##### 基于遗传算法求解生产过程中的决策问题

# 摘要

学习工厂成本与决策，对推动生产力发展有着重要意义。本文针对某工厂的生产过程中的决策问题，提出了使用遗传算法的优化模型，以提高生产效益并降低成本。首先本文对工厂的总体生产环境做出分析，设计决策序列来简化决策过程，通过代码模拟工厂生产产品并获得收益的过程来做出最优决策；其次，使用遗传算法来加速寻找最优决策，并对模型做出稳定性分析，保证结果的有效性和可行性；最后，模拟真实的产品可能出现的次品率以应付实际生产中更复杂的情况。

本文主要解决以下问题：

**针对问题1：**通过抽样方案设计，利用置信水平和假设检验标准的正态分布来判断是否接收和拒绝供应商提供的零件。并给出了具体的决策：在95%置信度下，抽样139个样品，若超过20个次品，则不接受该批零件；在90%置信度下，抽样98个样品，若不超过14个次品，则接受该批零件。

**针对问题2：**使用代码实现模拟工厂运行，穷举出6种情况的十六种可行方案，并针对每一个情况计算出收益最大化的一种方案。

**针对问题3：**同样使用代码实现模拟更复杂的工厂运行，并在此基础上使用遗传算法来优化决策过程，极大提高了决策速度。最后给出了在复杂工厂生产环境下的最优决策。

**针对问题4：**在问题1的基础上，针对问题2、3，引入次品率误差来更好的模拟现实生产环境，重新评估问题2、3，并给出了更贴合实际的决策。

本文所建立的模型具有快收敛，运行速度快的特点，以及在组合优化上具有通用性，便于推广，易于学习。

**关键词：**遗传算法 决策优化 置信度 抽样检测 生产模拟 利润分析

# 问题重述

随着现代科技的发展，携带电子技术的产品在市场经济的占额比逐步上升。某产品由两个零件装配而成，如果有某一件零件不合格，则判为是不合格产品；如果双零件都通过了检验，也不一定能够被判为合格产品。为解决这些不合格的产品，商家可以花费一些费用来拆解，进行维修并重新装配。现有一家商家生产某电子产品的多阶段分配问题需要解决。

请回答下列问题：

**问题一：**供应商生称该批零配件标称值为10%，要求为一下两种情况设计抽样检测方案。若在95%的置信水平下有充分的证据表明该批零配件次品率高于标称值，则可以拒绝接受零配件；若在90%的置信水平下有充足的证据表明该批零配件次品率低于标称值，则可以接收零配件。

**问题二：**企业生产某电子产品有四个阶段，给出零配件的次品率、购买单价、检测成本和组装成的成品的装配成本等数据，并给出6种不同的情况，要求对生产过程中的6种情况做出不同阶段下的决策。

**问题三：**在问题二的基础上，将面对m道工序、n个零配件的组装情况，给出2道工序和8个零配件在企业生产过程中遇到的情况，请为各个阶段做出决策优化，来使效益最大化。

**问题四：**假设通过抽样检测的方法，得到了问题2和问题3中的零配件、半成品和成品的次品率，需要重新求解问题2和问题3，给出决策优化。

# 问题分析

问题的研究对象是企业生产某种电子产品过程中的决策问题，生产过程中有检测、装配、拆解等阶段，需要调和各个阶段来实现效益最大化。

## 对问题一的分析

题目要求我们为企业设计出一种抽样检测方案来判断是接受供应商的配件还是拒绝配件。拒绝的情况是，在95%的置信区间内，配件的次品率如果超过标称值的10%，那么就不接受该批配件；接受的情况是，在90%的置信区间内，配件的次品率小于或等于供应商给的标称值10%，那么就可以接受这批配件。本题涉及到统计学问题，需要计算样本容量并通过假设检验来求解最合适的拒绝域和接受域。题干中并没有明确给出样本的容量，所以样本将面临两种情况，一是样本数量较少，小于30；二是样本数量较多，大于30。零配件只会出现是次品和不是次品的情况，符合二项分布情况，对于样本较少的情况，适用二项分布的精确求解来计算样本容量，但当样本不断增大，二项分布就会无限接近于正态分布的情况，这时适用正态分布来进行计算样本容量。然后通过假设校验方法来计算得出两种置信情况下的临界值，来得出两种置信情况下的检测方案。

## 对问题二的分析

题目中的企业定义某电子产品的生产四个阶段。第一阶段，从零配件1和零配件2作为起始，可选择检测和不检测，检测需要花费一定的成本，不检测则是直接丢弃；在第二阶段，零配件1和零配件2直接装配为成品，该生产阶段可以选择是否检测，同样需要花费一定的成本进行检测，不检测将直接进入市场；第三阶段，成品会进入到市场，市场中的用户会可能得到合格或不合格的产品，该阶段算作是产品中的一个检测环节；第四阶段，拿到不合格产品的用户可以选择调换产品，企业将产生调换损失，如果选择拆卸，将会产生拆解成本，但能将两种零配件进行回收。题目设计到统计学内容要求我们在不同的情况下的各种不同阶段中，以是否检测、是否拆解进行决策，来调和流程，将整条生产线中的生产成本降到最低，来将最终效益达到最高。下图是描绘的题目二的产品生产流程图。

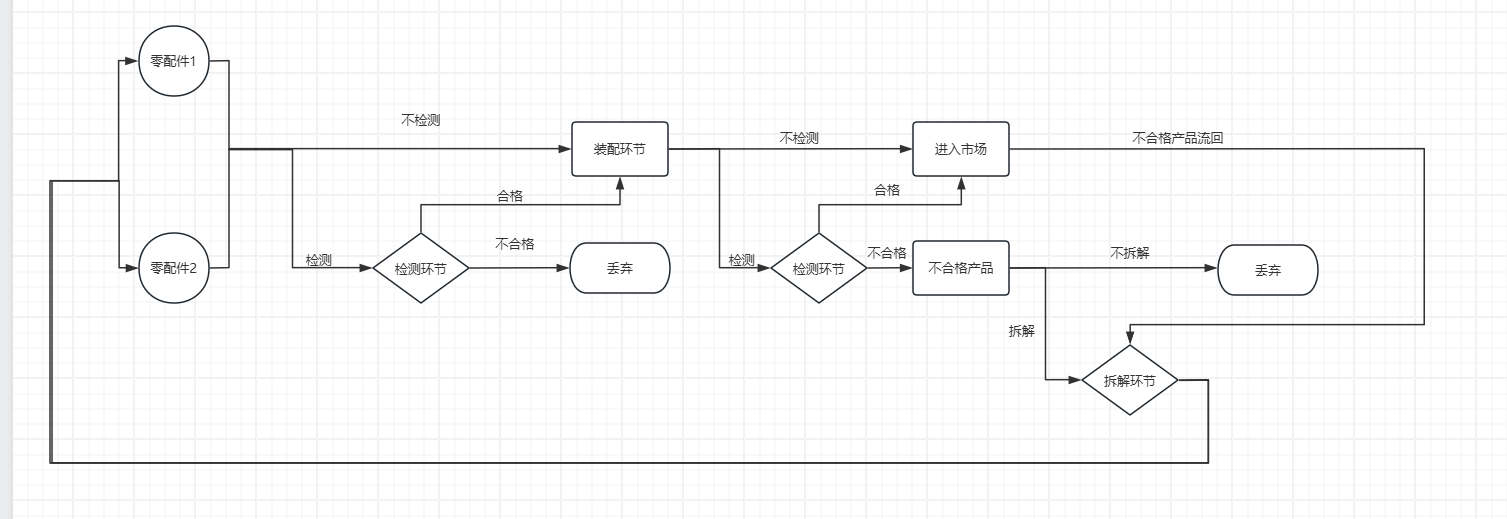


图1 问题二 产品生产阶段流程图

## 对问题三的分析

该问题三涉及了m道工序，n个零配件的多阶段生产流程决策。零配件、半成品、成品和成品之间存在着检测环节和拆解环节，是有关于多样式的组合优化问题。每个零配件、半成品和成品在制造和检测过程中都存在一定的次品率，需要我们去调配不同阶段的是否检测和是否拆解 。该题的核心是通过建立决策模型来优化生产过程中的各项成本，例如零配件的检测成本、成品的检测和拆解等费用。当涉及到多样且复杂的组合优化问题的时候，传统的方法显得难以求解，采用遗传算法进行优化是一种有效的解决方案。

## 对问题四的分析

问题四：已知问题2，问题3的零配件、半成品和成品的次品率是基于问题1通过抽样检测的方法得到。但在问题1中我们可知此次品率存在波动性。因此将波动性的次品率重新套入问题2，问题3的模型，重新得到决策。又因为波动性具有随机性，因此我们分别多次验算问题2和问题3，取出最优利润的计算

# 模型假设

* + 1. 假设可以接受的次品率误差为5%。
    2. 假设2，3 题目提供的次品率是理想的。
    3. 假设零件为一次性购买，此后不再购买。
    4. 假设流入市场的次品一定会被送回。
    5. 假设拆解过程不会对零配件造成损坏。

# 变量说明

|  |  |  |  |
| --- | --- | --- | --- |
| **符号** | **意义** | **符号** | **意义** |
|  | 零配件或成品的总数 |  | 零配件2的检测成本 |
|  | 零配件或成品次品率 |  | 零配件2的检测费用 |
|  | 检测或拆解成本 |  | 流入到装配环节的零配件2数量 |
|  | 标称次品率 |  | 装配成品的花费 |
|  | 实际次品率 |  | 能够组装成成品的数量 |
|  | 样本容量 |  | 两个零配件装配成一个成品的装配成本 |
|  | 样本中次品的数量 |  | 装配完成后不合格品的数量 |
|  | 零假设 |  | 装配完成后不合格品的数量 |
|  | 备择假设 |  | 检测成品的成本 |
|  | 与第一类错误的显著性水平a相关的  标准正态分布临界值 |  | 拆解的花费 |
|  | 第二类错误的显著性水平 |  | 拆解成本 |
|  | 总体的标准差 |  | 市场中的损失花费 |
|  | 可接受的误差范围 |  | 市场中的调换成本 |
|  | 零配件1的总数 |  | 每件成品的单价 |
|  | 零配件1的次品率 |  | 第i个零配件的总数 |
|  | 零配件1的检测成本 |  | 第i个零配件的次品率 |
|  | 零配件1的检测费用 |  | 第i个零配件的检测成本 |
|  | 流入到装配环节的零配件1数量 |  | 第i个零配件的检测费用 |
|  | 零配件2的总数 |  | 流入到下一个阶段的第i个零配件数量 |
|  | 零配件2的次品率 |  | 第一组零配件经过检测环节后能够组装成半成品的数量 |
|  | 第二个半成品的装配花费 |  | 第二组零配件经过检测环节后能够组装成半成品的数量 |
|  | 第三个半成品的装配花费 |  | 第三组零配件经过检测环节后能够组装成半成品的数量 |
|  | 重新评估后的零配件数量 |  | 半成品装配的花费 |
|  | 检测成品的花费 |  | 第一个半成品的装配花费 |
|  | 拆解的花费 |  |  |
|  | 最终获取的综合利润 |  |  |

# 模型的建立与求解

## 问题一：

为了设计出检测次数尽可能少的抽样检测方案，采用统计学中的假设检验方法。由题知，样本出现的情况只有两种，一种是该零配件是次品，二是该零配件不是次品，所以统计模型采用二项分布。

### 设定参数和假设：

* 标称次品率P0，真正的次品率P
* 次品的数量X，检测的样品容量为n
* 零假设 H0：零配件的次品率P≤P0
* 备择假设 H1：零配件的次品率P> P0

可得二项式分布式。

题目中并没有给出抽样检测中的样本容量，所以需要分两种情况：

1. 样本容量较小，小于30
2. 样本容量足够大，大于30

### 在样本容量较小的情况，小于30：

使用二项分布的精确计算方法保证样本容量的准确性，逆累积分布函数

最终求解的出的结果：

要想在95%置信区间下认定零配件次品率高过标称值，则拒收这批零配件，至少需要抽样1次，即样本容量n=1。

要想在90%置信区间下认定零配件次品率低于标称值，则接收这批零配件，至少需要抽样1次，即样本容量n=1。

由此可知，该题的样本容量大小不会是在小于30的情况。

### 在样本容量足够大的情况，大于30:

当样本的容量增加到一定的数量，根据中心极限定理可知，二项分布以正态分布为极限。

由De Moivre Laplace 定理可知，

积分极限的定理：对于任意的，恒有

这个定理说明了，当二项分布的样本容量n充分大的时候，二项分布的极限是向正态分布收敛的。

所以在样本容量有限大的情况下，使用正态近似比例的假设检验。

正态分布：

其中：是样本中的次品率估计值

求解样本容量：

使用正态分布的z校验的样本容量n计算公式

是与第一类错误的显著性水平α相关的标准正态分布的临界值。

是总体的标准差。

是可接受的误差范围，也就是效应量，期望次品率和实际次品率的差值。

由题目可知：

1. 在95%的置信区间下就拒绝接受这批零配件，认为这些零配件的次品率超过标称值。
2. 在90%的置信区间下就接受这批零配件，认为这些零配件的次品率不超过标称值。

对于95%置信区间的拒绝接受条件，样本容量可计算：是1.96，β设定0.2

对于90%置信区间的接收条件，样本容量可以计算：是1.645，β设定0.2

在高95%置信区间下拒绝这批零配件的情况下，高置信度意味着需要更高的确信度来认为实际次品率确实高于标称次品率，所以假设真实的次品率 为15%。

计算得出在95%置信度之下，当检测到139个零配件时候，如果次品率超过了10%，那么可以决绝接收该批零配件。

在高90%置信区间下拒绝这批零配件的情况下，较低的置信度意味着接收这批零配件的准测较为的宽松，所以假设真实的次品率 为5%。

计算得出在95%置信度之下，当检测到98个零配件时候，如果次品率超过了10%，那么可以接收该批零配件。

求解出在不同的允许误差范围内的样本容量n值，如图1和图2

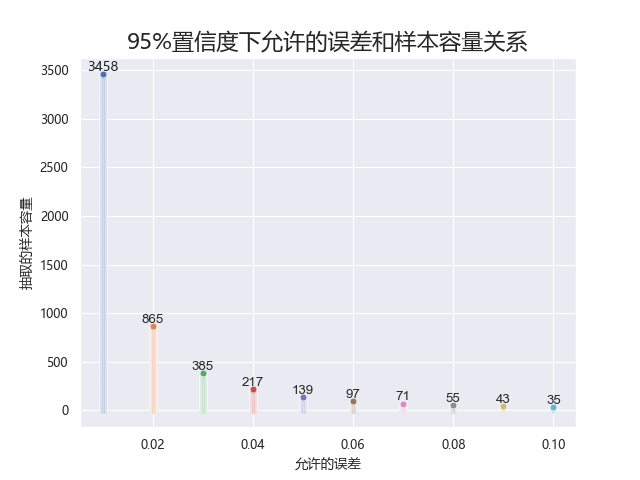


图1 95%置信度下允许的误差和样本容量关系

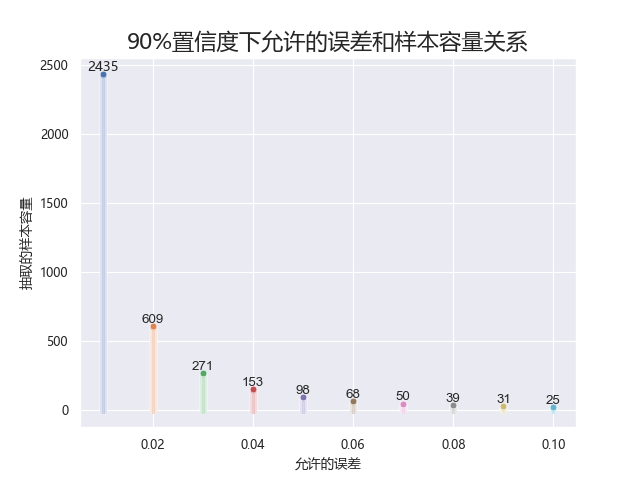


图2 90%置信度下允许的误差和样本容量关系

计算临界值选择统计检验的正态分布Z校验：

对于给定的显著性水平a，根据

若，则拒绝，接受；

若，则接受

求得95%置信度下临界值k为20，求得90%置信度下临界值k为14。

结论：

在95%的置信度下认为，抽样139个样品中，如果次品数超过20个，则拒绝这批零件。

在90%的置信度下认为，抽样98个样品中，如果次品数超过14个，则拒绝这批零件。

## 问题二：

由于题目中4个不同的成产阶段，各个阶段都需要进行决策评估来求得最优的生产方式，需要对各种可能出现的情况进行评估，所以使用工厂生产流程优化蒙特卡洛仿真模型来进行模拟不同的情况下的最优效益，最后得到的生产方式即为各种情况下的最有生产。蒙特卡洛仿真模拟可以利用随机数生成模拟现实中不去确定的因素，适合本题中 的多种情况下的多阶段生产模拟。

### 对零配件检测阶段：

**描述流程**

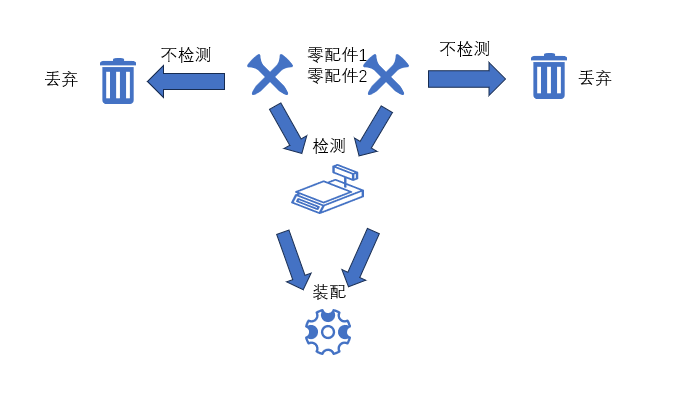


图3 零配件到装配的流程图

在零配件到装配的该阶段中，对零配件1和零配件2都要进行处理判断，零配件到生产线上有两种去向：

不检测：零配件直接进入装配环节，参与到成品的装配。不产生检测成本，但会对后续的成品装配造成不合格品

检测：零配件经过检测之后进入装配环节。会产生检测成本，但可以除去次品

**建立数学模型**

假设零配件1总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数

零配件2总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数 ，表示该阶段的花费。

该阶段的检测费用可以表示为：

进行该阶段后：

进行了检测，零配件1流入到装配环节的零配件数；未进行检测，

进行了检测，零配件2流入到装配环节的零配件数；未进行检测。

### 对零配件进入装配环节

**描述流程**

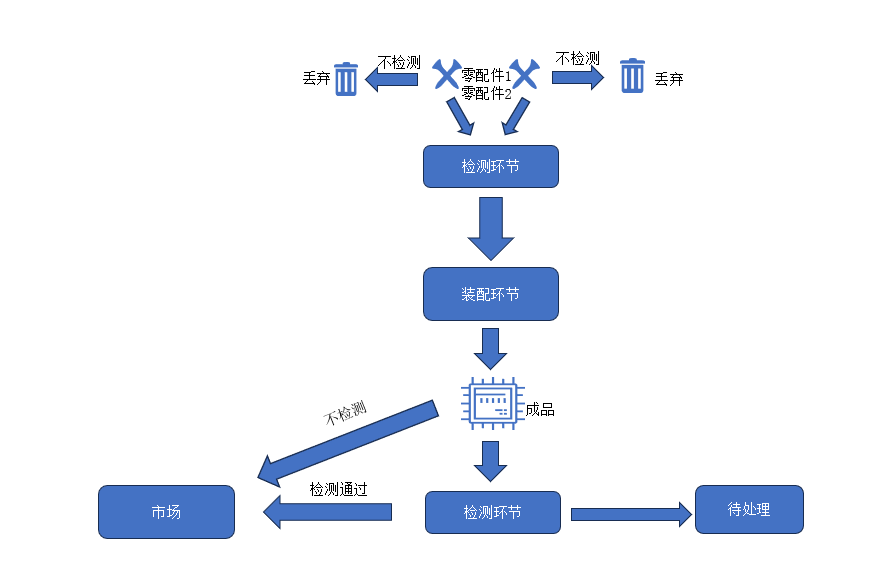
****

图4 零配件到装配环节流程图

零配件经过装配环节后，会产生合格品和不合格品，产品有两种去向：

不检测：产品直接进入市场。不会产生检测成本，但会有次品风险

检测：产品先进行检测，检测过后，只有合格品才能进入到市场，不合格品需要进行处理。检测会产生检测成本，但可以消除次品风险

**建立数学模型**

假设为零配件1和零配件2经过检测环节后，最终能够组装成成品的成品数量，则根据木桶效应[1]，可以表示为：

设 为成品数量花费的组装费用，为两个零配件装配成一个成品的装配成本，则可以表示为：

假设为两个零配件同时为正品的概率，是综合次品率，即从零配件检测到装配完成的次品概率，是装配产生的次品率，装配完成后不合格品的数量，则有

设检测成品的花费为，设检测成品的成本为，则有:

该阶段完成后，流入到拆解阶段的成品数量为

### 对装配出不合格成品的拆解阶段

**流程描述**

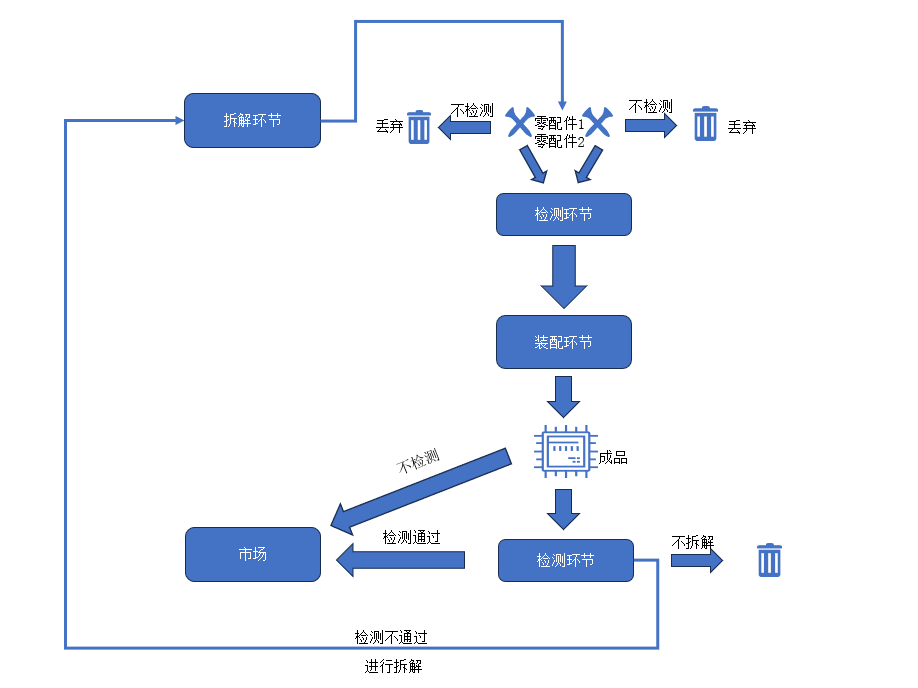


图5 不合格品拆解流程图

成品如果检测出不合格品，不合格品的去向有两种：

不拆解：直接丢弃。不会产生拆解成本。

拆解：拆解完后的产品会变为零配件1和零配件2。会产生拆解成本，但是零配件可以重新利用。

假设拆解成本为：，该阶段产生花费，因此有：

否则：

拆解完成后，一个成品会变为两个零配件，重新进行循环，回到零配件检测阶段。同时零配件1和零配件2的数量将发生变化。

### 对市场中的阶段

**流程描述**

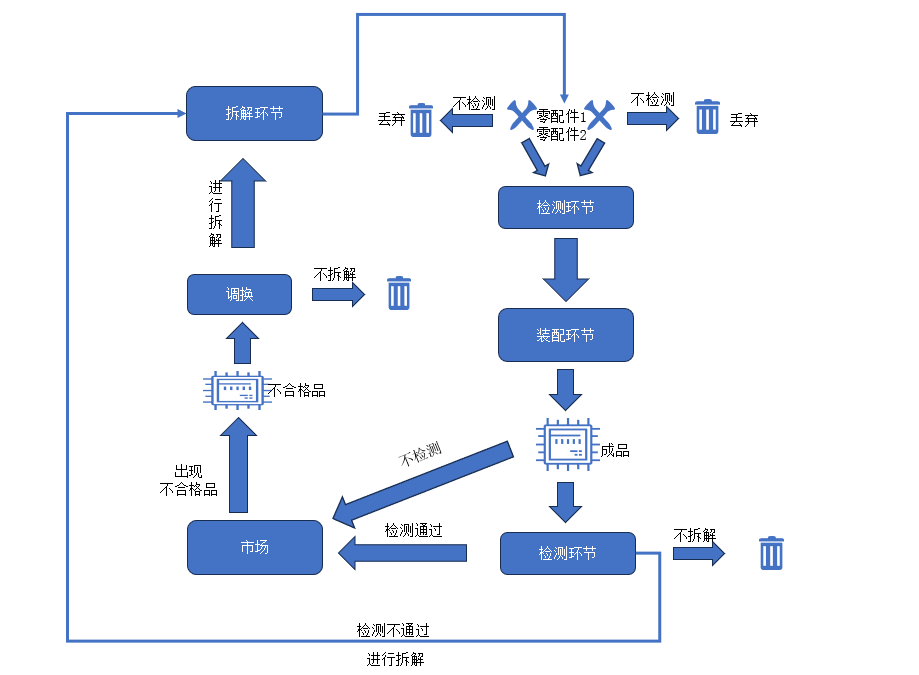


图6 市场调换和拆解流程图

成品流入市场后，用户可能会获得不合格的产品，此阶段可看作一个市场检测和反馈的环节，出现该情况，将会对企业造成物流成本、企业信誉等损失。

调换回来的不合格产品有两个去向：

拆解：拆解完后的产品会变为零配件1和零配件2。会产生拆解成本，但是零配件可以重新利用。

不拆解：直接丢弃。不会产生拆解成本。

该阶段后，如果选择拆解，那么零配件1和零配件2的剩余数量会发生改变，

如果拆卸， 的值更改为: ，的值更改为: 。

如果不拆卸，的值为旧值，的值为旧值。

**计算结果效益得出决策**

在成品流向市场的阶段，如果进行检测成品，则不会出现成品的调换，因此   
和只会出现一个。因此使用来代替两者中的一个。

假设最终获取的综合利润，每件成品单价为，则有：

通过不同的随机数生成不同的情况，使用以上蒙特卡洛仿真模型进行模拟生产，得到不同的生产方案，比较之间的综合利润，选出最效益最高的生产方式。

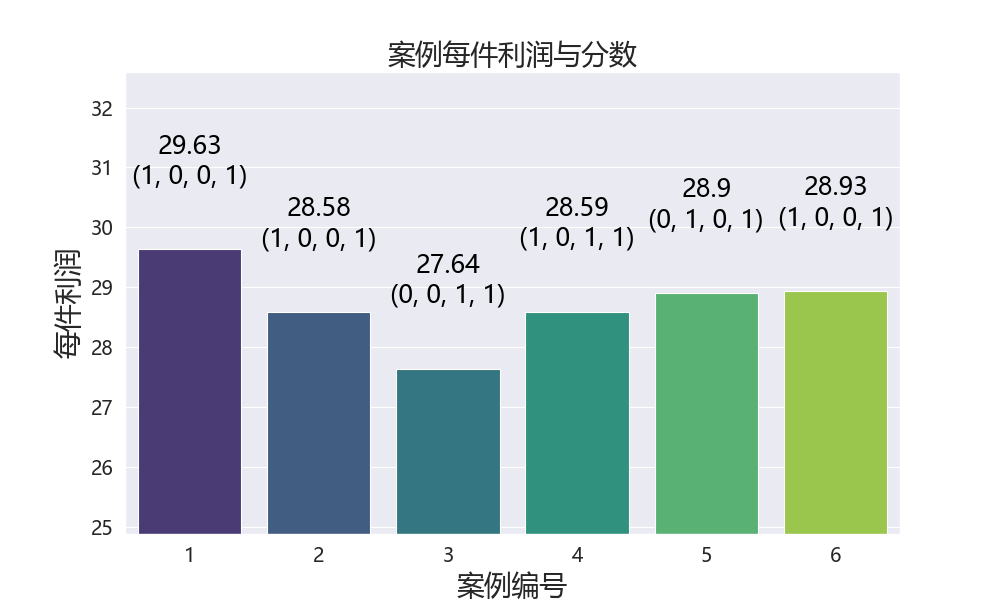
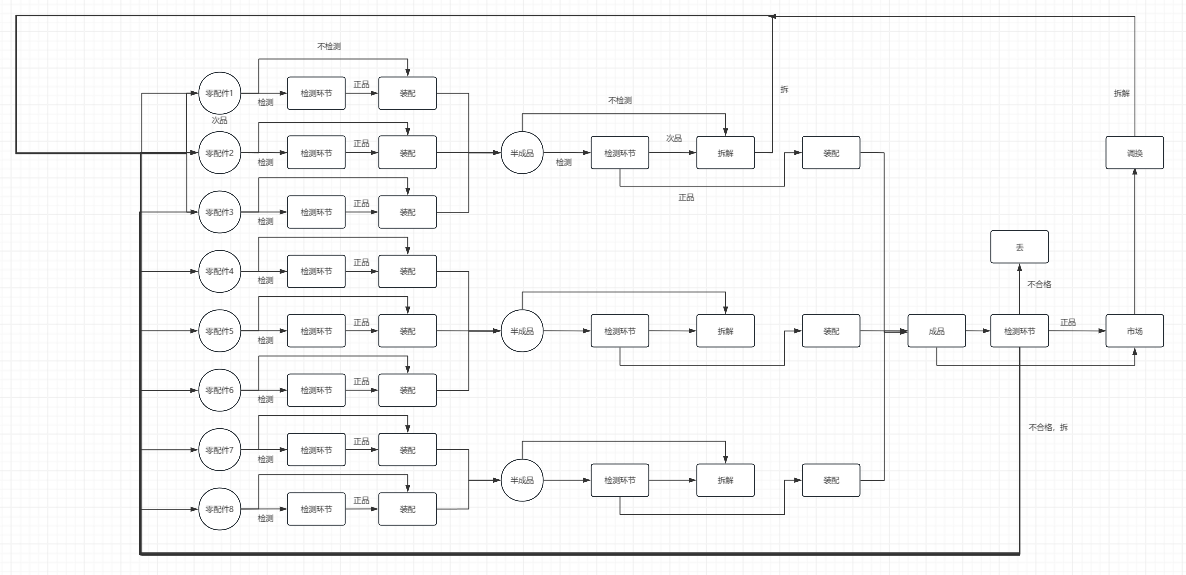


图7 6中情况下的最优解

## 问题三

本题有16个不同的阶段，有216种组合，属于求解复杂的多变量组合优化问题，属于NP难问题，使用传统的方法难以快速求解，使用基于遗传机制的遗传算法能够帮助我们解决复杂搜索问题。以下创建一个对应的工厂模型，来进行底层的分数运算，外层使用遗传算法来对算法进行优化，从而实现快速的求解不同情况下的最优解。



### 零配件检查阶段

**流程描述**

零配件检查阶段会对8个零配件进行检测，每个零配件都有两个去向：

不检测：零配件直接进入装配环节，参与到成品的装配。不产生检测成本，但会对后续的成品装配造成不合格品

检测：零配件经过检测之后进入装配环节。会产生检测成本，但可以除去次品

**建立数学模型**

假设零配件1总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数 ，表示该阶段的花费。

零配件2总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数。

零配件3总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数。

…...

零配件8总数为，次品率为，检测成本，检测费用，流入到下一个阶段的零配件数。

该阶段的检测费用可以表示为：

该阶段后,流入到下一个阶段的各种零配件数量为：

### 零配件装配成半成品阶段

**流程描述**

零配件经过装配成半成品环节后，会产生合格品和不合格品，产品有两种去向：

不检测：半成品直接进入成品装配环节。不会产生检测成本，但会有次品风险

检测：半成品先进行检测，检测过后，只有合格品才能进入到成品装配，不合格品需要进行处理。检测会产生检测成本，但可以消除次品风险

**建立数学模型**

假设为零配件1、零配件2和零配件3 经过检测环节后，最终能够组装成半成品的数量，则根据木桶效应[1]，可以表示为：

同理有：

设 为该阶段半成品花费的组装费用，为1，2，3零配件装配成一个半成品的装配成本，则设来表示第一个半成品的装配花费：

同理能够假设第二个半成品的装配花费和4，5，6零配件装配成第二个半成品的装配成本；第三个半成品的装配花费和7，8零配件装配成第三个半成品的装配成本。第二个半成品和第三个半成品的装配花费

假设该阶段中半成品的装配花费为：

该阶段完成后，流入到拆解阶段的成品数量为

### 半成品装配成品阶段

**流程描述:**

半成品装配完成后，有检测环节，需要选择是否要进行检测：

检测：产生检测成本费用，但可以消除次品风险

不检测：不产生检测成本，但需要承担次品风险

**计算装配完成为半成品后，半成品中的次品数量**

假设为1，2，3零配件同时为正品的概率，是综合次品率，即从零配件检测到装配完成的次品概率；是装配产生的次品率，装配完成后不合格品的数量，则有

同理得到第二和第三个半成品在的综合次品率

和不合格的半成品数量

**计算半成品检测费用**

设第一个半成品检测的花费为，设检测成品的成本为，则有:

同理得到第二和第三个半成品得检测费用和

则该阶段半成品得总检测费用:

**流程描述**

在检测完成后，有拆解环节，这个环节，不合格的半成品有两个去向：

不拆解：半成品直接丢弃。没有产生拆解费用，但零配件失去了使用价值。

拆解：不合格半成品进入拆解环节。产生拆解费用，但拆解后的零配件能够再次利用。

假设拆解第一种半成品成本为：，该阶段产生花费，因此有：

同理：第二种半成品成本为：，该阶段产生花费，因此有：

第三种半成品成本为：，该阶段产生花费，因此有：

得该阶段拆解费用:

拆解完成后，一个半成品会变为零配件，重新进行循环，回到零配件检测阶段。零配件数量发生变化。

**流程描述**

半成品进入到装配环节装配成正品

经过检测环节后，半成品的数量：

假设成品的数量为，根据水桶效应得

半成品装配成成品的装配成本，该阶段半成品装配成成品的费用，：

### 成品到市场的阶段

**流程描述**

成品到市场的阶段有一道检测环节，成功在这个环节有两种走向：

检测：成品只有合格才能进入市场。产生检测成本，但消除了次品风险

不检测：成品直接进入市场。不会产生检测成本，但有次品风险

**计算检测成本：**

假设检测成品的成本是，检测的费用是，则有：

**流程描述**

成品经过检测过后，会出现不合格的产品，不合格的产品的去向有两个:

拆解：拆解过后，零配件重新回到生产线中。产生拆解成本，但可以重新利用零配件

不拆解： 检测到不合格成品，直接丢弃。不会产生拆解成本，但会是使零配件失去使用价值

**计算成品的拆解费用**：

假设装配的次品率，成品的综合次品率，成品的不合格数量，拆解一个不合格品的费用，该阶段拆解成品的费用为则有，

**流程描述**

如果不进行检测，就会直接流入市场，市场可以看作一个企业市场检测和反馈的环节，出现该情况，将会对企业造成物流成本、企业信誉等损失。

调换回来的不合格产品有两个去向：

拆解：拆解完后的产品会变为零配件1和零配件2。会产生拆解成本，但是零配件可以重新利用。

不拆解：直接丢弃。不会产生拆解成本。

假设调换成本为，该阶段产生的花费，则有：

### 计算结果得出决策

在成品流向市场的过程中，当进行成品检测，市场中将不会出现不合格品。相反，如果不进行成品检测，市场中将可能会出现不合格品，就需要企业进行对用户的产品调换。所以调换花费和成品检测过程中的拆解花费两者只会有一者出现。使来代替两者中的一个。

假设最终获取的综合利润，每件成品单价为，则有：

以上，工厂的模型构建完毕，接下来，将使用遗传算法来进行模型的优化。

### 模型优化:

由于次品率、装配、检测和拆解等成本的多样性和复杂性，工厂生产过程的优化决策问题是一个复杂的组合问题，使用基于遗传机制的遗传算法[2]可以有效的应对这种问题。以下是遗传算法的具体实现步骤：

1. 初始化种群编码

个体长度设置：每个个体的长度为16位，其中包含了零配件、半成品和成品的检测和拆解等决策。

种群大小：设置较合适的种群大小为100

使用二进制编码来表示决策变量，编码方式如下：

* 零配件的检测用1进行表示，不检测用0进行表示。
* 半成品的检测用1进行表示，不检测用0进行表示。
* 成品的检测用1进行表示，不检测用0进行表示。
* 半成品和成品的拆解使用1进行标识，不拆解用0进行表示。

设置序列详细：  
0 - 7 ： 1-8号零件是否检测  
8 - 10： 1-3号半成品是否检测  
11：成品是否检测  
12 - 14： 1-3号是否拆解  
15：成品是否拆解

1. 适应度函数定义

适应度的函数使用的的是已经建立完成的模拟工厂函数，使用分数和利润来对每个基因个体进行评分，从而挑选出最好的个体。

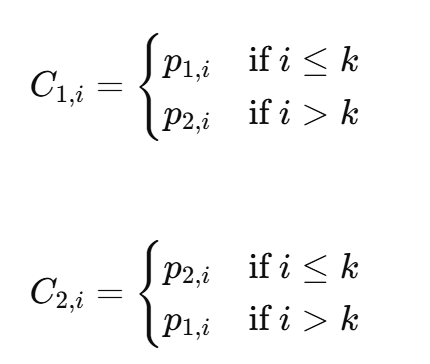
这里使用该条公式综合考虑各环节的成本，评估每一个个体的效益，计算出工厂中最后模拟出的总利润或分数。

1. 选择

依据锦标赛选择方法来使适应度函数选出相对优的个体，随机选取出若干个个体进行”对决”，拿出最具有优势的个体作为父系进行繁殖，使得优势基因在群众的比例有效保留。

1. 交叉

选择两条父母代的个体进行单点交叉操作，交换部分基因，产生多样性，得出具有新特性的基因，促进种群的进化。



1. 变异

通过基因的翻转这种变异方式，使得0和1进行互相转换。变异的操作有利于增加基因的多样性，防止最后得到的结果仅仅是局部最优解。

1. 适应度评估于迭代

计算每个个体的适应度函数来选择更加具有目标优势的个体进入到进入到下一代中，通过选择、交叉和变异的操作来使种群逐步淘汰掉低优势的个体，保留优势基因，逐渐收敛至最优解。

1. 终止

设置迭代停止的数值，防止无意义的无限迭代下去，当种群进行了多次的选择、变异等操作过后，开始逐渐收敛，就应当进行停止，节约时间等资源。

这里通过多次的测试，设置20次的迭代次数就可以达到较好的收敛。

1. 遗传算法的最终目标

通过遗传算法的多次迭代和变异，不断地将决策变量的取值进行优化，使得工厂生产中的成本降到最低，利润提到最高，为企业提供稳定、有效的生产方案。

### 结果展示

在理想情况下，最优决策为：

零件1 – 8号 都检测

半成品 1 - 3号 都不检测

成品需要检测

半成品1 – 3 号 不需要拆解

成品需要拆解

综合每件利润： 58.73元

产品综合次品率： 34.39 %

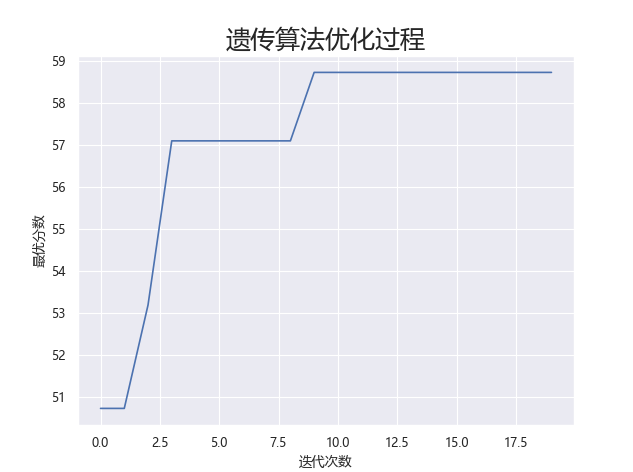


图8 遗传算法的优化

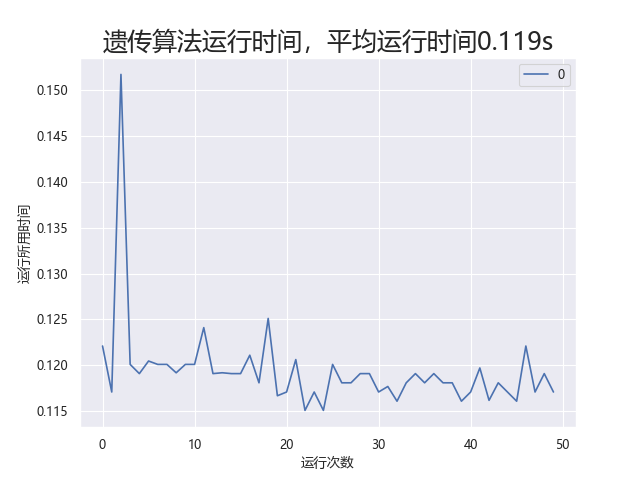


图9 遗传算法的平均运行时间

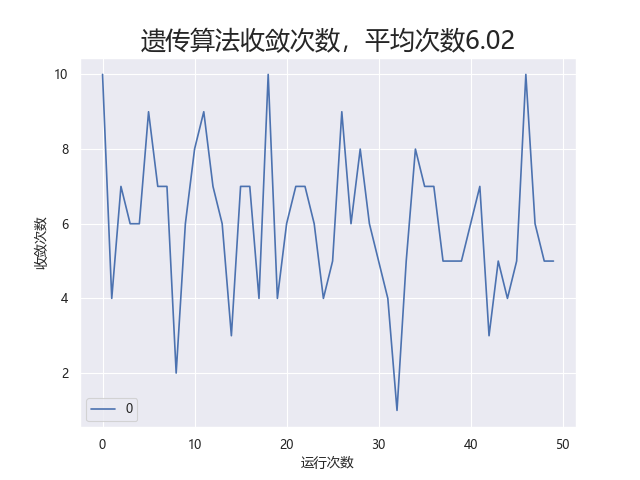


图9 遗传算法的平均收敛次数

遗传算法与暴力求解对比：

暴力求解：

暴力枚举所有情况运行时间运行时间： 1.3558 s

最优每件利润: 58.73

最优设置: (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1)

产品综合次品率： 34.39 %

遗传算法：

遗传算法运行时间： 0.0968 s

收敛次数： 5

最优设置: [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1]

最优每件利润: 58.73

总体提升效率： 14 倍

## 问题四

### 定义符号：

∆:可接受的误差范围，也就是效应量，期望次品率和实际次品率的差值,即置信区间半宽度

p：样本次品率

n：样本量

z\_{α/2}：与置信水平对应的标准正态分布临界值

置信区间公式：

代入已知条件：

假设n样本量为100，根据第一问我们可以假设可知∆ = 0.05，因此p的上限为p + 0.05，p的下限区间p - 0.05，即波动范围（p +0.05，p-0.05），最后我们假设在波动范围中随机到的概率设为p̂作为其样本次品率，进行求解出最优决策，考略到样本次品率p̂ 具有随机性和数据的庞大化因此，将对问题二进行200次循环，对问题三进行50次循环，取各个不同阶段决策后的的最优利润，并判断最优设置分布的各个次数

### 对于问题二的重新求解后所得的结果

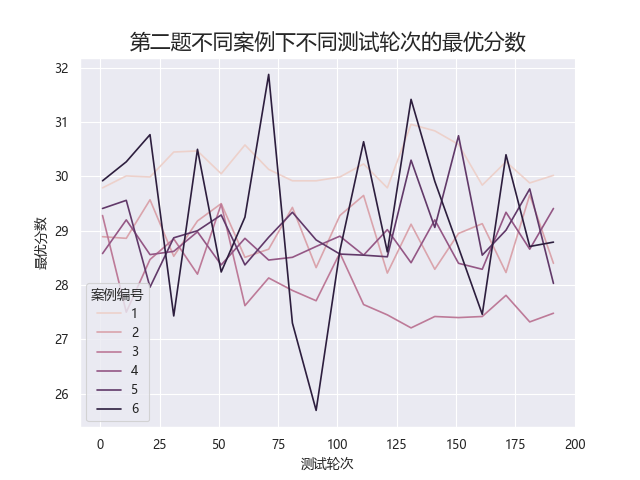


图10 题二不同案例不同测试轮次的最优分数

解释：通过改变不同的次品率，对不同案例，不同的测试轮次，所得到的最优分数即最优利润有所不同，得到最后结果的可能值

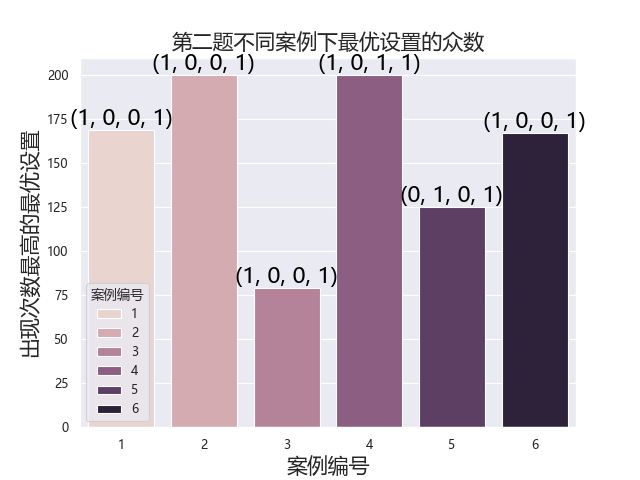


图11 不同案例下最优设置的众数

解释：通过改变不同的次品率，对于不同案例同样进行200次循环后，通过每次循环得到的一次最优解进行统计并取众数，从而从中判断出当次品率改变时，当前案例综合采取的策略。

**总结**：虽然最优利润变化值波动性强，但是对于不同案例，选择的最优策略还是不变。由此可得

案例一：对零件一进行检测，零件二不进行检测，成品不进行拆解，对合格品进行检验

案例二：对零件一进行检测，零件二不进行检测，成品不进行拆解，对合格品进行检验

案例三：对零件一进行检测，零件二不进行检测，成品不进行拆解，对合格品进行检验

案例四：对零件一进行检测，零件二不进行检测，成品进行拆解，对合格品进行检验

案例五：对零件一不进行检测，零件二进行检测，成品不进行拆解，对合格品进行检验

案例六：对零件一进行检测，零件二不进行检测，成品不进行拆解，对合格品进行检验。

### 对于问题三的重新求解后所得的结果

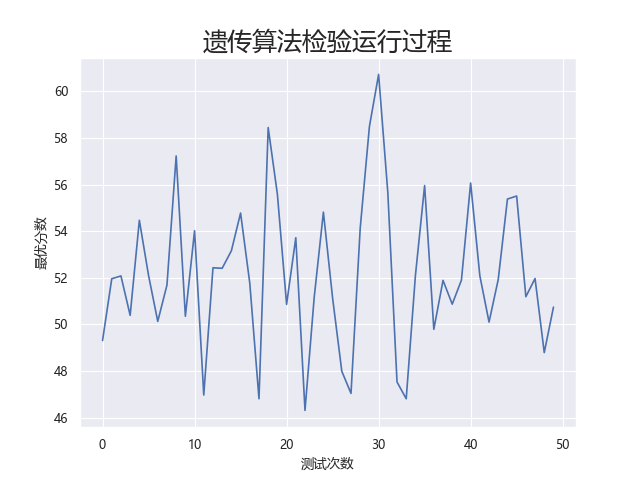


图12 遗传算法检验运行过程

表1 遗传算法出现次数较多的最优设置

|  |  |
| --- | --- |
| **最优设置** | **轮次** |
| (1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1) | 2 |
| (1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1) | 2 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1) | 6 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1) | 4 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1) | 3 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1) | 9 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1) | 4 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1) | 5 |
| (1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1) | 3 |
| **总计** | **50** |

同第二问处理方法一样，考虑到数据的复杂性，采用遗传算法模型，提高搜索效率，更加真实模拟真实环境。据上述图三和图四，采取最优设置的众数作为最优策略。

### 因此最优策略：

对零件一进行检测，零件二进行检测，零件三进行检测，零件四进行检测，零件五进行检测，零件六进行检测，零件七进行检测，零件八进行检测，对半成品一不进行检测，对半成品二不进行检测，对半成品三不进行检测，对半成品一进行拆解，对半成品二不进行拆解，对半成品三进行拆解，对成品进行检测，对成品进行拆解

# 模型的评价、改进与推广

## 模型的优点

1. 可以根据需要调整次品率误差值，灵活适应不同的分析需求。
2. 使用遗传算法来解决工厂配置的组合优化问题，有效解决传统办法解决组合优化问题时的效率问题
3. 通过多次实验测试评估遗传算法的稳定性，运行时间短，收敛快。
4. 通过代码实现全面模拟工厂装配成品，可以检验决策所带来的总体利润。
5. 基于误差值分析不同次品率所带来的利润上影响，帮助决策者做出决策。

## 模型的缺点

1. 所有的算法参数（如变异率、交叉率）都是固定的，可能需要调整以优化性能。
2. 次品率对于最优设置产生的幅度比较大。

## 模型的改进

1. 结合实际数据进行验证，调整模型以更好地适应实际情况。
2. 设置设置不同参数，来更好的检验模型。

## 模型的推广

1. 将此优化模型应用于生产调度、资源分配等，并进行实际验证。
2. 开发一个通用的遗传算法工具，能够处理各种组合优化问题，并提供详细的可视化和参数调优功能。

# 参考文献

1. 石磊.木桶效应:一个人、一个组织都存在一种致命的效应,那就是木桶效应[J].地震出版社, 2004.
2. 周明,孙树栋.遗传算法原理及应用[M].国防工业出版社,1999.

# 附录

|  |
| --- |
| **附录1** |
| 介绍：该代码用于选取90%置信度下的样本容量 |
| import math import numpy as np import matplotlib.pyplot as plt import seaborn as sns from scipy.stats import norm  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False   # 样本容量确定（二项分布） def get\_sample\_size(a, p, d):  # a是置信度  # d是允许的误差  # p是次品概率  z = norm.ppf((1 - a) / 2) # z是正态分布的a/2 的分位数  return math.ceil(z \* z \* p \* (1 - p) / (d \* d))   # 允许误差和样本容量关系 for i in range(1, 11):  dd = i \* 0.01 # 允许的误差  sample\_size = get\_sample\_size(0.90, 0.1, dd)  print(dd, sample\_size)  # 使用seaborn的散点图绘制  # 显示每个点的数值  plt.text(dd, sample\_size, str(sample\_size), ha='center', va='bottom')  plt.title('90%置信度下允许的误差和样本容量关系', fontsize=16)  plt.xlabel('允许的误差')  plt.ylabel('抽取的样本容量')  sns.set\_style("whitegrid")  sns.scatterplot(x=[dd] \* sample\_size, y=np.arange(sample\_size))  plt.savefig('90%置信度下允许的误差和样本容量关系.png') plt.show() |

|  |
| --- |
| **附录2** |
| 介绍：该代码用于选取90%置信度下的样本容量 |
| import math import numpy as np import matplotlib.pyplot as plt import seaborn as sns from scipy.stats import norm  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False   # 样本容量确定（二项分布） def get\_sample\_size(a, p, d):  # a是置信度  # d是允许的误差  # p是次品概率  z = norm.ppf((1 - a) / 2) # z是正态分布的a/2 的分位数  return math.ceil(z \* z \* p \* (1 - p) / (d \* d))   # 允许误差和样本容量关系 for i in range(1, 11):  dd = i \* 0.01 # 允许的误差  sample\_size = get\_sample\_size(0.95, 0.1, dd)  print(dd, sample\_size)  # 使用seaborn的散点图绘制  # 显示每个点的数值  plt.text(dd, sample\_size, str(sample\_size), ha='center', va='bottom')  plt.title('95%置信度下允许的误差和样本容量关系', fontsize=16)  plt.xlabel('允许的误差')  plt.ylabel('抽取的样本容量')  sns.set\_style("whitegrid")  sns.scatterplot(x=[dd] \* sample\_size, y=np.arange(sample\_size))  plt.savefig('95%置信度下允许的误差和样本容量关系.png') plt.show() |

|  |
| --- |
| **附录3** |
| 介绍：该代码用于选取两种置信度下的抽样方案 |
| import math from scipy.stats import norm  # 样本容量 n95 = 139 n90 = 98  # 次品率 p = 0.1  # 求出临界值 # 95% 置信度 k95 = n95 \* p + norm.ppf(0.95) \* math.sqrt(n95 \* p \* (1 - p)) k95 = math.ceil(k95) # 向上取整 print('95%置信度下临界值: ', k95)  k90 = n90 \* p + norm.ppf(0.90) \* math.sqrt(n90 \* p \* (1 - p)) k90 = math.ceil(k90) print('90%置信度下临界值: ', k90) |

|  |
| --- |
| **附录4** |
| 介绍： 该代码用于生产模拟，计算最大利润 |
| # 这个文件是工厂类，负责模拟工厂生产产品，并计算相对应的利润（score） # 设定工厂的流水线： # 流水线有1，2两种零件，将两种零件加工成成品 # 现在设置检测环节，可以选择是否检测1，2，成品，以及失败成品是否拆解 # 对所有情况进行评分处理。  class Factory:  def \_\_init\_\_(self):  self.part\_1 = 0 # 第一种零件总数  self.part\_1\_defect = 0 # 第一种零件不合格率  self.part\_1\_price = 0 # 第一种零件价格  self.part\_1\_check = 0 # 第一种零件检测费用   self.part\_2 = 0 # 第二种零件总数  self.part\_2\_defect = 0 # 第二种零件不合格率  self.part\_2\_price = 0 # 第二种零件价格  self.part\_2\_check = 0 # 第二种零件检测费用   self.product = 0 # 成品总数  self.product\_defect = 0 # 成品实际不合格率  self.product\_assemble\_defect\_rate = 0 # 成品组装不合格率  self.product\_price = 0 # 成品组装成本  self.product\_check = 0 # 成品检测费用  self.product\_sell = 0 # 成品销售价  self.product\_return = 0 # 调换损失  self.product\_dismantle = 0 # 拆解成本   self.settings = [] # 设置(1,2,成品，是否检测，失败是否拆解)  self.score = 0 # 当前设置下的成本   # 设置各零件的不合格率  def set\_defect\_rate(self, part\_1\_defect, part\_2\_defect, product\_defect) -> None:  self.part\_1\_defect = part\_1\_defect  self.part\_2\_defect = part\_2\_defect  self.product\_assemble\_defect\_rate = product\_defect   # 设置各零件成本  def set\_price(self, part\_1\_price, part\_2\_price, product\_price, product\_sell) -> None:  self.part\_1\_price = part\_1\_price  self.part\_2\_price = part\_2\_price  self.product\_price = product\_price  self.product\_sell = product\_sell   # 设置检测费用  def set\_check(self, part\_1\_check, part\_2\_check, product\_check) -> None:  self.part\_1\_check = part\_1\_check  self.part\_2\_check = part\_2\_check  self.product\_check = product\_check   # 设置返工成本  def set\_return(self, product\_return, product\_dismantle) -> None:  self.product\_return = product\_return  self.product\_dismantle = product\_dismantle   # 设置检测关卡以及是否拆解  def set\_settings(self, settings: list) -> None:  self.settings = settings  self.settings = list(settings)   def get\_score(self) -> float:  # 初始化  self.score = 0 # 分数  self.part\_1 = 100 # 第一种零件总数  self.part\_2 = 100 # 第二种零件总数  self.product = 0   cost\_1 = 0 # 检测零件1成本  cost\_2 = 0 # 检测零件2成本  cost\_product = 0 # 组装成本  defect\_cost = 0 # 不合格品所产生的成本  total\_product = 0 # 成品总数   # 购买零件成本  self.score -= self.part\_1 \* self.part\_1\_price  self.score -= self.part\_2 \* self.part\_2\_price   # 根据设置来模拟工厂运行  while self.part\_1 >= 1 and self.part\_2 >= 1: # 终止条件，只要有一个零部件用完了就结束  # print("part\_1:", self.part\_1, "part\_2:", self.part\_2)  # 是否检测零件一二次品  if self.settings[0] == 1:  cost\_1 += self.part\_1 \* self.part\_1\_check \* self.settings[0] # 检测零件1成本  self.part\_1 -= self.part\_1 \* self.part\_1\_defect # 去除不合格品  self.part\_1\_defect = 0 # 此时次品率为0  self.settings[0] = 0 # 检测后不再检测  if self.settings[1] == 1:  cost\_2 += self.part\_2 \* self.part\_2\_check \* self.settings[1] # 检测零件2成本  self.part\_2 -= self.part\_2 \* self.part\_2\_defect # 去除不合格品  self.part\_2\_defect = 0 # 此时次品率为0  self.settings[1] = 0 # 检测后不再检测   # 组装：  self.product = min(self.part\_1, self.part\_2)  self.part\_1 -= self.product  self.part\_2 -= self.product  cost\_product = self.product \* self.product\_price # 组装成本   # 两个零件同时为正品的概率：  correct\_rate = (1 - self.part\_1\_defect) \* (1 - self.part\_2\_defect)   # 综合次品率  defect\_rate = 1 - correct\_rate \* (1 - self.product\_assemble\_defect\_rate)  # print("综合次品率： ", defect\_rate)   # 不合格品的个数：  defect\_product = self.product \* defect\_rate   # 是否检测成品  if self.settings[2] == 1:  # 对所有产品进行检测  defect\_cost += self.product\_check \* self.product # 检测全部产品所产生的费用  self.product -= defect\_product # 检测后正常的产品  else:  # 不合格品的调换成本：  defect\_cost += defect\_product \* self.product\_return # 调换成本  self.product -= defect\_product # 调换后还需补上正常的产品   # 不合格品拆解成本  if self.settings[3] == 1:  defect\_cost += defect\_product \* self.product\_dismantle # 拆解成本  self.part\_1 += defect\_product # 拆解后需要重新生产  self.part\_2 += defect\_product  # 不论如何，不合格品数目最后会变为0   total\_product += self.product # 统计总产品数  self.product = 0   # 综合利润：  self.score += total\_product \* self.product\_sell - cost\_1 - cost\_2 - cost\_product - defect\_cost   return self.score |

|  |
| --- |
| **附录5** |
| 介绍：该代码用于生成每个情况最优的决策 |
| import itertools from matplotlib import pyplot as plt import pandas as pd from factory\_class import Factory from case\_file import cases import seaborn as sns  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False plt.rcParams['font.size'] = 18 # 设置全局字体大小   # 生成所有组合的函数 def generate\_combinations(n):  return list(itertools.product([0, 1], repeat=n))   if \_\_name\_\_ == '\_\_main\_\_':  i = 1 # 遍历案例的编号  best\_score\_logs = [] # 最优的分数记录  best\_settings\_logs = [] # 最优的设置记录   log = open("log.csv", "w")  log.write("//best\_setting含义：第一个是否检测，第二个是否检测，产品是否检测，产品是否拆解\n")  # 遍历6个案例  for case in cases:  print("案例: ", i)  factory = Factory() # 实例化流水线  # 初始化设置  factory.set\_defect\_rate(  case["part\_1\_defect"],  case["part\_2\_defect"],  case["product\_defect"]  )  factory.set\_price(  case["part\_1\_price"],  case["part\_2\_price"],  case["product\_price"],  case["product\_sell"]  )  factory.set\_check(  case["part\_1\_check"],  case["part\_2\_check"],  case["product\_check"]  )  factory.set\_return(  case["product\_return"],  case["product\_dismantle"]  )  # 生成所有可能的组合  settings = generate\_combinations(4)  best\_setting = [] # 最优的设置  max\_score = 0 # 最优的分数  for setting in settings:  factory.set\_settings(setting)  score = factory.get\_score()  if score > max\_score:  max\_score = score  best\_setting = setting  # print(setting, score)  print("max\_score:", round(max\_score, 2))  print("best\_setting:", best\_setting)   best\_score\_logs.append(round(max\_score, 2))  best\_settings\_logs.append(str(best\_setting))   log.write("case: " + str(i) + "\t")  log.write("max\_score: " + str(round(max\_score, 2)) + "\t")  log.write("best\_setting: " + str(best\_setting) + "\n")  i += 1   # 将分数除以100,得到平均利润  for i in range(len(best\_score\_logs)):  best\_score\_logs[i] /= 100  best\_score\_logs[i] = round(best\_score\_logs[i], 2)   # 使用sns绘制条形图  df = {  "案例编号": ['1', '2', '3', '4', '5', '6'],  "最佳分数": best\_score\_logs,  "最佳设置": best\_settings\_logs  }   df = pd.DataFrame(df)  print(df)  df.to\_csv("best\_setting.csv")   # 绘制条形图  plt.figure(figsize=(10, 6))  bar\_plot = sns.barplot(x="案例编号", y="最佳分数", data=df, palette="viridis")  plt.xticks(fontsize=14) # 设置 x 轴刻度标签的字体大小  plt.yticks(fontsize=14) # 设置 y 轴刻度标签的字体大小   # 添加最佳分数数据和设置标签  for index, row in df.iterrows():  bar\_plot.text(index, row["最佳分数"] + 1, f"{row['最佳分数']}\n{row['最佳设置']}",  color='black', ha="center", va="bottom")   # 这里更新了y轴的显示范围  y\_start = min(best\_score\_logs) \* 0.9  y\_end = max(best\_score\_logs) \* 1.1  plt.ylim(y\_start, y\_end)   plt.title("案例每件利润与分数", fontsize=20)  plt.xlabel("案例编号", fontsize=20)  plt.ylabel("每件利润", fontsize=20)  plt.savefig("best\_setting.png") # 保存结果图片  plt.show() |

|  |
| --- |
| **附录6** |
| 介绍：该代码用于模拟复杂生产环节，并计算当前决策的利润 |
| # *todo 设置详细：* # *todo 0 - 7 ： 1-8号零件是否检测* # *todo 8 - 10： 1-3号半成品是否检测* # *todo 11：成品是否检测* # *todo 12 - 14： 1-3号是否拆解* # *todo 15：成品是否拆解* CHECK\_PRODUCT = 11 # 成品是否检测 DISMANTLE\_PRODUCT = 15 # 成品是否拆解 CHECK\_SETTINGS = 8 # 偏移量来确定是否需要检测半成品，如序号0 的半成品想知道他是否要检测，则选择查看settings[8] DISMANTLE\_SETTINGS = 12 # 偏移量来确定是否需要拆解半成品 # ASSEMBLE\_FAILED\_RATE = 0.1 # 组装的失败率，都是0.1   # 零件 class Part:  def \_\_init\_\_(self, price, num, check\_price, defect\_rate):  self.price = price # 价格  self.num = num # 数量  self.check\_price = check\_price # 检测成本  self.defect\_rate = defect\_rate # 不合格率   def get\_num(self):  return self.num   # 产生次品  def clean\_defect(self):  self.num -= self.num \* self.defect\_rate   def set\_defect\_rate(self, defect):  self.defect\_rate = defect   def \_\_str\_\_(self):  return f"零件, 数目：{self.num}，价格：{self.price}，检测成本：{self.check\_price}，不合格率：{self.defect\_rate}"   # 半成品 class Semi\_product:  def \_\_init\_\_(self, num, price, check\_price, defect\_rate, product\_dismantle):  self.num = num # 数量  self.price = price # 组装成本  self.check\_price = check\_price # 检测成本  self.num\_defect = 0 # 不合格品数量  self.product\_dismantle = product\_dismantle # 拆解成本  self.defect\_rate = 0 # 实际的不合格率  self.assemble\_defect\_rate = defect\_rate # 组装时的不合格率   def clean\_defect(self):  self.num\_defect += self.num \* self.defect\_rate  self.num -= self.num\_defect   def set\_defect\_rate(self, defect):  self.defect\_rate = defect   def \_\_str\_\_(self):  return f"半成品, 数目：{self.num}，组装成本：{self.price}，检测成本：{self.check\_price}，不合格率：{self.defect\_rate}"   # 成品 class Product:  def \_\_init\_\_(self, num, price, check\_price, defect\_rate, sell\_price, product\_return, product\_dismantle):  self.num = num # 数量  self.price = price # 组装成本  self.check\_price = check\_price # 检测成本  self.sell\_price = sell\_price # 销售价  self.product\_return = product\_return # 调换损失  self.product\_dismantle = product\_dismantle # 拆解成本  self.num\_defect = 0 # 不合格品数量  self.defect\_rate = 0 # 实际的不合格率  self.assemble\_defect\_rate = defect\_rate # 组装时的不合格率   def \_\_str\_\_(self):  return f"成品, 数目：{self.num}，组装成本：{self.price}，检测成本：{self.check\_price}，不合格率：{self.defect\_rate}"   def set\_defect\_rate(self, defect):  self.defect\_rate = defect   def clean\_defect(self):  self.num\_defect += self.num \* self.defect\_rate  self.num -= self.num\_defect   # 流水线 class Factory:  def \_\_init\_\_(self):  self.parts = []  self.semi\_products = []  self.product = Product(0, 0, 0, 0, 0, 0, 0)   self.settings = [] # 设置(1-8号零件是否检测，1-3号半成品是否检测，成品是否检测，1-3号是否拆解，成品是否拆解）  # *todo 详细：* # *todo 0 - 7 ： 1-8号零件是否检测* # *todo 8 - 10： 1-3号半成品是否检测* # *todo 11：成品是否检测* # *todo 12 - 14： 1-3号是否拆解* # *todo 15：成品是否拆解* self.score = 0 # 当前设置下的成本   def set\_all\_part\_num(self, num):  for part in self.parts:  part.num = num   def set\_all\_semi\_num(self, num):  for semi in self.semi\_products:  semi.num = num   def set\_product\_num(self, num):  self.product.num = num   # 导入零件信息  def set\_part(self, parts: list[dict]) -> None:  for part in parts:  self.parts.append(Part(  part["price"],  part["num"],  part["check\_price"],  part["defect\_rate"]  ))   def set\_semi\_product(self, semi\_products: list[dict]) -> None:  for semi in semi\_products:  self.semi\_products.append(Semi\_product(  semi["num"],  semi["price"],  semi["check\_price"],  semi["defect\_rate"],  semi["product\_dismantle"]  ))   def set\_product(self, product: dict) -> None:  self.product = Product(  product["num"],  product["price"],  product["check\_price"],  product["defect\_rate"],  product["sell\_price"],  product["product\_return"],  product["product\_dismantle"]  )   # 设置检测关卡以及是否拆解  def set\_settings(self, settings: list) -> None:  self.settings = settings  self.settings = list(settings)   # 设置运行状态  def part\_running(self) -> bool:  for part in self.parts:  if part.num < 1:  return False  return True   def semi\_running(self) -> bool:  for semi in self.semi\_products:  if semi.num < 1:  return False  return True   def get\_score(self) -> dict:   total\_product\_num = 0  # 初始化  self.score = 0  self.set\_all\_part\_num(100) # 初始化所有零件数量  self.set\_all\_semi\_num(0)  self.set\_product\_num(0)   # 成本  parts\_cost = 0 # 零件成本  semi\_product\_cost = 0 # 半成品成本  product\_cost = 0 # 成品成本  product\_defect\_rate = 0 # 成品不合格率   for part in self.parts:  parts\_cost += part.price \* part.num # 购买成本   # 根据设置来模拟工厂运行  # 生产半成品  while self.part\_running():  # 是否检测零件次品  for i in range(0, 8):  if self.settings[i] == 1:  # 检测零件  parts\_cost += self.parts[i].check\_price \* self.parts[i].num  self.parts[i].clean\_defect() # 清除次品  self.parts[i].set\_defect\_rate(0) # 设置零件次品率为0，没有次品了  self.settings[i] = 0 # 之后便不需要检测了，因为次品已经被检测清除了   # 组装： 1-3号组成一个半成品， 4-6号组成半成品， 7-8号组成一个半成品  # 1-3号组成一个半成品  self.semi\_products[0].num += min( # 半成品数量由最少的一个零件决定  self.parts[0].num,  self.parts[1].num,  self.parts[2].num  )  # 组装成本  semi\_product\_cost += self.semi\_products[0].price \* self.semi\_products[0].num   part\_correct\_rate = 1 # 全部零件正确率  for i in range(0, 3):  self.parts[i].num -= self.semi\_products[0].num # 消耗零件  part\_correct\_rate \*= (1 - self.parts[i].defect\_rate) # 更新全部零件正确率   # 更新1号半成品的实际次品率  defect\_rate1 = 1 - part\_correct\_rate \* (1 - self.semi\_products[0].assemble\_defect\_rate) # 1号半成品的次品率  self.semi\_products[0].set\_defect\_rate(defect\_rate1)   # 4-6号组成半成品  self.semi\_products[1].num += min(  self.parts[3].num,  self.parts[4].num,  self.parts[5].num  )  # 组装成本  semi\_product\_cost += self.semi\_products[1].price \* self.semi\_products[1].num   part\_correct\_rate = 1 # 零件正确率  for i in range(3, 6):  self.parts[i].num -= self.semi\_products[1].num  part\_correct\_rate \*= (1 - self.parts[i].defect\_rate)   # 更新2号半成品的实际次品率  defect\_rate2 = 1 - part\_correct\_rate \* (1 - self.semi\_products[1].assemble\_defect\_rate)  self.semi\_products[1].set\_defect\_rate(defect\_rate2)   # 7-8号组成一个半成品  self.semi\_products[2].num += min(  self.parts[6].num,  self.parts[7].num  )  # 组装成本  semi\_product\_cost += self.semi\_products[2].price \* self.semi\_products[2].num   part\_correct\_rate = 1 # 零件正确率  for i in range(6, 8):  self.parts[i].num -= self.semi\_products[2].num  part\_correct\_rate \*= (1 - self.parts[i].defect\_rate)   # 更新3号半成品的实际次品率  defect\_rate3 = 1 - part\_correct\_rate \* (1 - self.semi\_products[2].assemble\_defect\_rate)  self.semi\_products[2].set\_defect\_rate(defect\_rate3)   # 生产成品  while self.semi\_running():  # 是否检测半成品次品  for i in range(0, 3):  if self.settings[i + CHECK\_SETTINGS] == 1:  # 检测成本  semi\_product\_cost += self.semi\_products[i].check\_price \* self.semi\_products[i].num  self.semi\_products[i].clean\_defect() # 区分次品  self.semi\_products[i].set\_defect\_rate(0) # 当前的半成品次品率为0  # 半成品和零件不一样，之后仍然需要检测  # 还需判断是否要拆解：  if self.settings[i + DISMANTLE\_SETTINGS] == 1:  # 拆解费用  semi\_product\_cost += (  self.semi\_products[i].product\_dismantle  \* self.semi\_products[i].num\_defect  )  # 根据半成品号数来拆解  if i == 0:  self.parts[0].num += self.semi\_products[i].num\_defect  self.parts[1].num += self.semi\_products[i].num\_defect  self.parts[2].num += self.semi\_products[i].num\_defect  elif i == 1:  self.parts[3].num += self.semi\_products[i].num\_defect  self.parts[4].num += self.semi\_products[i].num\_defect  self.parts[5].num += self.semi\_products[i].num\_defect  elif i == 2:  self.parts[6].num += self.semi\_products[i].num\_defect  self.parts[7].num += self.semi\_products[i].num\_defect  # 不论是否拆解，残次半成品数目清零  self.semi\_products[i].num\_defect = 0  self.semi\_products[i].set\_defect\_rate(0) # 更新成品的实际残次率   else:  # 保持当前次品率继续  pass   # 组装： 1-3号半成品组成成品  self.product.num += min(  self.semi\_products[0].num,  self.semi\_products[1].num,  self.semi\_products[2].num  )  # 计算成品正确率, 更新半成品数量  semi\_product\_correct\_rate = 1  for i in range(0, 3):  semi\_product\_correct\_rate \*= (1 - self.semi\_products[i].defect\_rate)  self.semi\_products[i].num -= self.product.num   # 更新成品的实际残次率  # 实际残次率源于半成品的残次率和自身残次率  defect\_rate = 1 - semi\_product\_correct\_rate \* (1 - self.product.assemble\_defect\_rate)  self.product.set\_defect\_rate(defect\_rate)  product\_defect\_rate = defect\_rate   # 是否检测成品：  product\_cost += self.product.price \* self.product.num # 组装成本  if self.settings[CHECK\_PRODUCT] == 1:  product\_cost += self.product.check\_price \* self.product.num # 检测成本  self.product.clean\_defect() # 区分次品, 这里会修改num和num\_defect 的值  self.product.set\_defect\_rate(0) # 综合次品率 0%   else:  # 流入市场，并承担调换费用， 保持综合次品率不变  self.product.clean\_defect() # 区分次品, 这里会修改num和num\_defect 的值  product\_cost += self.product.num\_defect \* self.product.product\_return # 调换费用   total\_product\_num += self.product.num # 生产产品正品数保存  # 是否拆解残次成品  if self.settings[DISMANTLE\_PRODUCT] == 1:  # 拆解费用  product\_cost += (  self.product.product\_dismantle  \* self.product.num\_defect  )  # 拆解成半成品  self.semi\_products[0].num += self.product.num\_defect  self.semi\_products[1].num += self.product.num\_defect  self.semi\_products[2].num += self.product.num\_defect  self.product.num\_defect = 0 # 清零  self.product.num = 0   # 综合利润：  self.score = total\_product\_num \* self.product.sell\_price - product\_cost - semi\_product\_cost - parts\_cost  return {  "score": round(self.score, 2),  "total\_product\_num": round(total\_product\_num, 2),  "product\_cost": round(product\_cost, 2),  "semi\_product\_cost": round(semi\_product\_cost, 2),  "parts\_cost": round(parts\_cost, 2),  "product\_defect\_rate": product\_defect\_rate  } |

|  |
| --- |
| **附录7** |
| 介绍：该代码用于遗传算法的搭建 |
| # 使用遗传算法来优化选择模型 import random from factory\_class\_pro import Factory from product\_file import parts, semi\_products, product  # 全局设置 INDIVIDUAL\_SIZE = 16 # 个体长度 POPULATION\_SIZE = 100 # 种群大小 MAX\_GENERATION = 20 # 最大迭代次数 MUTATION\_RATE = 0.05 # 变异概率 POPULATION\_RATE = 0.5 # 繁衍比率 CROSS\_RATE = 0.5 # 交叉概率   # 随机生成初始化种群 def generate\_population(individual\_size, population\_size):  population\_ = []  for n\_ in range(population\_size):  li = []  for i\_ in range(individual\_size):  li.append(random.randint(0, 1))  population\_.append(li)  return population\_   # 评估存活率函数 def calculate\_score(individual: list) -> float:  factory = Factory()  factory.set\_settings(individual)  factory.set\_part(parts)  factory.set\_semi\_product(semi\_products)  factory.set\_product(product)  data = factory.get\_score()  return round(data['score'], 2)   # 交叉操作, 两个父母生两个娃，保证群体数目不波动 def crossover(parent1: dict, parent2: dict) -> (dict, dict):  child\_1 = {'individual': [], 'score': 0}  child\_2 = {'individual': [], 'score': 0}  for i\_\_ in range(len(parent1['individual'])):  if random.random() < 0.5:  child\_1['individual'].append(parent1['individual'][i\_\_])  child\_2['individual'].append(parent2['individual'][i\_\_])  else:  child\_1['individual'].append(parent2['individual'][i\_\_])  child\_2['individual'].append(parent1['individual'][i\_\_])  return child\_1, child\_2   # 变异操作 def mutate(individual\_: dict) -> dict:  for i\_ in range(len(individual\_)):  if random.random() < MUTATION\_RATE:  individual\_['individual'][i\_] = 1 - individual\_['individual'][i\_]  return individual\_   # 种群繁衍，假设都会繁衍 def breed(population\_: list) -> list:  new\_population = population\_  for i\_ in range(int(len(population\_) \* POPULATION\_RATE)):  # 随机选择一对父母  parent1 = random.choice(population\_)  parent2 = random.choice(population\_)  child\_1, child\_2 = crossover(parent1, parent2) # 交叉  child\_1 = mutate(child\_1) # 变异  child\_2 = mutate(child\_2)  # 添加新成员  new\_population.append(child\_1)  new\_population.append(child\_2)  return new\_population   # 进行自然选择，淘汰一半个体 def select(population\_: list) -> list:  *"""  :param population\_: 种群  :return: 选出前50%的个体  解释一下population\_的数据结构：  [  {'individual': 个体, 'score': 分数},  {'individual': 个体, 'score': 分数},  ....  ]  """* for y in population\_:  y["score"] = calculate\_score(y["individual"])  population\_.sort(key=lambda x: x["score"], reverse=True) # 按分数排序  return population\_[:int(len(population\_) \* POPULATION\_RATE)] # 选出前50% |

|  |
| --- |
| **附录8** |
| 介绍：使用遗传算法求解最优组合 |
| import pandas as pd from matplotlib import pyplot as plt from generation import \* import seaborn as sns import time  """  解释一下population的数据结构：  [  {'individual': 个体, 'score': 分数},  {'individual': 个体, 'score': 分数},  ....  ] """  # *todo 设置序列详细：* # *todo 0 - 7 ： 1-8号零件是否检测* # *todo 8 - 10： 1-3号半成品是否检测* # *todo 11：成品是否检测* # *todo 12 - 14： 1-3号是否拆解* # *todo 15：成品是否拆解* # *todo 全局设置* INDIVIDUAL\_SIZE = 16 # 个体长度 POPULATION\_SIZE = 100 # 种群大小 MAX\_GENERATION = 20 # 最大迭代次数 MUTATION\_RATE = 0.05 # 变异概率 POPULATION\_RATE = 0.5 # 繁衍比率 CROSS\_RATE = 0.5 # 交叉概率  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False plt.rcParams['font.size'] = 18 # 设置全局字体大小   def plot\_show(times, score\_logs):  # 使用seaborn来绘制优化过程  sns.lineplot(x=range(times), y=score\_logs)  plt.xlabel("迭代次数")  plt.ylabel("最优分数")  plt.title("遗传算法优化过程", fontsize=18)  plt.savefig("遗传算法优化过程.png")  plt.show()   if \_\_name\_\_ == "\_\_main\_\_":  population = [] # 种群  best\_score = 0 # 最高分数  best\_settings = [] # 最优设置  best\_score\_logs = [] # 最高分数记录  best\_settings\_logs = [] # 最优设置记录  shou\_lian\_times = 0 # 收敛所用次数  start\_time = time.time() # 开始时间  log = [] # 运行日志   # 初始化种群  individuals = generate\_population(INDIVIDUAL\_SIZE, POPULATION\_SIZE)  print("初始化种群")   # 更改数据结构  for i in individuals:  population.append({"individual": i, "score": 0})   # 模拟种群运行  for i in range(MAX\_GENERATION):  population = breed(population) # 繁衍  population = select(population) # 选择  if population[0]["score"] > best\_score:  best\_score = population[0]["score"] # 最高分数  shou\_lian\_times = i + 1  best\_settings = population[0]["individual"] # 记录最优设置  best\_score\_logs.append(best\_score) # 记录最高分数  best\_settings\_logs.append(best\_settings) # 记录最优设置  # print(f"第{i+1}代, 最优分数为{best\_score}, 最优设置: {best\_settings}， 人口大小： {len(population)}")  log.append({  "generation": i + 1,  "best\_score": best\_score,  "best\_settings": best\_settings,  "population\_size": len(population),  })   end\_time = time.time() # 结束时间   log = pd.DataFrame(log)  log.to\_csv("遗传算法运行日志.csv", index=False)   print("遗传算法运行时间：", round(end\_time - start\_time, 4), 's')  print("收敛次数：", shou\_lian\_times)  print("最优设置:", best\_settings)  print("最优每件利润:", round(best\_score / 100, 2))   # 将分数除以100,得到平均利润  for i in range(len(best\_score\_logs)):  best\_score\_logs[i] /= 100  best\_score\_logs[i] = round(best\_score\_logs[i], 2)   # 使用seaborn来绘制优化过程  plot\_show(MAX\_GENERATION, best\_score\_logs) |

|  |
| --- |
| **附录9** |
| 介绍：比较暴力求解的结果 |
| import itertools import time  from factory\_class\_pro import Factory from product\_file import parts, semi\_products, product   # *todo 设置序列详细：* # *todo 0 - 7 ： 1-8号零件是否检测* # *todo 8 - 10： 1-3号半成品是否检测* # *todo 11：成品是否检测* # *todo 12 - 14： 1-3号是否拆解* # *todo 15：成品是否拆解* def generate\_combinations(n):  return list(itertools.product([0, 1], repeat=n))   if \_\_name\_\_ == '\_\_main\_\_':  start\_time = time.time() # 开始时间  # 生成所有可能的组合  settings = generate\_combinations(16) # 16个参数 65536 种组合  best\_setting = [] # 最优的设置  max\_score = 0 # 最优的分数  product\_defect\_rate = 0 # 产品综合次品率  data = []   for setting in settings:  factory = Factory()  # 重置工厂属性  factory.set\_part(parts)  factory.set\_semi\_product(semi\_products)  factory.set\_product(product)  factory.set\_settings(setting)   data = factory.get\_score()  if data["score"] > max\_score:  max\_score = data["score"]  best\_setting = setting  product\_defect\_rate = round(data["product\_defect\_rate"], 4)  # print(setting, data["score"])   end\_time = time.time() # 结束时间  run\_time = round(end\_time - start\_time, 4)  print("暴力枚举所有情况运行时间运行时间：", run\_time, 's')  print("最优每件利润:", round(max\_score / 100, 2))  print("最优设置:", best\_setting)  print('产品综合次品率：', product\_defect\_rate \* 100, '%') |

|  |
| --- |
| **附录10** |
| 介绍：测试遗传算法的稳定性 |
| import pandas as pd from matplotlib import pyplot as plt from generation import \* import seaborn as sns import time  # *todo 全局设置* INDIVIDUAL\_SIZE = 16 # 个体长度 POPULATION\_SIZE = 100 # 种群大小 MAX\_GENERATION = 20 # 最大迭代次数 MUTATION\_RATE = 0.05 # 变异概率 POPULATION\_RATE = 0.5 # 繁衍比率 CROSS\_RATE = 0.5 # 交叉概率 TEST\_TIMES = 50 # 测试次数  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False plt.rcParams['font.size'] = 18 # 设置全局字体大小   def plot\_show(times, score\_logs):  # 使用seaborn来绘制优化过程  sns.lineplot(x=range(times), y=score\_logs)  plt.xlabel("迭代次数")  plt.ylabel("最优分数")  plt.title("遗传算法优化过程", fontsize=18)  plt.savefig("遗传算法优化过程.png")  plt.show()   def main():  global log # 记录日志  population = [] # 种群  best\_score = 0 # 最高分数  best\_settings = [] # 最优设置  best\_score\_logs = [] # 最高分数记录  best\_settings\_logs = [] # 最优设置记录  shou\_lian\_times\_ = 0 # 收敛所用次数  start = time.time()  # 初始化种群  individuals = generate\_population(INDIVIDUAL\_SIZE, POPULATION\_SIZE)   # 更改数据结构  for i in individuals:  population.append({"individual": i, "score": 0})   # 模拟种群运行  for i in range(MAX\_GENERATION):  population = breed(population) # 繁衍  population = select(population) # 选择  if population[0]["score"] > best\_score:  best\_score = population[0]["score"] # 最高分数  shou\_lian\_times\_ = i + 1  best\_settings = population[0]["individual"] # 记录最优设置  best\_score\_logs.append(best\_score) # 记录最高分数  best\_settings\_logs.append(best\_settings) # 记录最优设置  # print(f"第{i+1}代, 最优分数为{best\_score}, 最优设置: {best\_settings}， 人口大小： {len(population)}")   end = time.time()   log.append({  "best\_settings": str(best\_settings),  "best\_score": round(best\_score / 100, 2),  })  # print("收敛次数：", shou\_lian\_times\_)  # print("最优设置:", best\_settings)  # print("最优分数:", best\_score)   return shou\_lian\_times\_, end - start   if \_\_name\_\_ == "\_\_main\_\_":  shou\_lian\_times\_log = [] # 收敛次数记录  run\_time\_log = [] # 运行时间记录  log = [] # 记录数据   for \_ in range(TEST\_TIMES):  shou\_lian\_times, run\_time = main()  shou\_lian\_times\_log.append(shou\_lian\_times)  run\_time\_log.append(run\_time)   shou\_lian\_times = sum(shou\_lian\_times\_log)  run\_time = sum(run\_time\_log)   print("平均运行时间：", run\_time / TEST\_TIMES, 4)  print("平均收敛次数：", shou\_lian\_times / TEST\_TIMES)   # 日志保存：  print(log)  log = pd.DataFrame(log)  log.to\_csv("遗传算法稳定性检验数据.csv", index=False)   df = pd.DataFrame(shou\_lian\_times\_log)   # 使用seaborn来绘制优化过程  sns.lineplot(data=df)  plt.xlabel("运行次数")  plt.ylabel("收敛次数")  plt.title(f"遗传算法收敛次数，平均次数{shou\_lian\_times / TEST\_TIMES}", fontsize=18)  plt.savefig("遗传算法迭代次数.png")  plt.show()   df = pd.DataFrame(run\_time\_log)  # 使用seaborn来绘制优化过程  sns.lineplot(data=df)  plt.xlabel("运行次数")  plt.ylabel("运行所用时间")  plt.title(f"遗传算法运行时间，平均运行时间{round(run\_time / TEST\_TIMES, 3)}s", fontsize=18)  plt.savefig("遗传算法运行时间.png")  plt.show() |

|  |
| --- |
| **附录11** |
| 介绍：第四题：对第二题重新检验 |
| import itertools import random from matplotlib import pyplot as plt import pandas as pd from 第二题 import factory\_class, case\_file import seaborn as sns  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False plt.rcParams['font.size'] = 16 # 设置全局字体大小   # 生成所有组合的函数 def generate\_combinations(n):  return list(itertools.product([0, 1], repeat=n))   test\_times = 200 # 测试次数 cases = case\_file.cases # 读取所有案例 E = 0.05 # 允许的误差 log = []  if \_\_name\_\_ == '\_\_main\_\_':  i = 1 # 遍历案例的编号  best\_score\_logs = [] # 最优的分数记录  best\_settings\_logs = {  '案例1': {},  '案例2': {},  '案例3': {},  '案例4': {},  '案例5': {},  '案例6': {}  } # 最优的设置记录  for case in cases:  for t in range(test\_times):  test\_case = case.copy() # 复制一个实例  # 基于原有次品率随机生成各零件的次品率  test\_case['part\_1\_defect'] = round(random.uniform(case['part\_1\_defect'] - E, case['part\_1\_defect'] + E), 4)  test\_case['part\_2\_defect'] = round(random.uniform(case['part\_2\_defect'] - E, case['part\_2\_defect'] + E), 4)  test\_case['product\_defect'] = round(random.uniform(case['product\_defect'] - E, case['product\_defect'] + E), 4)  factory = factory\_class.Factory()  # 初始化设置  factory.set\_defect\_rate(  test\_case["part\_1\_defect"],  test\_case["part\_2\_defect"],  test\_case["product\_defect"]  )  factory.set\_price(  test\_case["part\_1\_price"],  test\_case["part\_2\_price"],  test\_case["product\_price"],  test\_case["product\_sell"]  )  factory.set\_check(  test\_case["part\_1\_check"],  test\_case["part\_2\_check"],  test\_case["product\_check"]  )  factory.set\_return(  test\_case["product\_return"],  test\_case["product\_dismantle"]  )  # 生成所有可能的组合  settings = generate\_combinations(4)  best\_setting = [] # 最优的设置  max\_score = 0 # 最优的分数  for setting in settings:  factory.set\_settings(setting)  score = factory.get\_score()  if score > max\_score:  max\_score = score  best\_setting = setting  # print(f"第{i}个案例，第{t + 1}次测试，最优设置：{best\_setting}，最优分数：{round(max\_score, 2)}")  log.append({  '案例编号': i,  '测试轮次': t + 1,  '最优设置': tuple(best\_setting),  '最优分数': round(max\_score, 2)  })  try:  best\_settings\_logs[f'案例{i}'][tuple(best\_setting)] += 1  except KeyError:  best\_settings\_logs[f'案例{i}'][tuple(best\_setting)] = 1   i += 1  log = pd.DataFrame(log)  log.to\_csv('第二题重新检验.csv', index=False)  # 减少采样数据点  log\_sampled = log.groupby('案例编号').apply(lambda x: x.iloc[::10, :]).reset\_index(drop=True)  sns.lineplot(data=log\_sampled, x='测试轮次', y='最优分数', hue='案例编号')  plt.title('第二题不同案例下不同测试轮次的最优分数', fontsize=15)  plt.savefig('第二题不同案例下不同测试轮次的最优分数.png')  plt.show()   # 画出不同案例下最优分数中位数  grouped\_log = log.groupby('案例编号')['最优分数'].median()  print(grouped\_log)  sns.barplot(x=grouped\_log.index, y=grouped\_log.values)  plt.title('第二题不同案例下最优分数中位数', fontsize=15)  plt.savefig('第二题不同案例下最优分数中位数.png')  plt.show()   # 画出不同案例下最优设置的众数  li = []  y = 1  for test\_case in best\_settings\_logs.values():  setting, count = max(test\_case.items(), key=lambda item: item[1])  li.append({  '案例编号': y,  '最优设置': setting,  '出现次数': count  })  y += 1  li = pd.DataFrame(li)  print(li)  bar\_plot = sns.barplot(data=li, x='案例编号', y='出现次数', hue='案例编号')  # 添加最佳分数数据和设置标签  for index, row in li.iterrows():  bar\_plot.text(index, row["出现次数"], f"{row['最优设置']}",color='black', ha="center", va="bottom")  plt.title('第二题不同案例下最优设置的众数', fontsize=15)  plt.xlabel('案例编号', fontsize=15)  plt.ylabel('出现次数最高的最优设置', fontsize=15)  plt.savefig('第二题不同案例下最优设置的众数.png')  plt.show() |

|  |
| --- |
| **附录12** |
| 介绍：第四题：对第三题重新检验 |
| import numpy as np import pandas as pd from matplotlib import pyplot as plt from generation import \* import seaborn as sns import time from product\_file import parts, semi\_products, product  # *todo 全局设置* INDIVIDUAL\_SIZE = 16 # 个体长度 POPULATION\_SIZE = 100 # 种群大小 MAX\_GENERATION = 20 # 最大迭代次数 MUTATION\_RATE = 0.05 # 变异概率 POPULATION\_RATE = 0.5 # 繁衍比率 CROSS\_RATE = 0.5 # 交叉概率 TEST\_TIMES = 50 # 测试次数 E = 0.05 # 误差  # 设置sns字体 sns.set(font=['Microsoft YaHei']) sns.set\_context("paper") plt.rcParams['font.sans-serif'] = ['Microsoft YaHei'] plt.rcParams['axes.unicode\_minus'] = False plt.rcParams['font.size'] = 18 # 设置全局字体大小   def plot\_show(times, score\_logs):  # 使用seaborn来绘制优化过程  sns.lineplot(x=range(times), y=score\_logs)  plt.xlabel("测试次数")  plt.ylabel("最优分数")  plt.title("遗传算法检验运行过程", fontsize=18)  plt.savefig("遗传算法检验运行过程.png")  plt.show()   def main(parts\_, semi\_products\_, product\_):  population = [] # 种群  best\_score = 0 # 最高分数  best\_settings = [] # 最优设置  best\_score\_logs = [] # 最高分数记录  best\_settings\_logs = [] # 最优设置记录  # 初始化种群  individuals = generate\_population(INDIVIDUAL\_SIZE, POPULATION\_SIZE)   # 更改数据结构  for i in individuals:  population.append({"individual": i, "score": 0})   # 模拟种群运行  for i in range(MAX\_GENERATION):  population = breed(population) # 繁衍  population = select(population, parts\_, semi\_products\_, product\_) # 选择  if population[0]["score"] > best\_score:  best\_score = population[0]["score"] # 最高分数  best\_settings = population[0]["individual"] # 记录最优设置  best\_score\_logs.append(best\_score) # 记录最高分数  best\_settings\_logs.append(best\_settings) # 记录最优设置   return best\_settings, best\_score   if \_\_name\_\_ == "\_\_main\_\_":  best\_settings\_log = [] # 最优设置记录  best\_score\_log = [] # 最高分数记录  log = [] # 测试记录   for \_ in range(TEST\_TIMES):  # 基于随机数据进行测试  parts\_, semi\_products\_, product\_ = parts.copy(), semi\_products.copy(), product.copy()  for part\_ in parts\_:  part\_['defect\_rate'] = random.uniform(0.1 - E, 0.1 + E)  for semi\_product\_ in semi\_products\_:  semi\_product\_['defect\_rate'] = random.uniform(0.1 - E, 0.1 + E)   product\_['defect\_rate'] = random.uniform(0.1 - E, 0.1 + E)   best\_settings, best\_score = main(parts\_, semi\_products\_, product\_)  best\_settings\_log.append(tuple(best\_settings))  best\_score\_log.append(best\_score)  log.append({  "测试轮次": \_ + 1,  "最优设置": tuple(best\_settings),  "最优设置对应的最优利润": round(best\_score / 100, 2),  })   # 计算每件利润  for i in range(len(best\_score\_log)):  best\_score\_log[i] = round(best\_score\_log[i] / 100, 2)   # 生成数据框  df = pd.DataFrame(log)  df.to\_csv("第三问测试结果.csv", index=False)  plot\_show(TEST\_TIMES, best\_score\_log)   print("平均最优分数： ", round(sum(best\_score\_log) / len(best\_score\_log), 2))   setting\_counter = {} # 记录最优设置数目  for setting in best\_settings\_log:  if setting in setting\_counter:  setting\_counter[setting] += 1  else:  setting\_counter[setting] = 1  print("最优设置分布： ")  for setting, count in setting\_counter.items():  print(setting, count) |