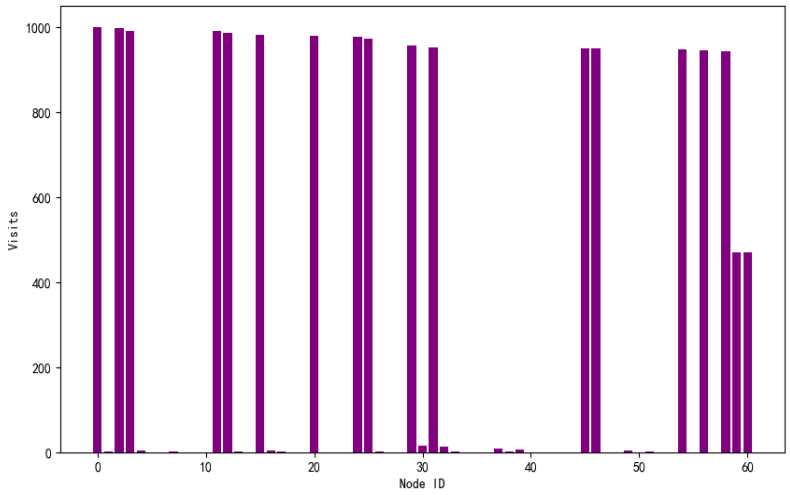

图 10　节点访问次数柱状图

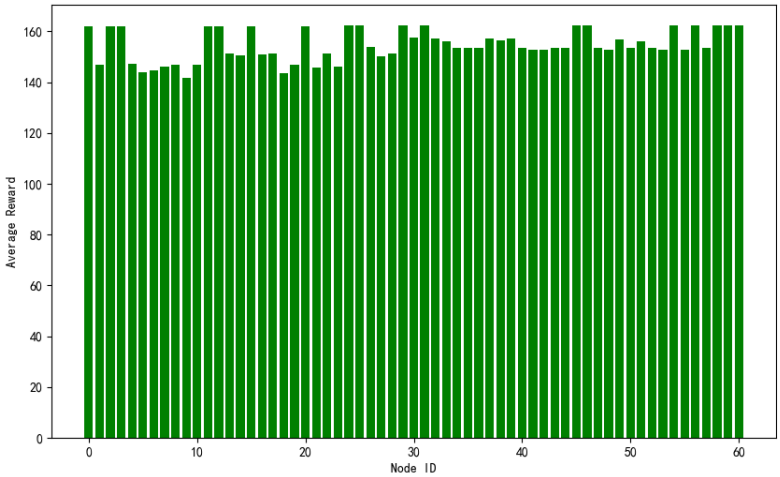

图 11　节点平均收益分布柱状图

由上图的分析可知：

(1) 从节点访问次数可以看出，算法可能已经很好地平衡了探索（尝试不同的行动以发现高回报的路径）和利用（利用已知的信息以优化回报。

(2) 节点 40 的访问次数下降可能指出了某些决策路径不是最优的，或者模拟中使用的启发式策略自然避开了那些看起来回报较低的路径,这进一步说明了不同决策路径对于收益有较大的决定因素。

(3) 平均收益的一致性表明了您的策略在不同的情况下能够保持相对稳定的表现，这是高效决策系统的一个良好特性。

(4) 大多数节点的平均收益维持在150 附近，这表明在这些节点上的决策通常能带来较高的回报。

综上所述，并结合蒙特卡洛搜索树算法求得的结果，可以得出：

• 最优决策路径: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]，即对于所有的零配件，最优策略都是"装配"；对于半成品和成品，最优策略是"销售"。

• 预期收益在150 上下浮动，与迭代法得到的结果相一致。

4.4 问题四模型建立与求解

4.4.1 贝叶斯自适应马尔可夫决策模型建立

基于问题四的背景，建立贝叶斯自适应马尔可夫决策过程（$BAMDP$）模型，该模型结合了问题三中的马尔可夫决策过程（MDP）模型，并引入了次品率参数的不确定性。在该模型中，决策者可以在不完全了解环境的情况下作出选择，使用贝叶斯方法来更新对环境的认知和优化决策策略。

(1)模型框架

$BAMDP$ 模型由五元组(Θ, S, A, P, R)定义：

• 参数空间Θ：包括所有不确定参数的概率分布。主要是次品率θ，其不确定性由贝塔分布$\text{Beta}(\alpha, \beta)$表示。

• 状态空间S：包括生产过程中的状态和次品率的当前信念状态。状态定义为$s = (s_1, s_2, ..., s_{12}, \theta)$。

• 行动空间A：包括对零配件、半成品或成品进行检测、不检测、装配、销售和拆解的行动。

• 状态转移概率$P(s'|s, a, \theta)$：根据行动a、当前状态s和参数θ下系统转移到下一个状态s'的概率。

• 奖励函数$R(s, a, \theta)$：基于行动a、当前状态s和参数θ下的收益或成本。

(2)数学表达及详细定义

• 参数空间Θ

次品率θ：由先验贝塔分布表示，例如，$\theta \sim \text{Beta}(\alpha, \beta)$。

• 状态空间S

状态定义：

$$s = (x_1, x_2, ..., x_8, y_1, y_2, y_3, z, \theta) \tag{31}$$

其中x_i表示零配件i的状态，y_j表示半成品j的状态，z表示成品状态。

• 行动空间A

行动包括：

-Inspect_i：检测指定部件

-DoNotInspect_i：不检测直接使用

-$\text{Assemble}_{i,j}$：装配部件

-Sell_{12}：销售成品

-Disassemble_{12}：拆解不合格成品

• 状态转移概率P和奖励函数R

i. 状态转移概率P：

$$P(s'|s, a, \theta) = \begin{cases} p_{\text{检测正确}} & \text{if } a = \text{Inspect}_i \\ p_{\text{装配成功}}(\theta) & \text{if } a = \text{Assemble}_{i,j} \\ 1 & \text{if } a = \text{DoNotInspect}_i \text{ or } \text{Sell}_{12} \\ p_{\text{拆解成功}} & \text{if } a = \text{Disassemble}_{12} \end{cases} \tag{32}$$

ii. 奖励函数R：

$$R(s, a, \theta) = \text{收益} - (\text{购买成本} + \text{检测成本} + \text{装配成本} + \text{拆解费用} + \text{调换损失}) \tag{33}$$

(3)求解最优策略π^*

i.初始化：所有状态的价值函数$V(s,\theta)=0$，并设定初始θ的分布。

ii.贝叶斯更新：基于观测数据更新θ的后验分布。

iii.策略迭代：

• 策略评估：使用贝尔曼方程(见式27)更新值函数。

• 策略改进：找到新的最优策略。

iv.迭代直到收敛：重复策略评估和改进过程直到V收敛。

通过$BAMDP$模型，可以在生产过程中不断更新对次品率的信念，并调整决策以最大化期望收益，同时考虑不确定性的影响。

4.4.2 蒙特卡洛树搜索算法求解

首先根据序贯分析法，计算生产过程中不同阶段产品的次品率及其置信区间的函数。这个过程从零件的次品率计算开始，扩展到半成品和最终产品的次品率计算，经过计算得出下表：

表 2　零配件次品率

名称	次品率
零配件 1	10.00%
零配件 2	8.20%
零配件 3	5.50%
零配件 4	10.70%
零配件 5	7.70%
零配件 6	9.00%
零配件 7	6.60%
零配件 8	10.00%
半成品 1	21.92%
半成品 2	24.99%
半成品 3	15.94%
成品	50.77%

在已知次品率的前提下，问题四就转换成问题三的求解，根据问题三中算法求解步骤得出最大收益为104.08元。

4.4.3 可视化结果分析

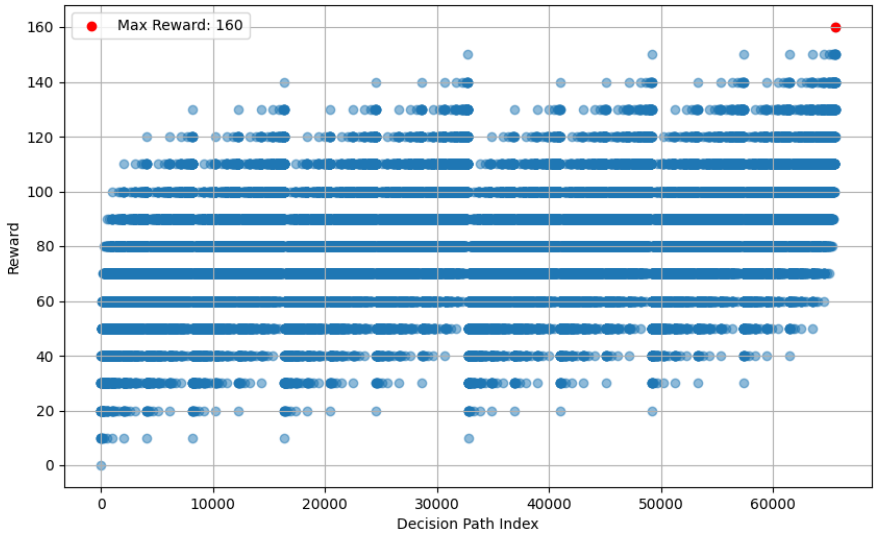

图 12　迭代全路径决策散点图

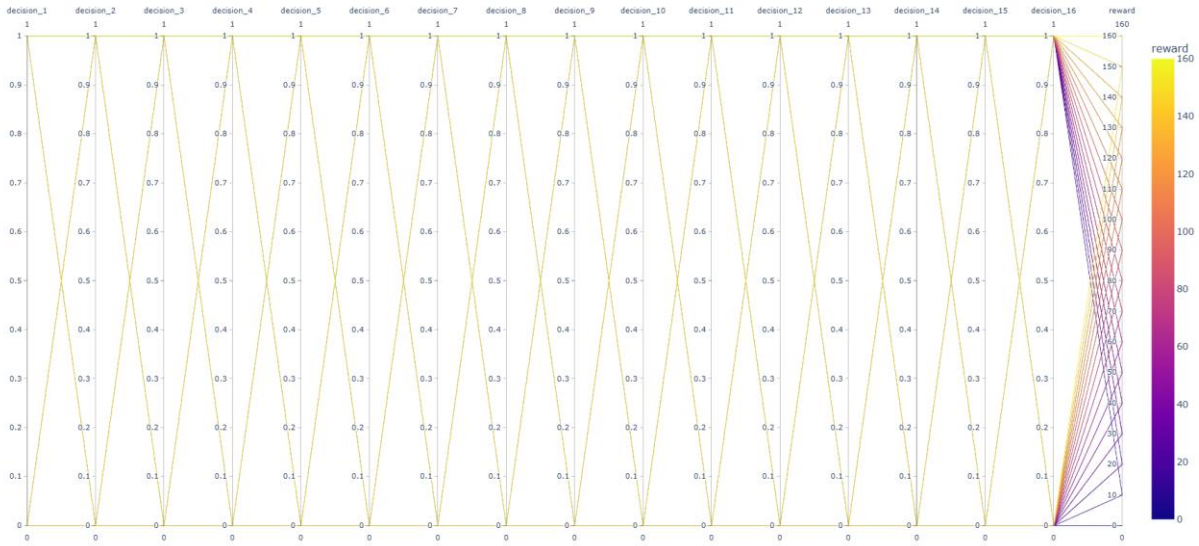

图 13　决策路径与收益的平行坐标轴

在问题中，我们考虑了抽样检测的不确定性,由第一问序贯分析法大致确定了各个阶段的次品率，分析上图可知：

(1)大部分决策路径在收益维度上显示出一定的集中趋势，这意味着收益是可预测且相对稳定的。

(2)高收益路径往往集中且颜色更亮，说明这些路径是较为有效的决策组合。

低收益路径的颜色较暗，并且在平行坐标图的最后一个坐标（*reward*）中，线条聚集在较低的收益值处。

(3) 从图中可以看出，大部分决策路径都走向较高的收益值，线条颜色的变化（从深紫到黄色）代表收益的变化，黄色线条对应较高的收益。

五、 模型分析检验

5.1 灵敏度分析

对于问题二，通过改变不同参数的值，迭代所有情况下的决策路径，进行灵敏度检验，得到全局决策路径下零配件次品率敏感性分析结果如下图所示，观察不同变化下对各种情况的最优决策影响。

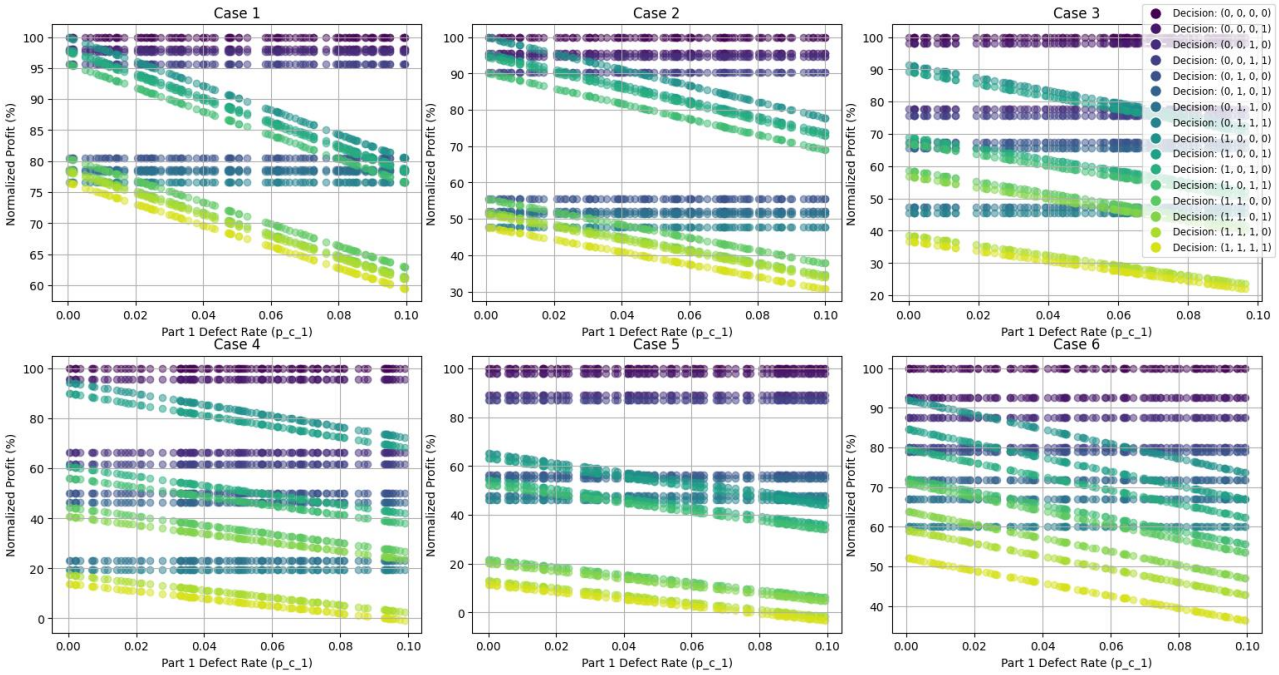

图 13　全局决策路径敏感性分析

由上图可知，通过观察全局决策路径在不同敏感度下的分布，可以得出总体上次品率波动与成本-利润波动成线性相关性，并且对最优路径的影响相对其他次优决策路径相比最小。

对于问题三的模型，灵敏度分析在蒙特卡洛树搜索($MCTS$)中的作用主要是评估探索参数c的变化对算法性能的影响，确保参数设置能够最大化奖励的获取。这种分析有助于理解参数设置如何影响决策模型的有效性和效率。

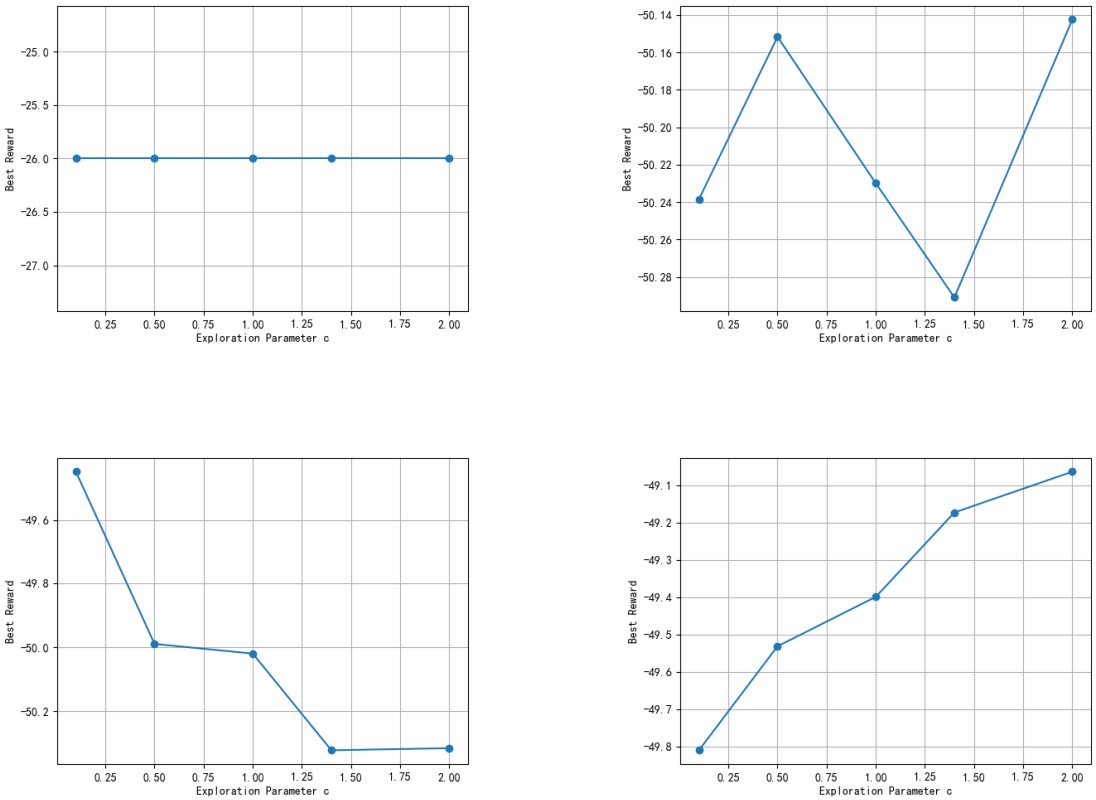

图 14　勘探参数 c 的敏感性分析

上图表明了不同探索参数c对最优奖励的影响，图中横轴表示探索参数c，纵轴表示通过*MCTS*算法找到的最优奖励值。各点标记了不同c值下的奖励，通过这些点的连接可以看出c值的变化如何影响算法的性能，可以更精确地调整算法参数，以适应特定的问题和环境，确保获得最佳的决策结果。

• 探索与利用的平衡：参数c在 MCTS 中控制着探索与利用之间的平衡。较低的c值倾向于利用已知的有利路径，而较高的c值鼓励探索尚未充分探索的路径。

• 敏感性趋势：如果图中显示随着c值增加，奖励先增加后减少，这表明存在一个最优的c值，能够在新的探索和已知路径之间达到最佳平衡。

六、 模型评价与推广

6.1 问题一模型的评价与推广

• 优点：

(1)该模型通过随机抽样和假设检验的结合来设计最优的检测方案，能满足题设要求在检测次数尽量少的前提下满足抽样次数少。

(2) 根据中心极限定理，许多统计量在大样本条件下会趋向于正态分布。因此，正态近似假设在处理大样本时非常有效，甚至在原始数据分布不是正态分布的情况下也能提供合理的结果。

(3)利用序贯分析法优化模型，在达到统计显著性时立即停止试验，在保持统计效力的同时减少所需的样本数量。

• 不足：

(1)模型未考虑实际检测过程中本身的误差，可能导致检测结果的过度信任。

(2)实际生产中检测，购买及装配的成本可能会随市场变化，可能会对估算结果产生一定影响

• 模型的推广：

引入自适应分布假设的方法可以根据数据的特性动态调整检验模型。例如，混合分布模型可以将数据拟合为多个分布的组合，而非单一正态分布假设。

6.2 问题二模型的评价与推广

• 优点：

(1) 多阶段动态决策模型能够处理随时间变化的决策问题，特别适用于决策在多个时间点上进行的场景。每个决策不仅影响当前的结果，还会影响未来的决策和收益。

(2) 多阶段动态决策模型的核心目标是通过优化算法（如动态规划、贝尔曼方程）找到最优策略，以使得整个过程的累积收益最大化。这对于长期收益或成本最小化尤为重要。

(3) 多阶段成本-利润决策模型综合考虑了生产过程中的各个环节，包括零配件购买及检测、产品装配、成品检测以及不合格品是否处理，建立了一个全面系统的模型框架。

• 不足：

(1) 当问题的状态空间或决策空间非常大时，计算最优策略的复杂性会急剧增加，导致“状态空间爆炸”问题。

(2) 对于大规模的问题，优化过程可能需要大量的计算时间，甚至在某些情况下无法找到最优解。因此，动态规划和强化学习等方法在处理大规模决策问题时常需采用近似算法。

• 模型的推广：

多阶段动态决策模型可应用于许多不同领域，尤其是在需要长期规划、动态决策和处理不确定性的场景中。推广时可以根据各个领域的具体需求和特性，定制模型应用方案，例如资产配置、风险管理、算法交易等，可以应用动态决策模型优化长期投资收益，平衡短期波动和长期回报。

6.3 问题三模型的评价与推广

• 优点：

(1) 该模型能够根据当前状态和时间变化进行决策调整，适应系统的动态特性，提供适时的策略更新，从而优化长期收益。

(2) 在不完全知道*MDP*所有细节的情况下，通过对未来可能行动的探索，来进行决策优化。这使其非常适合解决具有高度不确定性和复杂动态环境的决策问题。

(3) 蒙特卡洛树搜索算法可以在不完全知道*MDP*所有细节的情况下，通过对未来可能行动的探索，来进行决策优化。这使其非常适合解决具有高度不确定性和复杂动态环境的决策问题。

• 不足：

(1) 多维马尔可夫决策过程中的策略可能在某些条件下不稳定，尤其是在环境变化剧烈或不确定性较高的场景下。策略可能需要频繁调整，这可能导致模型的输出在实践中的可用性下降。

(2) 为了应对状态空间爆炸和计算复杂性，很多应用中会采用近似算法（如蒙特卡洛方法、值迭代、策略迭代等）。然而，近似算法会带来一定的偏差，无法保证求解的策略是最优的，有时会影响决策质量。

• **模型的推广：**

在大状态空间中，可以使用诸如蒙特卡洛方法、深度强化学习($DeepRL$)等近似算法来简化计算。这样能够在不显著牺牲精度的情况下，减少计算资源的需求，使模型能更高效地处理复杂问题。

6.4 问题四模型的评价与推广

• **优点：**

(1)通过贝叶斯方法可以显式地建模环境不确定性，使决策更加鲁棒。

(2)模型能够基于收集到的新信息不断更新其策略，适应环境的变化。

(3)提供了一种系统的方式来评估和改进决策策略，通过不断学习优化性能。

• **不足：**

(1)贝叶斯方法通常涉及到复杂的概率计算，尤其是在状态空间大或模型复杂时，计算负担较重。

(2)相对于传统的马尔可夫决策过程，$BAMDP$ 的实现更为复杂，需要高级的算法和技术支持。

• **模型的推广：**

i.机器人技术：在机器人的导航和任务规划中，使用 BAMDP 可以实现更精准的环境感知和决策。

ii.自动化驾驶：自动驾车系统中，BAMDP 有助于处理复杂的交通环境和动态障碍。健康医疗：在个性化医疗和治疗计划设计中，通过 BAMDP 模型进行风险评估和治疗方案优化。

参考文献

[1] 刘芳,朱天贺,苏卫星,等.基于高斯隐马尔可夫模型的人机共享控制区域化决策算法[J].电子学报,2022,50(11):2659-2667.

[2] 王嘉豪,张海军,方海.基于决策树-马尔可夫模型的轮状病毒疫苗纳入国家免疫规划的成本效益分析[J].中国疫苗和免疫,2022,28(03):309-316.DOI:10.19914/j.CJVI.2022060.

[3] 李金源,朱发新,滕宪斌,等.基于贝叶斯优化的时间卷积网络船舶航迹预测[J/OL].中国舰船研究,1-10[2024-09-08].https://doi.org/10.19693/j.issn.1673-3185.03755.

[4] 周翼男,崔桂梅,皮理想,等.基于贝叶斯优化 GBDT 的转炉炼钢终点预测[J].中国测试,2024,50(07):33-39.

[5] Yang, WH., Pan, MX., Zhou, Y. *et al.* Meaningful Update and Repair of Markov Decision Processes for Self-Adaptive Systems. *J. Comput. Sci. Technol.* **37**, 106–127 (2022).

附录

问题一源程序

```
import math
from scipy.stats import norm
# 计算样本量的函数
def calculate_sample_size(p_hat, error_margin, confidence_level):
    # 获取正态分布临界值 z_score
    z_score = norm.ppf(1 - (1 - confidence_level) / 2)

    # 计算样本量
    n = (z_score**2 * p_hat * (1 - p_hat)) / (error_margin**2)

    return math.ceil(n)


# 样本检测的函数
def check_acceptance_rejection(
    sample_size, observed_defect_rate, p_nominal, confidence_level, accept=True
):
    z_score = norm.ppf(1 - (1 - confidence_level) / 2)
    # 计算临界值
    critical_value = p_nominal + z_score * math.sqrt(
        (p_nominal * (1 - p_nominal)) / sample_size
    )

    if accept:
        return observed_defect_rate <= critical_value
    else:
        return observed_defect_rate > critical_value


# 参数设定
p_nominal = 0.10  # 标称次品率 10%
confidence_95 = 0.95  # 95% 信度
```

```
confidence_90 = 0.90  # 90% 信度
error_margin = 0.05  # 误差容忍度

# 计算拒收方案 (95% 信度)
sample_size_95 = calculate_sample_size(p_nominal, er-
ror_margin, confidence_95)
print(f"在 95%的信度下拒收零配件所需的样本量: {sam-
ple_size_95}")

# 计算接收方案 (90% 信度)
sample_size_90 = calculate_sample_size(p_nominal, er-
ror_margin, confidence_90)
print(f"在 90%的信度下接收零配件所需的样本量: {sam-
ple_size_90}")

# 假设我们从样本中得到了一个实际的次品率
observed_defect_rate = 0.12  # 实际观察到的次品率 12%

# 检查是否拒收 (95% 信度)
is_reject_95 = check_acceptance_rejection(
    sample_size_95, observed_defect_rate, p_nominal, confi-
dence_95, accept=False
)
print(f"在 95%的信度下，是否拒收零配件: {'是' if is_reject_95
else '否'}")

# 检查是否接收 (90% 信度)
is_accept_90 = check_acceptance_rejection(
    sample_size_90, observed_defect_rate, p_nominal, confi-
dence_90, accept=True
)
print(f"在 90%的信度下，是否接收零配件: {'是' if is_accept_90
else '否'}")
```

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

```
# Function to calculate the minimum number of samples re-
quired for SPR testing
def min_samples_sprt(p0, p1, alpha, beta):
    log_alpha = np.log(beta / (1 - alpha))
    log_beta = np.log((1 - beta) / alpha)
    n = 0  # Initial number of samples
    log_likelihood_ratio = 0

    # Incrementally calculate likelihood ratios until a de-
cision threshold is met
    while True:
        n += 1
        # Calculate probability of seeing a defective part
under H0 and H1
        likelihood_0 = binom.pmf(1, 1, p0)  # Probability of
defective under H0
        likelihood_1 = binom.pmf(1, 1, p1)  # Probability of
defective under H1
        # Update log likelihood ratio
        log_likelihood_ratio += np.log(likelihood_1 / like-
lihood_0)

        # Check if either decision threshold is met
        if log_likelihood_ratio >= log_beta or log_likeli-
hood_ratio <= log_alpha:
            break

    return n


# Given values
p0 = 0.10
p1 = 0.15  # Alternative hypothesis slightly higher defect
rate

# Scenario 1: Alpha = 5%, Beta = 10%
```

```
n_samples_1 = min_samples_sprt(p0, p1, alpha=0.000005,
beta=0.10)

# Scenario 2: Alpha = 10%, Beta = 5%
n_samples_2 = min_samples_sprt(p0, p1, alpha=0.000010,
beta=0.05)

# Print the results
print(f"Minimum samples required for scenario 1 (95% confi-
dence): {n_samples_1}")
print(f"Minimum samples required for scenario 2 (90% confi-
dence): {n_samples_2}")
```

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

# Function to calculate critical value based on sample
size, defect rate, and confidence level
def calculate_critical_value(sample_size, p_nominal, confi-
dence_level):
    z_score = norm.ppf(1 - (1 - confidence_level) / 2)
    critical_value = p_nominal + z_score * math.sqrt(
        (p_nominal * (1 - p_nominal)) / sample_size
    )
    return critical_value


# Range of sample sizes
sample_sizes = range(50, 1001, 50)  # From 50 to 1000 in
steps of 50

# Parameters
p_nominal = 0.10  # Nominal defect rate of 10%
confidence_levels = [0.90, 0.95, 0.99]  # Various confi-
dence levels
```

```
# Plotting
plt.figure(figsize=(10, 6))

for confidence_level in confidence_levels:
    critical_values = [
        calculate_critical_value(n, p_nominal, confi-
dence_level) for n in sample_sizes
    ]
    plt.plot(
        sample_sizes, critical_values, label=f"Confidence
{int(confidence_level*100)}%"
    )

plt.title("Critical Value vs Sample Size")
plt.xlabel("Sample Size")
plt.ylabel("Critical Value")
plt.legend()
plt.grid(True)
plt.show()
```

问题二源程序

```
import os

# 将工作目录更改为脚本所在目录
os.chdir(os.path.dirname(__file__))
print("Current working directory:", os.getcwd())
import json
import matplotlib.pyplot as plt


# 从 JSON 文件中读取数据
def load_data_from_json(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        data = json.load(f)
    return data
```

```python
# 计算决策路径的收益
def calculate_profit(
    n_c,
    p_c_1,
    p_c_2,
    c_p_1,
    c_p_2,
    c_d_1,
    c_d_2,
    p_f,
    c_s_f,
    c_d_f,
    s_f,
    c_s,
    c_a_f,
    x_1,
    x_2,
    y,
    z,
):
    effective_parts = n_c * (1 - x_1 * p_c_1) * (1 - x_2 * p_c_2)  # 有效零配件数量
    n_f = effective_parts * (1 - p_f)  # 成品数量计算

    # 各项成本计算
    C_c_p = n_c * (c_p_1 + c_p_2)  # 零配件购买成本
    C_c_d = n_c * (x_1 * c_d_1 + x_2 * c_d_2)  # 零配件检测成本
    C_d_f = c_d_f * y * n_f * (1 - p_f)  # 成品检测成本
    C_a_f = c_a_f * z * n_f * p_f  # 成品拆解成本
    C_s = c_s * y * n_f * p_f  # 调换成本
    C_s_f = c_s_f * n_f  # 成品装配成本
    S = s_f * n_f  # 利润

    # 目标函数（总成本）
    Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S
```

```
    return -Z  # 收益，目标函数的负值


# 分支定界法求解
def branch_and_bound(data):
    global_best_profit = float("-inf")  # 初始化全局最优收益
    global_best_decision = None  # 初始化全局最优决策路径
    all_scenarios = []  # 存储所有情况下的决策路径及收益
    decision_paths = []  # 存储所有的决策路径

    # 假设的零配件总数量
    n_c = 1  # 零配件总数量（假设值，可以调整）

    # 遍历所有情况
    for case_data in data["scenarios"]:
        case_number = case_data["case"]
        profits = []  # 存储当前情况下所有决策的收益
        best_profit = float("-inf")  # 当前情况下的最优收益
        best_decision = None  # 当前情况下的最优决策

        # 读取参数
        p_c_1 = case_data["part_1_defect_rate"]
        p_c_2 = case_data["part_2_defect_rate"]
        c_p_1 = case_data["part_1_purchase_price"]
        c_p_2 = case_data["part_2_purchase_price"]
        c_d_1 = case_data["part_1_detection_cost"]
        c_d_2 = case_data["part_2_detection_cost"]
        p_f = case_data["product_defect_rate"]
        c_s_f = case_data["assembly_cost"]
        c_d_f = case_data["product_detection_cost"]
        s_f = case_data["market_price"]
        c_s = case_data["replacement_loss"]
        c_a_f = case_data["disassembly_cost"]

        # 初始分支列表 (x_1, x_2, y, z)
        initial_branch = (0, 0, 0, 0)
        stack = [(initial_branch, 0)]  # 栈存储分支和层级
（depth）
```

```
        while stack:
            current_branch, depth = stack.pop()
            x_1, x_2, y, z = current_branch

            # 计算当前分支的收益
            profit = calculate_profit(
                n_c,
                p_c_1,
                p_c_2,
                c_p_1,
                c_p_2,
                c_d_1,
                c_d_2,
                p_f,
                c_s_f,
                c_d_f,
                s_f,
                c_s,
                c_a_f,
                x_1,
                x_2,
                y,
                z,
            )

            if depth == 4:  # 已到达决策的最后一层
                profits.append(profit)
                decision_path = f"({x_1}, {x_2}, {y}, {z})"
                if case_number == 1:
                    decision_paths.append(decision_path)  # 仅需添加一次决策路径

                # 更新当前情况下的最优决策路径
                if profit > best_profit:
                    best_profit = profit
                    best_decision = (x_1, x_2, y, z)
```

```
                # 更新全局最优决策路径
                if profit > global_best_profit:
                    global_best_profit = profit
                    global_best_decision = (x_1, x_2, y, z,
case_number)

            else:
                # 生成新的分支（扩展当前节点）
                next_depth = depth + 1
                for next_decision in [0, 1]:
                    new_branch = (
                        current_branch[:depth]
                        + (next_decision,)
                        + current_branch[depth + 1 :]
                    )
                    # 计算上界（可以用一个更好的估计方法）
                    # 简化示例中，假设不做其他调整，直接扩展
                    stack.append((new_branch, next_depth))

        # 输出当前情况下的最优决策路径
        print(
            f"最优决策路径: (x_1={best_decision[0]},
x_2={best_decision[1]}, y={best_decision[2]}, z={best_deci-
sion[3]}) 在情况 {case_number} 中 -> 最终收益:
{best_profit:.2f}"
        )

        # 将当前情况的收益存储到全局数据中
        all_scenarios.append(
            {
                "case": case_number,
                "profits": profits,
                "best_index": profits.index(best_profit),
            }
        )

    # 使用 SCI 常用的颜色方案
    colors = [
```

```
        "tab:blue",
        "tab:green",
        "tab:red",
        "tab:purple",
        "tab:orange",
        "tab:cyan",
    ]
    plt.figure(figsize=(12, 8))

    for idx, scenario in enumerate(all_scenarios):
        # 绘制折线图
        plt.plot(
            decision_paths,
            scenario["profits"],
            color=colors[idx],
            marker="o",
            label=f'Case {scenario["case"]}',
        )
        # 重点标记最优决策路径点
        plt.scatter(
            scenario["best_index"],
            scenario["profits"][scenario["best_index"]],
            color=colors[idx],
            edgecolor="black",
            s=150,
            zorder=5,
            label=f"Optimal Point Case {scenario['case']}",
        )

    # 图表设置
    plt.title("", fontsize=16)
    plt.xlabel("Decision Path", fontsize=14)
    plt.ylabel("Profit", fontsize=14)

    # 将决策路径标签旋转 45 度，并设置右对齐
    plt.xticks(rotation=45, ha="right")
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.legend()
```

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

    # 输出全局最优决策路径和对应的收益
    print(
        f"/n 全局最优决策路径: (x_1={global_best_deci-
sion[0]}, x_2={global_best_decision[1]}, y={global_best_de-
cision[2]}, z={global_best_decision[3]}) 在情况
{global_best_decision[4]} 中 -> 最终收益:
{global_best_profit:.2f}"
    )


# 从 JSON 文件中加载数据 (假设 JSON 文件名为'data.json')
data = load_data_from_json("./data.json")
branch_and_bound(data)
```

```
import json


# 从 JSON 文件中读取数据
def load_data_from_json(file_path, case_number):
    with open(file_path, "r", encoding="utf-8") as f:
        data = json.load(f)

    # 查找对应的情况
    for scenario in data["scenarios"]:
        if scenario["case"] == case_number:
            return scenario

    return None


# 初始化变量
best_profit = float("-inf")  # 初始化为负无穷大
best_decision = None  # 初始化最优决策路径为空
```

```
# 从 JSON 文件中加载数据 (假设 JSON 文件名为'parameters.json')
case_number = 2  # 选择要读取的情况
data = load_data_from_json("data.json", case_number)

if data:
    # 读取参数
    p_c_1 = data["part_1_defect_rate"]
    p_c_2 = data["part_2_defect_rate"]
    c_p_1 = data["part_1_purchase_price"]
    c_p_2 = data["part_2_purchase_price"]
    c_d_1 = data["part_1_detection_cost"]
    c_d_2 = data["part_2_detection_cost"]
    p_f = data["product_defect_rate"]
    c_s_f = data["assembly_cost"]
    c_d_f = data["product_detection_cost"]
    s_f = data["market_price"]
    c_s = data["replacement_loss"]
    c_a_f = data["disassembly_cost"]

    # 假设的零配件总数量
    n_c = 1  # 零配件总数量（假设值，可以调整）

    # 遍历所有可能的决策组合 (2^4 = 16 种)
    for x_1 in [0, 1]:  # 是否对零配件 1 进行检测
        for x_2 in [0, 1]:  # 是否对零配件 2 进行检测
            for y in [0, 1]:  # 是否对成品进行检测
                for z in [0, 1]:  # 是否对不合格成品进行拆解
                    # 计算有效零配件的数量
                    effective_parts = (
                        n_c * (1 - x_1 * p_c_1) * (1 - x_2 *
p_c_2)
                    )  # 有效零配件数量

                    # 成品数量计算
                    n_f = effective_parts * (1 - p_f)

                    # 各项成本计算
```

```
                    C_c_p = n_c * (c_p_1 + c_p_2)  # 零配件购
买成本
                    C_c_d = n_c * (x_1 * c_d_1 + x_2 *
c_d_2)  # 零配件检测成本
                    C_d_f = c_d_f * y * n_f * (1 - p_f)  #
成品检测成本
                    C_a_f = c_a_f * z * n_f * p_f  # 成品拆解
成本
                    C_s = c_s * y * n_f * p_f  # 调换成本
                    C_s_f = c_s_f * n_f  # 成品装配成本
                    S = s_f * n_f  # 利润

                    # 目标函数（总成本）
                    Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s
+ C_s_f - S
                    profit = -Z  # 收益，目标函数的负值

                    # 输出当前组合的决策路径和收益
                    print(
                        f"决策路径 (x_1={x_1}, x_2={x_2},
y={y}, z={z}) -> 收益: {profit:.2f}"
                    )

                    # 更新最优决策路径
                    if profit > best_profit:
                        best_profit = profit
                        best_decision = (x_1, x_2, y, z)

    # 输出最优决策路径和对应的收益
    print(
        f"\n 最优决策路径: (x_1={best_decision[0]},
x_2={best_decision[1]}, y={best_decision[2]}, z={best_deci-
sion[3]}) -> 最终收益: {best_profit:.2f}"
    )
else:
    print(f"未找到对应的情况 {case_number}")
```

```
import os
import json
import matplotlib.pyplot as plt

# 将工作目录更改为脚本所在目录
os.chdir(os.path.dirname(__file__))
print("Current working directory:", os.getcwd())


# 从 JSON 文件中读取数据
def load_data_from_json(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        data = json.load(f)
    return data


# 计算决策路径的收益
def calculate_profit(
    n_c,
    p_c_1,
    p_c_2,
    c_p_1,
    c_p_2,
    c_d_1,
    c_d_2,
    p_f,
    c_s_f,
    c_d_f,
    s_f,
    c_s,
    c_a_f,
    x_1,
    x_2,
    y,
    z,
):
    effective_parts = n_c * (1 - x_1 * p_c_1) * (1 - x_2 *
p_c_2)  # 有效零配件数量
    n_f = effective_parts * (1 - p_f)  # 成品数量计算
```

```
    # 各项成本计算
    C_c_p = n_c * (c_p_1 + c_p_2)  # 零配件购买成本
    C_c_d = n_c * (x_1 * c_d_1 + x_2 * c_d_2)  # 零配件检测成
本
    C_d_f = c_d_f * y * n_f * (1 - p_f)  # 成品检测成本
    C_a_f = c_a_f * z * n_f * p_f  # 成品拆解成本
    C_s = c_s * y * n_f * p_f  # 调换成本
    C_s_f = c_s_f * n_f  # 成品装配成本
    S = s_f * n_f  # 利润

    # 目标函数（总成本）
    Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S
    return -Z  # 收益，目标函数的负值


# 分支定界法求解
def branch_and_bound(data):
    global_best_profit = float("-inf")  # 初始化全局最优收益
    global_best_decision = None  # 初始化全局最优决策路径
    all_scenarios = []  # 存储所有情况下的决策路径及收益

    n_c = 1  # 零配件总数量（假设值，可以调整）

    for case_data in data["scenarios"]:
        case_number = case_data["case"]
        profits = []  # 存储当前情况下所有决策的收益
        best_profit = float("-inf")  # 当前情况下的最优收益
        best_decision = None  # 当前情况下的最优决策

        # 读取参数
        p_c_1 = case_data["part_1_defect_rate"]
        p_c_2 = case_data["part_2_defect_rate"]
        c_p_1 = case_data["part_1_purchase_price"]
        c_p_2 = case_data["part_2_purchase_price"]
        c_d_1 = case_data["part_1_detection_cost"]
        c_d_2 = case_data["part_2_detection_cost"]
        p_f = case_data["product_defect_rate"]
```

```
        c_s_f = case_data["assembly_cost"]
        c_d_f = case_data["product_detection_cost"]
        s_f = case_data["market_price"]
        c_s = case_data["replacement_loss"]
        c_a_f = case_data["disassembly_cost"]

        initial_branch = (0, 0, 0, 0)
        stack = [(initial_branch, 0)]

        while stack:
            current_branch, depth = stack.pop()
            x_1, x_2, y, z = current_branch

            # 计算当前分支的收益
            profit = calculate_profit(
                n_c,
                p_c_1,
                p_c_2,
                c_p_1,
                c_p_2,
                c_d_1,
                c_d_2,
                p_f,
                c_s_f,
                c_d_f,
                s_f,
                c_s,
                c_a_f,
                x_1,
                x_2,
                y,
                z,
            )

            if depth == 4:
                profits.append(profit)

                if profit > best_profit:
```

```
                    best_profit = profit
                    best_decision = (x_1, x_2, y, z)

                if profit > global_best_profit:
                    global_best_profit = profit
                    global_best_decision = (x_1, x_2, y, z,
case_number)

            else:
                next_depth = depth + 1
                for next_decision in [0, 1]:
                    new_branch = (
                        current_branch[:depth]
                        + (next_decision,)
                        + current_branch[depth + 1 :]
                    )
                    stack.append((new_branch, next_depth))

        print(
            f"最优决策路径: (x_1={best_decision[0]},
x_2={best_decision[1]}, y={best_decision[2]}, z={best_deci-
sion[3]}) 在情况 {case_number} 中 -> 最终收益:
{best_profit:.2f}"
        )

        all_scenarios.append(
            {
                "case": case_number,
                "best_profit": best_profit,
            }
        )

    # 绘制柱状图
    cases = [f"Case {scenario['case']}" for scenario in
all_scenarios]
    best_profits = [scenario["best_profit"] for scenario in
all_scenarios]
```

```
    plt.figure(figsize=(10, 6))
    plt.bar(cases, best_profits, color="skyblue")
    plt.title("", fontsize=16)
    plt.xlabel("Case", fontsize=14)
    plt.ylabel("Best Profit", fontsize=14)
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.tight_layout()
    plt.show()

    print(
        f"/n 全局最优决策路径: (x_1={global_best_deci-
sion[0]}, x_2={global_best_decision[1]}, y={global_best_de-
cision[2]}, z={global_best_decision[3]}) 在情况
{global_best_decision[4]} 中 -> 最终收益:
{global_best_profit:.2f}"
    )


# 从 JSON 文件中加载数据 (假设 JSON 文件名为'data.json')
data = load_data_from_json("./data.json")
branch_and_bound(data)
```

```
import os
import json
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
import matplotlib.cm as cm  # 导入 colormap

# 将工作目录更改为脚本所在目录
os.chdir(os.path.dirname(__file__))
print("Current working directory:", os.getcwd())


# 从 JSON 文件中读取数据
def load_data_from_json(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        data = json.load(f)
    return data
```

```python
# 计算决策路径的收益
def calculate_profit(
    n_c,
    p_c_1,
    p_c_2,
    c_p_1,
    c_p_2,
    c_d_1,
    c_d_2,
    p_f,
    c_s_f,
    c_d_f,
    s_f,
    c_s,
    c_a_f,
    decision,
):
    x_1, x_2, y, z = decision
    effective_parts = n_c * (1 - x_1 * p_c_1) * (1 - x_2 *
p_c_2)
    n_f = effective_parts * (1 - p_f)
    C_c_p = n_c * (c_p_1 + c_p_2)
    C_c_d = n_c * (x_1 * c_d_1 + x_2 * c_d_2)
    C_d_f = c_d_f * y * n_f * (1 - p_f)
    C_a_f = c_a_f * z * n_f * p_f
    C_s = c_s * y * n_f * p_f
    C_s_f = c_s_f * n_f
    S = s_f * n_f
    Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S
    return -Z


# Monte Carlo sensitivity analysis for each decision path
def monte_carlo_sensitivity_on_decisions(
    case_data, n_simulations=100, ax=None, colors=None
):
```

```
    decisions = list(product([0, 1], repeat=4))  # Generate
all decision combinations
    p_c_1_samples = np.random.uniform(0, 0.1, n_simula-
tions)
    all_profits = []

    results = {}
    for decision, color in zip(decisions, colors):
        profits = []
        for p_c_1 in p_c_1_samples:
            profit = calculate_profit(
                100,
                p_c_1,
                case_data["part_2_defect_rate"],
                case_data["part_1_purchase_price"],
                case_data["part_2_purchase_price"],
                case_data["part_1_detection_cost"],
                case_data["part_2_detection_cost"],
                case_data["product_defect_rate"],
                case_data["assembly_cost"],
                case_data["product_detection_cost"],
                case_data["market_price"],
                case_data["replacement_loss"],
                case_data["disassembly_cost"],
                decision,
            )
            profits.append(profit)
        results[decision] = profits
        all_profits.extend(profits)

    max_profit = max(all_profits)

    for decision, profits in results.items():
        normalized_profits = [
            (profit / max_profit) * 100 for profit in prof-
its
        ]  # Normalize
```

```
        ax.scatter(p_c_1_samples, normalized_profits, al-
pha=0.5, color=colors[decision])

    ax.set_title(f'Case {case_data["case"]}')
    ax.set_xlabel("Part 1 Defect Rate (p_c_1)")
    ax.set_ylabel("Normalized Profit (%)")
    ax.grid(True)


# Load data from JSON
data_path = "./data.json"
data = load_data_from_json(data_path)

# 创建 6 个子图
fig, axs = plt.subplots(2, 3, figsize=(18, 10))
axs = axs.flatten()  # 将子图对象展平为一维数组

# 设置颜色映射，每个决策变量对应一种颜色
decisions = list(product([0, 1], repeat=4))  # 决策变量
colors = {
    decision: plt.cm.viridis(i / len(decisions)) for i, de-
cision in enumerate(decisions)
}  # 使用 colormap 为决策变量分配颜色

# Perform sensitivity analysis on each case for all deci-
sion paths
for i, case_data in enumerate(data["scenarios"]):
    monte_carlo_sensitivity_on_decisions(case_data, 100,
ax=axs[i], colors=colors)

# 添加统一的图例，并确保图例颜色与散点图中的一致
handles = [
    plt.Line2D(
        [],
        [],
        marker="o",
        color="w",
        markerfacecolor=colors[decision],
```

```
        markersize=10,
        label=f"Decision: {decision}",
    )
    for decision in decisions
]

fig.legend(handles=handles, loc="upper right", font-
size="small")

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

问题三源程序

```
import random
from math import log
import matplotlib.pyplot as plt

plt.rcParams["font.sans-serif"] = ["SimHei"]  # 用黑体显示中
文
plt.rcParams["axes.unicode_minus"] = False  # 正确显示负号

import networkx as nx


def MCTS(root, iterations):
    for _ in range(iterations):
        node = root
        # 1. Selection
        while node.is_fully_expanded() and not
node.state.is_terminal():
            node = node.best_child()

        # 2. Expansion
        if not node.is_fully_expanded() and not
node.state.is_terminal():
```

```
            possible_moves = node.state.possible_moves()
            move = random.choice(possible_moves)
            new_state = node.state.move(move)
            child_node = Node(new_state, node)  # 加入 par-
ent 参数
            node.children.append(child_node)
            node = child_node

        # 3. Simulation
        reward = simulate(node.state)

        # 4. Backpropagation
        while node is not None:
            node.update(reward)
            node = node.parent


def simulate(state):
    # 这里需要定义一个模拟函数来评估从当前状态开始的随机玩法的结
果
    while not state.is_terminal():
        possible_moves = state.possible_moves()
        move = random.choice(possible_moves)
        state = state.move(move)
    return state.reward()


class Node:
    node_counter = 0  # 静态变量，给每个节点分配唯一 ID

    def __init__(self, state, parent=None):
        self.id = Node.node_counter
        Node.node_counter += 1
        self.state = state
        self.parent = parent
        self.value = 0
        self.visits = 0
        self.children = []
```

```
    def expand(self):
        for move in self.state.possible_moves():
            new_state = self.state.move(move)
            child_node = Node(new_state, self)
            self.children.append(child_node)

    def is_fully_expanded(self):
        return len(self.children) == len(self.state.possi-
ble_moves())

    def best_child(self, c_param=1.4):
        choices_weights = [
            (child.value / child.visits)
            + c_param * (2 * log(self.visits) / child.vis-
its) ** 0.5
            for child in self.children
        ]
        return self.children[choices_weights.in-
dex(max(choices_weights))]

    def update(self, reward):
        self.visits += 1
        self.value += reward


class MDPState:
    def __init__(self, decisions, parts, semi_products, fi-
nal_product, n_c):
        self.decisions = decisions
        self.parts = parts
        self.semi_products = semi_products
        self.final_product = final_product
        self.n_c = n_c

    def possible_moves(self):
        if len(self.decisions) < 16:  # 有 16 个决策
            return [0, 1]
        return []
```

```
    def move(self, decision):
        new_decisions = self.decisions[:]
        new_decisions.append(decision)
        return MDPState(
            new_decisions, self.parts, self.semi_products,
self.final_product, self.n_c
        )

    def is_terminal(self):
        return len(self.decisions) == 16

    def reward(self):
        # 计算零配件购买成本 C^c_p
        C_c_p = sum(
            n_c * c_p
            for c_p, n_c in zip(
                [part["purchase_price"] for part in
self.parts], self.n_c
            )
        )

        # 计算零配件检测成本 C^c_d
        C_c_d = sum(
            n_c * x * c_d
            for x, c_d, n_c in zip(
                self.decisions[:8],
                [part["inspection_cost"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品次品率 p^f
        p_f = self.final_product["defect_rate"]

        # 检测合格后的零配件数量
        n_c_post_inspection = sum(
```

```
            n_c * (1 - x * p_c)
            for x, p_c, n_c in zip(
                self.decisions[:8],
                [part["defect_rate"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品数量 n^f
        n_f = n_c_post_inspection * (1 - p_f)

        # 计算成品检测成本 C_d_f
        C_d_f = self.final_product["inspection_cost"] *
self.decisions[14] * n_f

        # 计算成品拆解成本 C_a_f
        C_a_f = self.final_product["disassembly_cost"] *
self.decisions[15] * n_f * p_f

        # 计算调换成本 C_s
        C_s = self.final_product["exchange_loss"] *
self.decisions[14] * n_f * p_f

        # 计算成品装配成本 C_s_f
        C_s_f = self.final_product["assembly_cost"] * n_f

        # 计算利润 S
        S = self.final_product["market_price"] * n_f

        # 计算总成本 Z
        Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S

        return -Z  # 返回负收益，因为我们是在最小化总成本


# 更新 parts, semi_products, final_product 和 n_c 的值
parts = [
```

```
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 1},  # 零配件 1
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 2
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 3
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 1},  # 零配件 4
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 5
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 6
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 7
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 8
]

semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 1
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 2
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 3
]
```

```
final_product = {
    "defect_rate": 0.10,
    "assembly_cost": 8,
    "inspection_cost": 6,
    "disassembly_cost": 10,
    "market_price": 200,
    "exchange_loss": 40,
}

n_c = [1 / 8] * 8  # 每种零配件数量均为 100

# 初始化状态
initial_state = MDPState([], parts, semi_products, fi-
nal_product, n_c)
root = Node(initial_state)

# 运行 MCTS
MCTS(root, iterations=1000)


# 可视化函数
def visualize_tree(root):
    G = nx.DiGraph()  # 创建一个有向图

    # 深度优先搜索，构建图
    def add_edges(node):
        for child in node.children:
            G.add_edge(node.id, child.id, visits=child.vis-
its, value=child.value)
            add_edges(child)

    add_edges(root)

    # 设置图形布局
    pos = nx.spring_layout(G)

    # 访问次数作为边的标签
```

```
    edge_labels = {(u, v): f'Visits: {d["visits"]}' for u,
v, d in G.edges(data=True)}

    # 绘制节点和边
    plt.figure(figsize=(12, 8))
    nx.draw(G, pos, with_labels=True, node_size=500,
node_color="skyblue", font_size=10)
    nx.draw_networkx_edge_labels(G, pos, edge_la-
bels=edge_labels)
    plt.title("MCTS 决策树结构")
    plt.show()


def visualize_node_visits(root):
    nodes = []
    visits = []

    # 记录每个节点的访问次数
    def collect_visits(node):
        nodes.append(node.id)
        visits.append(node.visits)
        for child in node.children:
            collect_visits(child)

    collect_visits(root)

    plt.figure(figsize=(10, 6))
    plt.bar(nodes, visits, color="purple")
    plt.xlabel("Node ID")
    plt.ylabel("Visits")
    plt.title("节点访问次数分布")
    plt.show()


def visualize_rewards(root):
    nodes = []
    rewards = []

    # 记录每个节点的收益
```

```
    def collect_rewards(node):
        nodes.append(node.id)
        rewards.append(node.value / node.visits if
node.visits > 0 else 0)
        for child in node.children:
            collect_rewards(child)

    collect_rewards(root)

    plt.figure(figsize=(10, 6))
    plt.bar(nodes, rewards, color="green")
    plt.xlabel("Node ID")
    plt.ylabel("Average Reward")
    plt.title("节点平均收益分布")
    plt.show()


# 可视化决策树、节点访问次数和收益
visualize_tree(root)
visualize_node_visits(root)
visualize_rewards(root)
```

```
import random
from math import log  # 导入 log 函数


def MCTS(root, iterations):
    for _ in range(iterations):
        node = root
        # 1. Selection
        while node.is_fully_expanded() and not
node.state.is_terminal():
            node = node.best_child()

        # 2. Expansion
        if not node.is_fully_expanded() and not
node.state.is_terminal():
            possible_moves = node.state.possible_moves()
```

```
            move = random.choice(possible_moves)
            new_state = node.state.move(move)
            child_node = Node(new_state, node)  # 加入 par-
ent 参数
            node.children.append(child_node)
            node = child_node

        # 3. Simulation
        reward = simulate(node.state)

        # 4. Backpropagation
        while node is not None:
            node.update(reward)
            node = node.parent


def simulate(state):
    # 这里需要定义一个模拟函数来评估从当前状态开始的随机玩法的结
果
    while not state.is_terminal():
        possible_moves = state.possible_moves()
        move = random.choice(possible_moves)
        state = state.move(move)
    return state.reward()


# 在 Node 类中添加 parent 属性
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.value = 0
        self.visits = 0
        self.children = []

    def expand(self):
        for move in self.state.possible_moves():
            new_state = self.state.move(move)
            child_node = Node(new_state, self)
```

```
            self.children.append(child_node)

    def is_fully_expanded(self):
        return len(self.children) == len(self.state.possi-
ble_moves())

    def best_child(self, c_param=1.4):
        # 使用 UCT 公式选择最佳子节点
        choices_weights = [
            (child.value / child.visits)
            + c_param * (2 * log(self.visits) / child.vis-
its) ** 0.5
            for child in self.children
        ]
        return self.children[choices_weights.in-
dex(max(choices_weights))]

    def update(self, reward):
        self.visits += 1
        self.value += reward


class MDPState:
    def __init__(self, decisions, parts, semi_products, fi-
nal_product, n_c):
        self.decisions = decisions
        self.parts = parts
        self.semi_products = semi_products
        self.final_product = final_product
        self.n_c = n_c

    def possible_moves(self):
        if len(self.decisions) < 16:  # 有 16 个决策
            return [0, 1]
        return []

    def move(self, decision):
        new_decisions = self.decisions[:]
```

```
        new_decisions.append(decision)
        return MDPState(
            new_decisions, self.parts, self.semi_products,
self.final_product, self.n_c
        )

    def is_terminal(self):
        return len(self.decisions) == 16

    def reward(self):
        # 计算零配件购买成本 C^c_p
        C_c_p = sum(
            n_c * c_p
            for c_p, n_c in zip(
                [part["purchase_price"] for part in
self.parts], self.n_c
            )
        )

        # 计算零配件检测成本 C^c_d
        C_c_d = sum(
            n_c * x * c_d
            for x, c_d, n_c in zip(
                self.decisions[:8],
                [part["inspection_cost"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品次品率 p^f
        p_f = self.final_product["defect_rate"]

        # 检测合格后的零配件数量
        n_c_post_inspection = sum(
            n_c * (1 - x * p_c)
            for x, p_c, n_c in zip(
                self.decisions[:8],
```

```
                [part["defect_rate"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品数量 n^f
        n_f = n_c_post_inspection * (1 - p_f)

        # 计算成品检测成本 C_d_f
        C_d_f = self.final_product["inspection_cost"] *
self.decisions[14] * n_f

        # 计算成品拆解成本 C_a_f
        C_a_f = self.final_product["disassembly_cost"] *
self.decisions[15] * n_f * p_f

        # 计算调换成本 C_s
        C_s = self.final_product["exchange_loss"] *
self.decisions[14] * n_f * p_f

        # 计算成品装配成本 C_s_f
        C_s_f = self.final_product["assembly_cost"] * n_f

        # 计算利润 S
        S = self.final_product["market_price"] * n_f

        # 计算总成本 Z
        Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S

        return Z  # 返回负收益，因为我们是在最小化总成本


# 更新 parts, semi_products, final_product 和 n_c 的值
parts = [
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 1},  # 零配件 1
```

```
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 2
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 3
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 1},  # 零配件 4
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 5
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 6
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 1},  # 零配件 7
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 2},  # 零配件 8
]

semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 1
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 2
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 3
]

final_product = {
```

```
    "defect_rate": 0.10,
    "assembly_cost": 8,
    "inspection_cost": 6,
    "disassembly_cost": 10,
    "market_price": 200,
    "exchange_loss": 40,
}

# n_c = [100] * 8  # 每种零配件数量均为 100
n_c = [1/8] * 8  # 每种零配件数量均为 100
# 初始化状态
initial_state = MDPState([], parts, semi_products, fi-
nal_product, n_c)
root = Node(initial_state)

# 运行 MCTS
MCTS(root, iterations=10000)

# 找到最优决策路径
best_node = max(
    root.children, key=lambda x: x.value / x.visits if
x.visits > 0 else float("-inf")
)


# 构建完整决策路径
def get_decision_path(node):
    decisions = []
    while node.parent is not None:
        decisions.append(node.state.decisions[-1])  # 获取最
新的决策
        node = node.parent
    return decisions[::-1]  # 反转顺序，得到从根节点到目标节点
的决策顺序


def get_full_decision_path(node):
    decisions = []
    while node.parent is not None:
```

```
        decisions.append(node.state.decisions[-1])  # 获取最新的决策
        node = node.parent
    decisions.reverse()  # 反转顺序，得到从根节点到目标节点的决策顺序
    # 确保决策数组长度为 16
    while len(decisions) < 16:
        decisions.append(0)  # 假设未指定的决策默认为 0
    return decisions


# 使用新的函数获取完整决策路径
full_decision_path = get_full_decision_path(best_node)

print(
    "最优决策路径:",
    full_decision_path,
    "预期收益:",
    -best_node.value / best_node.visits,
)
```

```
import random
from math import log
import matplotlib.pyplot as plt

plt.rcParams["font.sans-serif"] = ["SimHei"]  # 用黑体显示中文
plt.rcParams["axes.unicode_minus"] = False  # 正确显示负号

import networkx as nx


class Node:
    node_counter = 0  # 静态变量，给每个节点分配唯一 ID

    def __init__(self, state, parent=None):
        self.id = Node.node_counter
        Node.node_counter += 1
```

```
        self.state = state
        self.parent = parent
        self.value = 0
        self.visits = 0
        self.children = []

    def is_fully_expanded(self):
        return len(self.children) == len(self.state.possi-
ble_moves())

    def best_child(self, c_param=1.4):
        choices_weights = [
            (child.value / child.visits)
            + c_param * (2 * log(self.visits) / child.vis-
its) ** 0.5
            for child in self.children
        ]
        return self.children[choices_weights.in-
dex(max(choices_weights))]

    def update(self, reward):
        self.visits += 1
        self.value += reward


class MDPState:
    def __init__(self, decisions, parts, semi_products, fi-
nal_product, n_c):
        self.decisions = decisions
        self.parts = parts
        self.semi_products = semi_products
        self.final_product = final_product
        self.n_c = n_c

    def possible_moves(self):
        if len(self.decisions) < 16:  # 有 16 个决策
            return [0, 1]
        return []
```

```
    def move(self, decision):
        new_decisions = self.decisions[:]
        new_decisions.append(decision)
        return MDPState(
            new_decisions, self.parts, self.semi_products,
self.final_product, self.n_c
        )

    def is_terminal(self):
        return len(self.decisions) == 16

    def reward(self):
        # 计算成本和利润的模拟函数
        return -random.randint(0, 100)  # 返回负值模拟成本


def MCTS(root, iterations, c_param=1.4):
    for _ in range(iterations):
        node = root
        # 1. Selection
        while node.is_fully_expanded() and not
node.state.is_terminal():
            node = node.best_child(c_param=c_param)

        # 2. Expansion
        if not node.is_fully_expanded() and not
node.state.is_terminal():
            possible_moves = node.state.possible_moves()
            move = random.choice(possible_moves)
            new_state = node.state.move(move)
            child_node = Node(new_state, node)
            node.children.append(child_node)
            node = child_node

        # 3. Simulation
        reward = simulate(node.state)
```

```
        # 4. Backpropagation
        while node is not None:
            node.update(reward)
            node = node.parent


def simulate(state):
    # 这里需要定义一个模拟函数来评估从当前状态开始的随机玩法的结
果
    while not state.is_terminal():
        possible_moves = state.possible_moves()
        move = random.choice(possible_moves)
        state = state.move(move)
    return state.reward()


def get_best_reward(node):
    # 初始化最优奖励和对应节点
    best_reward = float("-inf")
    best_node = None

    # 使用递归函数来遍历所有节点
    def search_best(node):
        nonlocal best_reward, best_node
        if node.visits > 0:
            avg_reward = node.value / node.visits
            if avg_reward > best_reward:
                best_reward = avg_reward
                best_node = node
        for child in node.children:
            search_best(child)

    search_best(node)
    return best_reward


# 初始化状态
parts = [{"defect_rate": 0.10, "purchase_price": 2, "in-
spection_cost": 1}] * 8
```

```
semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    }
] * 3
final_product = {
    "defect_rate": 0.10,
    "assembly_cost": 8,
    "inspection_cost": 6,
    "disassembly_cost": 10,
    "market_price": 200,
    "exchange_loss": 40,
}
n_c = [1] * 8  # 每种零配件数量均为 100
initial_state = MDPState([], parts, semi_products, fi-
nal_product, n_c)
root = Node(initial_state)

# 运行 MCTS
MCTS(root, iterations=1000)


def sensitivity_analysis(root, iterations=1000, c_val-
ues=[0.1, 0.5, 1.0, 1.4, 2.0]):
    results = []
    for c in c_values:
        MCTS(root, iterations, c_param=c)
        best_reward = get_best_reward(root)
        results.append((c, best_reward))

    c_vals, rewards = zip(*results)
    plt.figure()
    plt.plot(c_vals, rewards, marker="o")
    plt.xlabel("Exploration Parameter c")
    plt.ylabel("Best Reward")
```

```
    plt.title("Sensitivity Analysis on Exploration Parameter c")
    plt.grid(True)
    plt.show()


# 调用敏感性分析函数
sensitivity_analysis(root)
```

```
import plotly.express as px
import pandas as pd

# 更新 parts, semi_products, final_product 和 n_c 的值
parts = [
    {"defect_rate": 0.10, "purchase_price": 2, "inspection_cost": 1},  # 零配件 1
    {"defect_rate": 0.10, "purchase_price": 8, "inspection_cost": 1},  # 零配件 2
    {"defect_rate": 0.10, "purchase_price": 12, "inspection_cost": 2},  # 零配件 3
    {"defect_rate": 0.10, "purchase_price": 2, "inspection_cost": 1},  # 零配件 4
    {"defect_rate": 0.10, "purchase_price": 8, "inspection_cost": 1},  # 零配件 5
    {"defect_rate": 0.10, "purchase_price": 12, "inspection_cost": 2},  # 零配件 6
    {"defect_rate": 0.10, "purchase_price": 8, "inspection_cost": 1},  # 零配件 7
    {"defect_rate": 0.10, "purchase_price": 12, "inspection_cost": 2},  # 零配件 8
]

semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
```

```
        "disassembly_cost": 6,
    },  # 半成品 1
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 2
    {
        "defect_rate": 0.10,
        "assembly_cost": 8,
        "inspection_cost": 4,
        "disassembly_cost": 6,
    },  # 半成品 3
]

final_product = {
    "defect_rate": 0.10,
    "assembly_cost": 8,
    "inspection_cost": 6,
    "disassembly_cost": 10,
    "market_price": 200,
    "exchange_loss": 40,
}

n_c = [1/8] * 8  # 每种零配件数量均为 100


class MDPState:
    def __init__(self, decisions, parts, semi_products, fi-
nal_product, n_c):
        self.decisions = decisions
        self.parts = parts
        self.semi_products = semi_products
        self.final_product = final_product
        self.n_c = n_c

    def possible_moves(self):
```

```
        if len(self.decisions) < 16:  # 共有 16 个决策
            return [0, 1]  # 每个决策有两个选项，0 或 1
        return []

    def move(self, decision):
        new_decisions = self.decisions[:]
        new_decisions.append(decision)
        return MDPState(
            new_decisions, self.parts, self.semi_products,
self.final_product, self.n_c
        )

    def is_terminal(self):
        return len(self.decisions) == 16  # 当决策长度达到 16
时，终止

    def reward(self):
        # 计算零配件购买成本 C^c_p
        C_c_p = sum(
            n_c * c_p
            for c_p, n_c in zip(
                [part["purchase_price"] for part in
self.parts], self.n_c
            )
        )

        # 计算零配件检测成本 C^c_d
        C_c_d = sum(
            n_c * x * c_d
            for x, c_d, n_c in zip(
                self.decisions[:8],
                [part["inspection_cost"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品次品率 p^f
```

```
        p_f = self.final_product["defect_rate"]

        # 检测合格后的零配件数量
        n_c_post_inspection = sum(
            n_c * (1 - x * p_c)
            for x, p_c, n_c in zip(
                self.decisions[:8],
                [part["defect_rate"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品数量 n^f
        n_f = n_c_post_inspection * (1 - p_f)

        # 计算成品检测成本 C_d_f
        C_d_f = self.final_product["inspection_cost"] *
self.decisions[14] * n_f

        # 计算成品拆解成本 C_a_f
        C_a_f = self.final_product["disassembly_cost"] *
self.decisions[15] * n_f * p_f

        # 计算调换成本 C_s
        C_s = self.final_product["exchange_loss"] *
self.decisions[14] * n_f * p_f

        # 计算成品装配成本 C_s_f
        C_s_f = self.final_product["assembly_cost"] * n_f

        # 计算利润 S
        S = self.final_product["market_price"] * n_f

        # 计算总成本 Z
        Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S

        return -Z  # 返回负收益，因为我们是在最小化总成本
```

```
# 初始化状态
initial_state = MDPState([], parts, semi_products, fi-
nal_product, n_c)
path_rewards = []


# 运行穷举法
def exhaustive_search(state, path_rewards, cur-
rent_path=[]):
    if state.is_terminal():
        reward = state.reward()
        # 将二进制决策列表转换为 16 进制字符串
        hex_path = "".join(format(x, "b") for x in cur-
rent_path).zfill(16)
        hex_path = f"{int(hex_path, 2):X}"  # 转换为 16 进制表
示
        path_rewards.append((hex_path, reward))
        return reward, current_path

    best_reward = float("-inf")
    best_decision_path = []

    # 穷举每一个可能的决策
    for move in state.possible_moves():
        next_state = state.move(move)
        reward, decision_path = exhaustive_search(
            next_state, path_rewards, current_path + [move]
        )

        if reward > best_reward:
            best_reward = reward
            best_decision_path = decision_path

    return best_reward, best_decision_path
```

```
best_reward, best_decision_path = exhaustive_search(ini-
tial_state, path_rewards)

# 准备数据用于 Plotly
data = pd.DataFrame(path_rewards, columns=["Decision Path",
"Reward"])

# 找到最大值所在的行
max_reward_idx = data["Reward"].idxmax()
data["Marker"] = ["Max" if i == max_reward_idx else "Other"
for i in data.index]

# 使用 Plotly 创建交互式散点图，并标记最大值
fig = px.scatter(
    data,
    x="Decision Path",
    y="Reward",
    color="Marker",  # 使用不同的颜色标记最大值
    labels={"Decision Path": "Decision Path (Hex)", "Re-
ward": "Reward"},
    title="Decision Paths and Their Rewards",
    color_discrete_map={"Max": "red", "Other": "blue"},  #
红色标记最大值
)

# 增加一个特殊的标记用于突出显示最大奖励值
fig.add_trace(
    px.scatter(
        data[data["Marker"] == "Max"],
        x="Decision Path",
        y="Reward",
        size=[10],  # 使用更大的点大小
        color="Marker",
        color_discrete_map={"Max": "red"},
    ).data[0]
)

# 在最大值点上添加文本标签
```

```
fig.add_annotation(
    x=data.loc[max_reward_idx, "Decision Path"],
    y=data.loc[max_reward_idx, "Reward"],
    text="最大奖励",
    showarrow=True,
    font=dict(family="Courier New, monospace", size=16,
color="#ffffff"),
    align="center",
    arrowhead=2,
    arrowsize=1,
    arrowwidth=2,
    arrowcolor="#636363",
    ax=20,
    ay=-30,
    bordercolor="#c7c7c7",
    borderwidth=2,
    borderpad=4,
    bgcolor="#ff7f0e",
    opacity=0.8,
)

fig.show()

print("Exhaustive Search Found Best Decision Path:",
best_decision_path)
print("Exhaustive Search Found Best Reward:", best_reward)
```

问题四源程序

```
import math
from scipy import stats


def calculate_defect_rate(defects, total_samples, confi-
dence_level=0.95):
    """
    计算次品率和置信区间
```

```
    :param defects: 次品数量
    :param total_samples: 样本总数
    :param confidence_level: 置信水平 (默认 95%)
    :return: 次品率, 置信区间 (下限, 上限)
    """
    defect_rate = defects / total_samples
    standard_error = math.sqrt((defect_rate * (1 - de-
fect_rate)) / total_samples)
    z_value = stats.norm.ppf(1 - (1 - confidence_level) /
2)
    lower_bound = defect_rate - z_value * standard_error
    upper_bound = defect_rate + z_value * standard_error
    lower_bound = max(0, lower_bound)
    upper_bound = min(1, upper_bound)

    return defect_rate, (lower_bound, upper_bound)


def calculate_half_product_defect_rate(component_de-
fect_rates):
    """
    根据多个零配件的次品率计算半成品的次品率
    假设零配件不合格则半成品不合格
    :param component_defect_rates: 零配件次品率的列表
    :return: 半成品次品率
    """
    combined_rate = 1
    for rate in component_defect_rates:
        combined_rate *= 1 - rate
    return 1 - combined_rate


def calculate_final_product_defect_rate(half_product_de-
fect_rates):
    """
    根据多个半成品的次品率计算成品的次品率
    :param half_product_defect_rates: 半成品次品率的列表
    :return: 成品次品率
    """
```

```
    combined_rate = 1
    for rate in half_product_defect_rates:
        combined_rate *= 1 - rate
    return 1 - combined_rate


# 示例使用：
# 假设 8 个零配件的抽样检测数据如下：
defects_components = [10.0, 8.2, 5.5, 10.7, 7.7, 9.0, 6.6,
10.0]
total_samples_components = [100, 100, 100, 100, 100, 100,
100, 100]

# 计算 8 个零配件的次品率
component_defect_rates = []
for defects, total_samples in zip(defects_components, to-
tal_samples_components):
    defect_rate, _ = calculate_defect_rate(defects, to-
tal_samples)
    component_defect_rates.append(defect_rate)

print("零配件次品率：", [f"{rate*100:.2f}%" for rate in com-
ponent_defect_rates])

# 假设半成品的组成如下（每个半成品由不同零配件组合而成）
# 半成品 1 由零配件 1, 2, 3 组成
# 半成品 2 由零配件 4, 5, 6 组成
# 半成品 3 由零配件 7, 8 组成
half_product_1_defect_rate = calculate_half_product_de-
fect_rate(
    [component_defect_rates[0], component_defect_rates[1],
component_defect_rates[2]]
)
half_product_2_defect_rate = calculate_half_product_de-
fect_rate(
    [component_defect_rates[3], component_defect_rates[4],
component_defect_rates[5]]
)
```

```
half_product_3_defect_rate = calculate_half_product_de-
fect_rate(
    [component_defect_rates[6], component_defect_rates[7]]
)

print(f"半成品 1 的次品率: {half_product_1_de-
fect_rate*100:.2f}%")
print(f"半成品 2 的次品率: {half_product_2_de-
fect_rate*100:.2f}%")
print(f"半成品 3 的次品率: {half_product_3_de-
fect_rate*100:.2f}%\n")

# 计算成品次品率 (由三个半成品装配成)
final_product_defect_rate = calculate_final_product_de-
fect_rate(
    [half_product_1_defect_rate, half_product_2_de-
fect_rate, half_product_3_defect_rate]
)

print(f"成品的次品率: {final_product_defect_rate*100:.2f}%")
```

```
import random
from math import log
from scipy.stats import beta  # 用于贝叶斯更新的 Beta 分布


# 贝叶斯自适应马尔可夫决策中的 Beta 分布
class BetaDistribution:
    def __init__(self, alpha=1, beta=1):
        self.alpha = alpha
        self.beta = beta

    def sample(self):
        # 从当前分布中采样
        return random.betavariate(self.alpha, self.beta)

    def update(self, success):
```

```
        # 更新贝叶斯分布：成功则增加 alpha，失败则增加 beta
        self.alpha += success
        self.beta += 1 - success


# 使用贝叶斯更新的 MDP 状态
class BAMDPState:
    def __init__(
        self, decisions, parts, semi_products, final_prod-
uct, n_c, prior_beliefs
    ):
        self.decisions = decisions
        self.parts = parts
        self.semi_products = semi_products
        self.final_product = final_product
        self.n_c = n_c
        self.prior_beliefs = prior_beliefs  # 不确定性信念

    def possible_moves(self):
        if len(self.decisions) < 16:  # 有 16 个决策
            return [0, 1]
        return []

    def move(self, decision):
        new_decisions = self.decisions[:]
        new_decisions.append(decision)
        # 产生新的状态
        return BAMDPState(
            new_decisions,
            self.parts,
            self.semi_products,
            self.final_product,
            self.n_c,
            self.prior_beliefs,
        )

    def is_terminal(self):
        return len(self.decisions) == 16
```

```
    def reward(self):
        # 用贝叶斯分布采样来更新我们对各零配件缺陷率的信念
        sampled_defect_rates = [
            self.prior_beliefs[i].sample() for i in
range(len(self.parts))
        ]

        # 计算零配件购买成本
        C_c_p = sum(
            n_c * c_p
            for c_p, n_c in zip(
                [part["purchase_price"] for part in
self.parts], self.n_c
            )
        )

        # 计算零配件检测成本
        C_c_d = sum(
            n_c * x * c_d
            for x, c_d, n_c in zip(
                self.decisions[:8],
                [part["inspection_cost"] for part in
self.parts],
                self.n_c,
            )
        )

        # 成品次品率
        p_f = self.final_product["defect_rate"]

        # 使用采样的次品率计算检测合格后的零配件数量
        n_c_post_inspection = sum(
            n_c * (1 - x * p_c)
            for x, p_c, n_c in zip(self.decisions[:8], sam-
pled_defect_rates, self.n_c)
        )
```

```
        # 成品数量
        n_f = n_c_post_inspection * (1 - p_f)

        # 计算其他相关成本
        C_d_f = self.final_product["inspection_cost"] *
self.decisions[14] * n_f
        C_a_f = self.final_product["disassembly_cost"] *
self.decisions[15] * n_f * p_f
        C_s = self.final_product["exchange_loss"] *
self.decisions[14] * n_f * p_f
        C_s_f = self.final_product["assembly_cost"] * n_f

        # 计算利润
        S = self.final_product["market_price"] * n_f

        # 总成本
        Z = C_c_p + C_c_d + C_d_f + C_a_f + C_s + C_s_f - S

        return Z

    # 贝叶斯更新每次根据观测值调整参数
    def update_beliefs(self, observed_results):
        for i, success in enumerate(observed_results):
            self.prior_beliefs[i].update(success)


# 在 Node 类中添加 parent 属性
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.value = 0
        self.visits = 0
        self.children = []

    def expand(self):
        for move in self.state.possible_moves():
            new_state = self.state.move(move)
```

```
            child_node = Node(new_state, self)
            self.children.append(child_node)

    def is_fully_expanded(self):
        return len(self.children) == len(self.state.possi-
ble_moves())

    def best_child(self, c_param=1.4):
        # 使用 UCT 公式选择最佳子节点
        choices_weights = [
            (child.value / child.visits)
            + c_param * (2 * log(self.visits) / child.vis-
its) ** 0.5
            for child in self.children
        ]
        return self.children[choices_weights.in-
dex(max(choices_weights))]

    def update(self, reward):
        self.visits += 1
        self.value += reward


# 模拟函数返回模拟的结果及观测值
def simulate(state):
    observed_results = []

    # 生成观测结果，使其长度与零配件数量一致
    for i in range(len(state.parts)):
        observed_results.append(random.choice([0, 1]))  # 随
机产生 0 或 1 表示观测到的结果

    while not state.is_terminal():
        possible_moves = state.possible_moves()
        move = random.choice(possible_moves)
        state = state.move(move)

    return state.reward(), observed_results
```

```
# MCTS 算法的修改
def MCTS(root, iterations):
    for _ in range(iterations):
        node = root
        # 1. Selection
        while node.is_fully_expanded() and not
node.state.is_terminal():
            node = node.best_child()

        # 2. Expansion
        if not node.is_fully_expanded() and not
node.state.is_terminal():
            possible_moves = node.state.possible_moves()
            move = random.choice(possible_moves)
            new_state = node.state.move(move)
            child_node = Node(new_state, node)
            node.children.append(child_node)
            node = child_node

        # 3. Simulation
        reward, observed_results = simulate(node.state)

        # 4. Backpropagation and Bayesian update
        while node is not None:
            node.update(reward)
            node.state.update_beliefs(observed_results)  #
更新贝叶斯信念
            node = node.parent


# 找到最优决策路径
def get_full_decision_path(node):
    decisions = []
    while node.parent is not None:
        decisions.append(node.state.decisions[-1])  # 获取最
新的决策
        node = node.parent
```

```
    decisions.reverse()  # 反转顺序，得到从根节点到目标节点的决策顺序
    # 确保决策数组长度为 16
    while len(decisions) < 16:
        decisions.append(0)  # 假设未指定的决策默认为 0
    return decisions


parts = [
    {"defect_rate": 0.10, "purchase_price": 2, "inspection_cost": 4.0},  # 零配件 1
    {"defect_rate": 0.10, "purchase_price": 8, "inspection_cost": 4.0},  # 零配件 2
    {"defect_rate": 0.10, "purchase_price": 12, "inspection_cost": 4.0},  # 零配件 3
    {
        "defect_rate": 0.10,
        "purchase_price": 2,
        "inspection_cost": 0.0,
    },  # 零配件 4 (无检测成本)
    {
        "defect_rate": 0.10,
        "purchase_price": 8,
        "inspection_cost": 0.0,
    },  # 零配件 5 (无检测成本)
    {
        "defect_rate": 0.10,
        "purchase_price": 12,
        "inspection_cost": 0.0,
    },  # 零配件 6 (无检测成本)
    {
        "defect_rate": 0.10,
        "purchase_price": 8,
        "inspection_cost": 0.0,
    },  # 零配件 7 (无检测成本)
    {
        "defect_rate": 0.10,
        "purchase_price": 12,
```

```
        "inspection_cost": 0.0,
    },  # 零配件 8 (无检测成本)
]

semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 1
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 2
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 3
]

final_product = {
    "defect_rate": 0.10,
    "assembly_cost": 8.0,
    "inspection_cost": 6.0,
    "disassembly_cost": 10.0,
    "market_price": 200,
    "exchange_loss": 40.0,
}


n_c = [1 / 8] * 8  # 每种零配件数量均为相同比例

# 初始化贝叶斯先验信念
```

```
prior_beliefs = [BetaDistribution(2, 3) for _ in parts]  # 设置每个零件的初始信念

# 初始状态
initial_state = BAMDPState([], parts, semi_products, final_product, n_c, prior_beliefs)
root = Node(initial_state)

# 运行 MCTS
MCTS(root, iterations=10000)

# 找到最优决策路径
best_node = max(
    root.children, key=lambda x: x.value / x.visits if x.visits > 0 else float("-inf")
)

# 输出最优决策路径及其预期收益
full_decision_path = get_full_decision_path(best_node)
print(
    "最优决策路径:",
    full_decision_path,
    "预期收益:",
    -best_node.value / best_node.visits,
)
```

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


# 假设 BAMDPState 是表示当前状态的类，允许根据路径进行决策和计算收益
class BAMDPState:
    def __init__(self, path, parts, semi_products, final_product, n_c, prior_beliefs):
        self.path = path  # 当前路径
        self.parts = parts  # 零配件列表
        self.semi_products = semi_products  # 半成品列表
```

```
        self.final_product = final_product  # 成品信息
        self.n_c = n_c  # 一些与状态相关的计数器或参数
        self.prior_beliefs = prior_beliefs  # 先验信息

    def reward(self):
        # 在这里计算路径的收益，使用 self.path 来计算
        # 示例收益计算（根据项目实际逻辑进行调整）
        reward = 0
        for decision in self.path:
            if decision == 1:
                reward += 10  # 示例逻辑，选择了这个决策会有一
定收益
        return reward


# 使用穷举法生成所有可能的决策路径
def generate_all_decision_paths():
    return list(itertools.product([0, 1], repeat=16))


# 计算所有路径的收益并存储结果
def evaluate_all_paths(root_state):
    decision_paths = generate_all_decision_paths()
    results = []

    for path in decision_paths:
        # 更新 state 为当前路径下的决策
        state = BAMDPState(
            list(path),
            root_state.parts,
            root_state.semi_products,
            root_state.final_product,
            root_state.n_c,
            root_state.prior_beliefs,
        )
        reward = state.reward()  # 计算当前路径的收益
        results.append((path, reward))

    return results
```

```python
# 将决策路径转换为可视化的数据格式
def paths_to_dataframe(results):
    paths = []
    rewards = []
    for path, reward in results:
        paths.append(list(path))
        rewards.append(reward)

    # 创建一个包含路径和收益的 DataFrame
    df = pd.DataFrame(paths, columns=[f"decision_{i+1}" for
i in range(16)])
    df["reward"] = rewards
    return df


# 绘制每种决策路径下的收益散点图，并标记最大值
def plot_reward_scatter(df):
    plt.figure(figsize=(10, 6))
    plt.scatter(range(len(df)), df["reward"], alpha=0.5)

    # 标记最大值
    max_reward_index = df["reward"].idxmax()
    max_reward_value = df["reward"].max()
    plt.scatter(
        max_reward_index,
        max_reward_value,
        color="red",
        label=f"Max Reward: {max_reward_value}",
    )

   # plt.title("Scatter Plot of Rewards for Each Decision
Path")
    plt.xlabel("Decision Path Index")
    plt.ylabel("Reward")
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
# 零件信息和初始状态（示例数据）
parts = [
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 4.0},  # 零配件 1
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 4.0},  # 零配件 2
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 4.0},  # 零配件 3
    {"defect_rate": 0.10, "purchase_price": 2, "inspec-
tion_cost": 0.0},  # 零配件 4
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 0.0},  # 零配件 5
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 0.0},  # 零配件 6
    {"defect_rate": 0.10, "purchase_price": 8, "inspec-
tion_cost": 0.0},  # 零配件 7
    {"defect_rate": 0.10, "purchase_price": 12, "inspec-
tion_cost": 0.0},  # 零配件 8
]

semi_products = [
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 1
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 2
    {
        "defect_rate": 0.10,
        "assembly_cost": 8.0,
```

```
        "inspection_cost": 4.0,
        "disassembly_cost": 6.0,
    },  # 半成品 3
]

final_product = {
    "defect_rate": 0.10,
    "assembly_cost": 8.0,
    "inspection_cost": 6.0,
    "disassembly_cost": 10.0,
    "market_price": 200,
    "exchange_loss": 40.0,
}

# 先验信息和其他状态参数
n_c = 3  # 示例值
prior_beliefs = [0.1, 0.1, 0.1]  # 示例值

# 创建初始状态
initial_state = BAMDPState([], parts, semi_products, fi-
nal_product, n_c, prior_beliefs)

# 使用穷举法评估所有路径
results = evaluate_all_paths(initial_state)

# 转换为 DataFrame 以便可视化
df = paths_to_dataframe(results)

# 绘制收益散点图，并标记最大值
plot_reward_scatter(df)
```