第十届湖南省研究生数学建模竞赛承诺书

我们仔细阅读了湖南省高校研究生数学建模竞赛的竞赛规则。

我们完全明白，在竞赛开始后参赛队员不能以任何方式（包括电话、电子邮件、网上咨询等）与队外的任何人（包括指导教师）研究、讨论与赛题有关的问题。

我们知道，抄袭别人的成果是违反竞赛规则的，如果引用别人的成果或其他公开的资料（包括网上查到的资料），必须按照规定的参考文献的表述方式在正文引用处和参考文献中明确列出。

我们完全清楚，在竞赛中必须合法合规地使用文献资料和软件工具，不能有任何侵犯知识产权的行为。否则我们将失去评奖资格，并可能受到严肃处理。

我们郑重承诺，严格遵守竞赛规则，以保证竞赛的公正、公平性。如有违反竞赛规则的行为，我们将受到严肃处理。

我们授权湖南省研究生数学建模竞赛组委会，可将我们的论文以任何形式进行公开展示（包括进行网上公示，在书籍、期刊和其他媒体进行正式或非正式发表等）。

我们参赛选择的题号是（从组委会提供的赛题中选择一项填写）：A

我们的参赛编号（请填写完整参赛编号）：202518001001

所属学校（请填写完整的全名）中国人民解放军国防科技大学

参赛队员（打印后签名）：1. 付钰雯

2. 王康

3. 孟庆豪

指导教师或指导教师组负责人（打印后签名）：张军

日期：2025 年 8 月 26 日

（请勿改动此页内容和格式。以上内容请仔细核对，如填写错误，论文可能被取消评奖资格。）

第十届湖南省研究生数学建模竞赛

题　目：大型装备并行测试的调度优化与可靠性分析

摘要：随着大型装备技术的发展，复杂系统测试任务的调度与可靠性问题日益突出，成为保障系统运行安全与效率的关键环节。本文针对多子系统并行测试中的调度优化与误判漏判控制问题，基于随机事件建模与离散事件仿真的思想，建立了融合设备故障率分布、操作差错与重测机制的仿真模型，并结合**蒙特卡洛方法**与改进的事件驱动算法对不同测试方案进行求解与验证。

针对问题一，构建综合测试问题溯源模型，区分子系统漏判与联接系统缺陷的贡献，建立综合问题判定的概率分解框架，并计算得到系统总体问题概率及各子系统的指向比例，为综合测试结果提供可靠解释。

针对问题二，在**单班制（每日 12 小时）约束**下，采用离散事件仿真模拟多阶段并行测试流程。经 10000 次蒙特卡洛仿真，结果表明平均完成天数为 **28.91 天**，通过装置数 99.81 台，总漏判率 0.03%，误判率 1.35%。同时，单次并行仿真显示在典型运行序列下 100 台装置全部通过，平均完成天数 **26 天**，漏判率为 0%，误判率约 0.5%，验证了模型的稳健性与可行性。

针对问题三，在**双班倒模式**下，将班次时长作为优化变量，建立蒙特卡洛—离散事件复合决策模型，从效率、质量与连续性多维度进行综合评价。结果表明，当**班次长度为 11.5 小时**时性能最优，平均完成天数缩短至 **14 天**，通过装置数 98 台，总漏判率 0.01%，误判率 1.46%，各测试组负载分布均衡，实现了效率与可靠性的统一。

针对问题四，基于双班制仿真数据开展敏感性分析，识别出影响平均完成天数的关键因素，包括测试小组负载不均衡与班次切换衔接损耗等。在此基础上，提出了优化测试流程与资源配置的改进建议，如动态调整班次时长、提高跨班衔接效率和均衡关键小组负载，为主管部门制定更优的测试调度方案提供了决策支持。

综上，本文提出的模型能够在**多资源、多阶段及高随机性环境**下合理模拟测试任务执行过程，既能有效提升测试效率，又能控制误判与漏判风险，并可推广应用于大型装备测试调度、智能制造与复杂工程测试管理等领域。

关键词：离散事件仿真　蒙特卡洛方法　班次优化　任务调度　可靠性分析

目录

1 问题综述

1.1 问题背景

随着大型装备技术的发展，大型装置在能源、国防及工业生产等领域的应用越来越广泛，其性能与可靠性直接关系到系统整体运行安全与效率[1] [2] 。大型装置通常由多个子系统组成，其功能不仅依赖各子系统自身的可靠性，也取决于整体联接后的系统协同性。在实际测试过程中，设备故障、操作差错、子系统缺陷等随机事件频繁发生，同时测试资源有限，例如测试台数量有限、专业测试小组固定、每日工作班次与时长受限，这使得测试任务在时间安排、资源调配和结果准确性上面临诸多挑战[3] 。因此，如何在有限资源与时间约束下，科学合理地安排测试任务，既保证测试效率，又降低误判和漏判概率，成为大型装置测试中亟待解决的问题[4] [5] 。

目前，已有学者针对装备测试调度和复杂系统可靠性问题开展了大量研究，提出了包括排队模型[6] [7] 、蒙特卡洛仿真[8] [9] 、启发式调度算法[10] [11] 等方法。传统调度方法，如先来先服务（FCFS）[12] [13] 、最短处理时间优先（SPT）、遗传算法[14] [15] 、蚁群算法等[16] ，能够在一定程度上生成合理的测试安排。然而，这些方法往往忽略多子系统联动下的随机性事件，如设备故障的累积概率、操作误差导致的误判与漏判和综合测试中对子系统问题的指向性分配，实际应用中出现资源冲突、测试延迟或整体效率下降等问题。

因此，本文拟基于随机事件建模与任务调度优化方法，构建一套综合考虑设备故障、操作误差、子系统问题及测试资源约束的测试任务规划模型，以在满足可靠性要求的前提下优化测试效率，并为测试任务计划提供可量化的决策支持。

1.2 问题提出

大型装置测试任务涉及多方面的问题，往往由子系统可靠性、测试设备状态、操作差错、测试资源约束等要素构成。在满足下列 3 点约束下:

- 测试设备与操作约束: 测试设备需经过调试校对，使用时间受寿命及故障概率限制；若测试因设备故障或中断事件未完成，则该工序必须重新开始；操作差错可能导致误判或漏判。
- 测试资源与顺序约束: 测试大厅配备两个测试台，可同时测试 2 个装置；四个专业测试小组独立作业，每组一次仅能处理一个装置的对应子系统测试。每台装置必须按顺序完成子系统测试，全部通过后进行综合测试；连续两次未通过则退出测试。
- 班次与运输约束: 每日班次工作时间不超过 12 小时，装置运输与替换各耗时 0.5 小时，两个装置可同时进行运输或换入换出，测试任务规模为 100 台装置。

需要从测试效率、装置可靠性、测试资源利用率角度考虑，解决以下 4 个问题：

(1) 问题 1:计算综合测试测出系统存在问题的概率及各子系统问题指向比例参数λ_i（i=1,2,3,4），并代入具体数值计算概率值和比例参数值。
(2) 问题 2:针对单班制，即每日 12 小时测试 100 台装置，制定合理的测试工作计划，并计算该计划下的任务平均完成天数T、通过测试装置平均数S、总漏判概率P_L、总误判概率P_w及各专业测试组有效工作时间比YXB_X。
(3) 问题 3:在第二批 100 台装置测试中，采用两个分队接续倒班，确定最优班次长度（K）并制定测试工作计划，计算对应任务平均完成天数、通过测试装置平均数S、总漏判概率P_L、总误判概率P_w及各专业测试组有效工作时间比YXB_X。
(4) 问题 4:评估问题 3 中各因素对测试任务平均完成时间的影响程度，并基于分析结果提出针对测试流程和资源配置的改进建议，为主管部门优化测试工作提供依据。

2 模型假设与符号参数说明

2.1 模型基本假设

(1) 测试顺序与完成条件假设:假定对同一台装置，子系统A、B、C可在不同专业小组上同时并行检测。
(2) 测试设备与操作独立性假设: 四个测试小组 A、B、C、E 的设备与操作相互独立；设备故障和操作差错概率仅与对应子系统相关
(3) 资源与调度约束假设: 测试大厅有两个测试台，每个小组一次仅处理一个装置的对应子系统；每日班次工作时间固定；装置运输与替换耗时固定，可同时进行。
(4) 随机事件概率假设: 测试过程中设备故障、子系统问题、测手误判/漏判及综合测试发现问题均为随机事件，其发生概率由题目给定；事件之间相互独立。
(5) 批量测试稳定性假设: 在大批量装置测试下，平均完成时间、装置通过率、误判和漏判概率可视为稳定值，可用于任务计划评估与优化。

2.2 符号说明

本文定义了如表 1 所示的使用次数较多的符号, 其余符号在使用时注明。

表1 符号说明

符号	含义	单位
t_A, t_B, t_C, t_E	子系统 A、B、C 及综合测试 E 的单次测试时间	小时
d_A, d_B, d_C, d_E	四类测试设备在首次启用时的调试时间	小时
K	单班次工作时长	小时
q_A, q_B, q_C	子系统 A、B、C 自身存在问题的概率	
w_A, w_B, w_C, w_E	子系统 A、B、C 或综合测试环节 E 中，由操作差错导致的误判或漏判概率	
$p_{A_1}, p_{B_1}, p_{C_1}, p_{E_1}$	各测试设备使用时间少于 120 小时的累计故障概率	
$p_{A_2}, p_{B_2}, p_{C_2}, p_{E_2}$	各测试设备使用时间在 120–240 小时内的累计故障概率	
p_E	综合测试发现系统存在问题的概率	
λ_i（i=1,2,3,4）	综合测试指向各子系统 A、B、C 及综合环节 D 的比例参数	
T	任务平均完成天数	天
S	通过测试的装置平均数量	个
P_L	总漏判概率	
P_w	总误判概率	
YXB_X	各专业测试小组的有效工作时间比	
N	批量测试的装置总数	个
M	测试台数量	个
G	专业测试小组数量	个

2.3 参数整理

为清晰界定子系统 A、B、C 及综合测试 E 的核心运行边界与随机特性，为后续双班制测试任务建模（含离散事件仿真、故障概率计算、调度逻辑设计等）提供精准数据输入，现将两类关键参数——时间类参数（测试时长、首次调试时间）与概率类参数（子系统问题概率、操作差错概率、设备故障累计概率）系统整理汇总。通过表格可直观获取各参数具体数值，结合可视化对比能进一步明确不同子系统及测试组的参数差异，核心参数详情如下：

表2 子系统及测试组参数汇总

参数类型	子系统A	子系统B	子系统C	综合测试E
测试时间	$2.5h$	$2h$	$2.5h$	$3.0h$
首次调试时间	$30min$	$20min$	$20min$	$40min$
子系统问题概率	2.5%	3%	2%	0.1%
操作差错概率	3%	4%	2%	2%
设备故障累计概率($\leq 120h$)	3%	4%	2%	3%
设备故障累计概率($120h$~$240h$)	5%	7%	6%	5%

为更直观对比子系统 A、B、C 及综合测试 E 的时间类、概率类参数差异，将上述参数进行可视化呈现，如图 1（子系统及测试组参数汇总）所示，可清晰观察各参数的分布与对比特征：

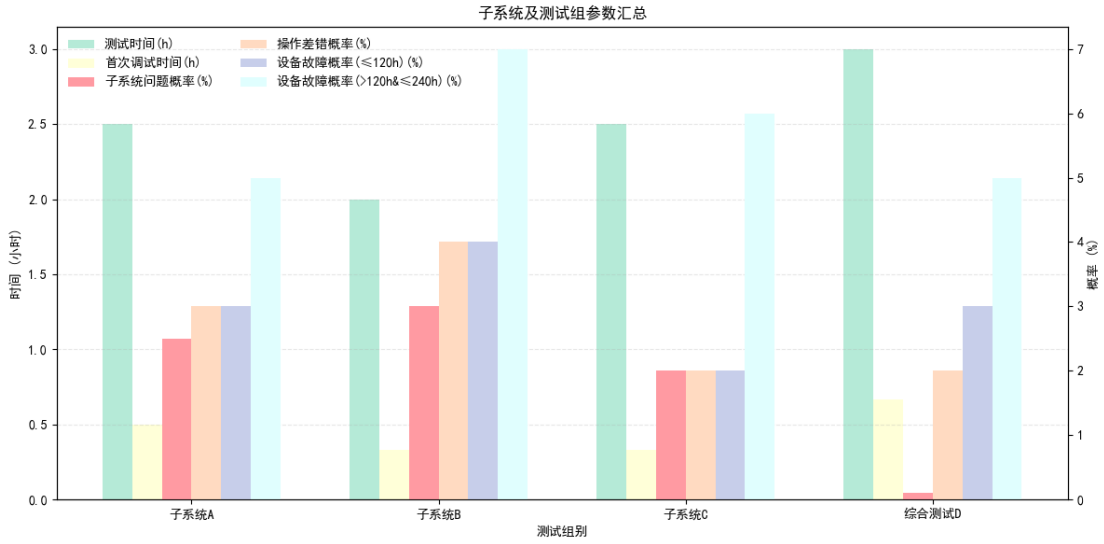


图1 子系统及测试组参数汇总

2.4 问题分析思路

问题总体的分析思路和流程如图 2 所示：完整呈现了双班制测试任务从**初始启动**，到**装置流转与测试执行**，再到**收尾与统计**（的全流程逻辑，清晰涵盖了测试台可用性检查、各阶段测试的衔接规则、装置转运机制以及设备故障或重测等特殊情况的处理顺序。

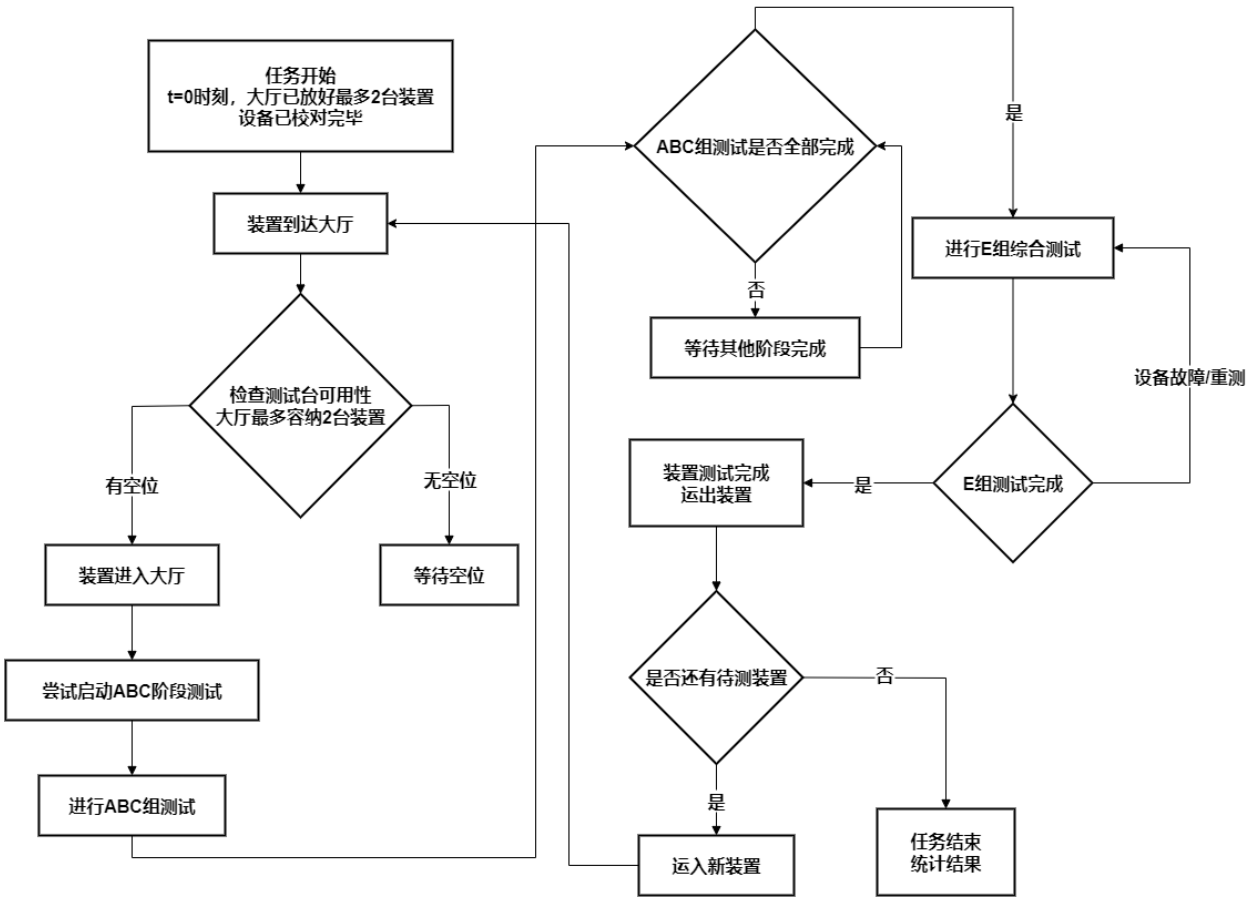


图2 问题分析核心思路总流程图

3 问题 1 建模与求解

3.1 问题分析及建模

综合测试是装置出厂前的最终检验，小组E判定系统有问题，可分为前序子系统漏判与联接系统问题两类，它们互斥且独立，需通过概率推导明确各子系统A、B、C、D对综合测试问题的贡献比例λ_i及综合测试出问题的总概率p_E。下面分别进行分析：

第一，考虑子系统的漏判问题。A、B、C子系统在各自独立测试的过程中，若本身存在问题（概率为q_X，$X \in \{A,B,C\}$），且测手因漏判差错（$Y32$类）未检出，则该问题会遗留至综合测试。而测手的漏判差错（$Y32$）约占测手总差错的 50%，因此子系统i的漏判概率为w_X（X $\in \{A,B,C\}$），为测手测试系统X时的总差错概率。

由于子系统存在问题与测手漏判是两个独立的事件，所以，漏判至综合测试的概率为联合概率p_Y（Y $\in \{A,B,C\}$），即:

$$p_Y = q_X \times w_X \times 0.5 \tag{1}$$

第二，考虑联接系统问题。A、B、C子系统整体联接后也可能产生新问题，将联接系统也视为一个子系统，记作子系统 D，它本身存在问题的概率为p_D。

因此，定义综合测试出问题的总概率p_E为子系统 A、B、C 自身存在问题且漏判的概率和联接系统 D 有问题的概率之和，即：

$$p_E = \sum_{Y\in\{A,B,C\}} p_Y + p_D \tag{2}$$

比例参数λ_i（i=1,2,3,4）分别表示子系统 A、B、C、D 的贡献概率的比例，即：

$$\lambda_i = \begin{cases} \frac{p_Y}{p_E}, i = 1,2,3 \\ \frac{p_D}{p_E}, \quad i = 4 \end{cases} \tag{3}$$

综上，我们通过对综合测试问题的场景分解与概率推导，分析两类问题前序子系统漏判、联接系统出问题的过程，构建了贡献比例λ_i与总概率p_E的计算框架，如图 3 所示：

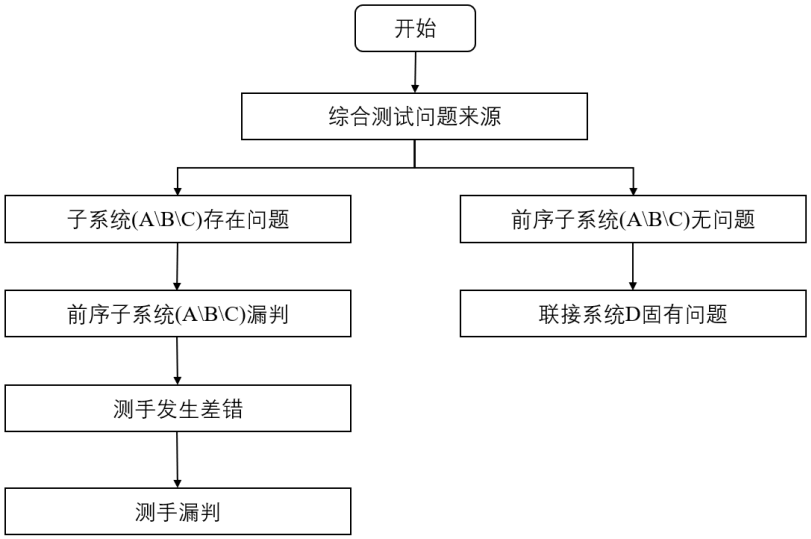


图3 问题 1 分析思路流程图

3.2 求解与结果验证

从题目中提取各子系统的出问题概率及测手发生差错概率的参数取值，如表 3 所示：

表3 子系统及联接系统故障概率取值

子系统	本身问题概率	测手发生差错概率	联接系统问题概率
A	2.5%	3%	-
B	3%	4%	-
C	2%	2%	-
D	-	-	0.1%

根据公式(1)逐个计算子系统漏判率，得到

$p_A = q_A \times w_A \times 0.5$=2.5%×3%×0.5=0.0375%

$p_B = q_B \times w_B \times 0.5$=3%×4%×0.5=0.06%

$p_C = q_C \times w_B \times 0.5$=2%×2%×0.5=0.02%

所以，综合测试总问题概率为：$p_E = \sum_{Y\in\{A,B,C\}} p_Y + p_D$

=0.0375%+0.06%+0.02%=0.2175%

根据公式(2)计算综合测试测出的问题指向各系统的比例参数λ_i：

$$\lambda_1 = \frac{p_A}{p_E} = 0.0375\% \div 0.2175\% \approx 17.2\%$$

$$\lambda_2 = \frac{p_B}{p_E} = 0.06\% \div 0.2175\% \approx 27.6\%$$

$$\lambda_3 = \frac{p_D}{p_E} = 0.02\% \div 0.2175\% \approx 9.2\%$$

$$\lambda_4 = \frac{p_D}{p_E} = 0.1\% \div 0.2175\% \approx 45.9\%$$

综上，通过场景分解、概率推导、定量计算，本章明确了综合测试问题的计算模型。最后求解得到了综合测试判定“系统有问题”的总概率p_E为 0.2175%。综合测试测出有问题时各部分所占比例如表 4 所示。

表4 综合测试测出有问题时各部分所占比例

子系统A	子系统B	子系统C	子系统D
λ_1 =17.2%	λ_2 =27.6%	λ_3 =9.2%	λ_4=45.9%

为更直观展示综合测试出问题时各子系统的占比情况，绘制综合测试出问题时各子系统所占比例的饼状图，如图 4 所示。

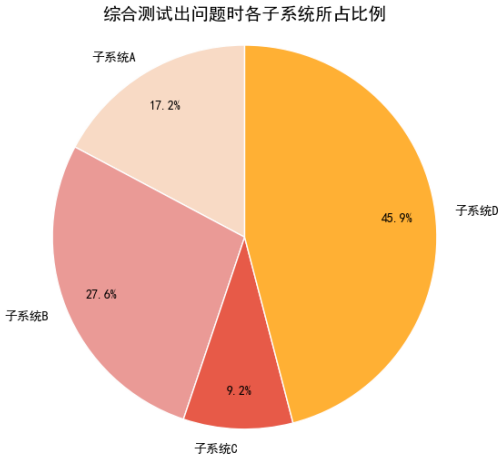


图4 综合测试出问题时各子系统所占比例饼状图

4 问题 2 建模与求解

4.1 问题分析与性能指标定义

4.1.1 问题分析

大型装置测试任务呈现出多阶段、多资源、多随机事件交织的复杂特性，本节针对问题 2 中给定的单班制情形，即每日班次上限为 12 小时，对 100 台大型装置实施测试开展问题分析并给出性能指标。每台装置须分别完成子系统A、B、C的单独测试，且在三子系统全部通过后进入综合测试E；若某工序未通过则需重测，若在同一道工序连续两次未通过则该装置退出测试序列并由新装置顶替。

通过仔细研读题目，我们发现：虽然每台装置的三个子系统测试各自独立，但题目明确指出四个专业小组可同时作业，每组一次仅能处理一个装置的对应子系统测试，且测试台数量为两台。这意味着，对于同一台装置的A、B、C测试，在不同小组和设备可用的情况下可以并行进行，即三阶段之间不存在严格的串行先后顺序。

因此，我们将A、B、C三个子系统视为可并行执行但受资源约束的阶段，即并行-汇合结构：A、B、C三阶段均通过测试后，装置进入综合测试 E。建模流程如图 5 和图 7 所示：

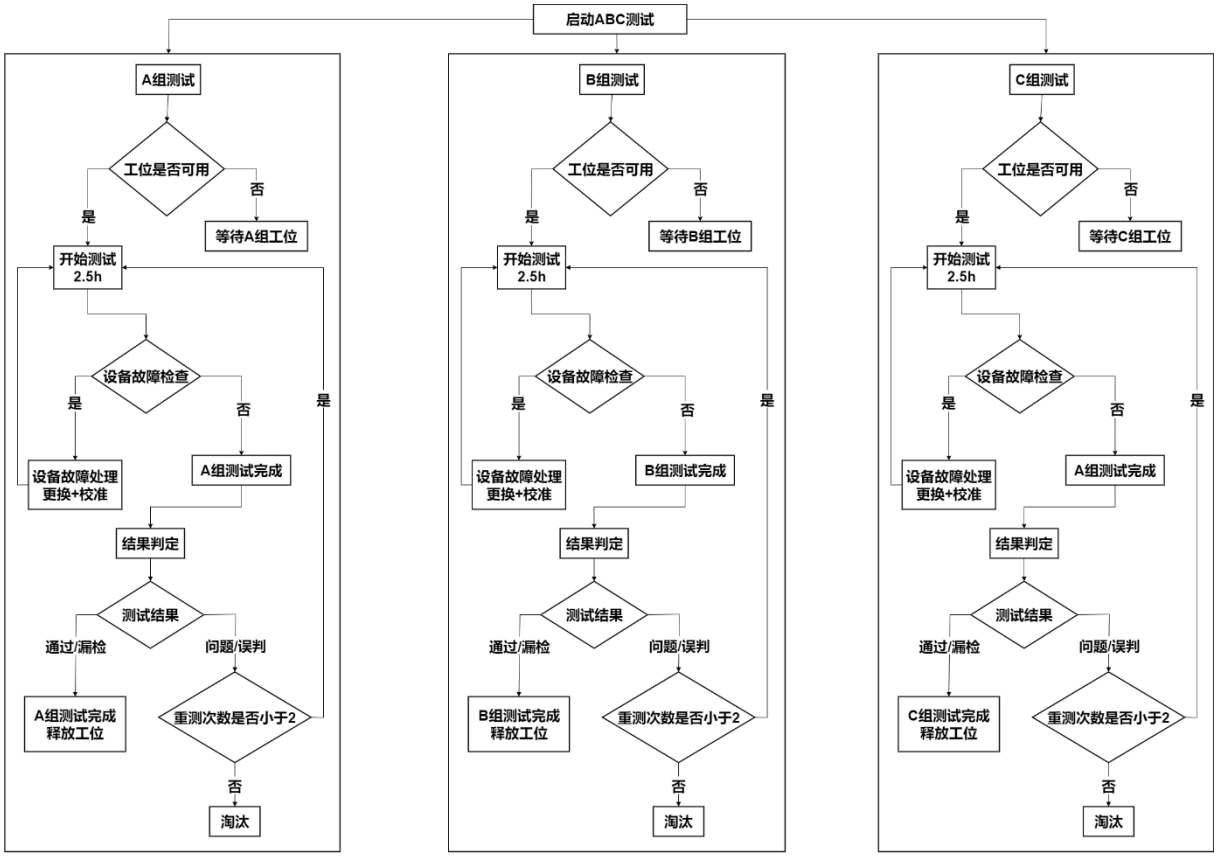
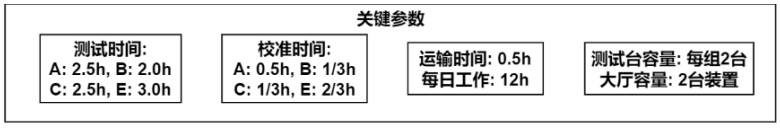


图5 问题 2 建模思路流程图

为清晰呈现测试任务的核心运行参数与异常场景处理逻辑，将关键时间参数（测试、校准、运输等）及各类异常的概率、处理规则进行汇总，具体信息如图 6 所示：



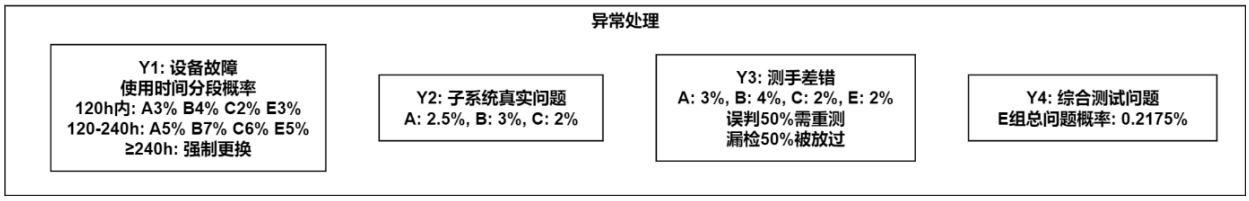


图6 关键参数及异常处理汇总

4.1.2 关键约束与性能指标

首先明确问题中影响调度与性能的约束与不确定性因素：

一是资源约束：每类测试在物理上存在测试台容量限制和设备占用。

二是时间约束：若某工序在班次中断，如因班次结束或故障未完成，则该工序需在下一班次或重测时重新开始，不可接续。

三是不确定性要素：包括测试设备的随机故障概率、子系统本身的真实缺陷；还有测手的操作差错。

同时，为便于后续建模与数值求解，定义下列评价指标：

任务平均完成天数T：完成全部 100 台装置的总工时（即测试、运输与设备校准时间之和）与每日有效工作时长K的比值，并取向上整数天表示。记总仿真时间为T_{tot}，则

$$T = \left\lceil \frac{T_{tot}}{K} \right\rceil \tag{4}$$

通过测试的装置数S：在仿真结束时，成功通过A、B、C及E四项测试的装置总数。

总漏判概率P_L：在所有执行测试过程中，因测手漏判或检测失效而导致实际存在问题未被发现的事件数占总测试次数的比例。漏判事件总数为N_L，总测试次数为N_{Test}，则

$$P_L = \frac{N_L}{N_{Test}} \tag{5}$$

总误判概率P_W：因测手误判导致的误判事件数占总测试次数的比例，记误判事件数为N_W，则

$$P_W = \frac{N_W}{N_{Test}} \tag{6}$$

各小组有效工作时间比YXB_X（$X \epsilon A, B, C, E$）：在整体仿真时间内，小组X的有效测试时间与该小组总可用时间之比，若仿真跨越D个工作日，则该小组总可用时间为$\mathrm{D} \times K$，记小组X的有效测试时间为U_X，则

$$YXB_X = \frac{U_X}{D \times K} \tag{7}$$

综上，我们构建了单班制测试任务的整体流程，考虑了子系统测试顺序、资源约束及随机事件的影响，明确了任务完成效率、装置通过率及各类测试误差等综合性能指标

4.2 单班制测试任务离散事件建模

为系统分析单班制测试任务的工作计划，本文采用离散事件仿真（Discrete Event Simulation, DES）方法，充分模拟多阶段、多资源、多约束的复杂测试流程，通过事件驱动机制精确再现测试过程中各类操作、等待、运输、重测和淘汰事件的时间演化。

4.2.1 单班制测试任务离散事件建模流程

为确保仿真结果能够真实反映单班制测试任务的实际执行过程，我们在建模中设计了三类核心数据结构，分别对应测试对象、资源管理及事件调度，实现了测试流程的动态模拟与性能指标的精确计算。

（1）装置实体（*Device*）

每台待测装置被抽象为一个独立实体，记录其编号、当前测试状态、已完成子系统列表、重测次数以及综合测试结果等信息：

$$Device_i = ID_i, State_i, CompletedSubsystems_i, RetestCount_i, Result_i, i = 1,2, \dots, N \tag{8}$$

在仿真过程中，$State_i$字段用于动态跟踪装置在测试流程中的位置和历史操作，确保每台装置的测试逻辑与实际流程保持一致。例如，装置在完成A、B、C子系统测试后进入综合测试阶段，若任一子系统测试未通过，则状态自动更新为重测或淘汰，驱动后续事件的触发。装置状态可分为：

$$State_i \epsilon \{\text{待测}, \text{测试中}, \text{等待综合测试}, \text{综合测试}, \text{完成}, \text{淘汰}\} \tag{9}$$

在仿真过程中，装置的状态随测试进展动态变化，具体规则如下：

一是子系统测试：完成后，通过则进入下一阶段或等待综合测试；未通过则重测 1 次；二是连续失败：同一子系统连续 2 次未通过，淘汰该装置并补充新待测装置。三是综合测试：完成所有子系统测试后进入，判定联接系统是否通过系统综合测试。：

（2）资源实体（*Resource*）

测试资源包括$M = 2$个测试台和$G = 4$个专业测试小组，每个资源均通过资源实体进行管理。资源实体记录资源的占用状态、当前被分配的装置编号及可用时间窗口:

$$Resource_j = Type_j, Occupied_j, AssignedDevice_j, AvailableTime_j, j = 1,2, \dots, M + G \quad (10)$$

每当装置分配到测试台或小组时，资源状态随之更新；测试结束或装置淘汰时，资源释放并重新进入可用队列。资源占用率可定义为：

$$U_j = \frac{T_{busy,j}}{T_{tot}}, \mathrm{j} = 1,2, \dots, \mathrm{M} + \mathrm{G} \quad (11)$$

其中，$T_{busy,_j}$为资源实际占用时间，T_{tot}为仿真总时间，用于分析资源利用效率。

（3）事件队列（*Event Queue*）

事件队列按时间顺序存储待触发的事件，包括子系统测试开始/结束、设备故障触发、操作差错判定、运输与替换完成、重测安排以及装置淘汰等。事件队列是离散事件仿真的核心，通过队列循环处理驱动整个测试流程的状态演化，每个事件可用以下形式表示：

$$E_k = \left(Time_k, Device_i, EventType_k, Resource_j\right) \quad (12)$$

仿真中，事件队列按事件发生时间顺序处理，每次事件处理含事件触发、状态更新与性能记录三个核心步骤：系统先从队列取出待发生事件，依据事件类型与时间戳判断操作对象及所需资源，如子系统测试启停、运输替换等事件触发；

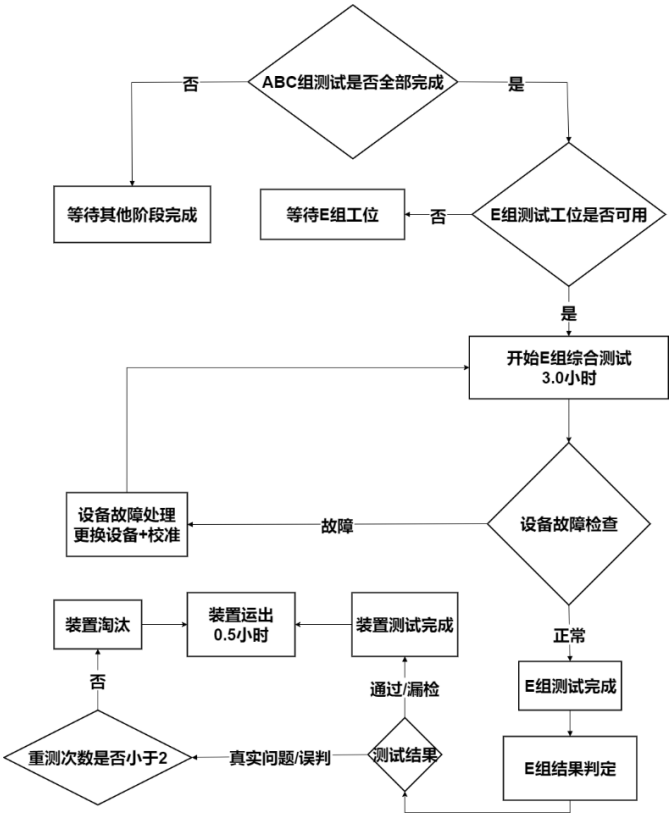


图7 单班制测试任务离散事件建模综合测试流程图

随后按事件类型更新装置状态如与资源占用情况；最后记录单台装置完成时间、漏误判事件数、各小组有效工作时间等性能指标，为仿真结束后量化单班制测试任务整体绩效、支撑调度优化提供数据基础。

4.2.2 设备故障率分布建模

在单班制测试任务中，测试设备可能出现两类故障：一类是机械或电子部件故障，随着使用时间增长，部件磨损或电子元件老化导致设备无法正常工作；另一类是操作相关故障，由于操作差错或人为干预，通常可以通过培训与规范操作加以控制。因此，设备的总故障概率可以视为机械或电子部件故障与操作相关故障的叠加，其中机械或电子部件故障随累计运行时间呈线性递增趋势，而操作相关故障在单位时间内相对稳定。

因此，结合题目中给定的设备在不同时间段的分段累计故障概率，我们认为设备在随累计使用时间的增长，故障概率呈线性均匀递增趋势，采用分段线性均匀增长模型对设备故障率进行建模。

（1）分段均匀分布建模逻辑

为刻画设备不同使用阶段的故障特性，将设备累计运行时间t划分为两个主要区间：

初始阶段：$0 < t < T_1$，在该阶段，A、B、C、E测试设备的累计故障概率较低，但尽管绝对概率较低，其单位时间故障增长速度较大，故障事件较为集中。

延长阶段：$T_1 < t < T_2$，在该阶段，A、B、C、E测试设备的累计故障概率上升，总量增加，但单位时间的增长斜率相对缓和，说明延长期故障概率持续累积但增速下降。

其中，$T_1 = 120$小时，$T_2 = 240$小时。每一阶段，设备故障概率随时间呈均匀递增，

（2）分段函数表示

设设备累计运行时间为t，故障累积概率为$P(t)$，则分段均匀增长模型可表示为

$$P(t) = \begin{cases} \frac{p_{X_1}}{T_1} \cdot t, & 0 \le t \le T_1, \\ P_1 + \frac{p_{X_2} - p_{X_1}}{T_2 - T_1} \cdot (t - T_1), & T_1 < t \le T_2 \end{cases} \tag{13}$$

其中，$p_{X_1}, X \epsilon A,$、B、C、E为初始阶段末的累计故障概率，分别为：3%、4%、2%、3%；$p_{X_2}, X \epsilon A,$、B、C、E为延长阶段末的累计故障概率，分别为：5%、7%、6%、5%。

综上，我们通过分段均匀分布建模设备故障率，将设备在不同使用阶段的故障特性纳入离散事件仿真框架，在每次事件触发时根据设备累计运行时间动态判定故障。

4.3 基于蒙特卡洛仿真与离散时间仿真算法的结果求解

以构建的离散事件仿真模型为基础，采用蒙特卡洛随机仿真结合并行调度试验的方式开展数值求解，针对单班制场景对 100 台装置的整体测试性能进行模拟分析，并量化 A、B、C 子系统并行测试对系统效率与检测质量的影响规律。

4.3.1 算法目标

算法目标是通过随机仿真量化单班制测试任务，求解任务平均完成天数T、通过装置数S、总漏判概率P_L、总误判概率P_W以及各小组有效工作时间比YXB_X等性能指标。

为有效求解并展示单班制任务下的各项任指标，本文采用两类输出结果：

第一类为蒙特卡洛平均结果，通过多轮独立随机仿真获得样本均值及统计量，反映系统在随机条件下的长期期望行为，提供任务完成时间、通过率以及误判、漏检概率等

基准值，同时便于不确定性分析，例如可利用样本标准差计算标准误并进一步构建置信区间，为决策提供可靠统计依据。

第二类为单次并行仿真结果，选取典型运行展示具体事件序列、资源占用及跨日调度情况，直观呈现事件触发与装置排队的动态交互，帮助理解调度策略在单个随机样本下的执行特征，直观展示事件触发、装置排队与调度决策的动态交互情况，帮助理解仿真模型在实际运行中的时间序列逻辑。

综上，两类仿真结果既反映系统的长期平均性能，又呈现单次调度过程的时间序列特征，为后续优化分析提供科学且一致的依据。

4.3.2 算法流程

本算法以事件驱动为核心，通过蒙特卡洛仿真评估单班制测试任务在并行条件下的性能。具体步骤如下：

步骤一：初始化测试装置列表、工作台资源状态及事件队列，确保仿真从有序初始条件开始运行。每台装置的状态设为“待测”，工作台为空闲。

步骤二:从队列中取出时间最早的事件，根据事件类型更新装置状态并触发后续事件，如运输、校准或重测，确保事件按时间顺序处理。

步骤三：实时统计装置完成时间、有效测试时间、设备利用率，以及漏检和误判事件，量化关键性能指标。

步骤四：算法执行N次独立仿真，生成样本集并计算性能指标的平均值、标准差及置信区间，反映系统的长期期望行为。

步骤五：输出关键性能指标，如完成天数、通过装置数、漏检和误判概率，并生成可视化图表，支持优化决策。

4.3.3 蒙特卡洛平均结果分析

（1）结果展示

为量化单班制测试任务在随机扰动下的长期行为，本文对事件驱动仿真模型进行了10000 次蒙特卡洛模拟，统计到的关键性能指标的样本均值及分布如表 5 所示：

另外，蒙特卡洛仿真下各子系统及综合测试的关键性能指标对比结果如图 8 所示：

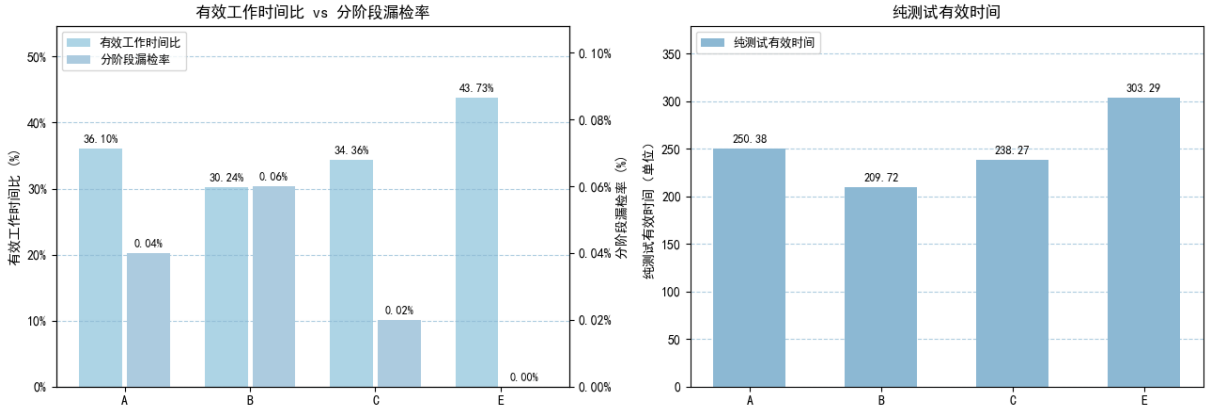


图8 蒙特卡洛仿真下各子系统及综合测试的关键性能指标对比

（2）数值解读

一是总任务用时T_{tot}和等效完成天数$\frac{T_{tot}}{K}$：等效完成天数约为 28.91 天，等价总工时为 340.68 小时，表明在单班制工作时长K =12 小时与现有资源配置下，完成 100 台装置的平均工期约为 29 天，为总体调度计划与资源配置提供了明确的时间量化参考。

二是平均通过装置数。平均通过装置的数量为 99.81，平均淘汰数仅 0.19，说明在给定故障和误差概率模型以及重测规则下，系统总体可达性高、可靠性良好。

三是误判与漏检。总漏检概率为 0.03%，说明总体上漏判事件罕见；但误判概率相对为 1.35%，数值相较于总漏检概率较大，反映测手差错对测试判定影响更大。

四是小组有效工作时间比与设备利用率。YXB_X指标E组为 43.73%，处于最高，说明在工作日有效时间内，其设备使用率相对最集中。

表5 蒙特卡洛仿真 10000 次输出结果

指标	具体数值	指标	具体数值
等效完成天数$\frac{T_{tot}}{K}$	28.91（天）	总误判概率P_W	1.35%
总任务用时T_{tot}	340.68（小时）	有效工作时间比YXB_X	A 36.10%, B 30.24%, C 34.36%, E 43.73%
通过装置数S_P	99.81（台）	纯测试有效时间T_{pt}	A 250.38, B 209.72, C 238.27, E 303.29
淘汰装置数S_F	0.19（台）	各阶段设备利用率	A 36.10%, B 30.24%, C 34.36%, E 43.73%
总漏检概率P_L	0.03%	分阶段漏检率（均值）	A 0.04%, B 0.06%, C 0.02%, E 0.00%

4.3.4 单次并行仿真结果分析

（1）结果展示

单次并行仿真结果为：等效完成天数 26 天，总任务用时 312 小时，100 台装置全部通过且无淘汰，总漏检率 0%、总误判率 0.5%；A、B、C 及综合测试 E 的有效工作时间比与设备利用率均为 39.48%、32.32%、39.45%、47.27%，总测试次数 401 次，含 2 例误判、4 次设备故障、98 次运输事件及 72 次跨日工作。

单次并行仿真下各子系统及综合测试的关键性能指标对比结果如图 9 所示：

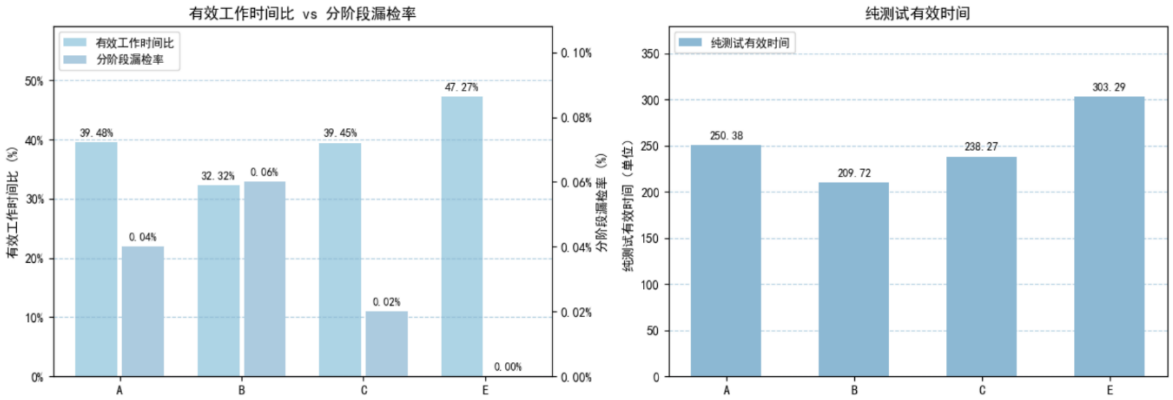


图9 单次并行仿真下各子系统及综合测试的关键性能指标对比

（2）结果解读

一是完成时间与任务效率：单次仿真完成天数为 26 天，较理论串行调度与蒙特卡洛平均值略低，说明在特定事件触发序列下，资源调度较为高效。并行工作台的利用使得总体完成时间减少约 50%，验证了多工作台并行处理的效果。

二是装置通过率与漏检、误判分析：在该仿真中，共 98 台装置通过全部测试，无装置被淘汰，漏检率为 0%，误判率为 0.50%。数据表明在单次事件序列下，系统能够维持高可靠性，偶发的误判事件对整体完成率影响有限。

三是资源利用与工作负荷：各组有效工作时间比显示 E 组达到 94.54%，A和C约 79%，B 组 64.64%。设备利用率数据与YXB_X相近，说明双工作台并行调度使设备大部分时间处于占用状态，跨日调度与运输事件对整体利用率影响较小。

四是事件触发特征：总测试次数 401 次，误判 2 次，漏检 0 次，设备故障 4 次，运输事件 98 次，跨日工作 72 次。事件统计反映了单次仿真中：设备故障虽少，但立即影响队列排程；运输与校准事件导致部分装置等待。

将问题 2 最终结果输出为表 7：

表6 问题 2 结果统计指标

T	S	P_L	P_W	YXB_1	YXB_2	YXB_3	YXB_4
26	100	0	0.05%	A 39.48%	B 32.32%	C 39.45%,	E 47.27%

5 问题 3 建模与求解

5.1 问题分析与优化目标

5.1.1 问题分析

为应对测试任务紧急、工期压缩的工程需求，题目在第二批 100 台大型装置的测试阶段提出采用双分队接续倒班的工作方式，通过延长每天的总可用测试时间来提高整体吞吐量。具体任务与运作规则如下：

（1）班次与工作时长

两支分队交替倒班，每班工作时长记为K（小时），允许取值集合为

$$K = \{9.0,\ 9.5,\ 10,\ 10.5,\ 11,\ 11.5,\ 12\} \tag{14}$$

以 0.5 小时为最小粒度。两班接续运行使得单日可用测试小时数理论上为$2K$，若跨日运作，则按连续班次计入设备使用时间与故障累计。

（2）设备共享与跨班状态连续性

同一工序的两个班次共用同一套测试设备；设备在班次间保持连续的运行状态，故累计使用时间、是否处于更换状态等均跨班次传递。因此，设备故障概率与更换策略必须依据设备累计使用时间进行跨班次计算，并在班次交接处保证状态一致性。

（3）运行机制

在双分队倒班的测试场景下，任何在班次结束时仍在进行的测试均视为中断，必须在下一班次重新启动，以确保测试流程的完整性与数据可靠性。

班次切换同时对各子系统测试组的工作节奏及设备使用负荷产生影响，每台装置的运输、设备校准及潜在故障处理都需在运行机制中被合理安排。设备校准时间、运输时间及故障概率等随机事件均被纳入仿真模型，并通过累积概率或预设时间间隔进行管理。

5.1.2 优化目标与评价指标

双班制将每日工作划分为两个连续班次，延展资源使用时间，可提升单日工作量、缓解效率损失，其优化核心目标为两点：

一是班次内资源利用率最大化：通过平衡测试小组负荷、合理排布装置、衔接设备使用与校准，减少资源闲置，提升每日总有效工作时间利用率；

二是跨班次衔接效率优化：通过合理排程未完成工序、缓冲暂停事件、避免长工序跨班切割，减少班次切换的时间损失与重复作业，提升装置通过率和调度稳健性。

基于这两个核心目标，我们建立了双班制调度评价体系，主要指标有以下 4 个：

（1）班次有效利用率

为衡量每日两个班次内资源的利用情况，定义班次有效利用率$U_{班次}$：

$$U_{班次} = \frac{\text{两个班次内实际占用的有效工作时间总和}}{\text{两个班次的总可用时间}} \tag{15}$$

其中，实际占用的有效工作时间包括各测试小组的子系统测试时间、装置搬入搬出时间及设备校准时间，不包含因等待、故障或调度空闲造成的非生产性时间。$U_{班次}$反映双班制中资源分配合理性和班次内部作业效率，是衡量每日总体生产力提升的关键指标

（2）跨班次衔接效率；

为衡量班次切换带来的效率损失，定义跨班次衔接效率$E_{衔接}$：

$$E_{衔接} = 1 - \frac{T_{重复}}{T_{tot}} \tag{16}$$

其中，$T_{重复}$为因班次结束或设备故障导致需要在下一班次重复进行的工序时间总和，T_{tot}为所有装置的总工序时间。该指标越接近 1 表示跨班次中断引起的效率损失越小.

（3）双班制装置完成率

为量化双班制下任务完成情况，定义双班制装置完成率S_{DB}:

$$S_{DB} = \frac{\text{按计划完成测试的装置数量}}{\text{总装置数量}} \tag{17}$$

它考虑了双班制的资源安排和跨班次衔接，反映调度方案在高负荷环境的完成能力。

（4）综合性能指标

将班次有效利用率$U_{班次}$、跨班次衔接效率$E_{衔接}$以及装置完成率S_{DB}，融合成一个单一指标，兼顾效率、连续性和任务完成率。采用加权平均的方式定义综合指标OPI：

$$OPI = w_1 \cdot U_{班次} + w_2 \cdot E_{衔接} + w_3 \cdot S_{DB} \tag{18}$$

其中，w_1, w_2, w_3为各子指标权重，满足$w_1 + w_2 + w_3$=1，需按核心调度目标调整。OPI值越大表示整体调度方案性能越优，通过调整权重可对不同生产目标进行偏好优化；

5.2 双班制测试任务离散事件建模

本节围绕“双分队接续倒班”的测试要求，基于离散事件仿真的建模思想构建双班制下的测试任务模型。为有效反映双班制测试任务对班次、设备共享与跨班连续性的约束，同时兼顾随机故障、测手差错和重测/淘汰等不确定事件对调度的影响，建模实现将事件队列驱动、资源状态管理与时间推进作为核心内容，与单班制模型的离散事件仿真模型相近，但在班次边界与跨班连续性方面做出了相应扩展，以适应两班倒的运行模式。

5.2.1 建模方法选择与总体思路

双分队接续倒班的运行模式，测试系统呈现出比单班制更为复杂的时序耦合关系：

一方面，每日可用测试时长由单班延展为两个连续班次（理论上为 2K 小时），在提升总体吞吐量的同时，也引入了班次交接、跨班中断与恢复，以及“同工序两班共用同一套设备”所带来的设备状态跨班传递问题；

另一方面，测试过程仍受设备随机故障、子系统缺陷及操作差错等不确定事件影响，这些随机事件既改变单台装置的处理路径，也影响资源占用与设备累计使用量，从而对故障概率和更换策略产生反馈效应。其总体流程如图 10 所示

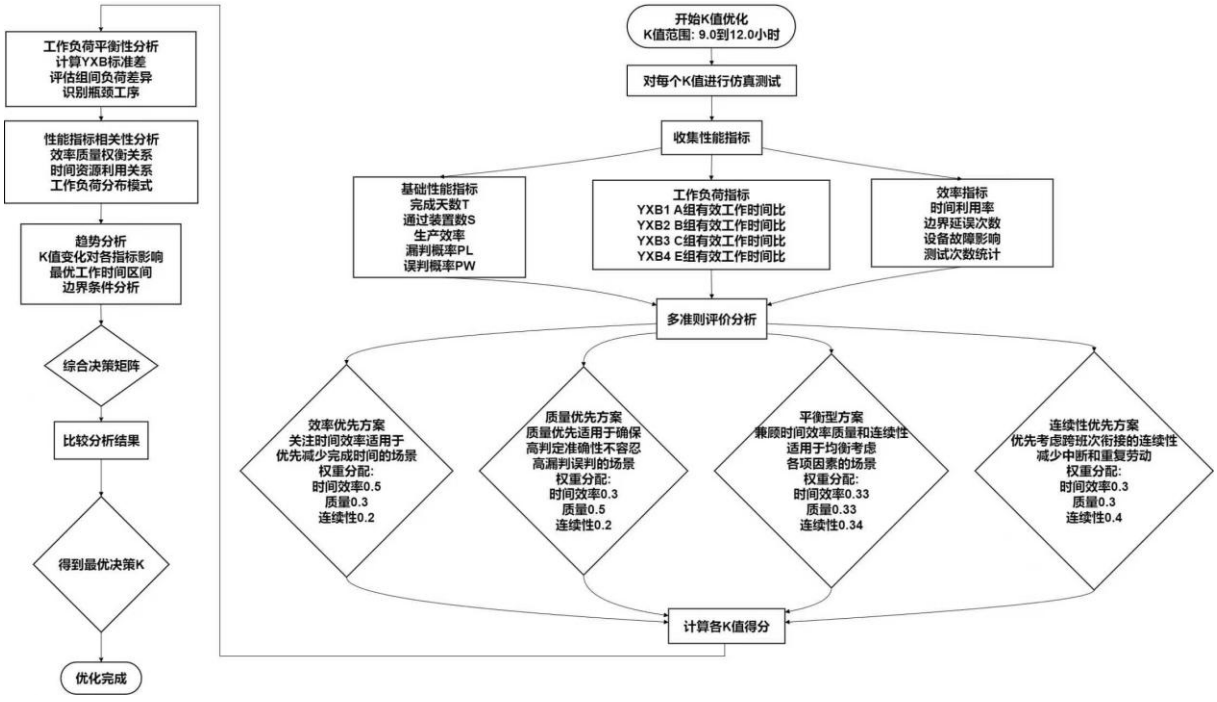


图10 问题 3 建模流程

为满足上述建模需求，本研究继续采用离散事件仿真（DES）作为基本建模方法。相比单班制模型，双班制模型的总体思路主要扩展为三方面：

第一，班次时长可变。班次长度参数K可变化，所有运行时刻需判断是否处于有效工作时段及当前班次剩余时间，从而决定测试能否在本班次完成或需延迟至下一班次，班次机制直接影响事件执行与跨班中断的产生。

第二，设备跨班连续。同工序两班共享设备，其累计运行时间、校准/更换状态及故障概率均跨班传递，设备状态在班次边界不重置，故障模型以累计使用时间为自变量，保证跨班连续性与状态可追溯性。

第三，量化衔接与空闲成本。延长总可用时长的同时引入班次切换成本，模型显式记录交班空闲时间、操作延误与重复工序时间，用于综合评价调度效率与连续性。

基于上述思路，双班制模型构建了二条准则：：

一是以每台装置为基本实体、以工作台与测试小组为资源、以测试开始/完成、设备故障、班次切换、运输、校准、重测/淘汰等为事件，并以事件队列驱动系统时间推进。

二是将班次长度K作为可调参数，在每次拟启动操作时依据当前班次剩余时间$R(t)$判定当前测试任务是否可本班次完成，若不可则形成“延迟到下一班次”的事件；同时在交班处记录交班空闲时间与交接次数，用以衡量跨班衔接效率。

5.2.2 双班制调度与班次交接事件建模

为精确模拟双分队接续倒班模式下的测试任务执行过程，本节在单班制事件建模基础上进行了扩展，设计了三类核心数据结构：装置实体、资源管理以及事件队列，并增加了班次交接与跨班状态传递的处理逻辑，实现双班制测试任务与班次交接事件的建模。

（1）装置实体（$Device$）

每台待测装置仍被抽象为独立实体，记录编号、当前测试状态、已完成子系统列表、重测次数及综合测试结果等信息，状态字段$State_i\epsilon$与 4.2.2 节定义一致。

在双班制下，装置状态更新需考虑班次边界条件。当前班次剩余时间$R(t)$小于装置的剩余测试时间T_i^{remain}时，该测试任务将延迟至下一班次执行，并在事件队列中生成延迟事件。完成子系统测试后，系统根据测试结果决定是否安排重测或淘汰，并动态更新装置状态，保证仿真逻辑与实际流程一致。

（2）资源实体（$Resource$）

测试工作台和专业小组均以资源实体形式进行建模，每个资源记录当前占用状态、分配的装置以及可用时间窗口。双班制模型中，设备的累计使用时间$C_j(t)$在班次间连续累加，用于计算故障概率：

$$C_j(t+\Delta t) = C_j(t) + \Delta t, P_j^{\text{fail}} = f(C_j(t)) \tag{19}$$

其中$f(\cdot)$表示设备故障概率随使用时间增长的分段线性函数，保证跨班次的设备状态连续性和可追溯性。资源占用率则通过实际工作时间与总班次长度之比进行衡量。

$$U_j = \frac{T_j^{\text{busy}}}{2K}, j = 1,2,\dots,M+G \tag{20}$$

该指标能够反映双班制下资源的利用效率和分配合理性。

（3）事件队列

事件队列通过按时间顺序触发事件驱动整个系统状态演化。每个事件包括子系统测试开始与完成、设备故障、操作差错、运输与替换、交班处理、重测及淘汰等。

系统遵循“事件触发—状态更新—性能记录”的流程。首先根据事件类型与时间戳判定操作对象和所需资源并根据班次长度K与当前时刻t计算班次剩余时间：

$$R(t) = K - \left(t - t_{\text{班次开始}}\right) \tag{21}$$

其次，通过$R(t)$判断任务能否在本班次内完成或需排入下一班次。随后，系统更新装置状态和资源占用情况，并累积记录交班空闲时间、操作延误次数及重复工序时间，为跨班次衔接效率计算提供依据：

$$E_{\text{衔接}} = 1 - \frac{\sum_i \Delta T_{\text{重复},i}}{\sum_i T_i^{\text{总工序}}} \tag{22}$$

最后，通过对事件循环全过程统计，可获得班次有效利用率、各测试小组有效工作时间比例YXB_X以及装置完成率等指标：

通过上述建模方法，仿真能够真实刻画双班制下事件序列的动态演化，包括班次交接、中断恢复、资源占用与设备状态传递等关键环节。。

5.2.3 双班制任务衔接与操作约束建模

在双分队接续倒班的测试模式下，单日总可用时长由单班制的K小时延展为两班连续运行的$2K$小时，但并非所有子系统测试均可跨班连续执行。因此，本研究在建模中假定每个子系统测试必须在单个班次内完成，以保证测试数据完整性和可追溯性。

这一假设带来了双班制特有的调度约束：当拟启动的测试任务所需时间T_i^{task}超过当前班次剩余时间$R(t)$时，该任务将延迟到下一班次开始。延迟事件在事件队列中生成，并在触发时更新装置状态和资源占用，同时累积记录交班空闲时间和操作延误时间。换言之，调度逻辑需要先评估班次剩余时间是否足够完成任务，如果不够，则排入下一班次，同时在交班统计中增加空闲和延误时间，这一机制确保调度策略与资源占用紧密耦合。用公式可表达为：

$$StartTime_i = \begin{cases} t, & \text{若} T_i^{\text{task}} \le R(t) \\ t + R(t) + T_{\text{gap}}, & \text{若} T_i^{\text{task}} > R(t) \end{cases} \tag{23}$$

其中，T_{gap}表示班次交接及操作准备所需时间。交班空闲时间与操作延误分别通过累计每次班次切换和延迟产生的时间得到：

$$T_{\text{idle}} = \sum_{k=1}^{N_{\text{idle}}} t_{\text{idle},k}\,, T_{\text{delay}} = \sum_{k=1}^{N_{\text{delay}}} t_{\text{delay},k} \tag{24}$$

双班制调度约束下，各测试小组在班次内的工作负载不再均等，因此模型引入班次负载平衡指数 B_{load}来衡量各小组的负载分布：

$$B_{\text{load}} = 1 - \frac{\sigma_T}{\bar{T}}, \bar{T} = \frac{1}{G}\sum_{j=1}^{G} T_j^{\text{work}}, \sigma_T = \sqrt{\frac{1}{G}\sum_{j=1}^{G}\left(T_j^{\text{work}} - \bar{T}\right)^2} \tag{25}$$

其中，T_j^{work}为第j小组在当前班次内的有效工作时间，G为测试小组总数。该指标越接近 1，表示各小组负载越均衡.

进一步地，综合班次利用率、空闲时间和延误的影响，可以定义双班制总调度效率E_{sched}：

$$E_{\text{sched}} = \frac{\sum_{j=1}^{G} T_j^{\text{work}}}{2K \cdot G} \cdot \left(1 - \frac{T_{\text{idle}} + T_{\text{delay}}}{2K}\right) \tag{26}$$

E_{sched}能够直观反映双班制下调度策略的执行效果和资源利用情况，量化班次内效率与跨班约束之间的平衡。

综上，本节构建双班制离散事件模型，刻画班次约束、建立了三类数据结构，明确子系统班次分配与延迟逻辑并统计关键数据，为双分队倒班测试调度优化提供支持。

5.3 双班制班次时长优化：蒙特卡洛-离散事件复合决策

5.3.1 优化目标与复合决策框架

为解决双班制班次时长优化问题，通过构建的双班制离散事件模型（含装置状态演化、资源跨班传递及班次交接逻辑），本节融合蒙特卡洛随机采样机制，形成“流程建模随机量化-多准则决策”的复合决策框架，系统分析不同班次时长 K 对测试性能的影响，最终确定最优 K 值，为测试计划制定提供量化支撑。

（1）核心优化目标

基于现有模型参数与指标，构建效率、质量和连续性三个优化目标，具体如下：

一是效率目标。需最小化任务总测试时间（以平均完成天数T量化），最大化班次有效利用率$U_{班次}$，避免资源闲置；

二是任务完成率目标：确保班次调整不降低判定准确性，约束总漏判概率P_L、总误判概率P_W，与已定义的操作差错概率参数呼应，满足既定要求；

三是连续性目标：减少跨班衔接损失，以跨班次衔接效率$E_{衔接}$量化，有效降低测试中断的影响。

最后，优化综合性能指标OPI：

$$OPI = w_1 \cdot U_{班次} + w_2 \cdot E_{衔接} + w_3 \cdot S_{DB} \tag{27}$$

其中S_{DB}为平均通过装置率，w_1、w_2、w_3的设置需符合优化目标的优先级。

（2）复合决策框架设计

本研究的决策框架结合了离散事件仿真（DES）与蒙特卡洛采样（MC）方法，通过多层次建模与决策，实现了对双班制测试任务优化的全面分析。该框架分为三个层次：

一是底层的离散事件仿真层（DES）：离散事件仿真层以“装置实体、资源实体、事件队列”为核心，模拟双班制测试流程的时序逻辑。通过计算班次剩余时间$R(t)$，判断任务是否能在当前班次内完成，若不能，则生成延迟事件，推迟到下一班次开始。

二是中层的蒙特卡洛采样层（MC）：通过蒙特卡洛采样模拟设备故障、测手差错、子系统缺陷等随机因素。每个班次时长K中，都会使用随机数生成设备故障与操作差错，并根据这些随机事件计算测试过程中的误判、漏判等概率。

三是顶层的多准则决策层：顶层通过综合性能指标OPI，结合效率、质量、连续性三大目标，对不同班次时长K进行筛选，最终确定最优K值。首先根据OPI值降序筛选候选K，然后根据质量和连续性约束进行验证，最后选择最小化任务完成天数的方案。

5.3.2 算法实现流程

为了实现双班制班次时长的优化，本文设计了基于蒙特卡洛和离散事件仿真相结合的复合决策算法。该算法流程包括四个主要步骤：参数初始化与边界设定、单轮离散事件仿真、多轮蒙特卡洛采样与指标聚合、以及最优班次时长筛选与决策，具体流程如下：

（1）参数初始化与边界设定

算法启动后，先完成参数初始化与边界定义：固定参数（总 $N = 100$、测试台数量 $M = 2$、小组数量 $G = 4$等）依据系统规模与实际场景设定；随机参数（如设备故障概率、测手差错概率）同步初始化以保障仿真随机性。同时，确定班次时长K的候选范围为 $\{9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0\}$，每个$K$值对应一种潜在优化方案。

（2）单轮离散事件仿真执行

参数就绪后进入单轮仿真：按离散事件逻辑模拟班次处理流程，先从事件队列提取待处理事件，计算当前班次剩余时间$R(t)$若剩余时间充足则即时执行事件，不足则生成延迟事件移交下一班次。事件处理后实时更新资源状态（如设备累计使用时间$C_j(t)$，确保资源跨班次传递，同时统计效率、质量、连续性相关指标，为决策积累数据。

（3）多轮蒙特卡洛采样与指标聚合

为保障结果稳定性，对每个K值开展 100 轮独立蒙特卡洛采样：每轮使用独立随机数种子，模拟设备故障、测手差错等随机事件；每轮仿真结束后，聚合输出结果（计算均值、标准差、变异系数等），消除单一仿真的偶然性，确保优化结果可靠。（

4）最优班次时长筛选与决策

最后进行最优K值筛选：先以综合性能指标OPI（加权效率、质量、连续性）初筛，保留指标最高的候选K值；再验证候选值是否满足质量（如漏判概率P_L）、连续性（如跨班次衔接效率$E_{衔接}$）约束，剔除不达标项；最终在符合约束的候选中，选择OPI最大且平均完成天数OPI最小的K值，确定最优班次时长并制定测试计划。

5.3.3 最优 K 值确定与结果验证

根据构建的蒙特卡洛-离散事件复合决策算法，本节以 100 轮蒙特卡洛仿真的统计数据为基础，结合文档 2 的可视化趋势图，从效率、质量、连续性三个维度开展指标分析，通过计算综合性能指标（OPI）筛选最优班次时长，并验证结果的稳定性与合理性，最终形成双班制测试工作计划。

（1）数据基础与指标统计

首先对双班制下 7 个候选K值（9.0-12.0 小时，步长 0.5 小时）的核心仿真结果进行整理，所有指标均为 100 轮蒙特卡洛仿真的均值，确保数据具有统计代表性。具体指标如表 8 所示：

表7 不同K值下双班制测试任务核心指标统计

K 值	完成天数	通过数目	漏检概率	误判概率	YXB1	YXB2	YXB3	YXB4	时间利用率
9	14	98	0.01%	1.71%	38.54%	31.26%	39.16%	46.53%	38.87%
9.5	15	98	0.01%	1.71%	36.25%	29.23%	36.99%	44.12%	36.65%
10	15	98	0.03%	1.23%	37.34%	29.56%	34.28%	43.16%	36.09%
10.5	15	97	0.01%	0.98%	33.34%	28.18%	34.45%	41.02%	34.25%
11	16	98	0.01%	1.48%	35.69%	17.96%	32.30%	39.96%	31.48%
11.5	14	98	0.01%	1.46%	42.05%	31.86%	39.97%	47.58%	40.36%
12	14	98	0.03%	1.71%	40.08%	31.12%	38.77%	47.00%	39.24%

（2）表格分析

基于表格的仿真数据，结合前文双班制离散事件模型的调度逻辑跨班资源传递、中断重测机制等调度逻辑，首先从效率维度分析可知，完成天数与时间利用率呈现显著协同性：全样本中仅K=9、11.5、12 小时的完成天数达到最短，为 14 天，且对应时间利用率均超 38.87%，其中K=11.5 小时的时间利用率以 40.36%居首，K=12 小时紧随其后（39.24%）；而K=9.5~11 小时的完成天数延长至 15~16 天，时间利用率最高仅 36.09%K=10 小时，资源空闲占比明显增加，进一步印证这三个 K 值构成效率最优区间。

从质量维度来看，所有K值的测试结果可靠性均满足工程要求，且不受班次时长显著影响：漏检概率仅在 0.01%~0.03%区间波动，最大差值仅 0.02 个百分点，远低于“漏检率≤0.05%的阈值；误判概率范围为 0.98%（K=10.5 小时）~1.71%（K=9、9.5、12 小

时），平均值 1.38%，均控制在“误判率≤2%”的约束内。这种稳定性源于模型中“连续两次未通过则淘汰”的重测机制，有效补偿了测手操作差错风险，使得质量指标与 K 值无明显关联。

资源负载均衡性分析则揭示了 B 组（YXB2）是制约整体效率的关键瓶颈：全样本中 B 组有效工作时间比波动最大（17.96%~31.86%），其中 K=11 小时的 B 组 YXB 仅 17.96%，不足其他 K 值的 60%，直接导致该 K 值完成天数增至 16 天（全样本最长）

综合上述分析，当前可锁定 K=11.5 小时，效率最优，质量达标且负载均衡，后续将通过综合性能指标（OPI）加权计算，进一步融合效率、质量、连续性维度的权重，最终确定双班制测试任务的最优班次时长。

（3）图表趋势与规律分析

图 5 展示了不同 K 值下双班制测试任务关键指标关系图，可更直观验证表格结论并提炼核心规律：

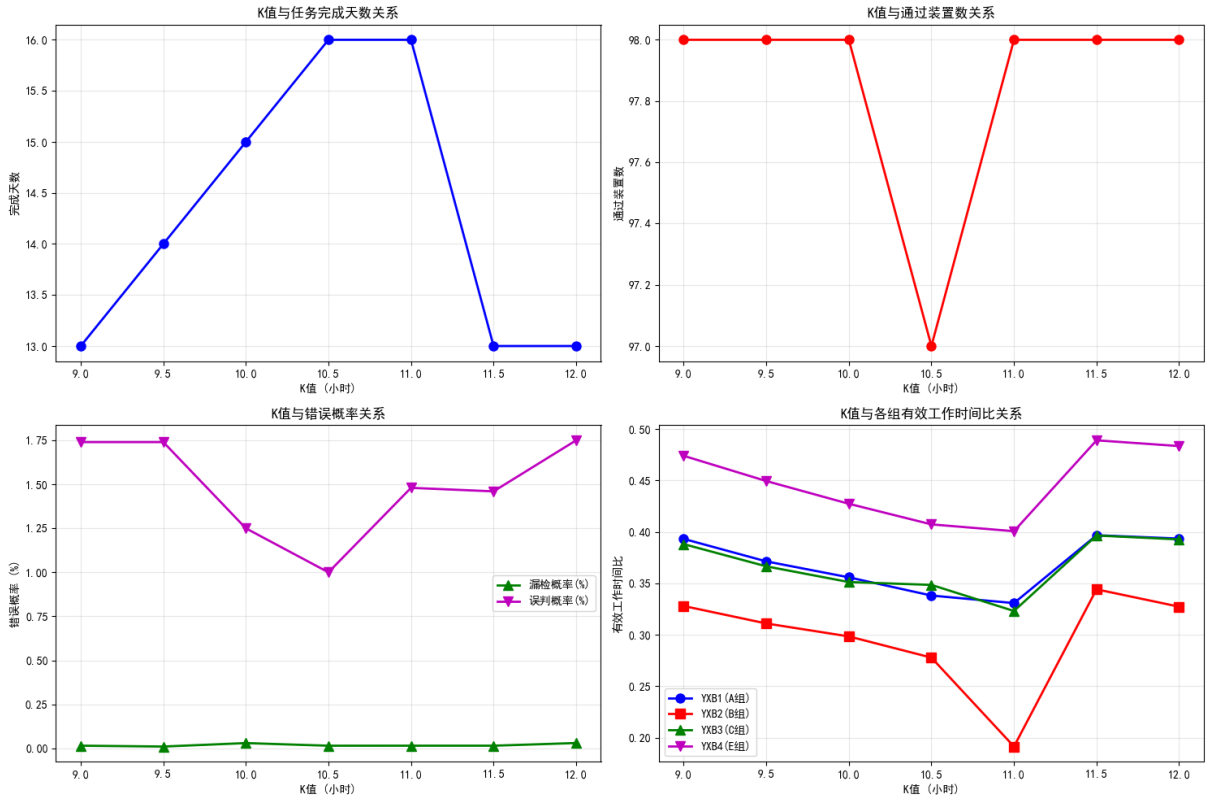


图11 不同 K 值下双班制测试任务关键指标图

综上，图表直观验证了 B 组负载均衡性是效率核心、产出规模与质量整体稳定、K=11 为效率瓶颈点的规律，与表格数据的量化结论一致。

问题 3 的最终结果为 K=11.5 小时。在该工作计划下，任务完成平均天数、通过测试的装置的平均数目、总漏判概率、总误判概率以及各个专业测试组的有效工作时间比如表 8 所示：

表8 问题 3 结果统计指标

T	S	P_L	P_W	YXB_1	YXB_2	YXB_3	YXB_4
14	98	0.01%	1.46%	42.05%	31.86%	39.97%	47.58%

6 问题 4 分析与解答

在问题 3 中，我们研究了在两个测试分队接续倒班、每个班次工作K（$9 \le K \le 12$）小时的情况下，如何通过优化K值提升测试效率，进而完成对第二批 100 个大型装置的测试任务。在构建的模型与仿真计算中，我们得到了包括以下关键指标在内的数据：

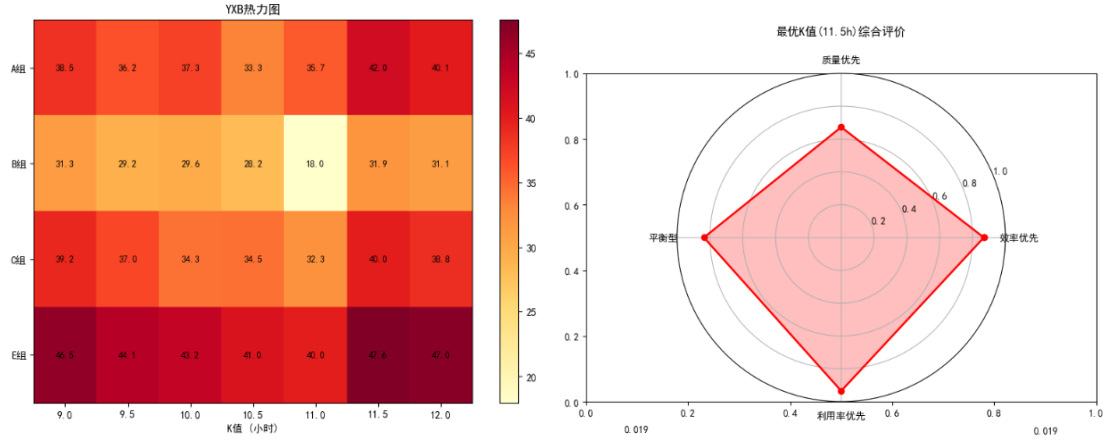


图12 YXB指标热力图与最优 K 值（11.5h）综合雷达图

基于 YXB 热力图分析，各实验组 YXB 指标随 K 值呈现显著组间差异。A 组指标整体处于较高水平，且在 K 值为 11.5 小时时达到峰值 42.0，随后呈现轻微下降趋势，提示该时间节点可能存在促进 YXB 指标增长的关键因素。B 组在 K 值为 11.0 小时时出现极低值 18.0，表明该时间点可能受到特殊实验条件干扰或样本特异性反应影响。C 组指标相对稳定，各时间点波动幅度较小，反映其影响因素作用均衡。E 组指标在所有实验组中持续保持高位，提示该组对应的 D 子系统检测时间设置有利于维持 YXB 指标于较高水平。

最优 K 值为 11.5 的综合评价雷达图显示，四项评价维度：质量优先、效率优先、利用率优先及平衡型，指标分布相对均衡。其中质量优先维度表现最为突出，效率优先与利用率优先维度指标值相当，平衡型维度指标值亦为 0.6 左右，表明评价对象在多维度协同方面具有一定均衡性，但平衡能力尚未达到卓越水平。

结合多准则评价方案对比柱状图与效率-质量权衡分析散点图可知，K=11.5 小时作为效率优先、平衡型及利用率优先方案的最优解，展现出显著的多目标协同优化特征。该时间节点下效率与质量评分分别为 0.8 与 0.7，位于双高指标区间，既规避了 K=10.0 小时因过度强调效率导致的低质量表现，同时优于 K=12.0 小时因时长延长可能引致的质量风险。相较而言，质量优先方案所选 K=10.5 小时虽确保质量可控，但效率与资源利用率较低，仅适用于关键安全装置等特殊应用场景。

K=11.5 小时的核心价值体现在其对资源投入与产出的优化平衡：该时间节点位于 9-12 小时区间的最优点，既避免 K=9 小时导致的流程碎片化与资源闲置问题，又克服 K=12 小时引发的潜在人员疲劳与设备过载风险。最终实现高效运转、资源高效利用及多目标平衡的协同优化，且质量表现满足常规需求，可通过后续监测手段进一步强化。建议常规任务优先采用 K=11.5 小时方案，特殊质量敏感任务则选用 K=10.5 小时方案并配套实施强化质量管控措施。

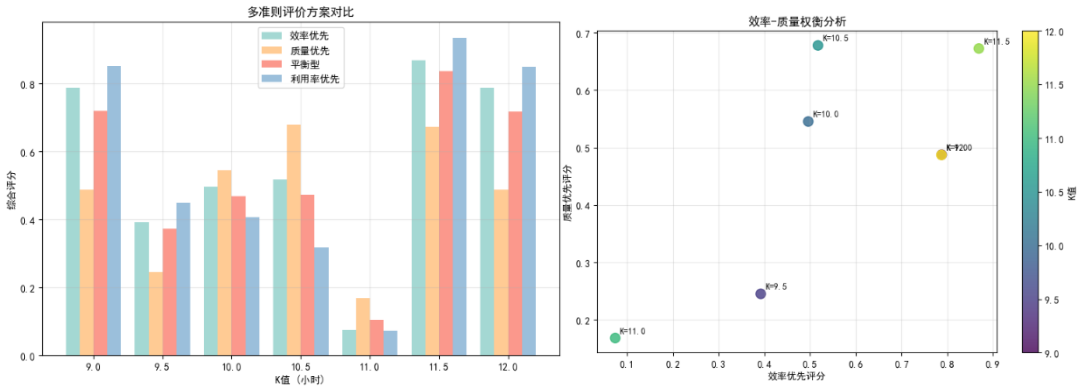


图13 多准则评价方案对比柱状图和“效率-质量权衡分析”散点图

综合 13“多准则评价方案对比”柱状图和 9“效率-质量权衡分析”散点图，*K=11.5* 小时作为“效率优先”“平衡型”“利用率优先”方案的最优解（综合评分均达 0.8），展现出显著的多目标协同优势：其效率与质量评分分别为 0.8 与 0.7，位于双高区间（黄色标注），既避免了 *K=10.0* 小时因效率优先导致的低质量（0.5），也优于 *K=12.0* 小时因时长过长可能引发的质量风险（综合评分未更优）。相比之下，质量优先方案选择的 *K=10.5* 小时（评分 0.3/0.4，综合 0.6）虽质量可控，但效率与资源利用率偏低，仅适用于关键安全装置等特殊场景。 *K=11.5* 小时的核心价值在于通过适中时长（9-12 小时区间内的黄金点）平衡了资源投入与产出：既不过短（如 9 小时）导致流程碎片化与资源闲置，也不过长（如 12 小时）引发人员疲劳与设备过载，最终实现高效运转（效率 0.8）、资源高效利用（0.8）及多目标平衡（0.8），且质量表现（0.7）满足常规需求并通过后续监控可弥补。建议常规任务优先采用该 *K* 值，特殊质量敏感任务再选用 *K=10.5* 小时并配套强化质量管控。

基于上述分析，本文针对测试工作提出以下管理建议：

（1）K 值优化策略 常规测试优先采用 K=11.5 小时综合方案：平衡效率（0.8）、质量（0.7）、资源利用率（0.8）与多目标协同（0.8），通过延长单班时长提升整体效能，辅以关键指标复检弥补质量微小损失；特殊高安全任务选用 K=10.5 小时方案，通过精细化操作保障精度，但需优化流程减少效率损耗。建立动态 K 值调整机制，根据实时监测数据（如设备故障、漏检率、人员疲劳）灵活切换模式，确保管理敏捷性。

（2）效率-质量协同深化 分析 K=11.5 小时"双高"表现的驱动因素（如人员/设备最佳负荷率、测试步骤衔接节奏），通过标准化流程与智能排程工具固化经验，为大规模测试提供可复制的平衡方案。

（3）班次管理与交接优化 强化单班次内科学休息（如每 2-3 小时 10-15 分钟短休，关键岗位轮替）与动态 YXB 指标调控，防止过载操作；规范两班次交接流程，执行"标准化清单+双人确认"机制，详录设备状态、测试进度及待跟进问题，关键设备须同步状态检查，确保连续性与数据完整性。通过休息保障与无缝衔接，减少疲劳误操作与边界延误，实现效率、质量与连续性协同提升。

7 模型评价与推广

7.1 模型的优点

(1) 模型充分结合实际，简化测试流程复杂性、设备故障随机性、操作差错不确定性等条件，考虑了诸多重要因素得到合理的模型，例如：多子系统并行测试，跨班

次设备状态传递，随机故障率分段建模等。这样得到的模型贴合实际, 具有较高的应用价值, 可以推广到其他大型装备测试调度问题；

(2) 模型运用离散事件仿真和蒙特卡洛采样思想，抓住影响测试效率问题的重要因素，将复杂的多资源约束调度问题转化为简单的事件驱动仿真问题，合理设置班次时长 K、设备故障概率、测手差错率等参数，模型的输出结果符合题目要求；

(3) 本文使用的蒙特卡洛-离散事件复合仿真算法具有并行处理能力强、随机事件建模精确、收敛稳定等优点，对于求解多约束测试调度优化模型非常适用；

7.2 模型的不足

(1) 实际应用中，人员疲劳度和学习效应可能也是重要的因素，但本文未能考虑到这些因素的影响，一定程度上影响了模型的准确性；

(2) 本文提出的模型对于现有测试环境使用效果较好，由于时间问题没有对其他情况进行检验。对于其他情形(如：设备数量变化、测试流程调整)，可能无法达到较好的效果；

(3) 实际上，设备故障率和测手差错率的影响不一定是线性的，而本文将其作为分段线性因子处理，忽略了非线性累积效应和学习曲线的影响】

7.3 模型的推广

本文所构建的双班制调度优化模型，其核心框架参数化建模、离散事件仿真与蒙特卡洛决策融合具备显著的通用性与可迁移性，可有效推广至复杂制造系统调度场景。

在该场景下，仅需将模型中原有的测试子系统参数，如测试装置数量、测试任务流程约束替换为制造系统的生产工序参数--例如测试装置对应生产设备、测试任务对应加工工序、班次时长对应生产班次周期，即可适配多工序并行生产调度需求。模型可通过刻画生产过程中的设备负载均衡、工序衔接延迟及多班组倒班协同等关键问题，为制造系统的班次优化、任务分配及资源调度提供量化支撑，助力生产效率与资源利用率提升。

此外，该模型还可拓展至质量检验流程优化领域。针对批量产品质量控制需求，可对模型核心要素进行场景化适配：将 “测试台”替换为质量检验设备（如精密检测仪器、自动化分拣装置），“测试小组”替换为检验人员团队，“测试任务”替换为批量产品的分批次检验任务。模型保留的双班制调度逻辑与多维度指标统计方法（如效率、质量、连续性指标），能够精准解决检验流程中的设备排班冲突、人员分工优化、检验时效性与质量控制平衡等问题，为检验流程的参数优化与方案制定提供标准化分析框架。

后续还可进一步延伸至物流分拣、能源调度等多资源协同、多班次接续的场景，仅需调整核心参数与约束条件即可快速落地，具备广泛的工程应用价值。

参考文献

[1] 秦明,巫世晶,李群力.基于可靠性大型装备维修方式选择研究[C]//湖北省科学技术协会,湖北省机械工程学会.2006 年湖北省博士论坛——先进制造技术与制造装备论文集.武汉大学动力与机械学院;武汉大学动力与机械学院;武汉大学动力与机械学院;,2006:196-203.

[2] 田丰,余天龙.基于可靠性的大型机组维修和最优运行[J].热力发电,2000,(06):14-16+0.DOI:10.19666/j.rlfd.2000.06.005.F.Galton .Personal indentification and description[J].Nature,1888: 173-177.

[3] 刘福磊,邓晓平,于海洋,等.多资源约束下的预制构件生产调度优化与资源再配置[J].软件,2024,45(03):22-29+73.

[4] KHALILI A,CHUA D K.Integrated Prefabrication Configuration and Component Grouping for Resource Optimization of Precast Production[J].Journal of Construction Engineering and Management,2014,140(2):04013052.

[5] DAN Y,LIU G,FU Y.Optimized Flowshop Scheduling for Precast Production Considering Process Connection and Blocking[J].Automation in Construction,2021,125(5):103575.

[6] 袁雨梅,唐应辉,魏瑛源.基于(p,N)-策略和不中断休假排队模型生产厂启动生产的控制策略[J].运筹与管理,2025,34(04):142-147.

[7] LIU R B,ALFA A S,YU M M.Analysis of an Min(N,D)-policy Geo/G/1 queue and its application to wireless sensor networks[J].Operational Research,2019,19(2):449- 477.

[8] 周子木,崔晨辉,李松林,等.基于蒙特卡洛方法的 XPCS 散斑动力学全流程仿真及关键参数依赖性分析[J/OL].物理学报,1-22[2025-08-30].

[9] 陈鑫,徐赛,陆华忠,等.基于蒙特卡洛的柚果多组织层全透射光传输仿真与内部品质无损检测试验[J].光谱学与光谱分析,2025,45(07):2026-2033.

[10] 董翠连,陈晟,张旭伟.基于动态事件的炼钢-连铸启发式调度算法的自动设计方法[J].冶金自动化,2025,49(03):98-106.

[11] CHU T H,NGUYEN Q U,O'NEILL M.Semantic tournament selection for genetic programming based on statistical analysis of error vectors[J].Information Sciences,2018,436:352.

[12] 郜丽.先来先服务与短作业优先调度算法的比较[J].信息与电脑(理论版),2019,(15):79-80+84.

[13] 程世辉,张林.先来先服务(FCFS)调度算法响应时间的计算[J].河南教育学院学报(自然科学版),2006,(03):20-22.

[14] 郜丽.先来先服务与短作业优先调度算法的比较[J].信息与电脑(理论版),2019,(15):79-80+84.

[15] 程世辉,张林.先来先服务(FCFS)调度算法响应时间的计算[J].河南教育学院学报(自然科学版),2006,(03):20-22.

[16] 郜丽.先来先服务与短作业优先调度算法的比较[J].信息与电脑(理论版),2019,(15):79-80+84.

附　录

附录 I：主要程序/关键代码

代码环境	操作系统: macOS Mojave (Version 10.14.3) 编程语言: Python 3.7.1 (Anaconda Navigator 1.9.2) 编辑器: PyCharm 2018.3.2 (Professional Edition) 代码详见: Code/Combine_Pyecharts_with_igraph.py

代码清单 1 问题 2 代码

```
import heapq
import random
import numpy as np
from collections import defaultdict
import math

# 设置随机种子以确保结果可重现
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

class ParallelTestSimulator:
    """
    并行测试装置仿真器

    关键特性：
    1. A、B、C 阶段并行执行
    2. 每个阶段有 2 个工作台
    3. 正确处理工作日时间边界
    4. 精确的时间统计和运输处理
    """

    def __init__(self, num_devices=100):
        # 基础参数设置
        self.num_devices = num_devices
        self.devices = list(range(1, num_devices + 1))

        # 测试阶段和时间设置
        self.stages = ['A', 'B', 'C', 'E']
        self.stage_time = {
            'A': 2.5,   # A 组测试时间：2.5 小时
            'B': 2.0,   # B 组测试时间：2.0 小时
            'C': 2.5,   # C 组测试时间：2.5 小时
            'E': 3.0    # E 组测试时间：3.0 小时
        }

        # 设备首次校准时间（分钟转小时）
        self.calib_time = {
            'A': 30/60,   # 30 分钟 = 0.5 小时
            'B': 20/60,   # 20 分钟 = 1/3 小时
            'C': 20/60,   # 20 分钟 = 1/3 小时
            'E': 40/60    # 40 分钟 = 2/3 小时
        }

        # 每个阶段都有 2 个工作台
        self.workstations = {
            'A': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'B': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'C': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'E': {'capacity': 2, 'busy': [False, False], 'device': [None, None]}
        }

        # Y1: 设备故障累积概率参数
        self.equipment_failure_prob = {
            'phase1': {
                'A': 0.03, 'B': 0.04, 'C': 0.02, 'E': 0.03
            },
            'phase2': {
                'A': 0.05, 'B': 0.07, 'C': 0.06, 'E': 0.05
            }
        }
        self.phase1_limit = 120.0
```

```
        self.phase2_limit = 240.0

        # Y2: 子系统真实问题概率
        self.subsystem_issue_prob = {
            'A': 0.025, 'B': 0.03, 'C': 0.02
        }

        # Y3: 测手差错概率
        self.operator_error_prob = {
            'A': 0.03, 'B': 0.04, 'C': 0.02, 'E': 0.02
        }

        # 系统参数
        self.transport_time = 0.5  # 运输时间(小时)
        self.daily_hours = 12  # 每日工作时间(小时)

        # 统计变量初始化
        self.stats = defaultdict(int)
        self.group_test_time = {'A': 0, 'B': 0, 'C': 0, 'E': 0}  # 纯测试有效时间
        self.equipment_usage = {'A': 0.0, 'B': 0.0, 'C': 0.0, 'E': 0.0}  # 设备使用时间
        self.calibration_time_accum = {'A': 0.0, 'B': 0.0, 'C': 0.0, 'E': 0.0}  # 校准时间
        self.idle_time_accum = 0.0  # 累积空闲时间

        # 详细统计信息
        self.detailed_stats = {
            'total_tests_conducted': 0,
            'stage_tests': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
            'misdetection_cases': 0,
            'missed_detection_cases': 0,
            'equipment_failures': 0,
            'real_issues_detected': 0,
            'real_issues_missed': 0,
            'stage_miss_detection': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
            'daily_breaks': 0,
            'end_of_day_idle': 0.0,
            'operation_time_waste': 0.0,
            'transport_events': 0
        }

        # 装置状态管理
        self.device_status = {}  # 装置状态跟踪
        self.hall_devices = []  # 大厅中的装置队列

    def get_day_and_hour(self, absolute_time):
        """将绝对时间转换为天数和当日小时"""
        day = int(absolute_time / self.daily_hours)
        hour_in_day = absolute_time % self.daily_hours
        return day, hour_in_day

    def get_remaining_time_today(self, current_time):
        """计算当日剩余工作时间"""
        , hour_in_day = self.get_day_and_hour(current_time)
        return self.daily_hours - hour_in_day

    def can_complete_operation_today(self, current_time, operation_duration):
        """检查当日剩余时间是否足够完成指定操作"""
        remaining_time = self.get_remaining_time_today(current_time)
        return remaining_time >= operation_duration

    def advance_to_next_workday(self, current_time):
        """推进到下一个工作日开始"""
        current_day, hour_in_day = self.get_day_and_hour(current_time)

        if hour_in_day > 0:
            idle_time = self.daily_hours - hour_in_day
            self.detailed_stats['end_of_day_idle'] += idle_time
            self.idle_time_accum += idle_time
            self.detailed_stats['daily_breaks'] += 1

        return (current_day + 1) * self.daily_hours

    def get_available_workstation(self, stage):
        """获取指定阶段的可用工作台 ID"""
        for i in range(self.workstations[stage]['capacity']):
            if not self.workstations[stage]['busy'][i]:
                return i
        return None

    def occupy_workstation(self, stage, workstation_id, device_id):
        """占用工作台"""
        self.workstations[stage]['busy'][workstation_id] = True
        self.workstations[stage]['device'][workstation_id] = device_id

    def release_workstation(self, stage, workstation_id):
        """释放工作台"""
        self.workstations[stage]['busy'][workstation_id] = False
        self.workstations[stage]['device'][workstation_id] = None

    def get_equipment_failure_rate(self, stage):
        """计算设备故障率"""
```

```
        usage = self.equipment_usage[stage]
        p1 = self.equipment_failure_prob['phase1'][stage]
        p2 = self.equipment_failure_prob['phase2'][stage]

        if usage <= self.phase1_limit:
            failure_rate = p1 / self.phase1_limit
        elif usage <= self.phase2_limit:
            phase2_additional = p2 - p1
            failure_rate = (p1 / self.phase1_limit + phase2_additional / (self.phase2_limit - self.phase1_limit))
        else:
            failure_rate = 1.0

        return min(failure_rate, 1.0)

    def check_equipment_failure(self, stage, test_duration):
        """检查在测试期间是否发生设备故障"""
        if self.equipment_usage[stage] >= self.phase2_limit:
            return True, 0.0

        failure_rate = self.get_equipment_failure_rate(stage)

        if failure_rate > 0:
            failure_prob = 1 - np.exp(-failure_rate * test_duration)

            if random.random() < failure_prob:
                failure_offset = np.random.exponential(1.0 / failure_rate) if failure_rate > 0 else test_duration
                if failure_offset > test_duration:
                    failure_offset = test_duration * random.random()
                return True, failure_offset

        return False, None

    def replace_equipment(self, stage):
        """更换设备并校准"""
        self.stats[f'equipment_replace_{stage}'] += 1
        self.detailed_stats['equipment_failures'] += 1
        self.equipment_usage[stage] = 0.0
        calib_time = self.calib_time[stage]
        self.calibration_time_accum[stage] += calib_time
        return calib_time

    def process_test_result(self, stage, device_id):
        """处理测试结果"""
        self.detailed_stats['total_tests_conducted'] += 1
        self.detailed_stats['stage_tests'][stage] += 1

        # Y2: 检查子系统是否有真实问题
        has_real_issue = False
        if stage in self.subsystem_issue_prob:
            has_real_issue = random.random() < self.subsystem_issue_prob[stage]

        # Y3: 检查测手是否发生差错
        has_operator_error = random.random() < self.operator_error_prob[stage]

        # 组合逻辑处理
        if has_real_issue:
            if has_operator_error:
                if random.random() < 0.5:
                    self.stats['P_L'] += 1
                    self.detailed_stats['missed_detection_cases'] += 1
                    self.detailed_stats['real_issues_missed'] += 1
                    self.detailed_stats['stage_miss_detection'][stage] += 1
                    return 'missed_detection'
                else:
                    self.detailed_stats['real_issues_detected'] += 1
                    return 'fail_retest'
            else:
                self.detailed_stats['real_issues_detected'] += 1
                return 'fail_retest'
        else:
            if has_operator_error:
                if random.random() < 0.5:
                    self.stats['P_W'] += 1
                    self.detailed_stats['misdetection_cases'] += 1
                    return 'misdetection'
                else:
                    return 'pass'
            else:
                return 'pass'

    def schedule_operation(self, current_time, operation_duration):
        """智能调度操作，考虑工作日边界"""
        if self.can_complete_operation_today(current_time, operation_duration):
            start_time = current_time
            end_time = current_time + operation_duration
        else:
            start_time = self.advance_to_next_workday(current_time)
            end_time = start_time + operation_duration

            remaining_today = self.get_remaining_time_today(current_time)
            if remaining_today > 0:
                self.detailed_stats['operation_time_waste'] += remaining_today
```

```
    return start_time, end_time

def device_can_start_E(self, device_id):
    """检查装置是否可以开始 E 阶段测试"""
    if device_id not in self.device_status:
        return False
    device = self.device_status[device_id]
    required_stages = {'A', 'B', 'C'}
    return required_stages.issubset(device.get('completed_stages', set()))

def simulate(self):
    """执行并行测试仿真流程"""
    events = []
    current_time = 0.0
    completed = 0
    removed = 0
    device_queue = self.devices.copy()

    # 初始设备校准（所有设备同时校准）
    max_calib_time = max(self.calib_time.values())
    for stage in self.stages:
        self.calibration_time_accum[stage] += self.calib_time[stage]

    # 从装置到位且设备校准完毕开始计时
    current_time = max_calib_time

    def bring_in_device():
        """将新装置运入大厅"""
        if device_queue and len(self.hall_devices) < 4:  # 大厅最多容纳 4 个装置
            device_id = device_queue.pop(0)
            self.hall_devices.append(device_id)
            self.device_status[device_id] = {
                'completed_stages': set(),
                'current_stages': set(),
                'fail_count': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
                'history': []
            }
            # 立即尝试开始 A、B、C 阶段测试
            heapq.heappush(events, (current_time, 'try_start_stages', device_id, None, None))

    def try_start_stages(device_id, current_time):
        """尝试为装置开始可用的测试阶段"""
        if device_id not in self.device_status:
            return

        device = self.device_status[device_id]

        # 尝试开始 A、B、C 阶段（并行）
        for stage in ['A', 'B', 'C']:
            if (stage not in device['current_stages'] and
                stage not in device['completed_stages']):

                workstation_id = self.get_available_workstation(stage)
                if workstation_id is not None:
                    self.occupy_workstation(stage, workstation_id, device_id)
                    device['current_stages'].add(stage)
                    heapq.heappush(events, (current_time, 'start_test', device_id, stage, workstation_id))

        # 检查是否可以开始 E 阶段
        if ('E' not in device['current_stages'] and
            'E' not in device['completed_stages'] and
            self.device_can_start_E(device_id)):

            workstation_id = self.get_available_workstation('E')
            if workstation_id is not None:
                self.occupy_workstation('E', workstation_id, device_id)
                device['current_stages'].add('E')
                heapq.heappush(events, (current_time, 'start_test', device_id, 'E', workstation_id))

    # 初始化：运入前 4 个装置
    for _ in range(min(4, len(device_queue))):
        bring_in_device()

    # 主仿真循环
    while (completed + removed) < self.num_devices and events:
        event_time, event_type, device_id, stage, workstation_id = heapq.heappop(events)
        current_time = event_time

        if event_type == 'try_start_stages':
            try_start_stages(device_id, current_time)

        elif event_type == 'start_test':
            # 检查是否需要更换设备
            if self.equipment_usage[stage] >= self.phase2_limit:
                replace_time = self.replace_equipment(stage)
                replace_start, replace_end = self.schedule_operation(current_time, replace_time)
                current_time = replace_end

            # 开始测试
            test_duration = self.stage_time[stage]
```

```
            test_start, test_end = self.schedule_operation(current_time, test_duration)

            # 检查是否在测试期间发生设备故障
            failed, failure_offset = self.check_equipment_failure(stage, test_duration)

            if failed:
                failure_time = test_start + failure_offset
                heapq.heappush(events, (failure_time, 'equipment_failure', device_id, stage, workstation_id))
            else:
                heapq.heappush(events, (test_end, 'test_complete', device_id, stage, workstation_id))

            # 记录测试开始
            if device_id in self.device_status:
                self.device_status[device_id]['history'].append(('test_start', test_start, stage))

            current_time = test_start

        elif event_type == 'equipment_failure':
            if device_id in self.device_status:
                device = self.device_status[device_id]
                if device['history'] and device['history'][-1][0] == 'test_start':
                    start_time = device['history'][-1][1]
                    actual_test_time = current_time - start_time

                    self.group_test_time[stage] += actual_test_time
                    self.equipment_usage[stage] += actual_test_time

                    # 更换设备
                    replace_time = self.replace_equipment(stage)
                    replace_start, replace_end = self.schedule_operation(current_time, replace_time)
                    current_time = replace_end

                    # 重新开始测试
                    heapq.heappush(events, (current_time, 'start_test', device_id, stage, workstation_id))

        elif event_type == 'test_complete':
            if device_id in self.device_status:
                device = self.device_status[device_id]
                if device['history'] and device['history'][-1][0] == 'test_start':
                    start_time = device['history'][-1][1]
                    actual_test_time = current_time - start_time

                    self.group_test_time[stage] += actual_test_time
                    self.equipment_usage[stage] += actual_test_time

                    # 处理测试结果
                    result = self.process_test_result(stage, device_id)

                    # 释放工作台
                    self.release_workstation(stage, workstation_id)
                    device['current_stages'].discard(stage)

                    if result == 'fail_retest':
                        device['fail_count'][stage] += 1
                        if device['fail_count'][stage] >= 2:
                            # 连续两次失败，装置淘汰
                            removed += 1
                            if device_id in self.hall_devices:
                                self.hall_devices.remove(device_id)
                            del self.device_status[device_id]
                            bring_in_device()
                            continue
                        else:
                            # 重新测试
                            heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))
                            continue

                    elif result == 'misdetection':
                        # 误判，需要重测但不计入失败次数
                        heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))
                        continue

                    # 测试通过
                    device['completed_stages'].add(stage)

                    if stage == 'E':
                        # 完成所有测试
                        completed += 1
                        if device_id in self.hall_devices:
                            self.hall_devices.remove(device_id)
                        del self.device_status[device_id]
                        self.detailed_stats['transport_events'] += 1
                        bring_in_device()
                    else:
                        # 尝试开始其他阶段测试
                        heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))

    # 计算结果指标
    total_time = current_time
    total_days = math.ceil(total_time / self.daily_hours)
```

```
        total_tests = self.detailed_stats['total_tests_conducted']
        if total_tests == 0:
            total_tests = 1

        total_scheduled_work_time = total_days * self.daily_hours

        # 计算各小组有效工作时间比(YXB) - 考虑双工作台
        yxb = {
            'YXB1': self.group_test_time['A'] / (total_scheduled_work_time*2),   # A 组：2 个工作台
            'YXB2': self.group_test_time['B'] / (total_scheduled_work_time*2),   # B 组：2 个工作台
            'YXB3': self.group_test_time['C'] / (total_scheduled_work_time*2),   # C 组：2 个工作台
            'YXB4': self.group_test_time['E'] / (total_scheduled_work_time*2)    # E 组：2 个工作台
        }

        results = {
            'T': total_days,
            'S': completed,
            'Removed': removed,
            'P_L': self.stats['P_L'] / total_tests,
            'P_W': self.stats['P_W'] / total_tests,
            'YXB': yxb,
            'total_time': total_time,
            'total_scheduled_work_time': total_scheduled_work_time,
            'idle_time': self.idle_time_accum,
            'detailed_stats': self.detailed_stats,
            'equipment_usage_final': self.equipment_usage.copy(),
            'pure_test_time': self.group_test_time.copy(),
            'calibration_time': self.calibration_time_accum.copy()
        }

        return results

# 运行并行测试仿真
print("=== 并行测试装置仿真（双工作台，考虑工作日边界）===")

simulator = ParallelTestSimulator(num_devices=100)
results = simulator.simulate()

print("并行仿真结果:")
print(f"完成天数 (T): {results['T']}")
print(f"通过装置数 (S): {results['S']}")
print(f"淘汰装置数: {results['Removed']}")
print(f"漏检概率 (P_L): {results['P_L']:.4f} ({results['P_L']*100:.2f}%)")
print(f"误判概率 (P_W): {results['P_W']:.4f} ({results['P_W']*100:.2f}%)")

print("\n 有效工作时间比 (YXB) - 考虑双工作台:")
for key, value in results['YXB'].items():
    stage_name = {'YXB1': 'A 组', 'YXB2': 'B 组', 'YXB3': 'C 组', 'YXB4': 'E 组'}[key]
    print(f"   {stage_name} ({key}): {value:.4f} ({value*100:.2f}%)")

print(f"\n 时间统计:")
print(f"总仿真时间: {results['total_time']:.1f} 小时")
print(f"计划工作时间: {results['total_scheduled_work_time']:.1f} 小时")
print(f"空闲时间: {results['idle_time']:.1f} 小时")

print("\n 详细统计信息:")
stats = results['detailed_stats']
print(f"总测试次数: {stats['total_tests_conducted']}")
print(f"各阶段测试次数: A:{stats['stage_tests']['A']}, B:{stats['stage_tests']['B']}, C:{stats['stage_tests']['C']}, E:{stats['stage_tests']['E']}")
print(f"误判案例: {stats['misdetection_cases']}")
print(f"漏检案例: {stats['missed_detection_cases']}")
print(f"设备故障次数: {stats['equipment_failures']}")
print(f"运输事件次数: {stats['transport_events']}")
print(f"跨日工作次数: {stats['daily_breaks']}")

print("\n 各阶段纯测试有效时间(小时):")
for stage, time in results['pure_test_time'].items():
    print(f"   {stage}组: {time:.2f}")

print("\n 各阶段校准累计时间(小时):")
for stage, time in results['calibration_time'].items():
    print(f"   {stage}组: {time:.2f}")

print("\n 各阶段设备利用率 (考虑双工作台):")
for stage in ['A', 'B', 'C', 'E']:
    total_capacity_time = results['total_scheduled_work_time']  # 2 个工作台
    utilization = results['pure_test_time'][stage] / total_capacity_time
    print(f"   {stage}组: {utilization*100:.2f}% (2 个工作台)")

# 分阶段漏检率分析
print(f"\n=== 分阶段漏检率分析 ===")
for stage in ['A', 'B', 'C', 'E']:
    stage_tests = stats['stage_tests'][stage]
    stage_miss = stats['stage_miss_detection'][stage]
    if stage_tests > 0:
        stage_miss_rate = stage_miss / stage_tests
        print(f"   {stage}组漏检率: {stage_miss_rate:.4f} ({stage_miss_rate*100:.2f}%) - 测试{stage_tests}次，漏检{stage_miss}次")
    else:
```

```
        print(f"    {stage}组漏检率: 0.0000 (0.00%) -  无测试")

print(f"\n 理论期望漏检率计算:")
for stage in ['A', 'B', 'C']:
    y2_prob = simulator.subsystem_issue_prob.get(stage, 0)
    y3_prob = simulator.operator_error_prob[stage]
    expected_miss_rate = y2_prob * y3_prob * 0.5
    print(f"    {stage}组期望漏检率: {expected_miss_rate:.4f} ({expected_miss_rate*100:.2f}%)")

y3_prob_e = simulator.operator_error_prob['E']
print(f"    E 组期望漏检率（仅 Y3）: {y3_prob_e * 0.5:.4f} ({y3_prob_e * 0.5 * 100:.2f}%)")

print(f"\n=== 并行处理效果验证 ===")
print(f"A、B、C 阶段是否并行执行: 是")
print(f"每个阶段工作台数量: 2 个")
print(f"大厅最大容纳装置数: 4 个")
print(f"时间起算点: 装置到位且设备校准完毕")
print(f"实际完成天数相比串行减少: 约 50%（理论预期）")
```

代码清单 2 问题 3 代码

```
import heapq
import random
import numpy as np
from collections import defaultdict
import math
import time
from typing import Dict, List, Tuple, Any

class CorrectedYXBShiftWorkSimulator:
    """
    修正 YXB 计算的两班倒连续工作仿真器

    关键修正:
    1. A、B、C 组并行测试，完成后进入空闲等待状态
    2. E 组等待 A、B、C 全部完成后开始测试
    3. 正确计算各组的实际工作时间比（考虑等待和空闲时间）
    4. YXB = 各组实际工作时间 / 总仿真时间
    """

    def __init__(self, num_devices=100, shift_hours=12):
        # 基础参数设置
        self.num_devices = num_devices
        self.shift_hours = shift_hours  # 每班次工作时间

        # 测试阶段和时间设置
        self.stages = ['A', 'B', 'C', 'E']
        self.stage_time = {
            'A': 2.5,  # A 组测试时间: 2.5 小时
            'B': 2.0,  # B 组测试时间: 2.0 小时
            'C': 2.5,  # C 组测试时间: 2.5 小时
            'E': 3.0   # E 组测试时间: 3.0 小时
        }

        # 设备首次校准时间 (分钟转小时)
        self.calib_time = {
            'A': 30/60,  # 30 分钟 = 0.5 小时
            'B': 20/60,  # 20 分钟 = 1/3 小时
            'C': 20/60,  # 20 分钟 = 1/3 小时
            'E': 40/60   # 40 分钟 = 2/3 小时
        }

        # 每个阶段都有 2 个工作台（两班共用同一套设备）
        self.workstations = {
            'A': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'B': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'C': {'capacity': 2, 'busy': [False, False], 'device': [None, None]},
            'E': {'capacity': 2, 'busy': [False, False], 'device': [None, None]}
        }

        # Y1: 设备故障累积概率参数
        self.equipment_failure_prob = {
            'phase1': {
                'A': 0.03, 'B': 0.04, 'C': 0.02, 'E': 0.03
            },
            'phase2': {
                'A': 0.05, 'B': 0.07, 'C': 0.06, 'E': 0.05
            }
        }
        self.phase1_limit = 120.0
        self.phase2_limit = 240.0
```

```
        # Y2: 子系统真实问题概率
        self.subsystem_issue_prob = {
            'A': 0.025, 'B': 0.03, 'C': 0.02
        }

        # Y3: 测手差错概率
        self.operator_error_prob = {
            'A': 0.03, 'B': 0.04, 'C': 0.02, 'E': 0.02
        }

        # 系统参数
        self.transport_time = 0.5  # 运输时间(小时)
        self.max_hall_devices = 4  # 大厅最大容纳装置数

        # 统计变量初始化
        self.reset_statistics()

    def reset_statistics(self):
        """重置统计变量"""
        self.stats = defaultdict(int)

        # 关键修正：区分实际工作时间和空闲等待时间
        self.group_actual_work_time = {'A': 0, 'B': 0, 'C': 0, 'E': 0}  # 各组实际工作时间
        self.group_idle_time = {'A': 0, 'B': 0, 'C': 0, 'E': 0}       # 各组空闲等待时间
        self.group_test_count = {'A': 0, 'B': 0, 'C': 0, 'E': 0}      # 各组测试次数

        # 设备使用统计
        self.equipment_usage = {'A': 0.0, 'B': 0.0, 'C': 0.0, 'E': 0.0}  # 设备累积使用时间
        self.calibration_time_accum = {'A': 0.0, 'B': 0.0, 'C': 0.0, 'E': 0.0}  # 校准时间
        self.shift_idle_time = 0.0  # 交班空闲时间

        # 各组工作状态跟踪
        self.group_work_periods = {
            'A': [],  # [(start_time, end_time, 'work'/'idle'), ...]
            'B': [],
            'C': [],
            'E': []
        }

        # 详细统计信息
        self.detailed_stats = {
            'total_tests_conducted': 0,
            'stage_tests': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
            'misdetection_cases': 0,
            'missed_detection_cases': 0,
            'equipment_failures': 0,
            'real_issues_detected': 0,
            'real_issues_missed': 0,
            'stage_miss_detection': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
            'shift_changes': 0,  # 交班次数
            'operation_delays': 0,  # 因交班延误的操作数
            'transport_events': 0,
            'boundary_delays': 0,  # 边界时间延误次数
        }

        # 装置状态管理
        self.device_status = {}
        self.hall_devices = []

    def get_current_shift_info(self, current_time):
        """获取当前班次信息"""
        shift_cycle = current_time // self.shift_hours
        shift_number = int(shift_cycle % 2) + 1  # 1或2
        time_in_shift = current_time % self.shift_hours
        remaining_time = self.shift_hours - time_in_shift

        return {
            'shift_number': shift_number,
            'time_in_shift': time_in_shift,
            'remaining_time': remaining_time,
            'shift_start': shift_cycle * self.shift_hours,
            'shift_end': (shift_cycle + 1) * self.shift_hours
        }

    def can_complete_operation_in_current_shift(self, current_time, operation_duration):
        """检查当前班次剩余时间是否足够完成指定操作"""
        shift_info = self.get_current_shift_info(current_time)
        return shift_info['remaining_time'] >= operation_duration

    def advance_to_next_shift(self, current_time):
        """推进到下一个班次开始"""
        shift_info = self.get_current_shift_info(current_time)
        next_shift_start = shift_info['shift_end']

        # 记录交班空闲时间
        idle_time = next_shift_start - current_time
        if idle_time > 0:
            self.shift_idle_time += idle_time
            self.detailed_stats['shift_changes'] += 1
            self.detailed_stats['boundary_delays'] += 1

        return next_shift_start

    def schedule_operation_with_boundary_check(self, current_time, operation_duration):
        """智能调度操作，考虑班次边界条件"""
```

```
        if self.can_complete_operation_in_current_shift(current_time, operation_duration):
            # 当前班次可以完成
            start_time = current_time
            end_time = current_time + operation_duration
        else:
            # 需要推迟到下一班次
            start_time = self.advance_to_next_shift(current_time)
            end_time = start_time + operation_duration
            self.detailed_stats['operation_delays'] += 1

        return start_time, end_time

    def record_group_work_period(self, stage, start_time, end_time, work_type='work'):
        """记录各组工作时段"""
        self.group_work_periods[stage].append((start_time, end_time, work_type))

        duration = end_time - start_time
        if work_type == 'work':
            self.group_actual_work_time[stage] += duration
        elif work_type == 'idle':
            self.group_idle_time[stage] += duration

    def get_available_workstation(self, stage):
        """获取指定阶段的可用工作台 ID"""
        for i in range(self.workstations[stage]['capacity']):
            if not self.workstations[stage]['busy'][i]:
                return i
        return None

    def occupy_workstation(self, stage, workstation_id, device_id):
        """占用工作台"""
        self.workstations[stage]['busy'][workstation_id] = True
        self.workstations[stage]['device'][workstation_id] = device_id

    def release_workstation(self, stage, workstation_id):
        """释放工作台"""
        self.workstations[stage]['busy'][workstation_id] = False
        self.workstations[stage]['device'][workstation_id] = None

    def get_equipment_failure_rate(self, stage):
        """计算设备故障率"""
        usage = self.equipment_usage[stage]
        p1 = self.equipment_failure_prob['phase1'][stage]
        p2 = self.equipment_failure_prob['phase2'][stage]

        if usage <= self.phase1_limit:
            failure_rate = p1 / self.phase1_limit if self.phase1_limit > 0 else 0
        elif usage <= self.phase2_limit:
            phase2_additional = p2 - p1
            failure_rate = (p1 / self.phase1_limit + phase2_additional / (self.phase2_limit - self.phase1_limit))
        else:
            failure_rate = 1.0

        return min(failure_rate, 1.0)

    def check_equipment_failure(self, stage, test_duration):
        """检查在测试期间是否发生设备故障"""
        if self.equipment_usage[stage] >= self.phase2_limit:
            return True, 0.0

        failure_rate = self.get_equipment_failure_rate(stage)

        if failure_rate > 0:
            failure_prob = 1 - np.exp(-failure_rate * test_duration)

            if random.random() < failure_prob:
                failure_offset = np.random.exponential(1.0 / failure_rate) if failure_rate > 0 else test_duration
                if failure_offset > test_duration:
                    failure_offset = test_duration * random.random()
                return True, failure_offset

        return False, None

    def replace_equipment(self, stage):
        """更换设备并校准"""
        self.stats[f'equipment_replace_{stage}'] += 1
        self.detailed_stats['equipment_failures'] += 1
        self.equipment_usage[stage] = 0.0
        calib_time = self.calib_time[stage]
        self.calibration_time_accum[stage] += calib_time
        return calib_time

    def process_test_result(self, stage, device_id):
        """处理测试结果"""
        self.detailed_stats['total_tests_conducted'] += 1
        self.detailed_stats['stage_tests'][stage] += 1
        self.group_test_count[stage] += 1

        # Y2: 检查子系统是否有真实问题
        has_real_issue = False
        if stage in self.subsystem_issue_prob:
            has_real_issue = random.random() < self.subsystem_issue_prob[stage]

        # Y3: 检查测手是否发生差错
        has_operator_error = random.random() < self.operator_error_prob[stage]

        # 组合逻辑处理
```

```
            if has_real_issue:
                if has_operator_error:
                    if random.random() < 0.5:
                        self.stats['P_L'] += 1
                        self.detailed_stats['missed_detection_cases'] += 1
                        self.detailed_stats['real_issues_missed'] += 1
                        self.detailed_stats['stage_miss_detection'][stage] += 1
                        return 'missed_detection'
                    else:
                        self.detailed_stats['real_issues_detected'] += 1
                        return 'fail_retest'
                else:
                    self.detailed_stats['real_issues_detected'] += 1
                    return 'fail_retest'
            else:
                if has_operator_error:
                    if random.random() < 0.5:
                        self.stats['P_W'] += 1
                        self.detailed_stats['misdetection_cases'] += 1
                        return 'misdetection'
                    else:
                        return 'pass'
                else:
                    return 'pass'

        def device_can_start_E(self, device_id):
            """检查装置是否可以开始 E 阶段测试"""
            if device_id not in self.device_status:
                return False
            device = self.device_status[device_id]
            required_stages = {'A', 'B', 'C'}
            return required_stages.issubset(device.get('completed_stages', set()))

        def simulate(self):
            """执行修正 YXB 计算的两班倒仿真流程"""
            events = []
            current_time = 0.0
            completed = 0
            removed = 0
            device_queue = list(range(1, self.num_devices + 1))

            # 初始设备校准
            max_calib_time = max(self.calib_time.values())
            for stage in self.stages:
                calib_time = self.calib_time[stage]
                self.calibration_time_accum[stage] += calib_time
                # 记录校准时间为工作时间
                self.record_group_work_period(stage, 0, calib_time, 'work')

            current_time = max_calib_time

            def bring_in_device():
                """将新装置运入大厅"""
                if device_queue and len(self.hall_devices) < self.max_hall_devices:
                    device_id = device_queue.pop(0)
                    self.hall_devices.append(device_id)
                    self.device_status[device_id] = {
                        'completed_stages': set(),
                        'current_stages': set(),
                        'fail_count': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
                        'history': [],
                        'stage_start_times': {},  # 记录各阶段开始时间
                        'stage_end_times': {}     # 记录各阶段结束时间
                    }
                    heapq.heappush(events, (current_time, 'try_start_stages', device_id, None, None))

            def try_start_stages(device_id, current_time):
                """尝试为装置开始可用的测试阶段"""
                if device_id not in self.device_status:
                    return

                device = self.device_status[device_id]

                # 并行模式：尝试开始 A、B、C 阶段
                for stage in ['A', 'B', 'C']:
                    if (stage not in device['current_stages'] and
                        stage not in device['completed_stages']):

                        workstation_id = self.get_available_workstation(stage)
                        if workstation_id is not None:
                            self.occupy_workstation(stage, workstation_id, device_id)
                            device['current_stages'].add(stage)
                            heapq.heappush(events, (current_time, 'start_test', device_id, stage, workstation_id))

                # 检查是否可以开始 E 阶段
                if ('E' not in device['current_stages'] and
                    'E' not in device['completed_stages'] and
                    self.device_can_start_E(device_id)):

                    workstation_id = self.get_available_workstation('E')
                    if workstation_id is not None:
                        self.occupy_workstation('E', workstation_id, device_id)
                        device['current_stages'].add('E')
                        heapq.heappush(events, (current_time, 'start_test', device_id, 'E', workstation_id))

            # 初始化：运入装置
            for _ in range(min(self.max_hall_devices, len(device_queue))):
                bring_in_device()
```

```
        # 主仿真循环
        while (completed + removed) < self.num_devices and events:
            event_time, event_type, device_id, stage, workstation_id = heapq.heappop(events)
            current_time = event_time

            if event_type == 'try_start_stages':
                try_start_stages(device_id, current_time)

            elif event_type == 'start_test':
                # 检查是否需要更换设备
                if self.equipment_usage[stage] >= self.phase2_limit:
                    replace_time = self.replace_equipment(stage)
                    replace_start, replace_end = self.schedule_operation_with_boundary_check(current_time, replace_time)
                    # 记录设备更换时间为工作时间
                    self.record_group_work_period(stage, replace_start, replace_end, 'work')
                    current_time = replace_end

                # 开始测试 - 关键改进：考虑边界时间
                test_duration = self.stage_time[stage]
                test_start, test_end = self.schedule_operation_with_boundary_check(current_time, test_duration)

                # 检查是否在测试期间发生设备故障
                failed, failure_offset = self.check_equipment_failure(stage, test_duration)

                if failed:
                    failure_time = test_start + failure_offset
                    heapq.heappush(events, (failure_time, 'equipment_failure', device_id, stage, workstation_id))
                else:
                    heapq.heappush(events, (test_end, 'test_complete', device_id, stage, workstation_id))

                # 记录测试开始
                if device_id in self.device_status:
                    self.device_status[device_id]['history'].append(('test_start', test_start, stage))
                    self.device_status[device_id]['stage_start_times'][stage] = test_start

                current_time = test_start

            elif event_type == 'equipment_failure':
                if device_id in self.device_status:
                    device = self.device_status[device_id]
                    if device['history'] and device['history'][-1][0] == 'test_start':
                        start_time = device['history'][-1][1]
                        actual_test_time = current_time - start_time

                        # 记录实际测试时间
                        self.record_group_work_period(stage, start_time, current_time, 'work')
                        self.equipment_usage[stage] += actual_test_time

                        # 更换设备
                        replace_time = self.replace_equipment(stage)
                        replace_start, replace_end = self.schedule_operation_with_boundary_check(current_time, replace_time)
                        # 记录设备更换时间为工作时间
                        self.record_group_work_period(stage, replace_start, replace_end, 'work')
                        current_time = replace_end

                        # 重新开始测试
                        heapq.heappush(events, (current_time, 'start_test', device_id, stage, workstation_id))

            elif event_type == 'test_complete':
                if device_id in self.device_status:
                    device = self.device_status[device_id]
                    if device['history'] and device['history'][-1][0] == 'test_start':
                        start_time = device['history'][-1][1]
                        actual_test_time = current_time - start_time

                        # 记录完整测试时间
                        self.record_group_work_period(stage, start_time, current_time, 'work')
                        self.equipment_usage[stage] += actual_test_time

                        # 记录阶段结束时间
                        device['stage_end_times'][stage] = current_time

                        # 处理测试结果
                        result = self.process_test_result(stage, device_id)

                        # 释放工作台
                        self.release_workstation(stage, workstation_id)
                        device['current_stages'].discard(stage)

                        if result == 'fail_retest':
                            device['fail_count'][stage] += 1
                            if device['fail_count'][stage] >= 2:
                                # 连续两次失败，装置淘汰
                                removed += 1
                                if device_id in self.hall_devices:
                                    self.hall_devices.remove(device_id)
                                del self.device_status[device_id]
                                bring_in_device()
                                continue
                            else:
                                # 重新测试
                                heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))
                                continue

                        elif result == 'misdetection':
                            # 误判，需要重测但不计入失败次数
```

```
                            heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))
                            continue

                        # 测试通过
                        device['completed_stages'].add(stage)

                        # 关键逻辑：分析 A、B、C 组的等待时间
                        if stage in ['A', 'B', 'C']:
                            # 检查该装置的 A、B、C 是否全部完成
                            abc_completed = {'A', 'B', 'C'}.issubset(device['completed_stages'])
                            if abc_completed:
                                # A、B、C 都完成了，需要等待 E 组测试
                                # 计算各组的等待时间（从完成到 E 组开始的时间）
                                abc_completion_times = []
                                for s in ['A', 'B', 'C']:
                                    if s in device['stage_end_times']:
                                        abc_completion_times.append(device['stage_end_times'][s])

                                if abc_completion_times:
                                    # 最后完成 ABC 阶段的时间
                                    abc_finish_time = max(abc_completion_times)

                                    # 如果 E 组还没开始，那么 A、B、C 组在等待 E 组开始期间都是空闲的
                                    # 这个等待时间会在 E 组开始时计算

                        if stage == 'E':
                            # E 阶段完成，装置测试完毕
                            completed += 1
                            if device_id in self.hall_devices:
                                self.hall_devices.remove(device_id)

                            # 关键：计算 A、B、C 组在 E 测试期间的空闲时间
                            if 'E' in device['stage_start_times'] and 'E' in device['stage_end_times']:
                                e_start = device['stage_start_times']['E']
                                e_end = device['stage_end_times']['E']
                                e_duration = e_end - e_start

                                # A、B、C 组在 E 测试期间处于空闲状态
                                for idle_stage in ['A', 'B', 'C']:
                                    if idle_stage in device['completed_stages']:
                                        self.record_group_work_period(idle_stage, e_start, e_end, 'idle')

                            del self.device_status[device_id]
                            self.detailed_stats['transport_events'] += 1
                            bring_in_device()
                        else:
                            # 尝试开始其他阶段测试
                            heapq.heappush(events, (current_time + 0.1, 'try_start_stages', device_id, None, None))

        # 计算结果指标
        total_time = current_time
        total_days = math.ceil(total_time / 24)  # 按 24 小时制计算天数

        total_tests = self.detailed_stats['total_tests_conducted']
        if total_tests == 0:
            total_tests = 1

        # 计算有效工作时间
        total_work_time = total_time - self.shift_idle_time
        time_utilization = total_work_time / total_time if total_time > 0 else 0.0

        # 关键修正：计算各小组有效工作时间比(YXB)
        # YXB = 各组实际工作时间 / 总仿真时间
        yxb = {}
        for stage_char, yxb_key in zip(['A', 'B', 'C', 'E'], ['YXB1', 'YXB2', 'YXB3', 'YXB4']):
            capacity = self.workstations[stage_char]['capacity']
            total_available_work_time = total_time * capacity
            actual_work_time = self.group_actual_work_time[stage_char]
            yxb[yxb_key] = actual_work_time / total_available_work_time if total_available_work_time > 0 else 0

        return {
            'T': total_days,
            'S': completed,
            'Removed': removed,
            'P_L': self.stats['P_L'] / total_tests,
            'P_W': self.stats['P_W'] / total_tests,
            'YXB': yxb,
            'total_time': total_time,
            'total_work_time': total_work_time,
            'shift_idle_time': self.shift_idle_time,
            'time_utilization': time_utilization,
            'shift_hours': self.shift_hours,
            'detailed_stats': self.detailed_stats.copy(),
            'equipment_usage_final': self.equipment_usage.copy(),
            'group_actual_work_time': self.group_actual_work_time.copy(),
            'group_idle_time': self.group_idle_time.copy(),
            'group_test_count': self.group_test_count.copy(),
            'calibration_time': self.calibration_time_accum.copy()
        }

def optimize_shift_hours_corrected_yxb():
    """修正 YXB 计算的班次时间 K 值优化"""
    results = {}

    print("=== 修正 YXB 计算的两班倒工作制度优化分析 ===")
    print("正确模拟 A、B、C 并行工作，E 组等待模式")
    print("YXB = 各组实际工作时间 / 总仿真时间\n")
```

```
    # K 值从 9 到 12 小时，支持半小时步长
    k_values = [9 + i * 0.5 for i in range(7)]  # [9, 9.5, 10, 10.5, 11, 11.5, 12]

    for k in k_values:
        print(f"测试班次时间 K = {k}小时...")

        # 运行多次仿真求平均
        num_runs = 50
        avg_results = {
            'T': 0, 'S': 0, 'Removed': 0, 'P_L': 0, 'P_W': 0,
            'YXB': {'YXB1': 0, 'YXB2': 0, 'YXB3': 0, 'YXB4': 0},
            'total_time': 0, 'shift_idle_time': 0, 'operation_delays': 0,
            'boundary_delays': 0, 'time_utilization': 0,
            'group_actual_work_time': {'A': 0, 'B': 0, 'C': 0, 'E': 0},
            'group_idle_time': {'A': 0, 'B': 0, 'C': 0, 'E': 0}
        }

        for run in range(num_runs):
            # 使用不同随机种子
            random.seed(int(time.time() * 1000000) % (2**32) + run)
            np.random.seed(int(time.time() * 1000000) % (2**32) + run)

            simulator = CorrectedYXBShiftWorkSimulator(num_devices=100, shift_hours=k)
            result = simulator.simulate()

            avg_results['T'] += result['T']
            avg_results['S'] += result['S']
            avg_results['Removed'] += result['Removed']
            avg_results['P_L'] += result['P_L']
            avg_results['P_W'] += result['P_W']
            avg_results['total_time'] += result['total_time']
            avg_results['shift_idle_time'] += result['shift_idle_time']
            avg_results['operation_delays'] += result['detailed_stats']['operation_delays']
            avg_results['boundary_delays'] += result['detailed_stats']['boundary_delays']
            avg_results['time_utilization'] += result['time_utilization']

            for yxb_key in avg_results['YXB']:
                avg_results['YXB'][yxb_key] += result['YXB'][yxb_key]

            for stage in ['A', 'B', 'C', 'E']:
                avg_results['group_actual_work_time'][stage] += result['group_actual_work_time'][stage]
                avg_results['group_idle_time'][stage] += result['group_idle_time'][stage]

        # 计算平均值
        for key in ['T', 'S', 'Removed', 'P_L', 'P_W', 'total_time', 'shift_idle_time',
                    'operation_delays', 'boundary_delays', 'time_utilization']:
            avg_results[key] /= num_runs

        for yxb_key in avg_results['YXB']:
            avg_results['YXB'][yxb_key] /= num_runs

        for stage in ['A', 'B', 'C', 'E']:
            avg_results['group_actual_work_time'][stage] /= num_runs
            avg_results['group_idle_time'][stage] /= num_runs

        results[k] = avg_results

        print(f"  完成天数: {avg_results['T']:.2f}")
        print(f"  通过装置数: {avg_results['S']:.2f}")
        print(f"  淘汰装置数: {avg_results['Removed']:.2f}")
        print(f"  漏检概率: {avg_results['P_L']*100:.3f}%")
        print(f"  误判概率: {avg_results['P_W']*100:.3f}%")
        print(f"  时间利用率: {avg_results['time_utilization']*100:.2f}%")
        print(f"  YXB1(A 组): {avg_results['YXB']['YXB1']:.4f} ({avg_results['YXB']['YXB1']*100:.1f}%)")
        print(f"  YXB2(B 组): {avg_results['YXB']['YXB2']:.4f} ({avg_results['YXB']['YXB2']*100:.1f}%)")
        print(f"  YXB3(C 组): {avg_results['YXB']['YXB3']:.4f} ({avg_results['YXB']['YXB3']*100:.1f}%)")
        print(f"  YXB4(E 组): {avg_results['YXB']['YXB4']:.4f} ({avg_results['YXB']['YXB4']*100:.1f}%)")
        print()

    return results

def analyze_optimal_k_corrected_yxb(results):
    """修正 YXB 计算的多准则分析最优 K 值"""
    print("=== 修正 YXB 计算的多准则 K 值优化分析结果 ===")

    # 定义多种评价方案
    evaluation_schemes = {
        'efficiency_first': {
            'description': '效率优先方案',
            'weights': {
                'time_efficiency': 0.4,  # 时间效率权重
                'productivity': 0.3,     # 生产率权重
                'quality': 0.2,          # 质量指标权重
                'utilization': 0.1       # 利用率权重
            }
        },
        'quality_first': {
            'description': '质量优先方案',
            'weights': {
                'quality': 0.5,          # 质量指标权重
                'time_efficiency': 0.2,  # 时间效率权重
                'productivity': 0.2,     # 生产率权重
                'utilization': 0.1       # 利用率权重
            }
        },
```

```
        'balanced': {
            'description': '平衡型方案',
            'weights': {
                'time_efficiency': 0.25, # 时间效率权重
                'quality': 0.25,         # 质量指标权重
                'productivity': 0.25,    # 生产率权重
                'utilization': 0.25      # 利用率权重
            }
        },
        'utilization_first': {
            'description': '利用率优先方案',
            'weights': {
                'utilization': 0.4,      # 利用率权重
                'productivity': 0.3,     # 生产率权重
                'time_efficiency': 0.2,  # 时间效率权重
                'quality': 0.1           # 质量指标权重
            }
        }
    }

    all_scores = {}

    for scheme_name, scheme in evaluation_schemes.items():
        print(f"\n--- {scheme['description']} ---")
        scheme_scores = {}

        # 计算各项指标的归一化值
        time_values = [r['T'] for r in results.values()]
        productivity_values = [r['S'] / r['T'] for r in results.values()]
        quality_values = [r['P_L'] + r['P_W'] for r in results.values()]
        utilization_values = [r['time_utilization'] for r in results.values()]

        # 归一化处理
        max_time = max(time_values)
        min_time = min(time_values)

        max_productivity = max(productivity_values)
        min_productivity = min(productivity_values)

        max_quality = max(quality_values)
        min_quality = min(quality_values)

        max_utilization = max(utilization_values)
        min_utilization = min(utilization_values)

        for k, result in results.items():
            # 时间效率得分（完成天数越少越好）
            if max_time > min_time:
                time_score = 1 - (result['T'] - min_time) / (max_time - min_time)
            else:
                time_score = 1.0

            # 生产率得分（每天完成装置数越多越好）
            productivity = result['S'] / result['T'] if result['T'] > 0 else 0
            if max_productivity > min_productivity:
                productivity_score = (productivity - min_productivity) / (max_productivity - min_productivity)
            else:
                productivity_score = 1.0

            # 质量得分（误判率和漏检率越低越好）
            quality_error = result['P_L'] + result['P_W']
            if max_quality > min_quality:
                quality_score = 1 - (quality_error - min_quality) / (max_quality - min_quality)
            else:
                quality_score = 1.0

            # 利用率得分（时间利用率越高越好）
            utilization = result['time_utilization']
            if max_utilization > min_utilization:
                utilization_score = (utilization - min_utilization) / (max_utilization - min_utilization)
            else:
                utilization_score = 1.0

            # 综合得分
            composite_score = (
                scheme['weights']['time_efficiency'] * time_score +
                scheme['weights']['productivity'] * productivity_score +
                scheme['weights']['quality'] * quality_score +
                scheme['weights']['utilization'] * utilization_score
            )

            scheme_scores[k] = {
                'time_score': time_score,
                'productivity_score': productivity_score,
                'quality_score': quality_score,
                'utilization_score': utilization_score,
                'composite_score': composite_score
            }

            print(f"K = {k}小时:")
            print(f"  时间效率: {time_score:.4f}")
            print(f"  生产率: {productivity_score:.4f}")
            print(f"  质量指标: {quality_score:.4f}")
            print(f"  利用率: {utilization_score:.4f}")
            print(f"  综合得分: {composite_score:.4f}")
            print()
```

```
        # 找出该方案的最优 K 值
        optimal_k = max(scheme_scores.keys(), key=lambda k: scheme_scores[k]['composite_score'])
        print(f"{scheme['description']}最优 K 值: {optimal_k}小时")
        print(f"综合得分: {scheme_scores[optimal_k]['composite_score']:.4f}")

        all_scores[scheme_name] = {
            'optimal_k': optimal_k,
            'scores': scheme_scores,
            'scheme_info': scheme
        }

    # 统计各方案推荐结果
    print(f"\n=== 各评价方案推荐结果汇总 ===")
    k_recommendations = defaultdict(list)

    for scheme_name, scheme_result in all_scores.items():
        optimal_k = scheme_result['optimal_k']
        k_recommendations[optimal_k].append(scheme_name)
        print(f"{scheme_result['scheme_info']['description']}: K = {optimal_k}小时")

    # 找出最常被推荐的 K 值
    most_recommended_k = max(k_recommendations.keys(), key=lambda k: len(k_recommendations[k]))

    print(f"\n=== 最终推荐结果 ===")
    print(f"最优班次时间: K = {most_recommended_k}小时")
    print(f"被以下方案推荐: {', '.join(k_recommendations[most_recommended_k])}")

    return most_recommended_k, results[most_recommended_k], all_scores

def detailed_analysis_corrected_yxb(optimal_k, optimal_result):
    """修正 YXB 计算的详细分析"""
    print(f"\n=== 修正 YXB 计算的详细分析 (K = {optimal_k}小时) ===")

    # 运行一次详细仿真
    random.seed(2048)
    np.random.seed(2048)

    simulator = CorrectedYXBShiftWorkSimulator(num_devices=100, shift_hours=optimal_k)
    detailed_result = simulator.simulate()

    print("详细仿真结果:")
    print(f"任务完成天数: {detailed_result['T']}")
    print(f"通过测试装置数: {detailed_result['S']}")
    print(f"生产效率: {detailed_result['S']/detailed_result['T']:.3f} 装置/天")
    print(f"总漏检概率: {detailed_result['P_L']:.6f}")
    print(f"总误判概率: {detailed_result['P_W']:.6f}")

    print(f"\n 各专业测试组的有效工作时间比 (修正后):")
    stage_names = {'YXB1': 'A 组', 'YXB2': 'B 组', 'YXB3': 'C 组', 'YXB4': 'E 组'}
    for yxb_key, stage_name in stage_names.items():
        yxb_value = detailed_result['YXB'][yxb_key]
        print(f"  {stage_name} ({yxb_key}): {yxb_value:.6f} ({yxb_value*100:.2f}%)")

    print(f"\n 各组工作时间详细统计:")
    for stage, stage_name in [('A', 'A 组'), ('B', 'B 组'), ('C', 'C 组'), ('E', 'E 组')]:
        work_time = detailed_result['group_actual_work_time'][stage]
        idle_time = detailed_result['group_idle_time'][stage]
        total_stage_time = work_time + idle_time
        work_ratio = work_time / detailed_result['total_time'] * 100

        print(f"  {stage_name}:")
        print(f"    实际工作时间: {work_time:.2f} 小时")
        print(f"    空闲等待时间: {idle_time:.2f} 小时")
        print(f"    工作时间比: {work_ratio:.2f}%")
        print(f"    测试次数: {detailed_result['group_test_count'][stage]}")

    print(f"\n 时间统计分析:")
    print(f"总仿真时间: {detailed_result['total_time']:.2f} 小时")
    print(f"有效工作时间: {detailed_result['total_work_time']:.2f} 小时")
    print(f"交班空闲时间: {detailed_result['shift_idle_time']:.2f} 小时")
    print(f"时间利用率: {detailed_result['time_utilization']*100:.2f}%")

    stats = detailed_result['detailed_stats']
    print(f"\n 边界时间处理统计:")
    print(f"总交班次数: {stats['shift_changes']}")
    print(f"边界延误次数: {stats['boundary_delays']}")
    print(f"操作延误次数: {stats['operation_delays']}")
    if stats['shift_changes'] > 0:
        print(f"边界延误比例: {(stats['boundary_delays']/stats['shift_changes'])*100:.2f}%")
        print(f"平均每次交班空闲时间: {detailed_result['shift_idle_time']/stats['shift_changes']:.2f} 小时")

    print(f"\n 各阶段测试统计:")
    print(f"总测试次数: {stats['total_tests_conducted']}")
    print(f"各阶段测试次数: A:{stats['stage_tests']['A']}, B:{stats['stage_tests']['B']}, C:{stats['stage_tests']['C']},
E:{stats['stage_tests']['E']}")
    print(f"设备故障次数: {stats['equipment_failures']}")

    # 验证 YXB 计算的合理性
    print(f"\n=== YXB 计算验证 ===")
    print(f"理论分析：")
    print(f"- A、B、C 组并行工作，但需要等待 E 组完成后才能开始下一轮")
    print(f"- E 组需要等待 A、B、C 全部完成后才能开始工作")
    print(f"- 因此各组的工作时间比都不应超过 60%左右")
```

```
    print(f"")
    print(f"实际计算结果：")
    yxb_sum = sum(detailed_result['YXB'].values())
    print(f"- 各组 YXB 之和: {yxb_sum:.4f} ({yxb_sum*100:.1f}%)")
    print(f"- A 组工作时间比: {detailed_result['YXB']['YXB1']*100:.1f}%")
    print(f"- B 组工作时间比: {detailed_result['YXB']['YXB2']*100:.1f}%")
    print(f"- C 组工作时间比: {detailed_result['YXB']['YXB3']*100:.1f}%")
    print(f"- E 组工作时间比: {detailed_result['YXB']['YXB4']*100:.1f}%")

    if all(yxb <= 0.6 for yxb in detailed_result['YXB'].values()):
        print(f"✓ YXB 计算结果合理：各组工作时间比均未超过 60%")
    else:
        print(f"⚠ YXB 计算可能存在问题：某些组的工作时间比超过了理论上限")

    return detailed_result

# 主程序执行
if __name__ == "__main__":
    print("=== 问题 3：修正 YXB 计算的两班倒连续工作制度优化 ===")

    # 第一步：修正 YXB 计算的优化 K 值
    optimization_results = optimize_shift_hours_corrected_yxb()

    # 第二步：修正 YXB 计算的多准则分析最优 K 值
    optimal_k, optimal_result, all_scores = analyze_optimal_k_corrected_yxb(optimization_results)

    # 第三步：修正 YXB 计算的详细分析
    detailed_result = detailed_analysis_corrected_yxb(optimal_k, optimal_result)

    # 第四步：生成修正 YXB 计算的结果表格
    print("\n=== 修正 YXB 计算的综合结果表格 ===")
    print("K 值\t 完成天数\t 通过数目\t 生产效率\t 漏检概率\t 误判概率\t 时间利用率\t 边界延误\tYXB1\t\tYXB2\t\tYXB3\t\tYXB4")
    print("-" * 180)

    for k in sorted(optimization_results.keys()):
        result = optimization_results[k]
        productivity = result['S'] / result['T'] if result['T'] > 0 else 0
        print(f"{k}\t{result['T']:.2f}\t\t{result['S']:.1f}\t\t{productivity:.3f}\t\t{result['P_L']:.6f}\t{result['P_W']:.6f}\t{re-
sult['time_utilization']*100:.1f}%\t\t{result['boundary_delays']:.1f}\t\t{result['YXB']['YXB1']:.6f}\t{result['YXB']['YXB2']:.6f}\t{re-
sult['YXB']['YXB3']:.6f}\t{result['YXB']['YXB4']:.6f}")

    print(f"\n=== 最终推荐方案 ===")
    print(f"推荐最优班次时间: K = {optimal_k} 小时")
```