

**软件可靠性课程实验报告**

**题 目:** 故障树的定性、定量分析

**院 系:** 计算机科学与技术学院

**专 业:** 软件工程

**学生姓名:**

**学 号:**

**二零二四 年 十二月 七日**

目录

[1、引言 3](#_Toc186231515)

[1．1编写目的 3](#_Toc186231516)

[2.算法原理 4](#_Toc186231517)

[2.1 故障树分析概念 4](#_Toc186231518)

[2.2 故障树分析流程 4](#_Toc186231519)

[2.3 故障树分析符号 5](#_Toc186231520)

[2.4 故障树构建 6](#_Toc186231521)

[3.算法实现 8](#_Toc186231522)

[3.1 算法流程 8](#_Toc186231523)

[3.2 算法实现 9](#_Toc186231524)

[3.3 数据来源 26](#_Toc186231525)

[3.4 结果展示 26](#_Toc186231526)

[4.算法结果分析 27](#_Toc186231527)

# 1、引言

## 1．1编写目的

通过构建故障树，以图形化和逻辑化的方式呈现软件系统中可能导致故障发生的各种因素及其相互关系。定性分析能够全面地确定系统所有可能的故障模式组合，即最小割集。这有助于软件测试人员、开发人员以及维护人员深入理解系统在哪些条件下会出现故障，从而提前识别潜在风险点，为针对性的测试用例设计、代码优化以及系统改进提供明确的方向。例如，在一个复杂的金融交易软件系统中，故障树分析可以揭示出诸如数据库连接错误、网络通信中断、交易逻辑错误等多种可能引发系统故障的因素及其组合方式，使相关人员清晰知晓系统的薄弱环节所在。

定量分析则在定性分析的基础上，进一步结合历史数据、专家经验以及相关概率模型，对故障树中的各个基本事件赋予发生概率，并通过逻辑运算计算出系统故障发生的概率以及各个故障模式对系统整体故障的贡献程度。这为软件系统的风险评估提供了精确的量化依据，使企业能够依据风险的大小合理分配资源进行风险管理。例如，对于一个航空航天软件系统，通过定量分析确定某个关键子系统故障导致整个系统失事的概率为极低水平，但一旦发生后果极其严重，那么企业就可以针对性地采取冗余设计、加强监控等措施来降低风险；而对于一些发生概率较高但影响相对较小的故障模式，则可以安排在常规维护中进行修复和优化。

基于故障树的定性与定量分析结果，能够制定出切实可行的系统可靠性提升策略。通过对最小割集的分析，可以确定哪些基本事件的改进对系统可靠性提升最为关键，从而优先进行处理。例如，如果发现某个软件模块的错误处理机制不完善是多个故障模式的共同原因，那么可以集中精力对该模块进行重新设计和优化。同时，定量分析得到的故障概率数据也有助于评估不同改进策略的效果，为选择最优方案提供支持。例如，比较增加硬件冗余和优化软件算法两种策略对系统可靠性提升的影响程度以及成本效益比，从而做出科学合理的决策，以实现系统可靠性的持续提升并确保软件在规定的条件下和规定的时间内能够正常运行，满足用户的需求并降低因软件故障带来的经济损失和安全风险。

在软件的整个生命周期中，故障树分析报告都能发挥重要作用。在需求分析阶段，可用于识别潜在的可靠性需求和风险，为制定合理的需求规格说明书提供参考；在设计阶段，帮助设计人员优化系统架构和模块设计，避免引入可能导致故障的因素；在测试阶段，指导测试用例的设计和优化，提高测试的覆盖率和有效性；在运维阶段，为故障诊断和修复提供线索，快速定位故障根源并采取有效的解决措施。总之，本报告能够为软件生命周期各个阶段的决策提供有力的数据支持和技术依据，确保软件项目的顺利推进和可靠运行。

# 2.算法原理

## 2.1 故障树分析概念

故障树分析是一种用于系统安全性和可靠性分析的方法。它从一个不希望出现的系统故障事件（顶事件）入手，通过逻辑推理来剖析导致该顶事件发生的各种直接和间接原因。在构建故障树的过程中，把原因事件逐步分解，这些事件包括中间事件和最终的基本事件。它的构建类似一个逆向推导过程，从结果找原因，用逻辑门来连接不同层级的事件。

在分析阶段，不仅要构建好故障树，还会进行定性分析，比如找出最小割集，即一组基本事件同时发生就会导致顶事件必然发生的集合，并且对这些最小割集进行重要性排序和比较。有时还会进行定量分析，在已知基本事件发生概率的情况下，计算顶事件的发生概率。

故障树分析可以用于故障诊断，能够帮助我们快速定位系统故障的根本原因。也可以用于系统设计阶段，用来发现系统的薄弱环节，从而对系统进行改进，提升系统的可靠性和安全性。还能用于风险评估，通过分析来评估系统故障所带来的风险程度，为采取风险控制措施提供依据。

## 2.2 故障树分析流程

故障树分析法的主要流程包括：分析了解系统、明确顶事件、构建故障树并进行定性、定量分析等。具体的步骤如图1所示。



图1 故障树分析流程图

1. 分析了解系统：首先要区分软件与人为因素给整个系统造成的影响，了解系统能够采用的不同状态模式及其与其他单元状态之间的对应关系，明确不同模式之间的互相转变。
2. 选择顶事件：完全了解系统和相关信息之后，把所有重要的故障事件逐一罗列并区分主次，之后结合分析目的和故障判断依据决定此次分析的顶事件。
3. 构建故障树：将顶事件放在故障树最上端的相应符号中，然后把导致顶事件的所有直接原因事件罗列在第二排的对应符号中，包括操作故障、配置错误、平台故障等，之后结合系统中故障间的逻辑关系采用合适的逻辑门将顶事件和上述直接原因事件相连接，按照这种方式，依照建树的原则依次逐级向下扩展，直至底层的全部原因事件都是底事件为止。
4. 故障树的简化：根据故障树的最小割集，在某些必要的合理假设下，可以由真实的系统图简化成一个表示主要逻辑关系的系统简化图。
5. 定性分析：定性分析是故障树分析方法的必要步骤，关键是得出故障树的最小割集和最小径集。其中最小割集代表了系统危险性，最小径集代表系统可靠性，后续章节中将详述求解方法。
6. 定量分析：定量分析是故障树分析方法的关键步骤，主要是计算概率重要度，得到顶事件的发生概率。通过定量分析的结果可以了解故障对系统造成的危害。

## 2.3 故障树分析符号

故障树当中经常用到的符号包括事件符号、转移符号以及逻辑门符号三类，经常能够使用到的包含下面这些：

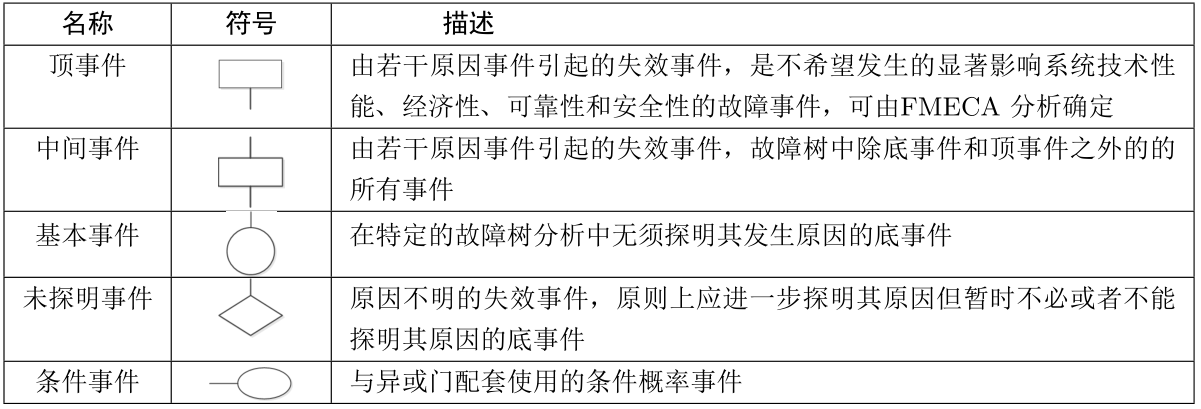
1. 事件符号（如表1）：

顶事件：故障树分析方法里面，首先需要确定的以及最需要关注的结果事件称作顶事件，放在故障树的最上面，为故障树分析的首要目的。

底事件：假如某一个原因事件仅仅能够引起另外别的事件出现，那么就叫做底事件，位于故障树最底端。

中间事件：在故障树分析方法中，处于顶事件与底事件之间的事件被称作中间事件，它既是当前逻辑门的输入事件，又是其他逻辑门的输出事件。

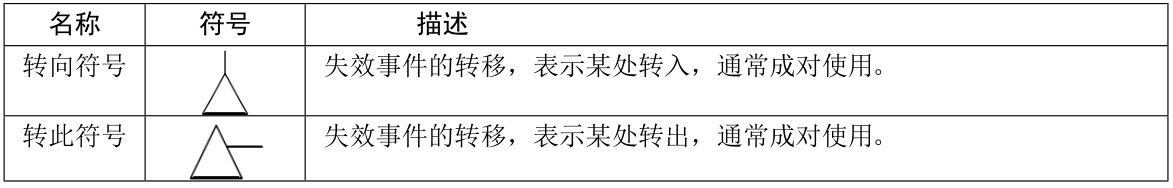
表1 故障树分析法的事件符号及描述



1. 转移符号（如表2）：

为了防止绘制故障树图的时候出现重复的现象，以及让图像更加简单明了，特意制定了转移符号。

表2 故障树分析法的转移符号及描述



## 2.4 故障树构建

由于大部分装备软件较为庞大复杂，如果根据程序各条语句的逻辑关系逐条分析来建立故障树，耗费成本巨大且绘制故障树非常困难，复杂的故障失效模式无法开展分析，因此以程序的各子模块为事件绘制故障树，对可能造成系统故障的各模块组合方式进行分析。

构建故障树要求建树者对系统及其组成部分有充分的了解，应由设计人员、使用维修人员、可靠性安全工程技术人员共同研究完成。构建故障树是为了利用建树的过程更加彻底地认识系统，确定薄弱环节，这也是故障树定性分析与定量分析的基础。可通过系统的结构及功能，剖析里面最不想让其产生的，明显导致系统故障的情形，明确此次分析的顶事件，然后建造故障树。建模树是一个多次反复、逐步深入完善的过程。建立故障树的根本原则：

1. 故障状况以及故障事件要进行明确的定义。
2. 对系统进行全面分析，在调查清楚故障的原因之后，明确故障树的顶事件是我们最不愿其产生的情形。
3. 遵循系统原则，进一步明确故障树的建树范围。
4. 需要表明各个矩形框里面应该放入的故障名称，与此同时确保该说明是正确的。
5. 在逐级分析矩形框中的故障事件时，如果故障事件由组件组成，则属于“部件故障状态”、反之，则故障事件属于“系统故障状态”。

在分析过程中，要找到造成故障情形产生的最简单而且是充分必要的直接原因，根据这种方式构造出来的故障树才能层次分明同时思维明确。

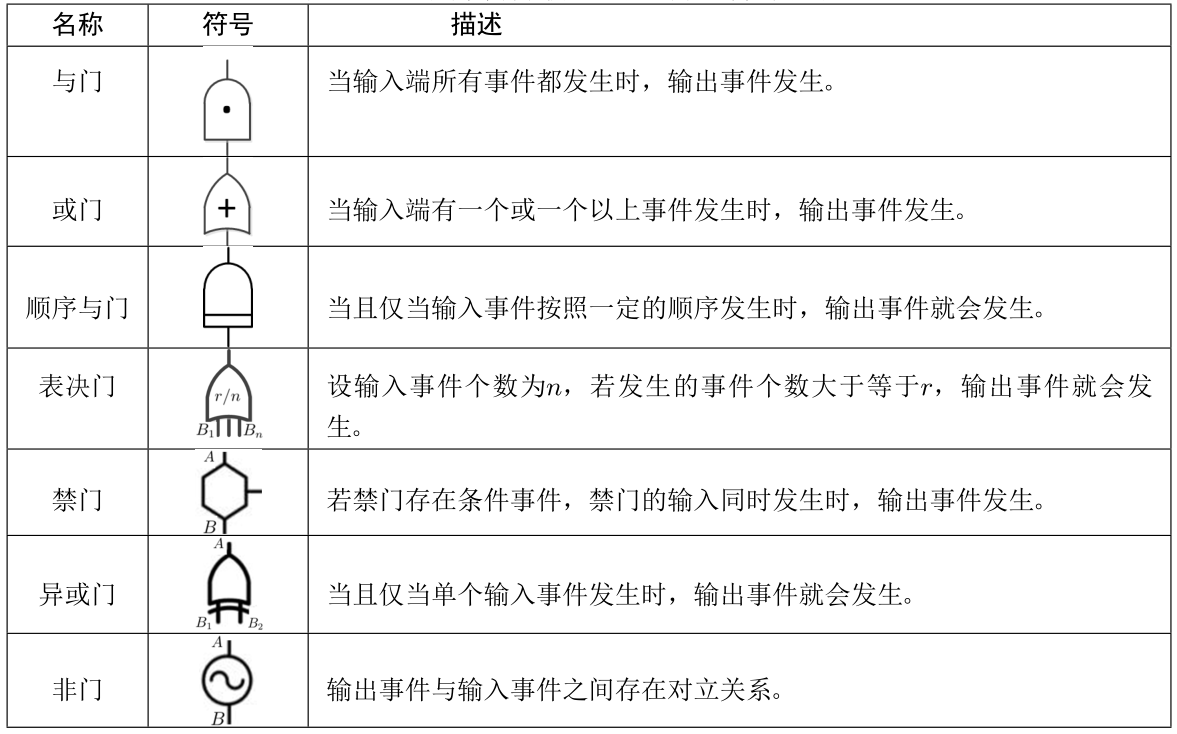
常用的建模方法为演绎法，从顶事件开始，由上而下，逐级进行分析，具体步骤如下：

1. 分析顶事件发生的直接原因：将顶事件作为逻辑门的输出事件，将所有引起顶事件发生的直接原因作为输入事件，根据它们之间的逻辑关系用适当的逻辑门连接起来。
2. 对每一个中间事件用同样的方法：逐级向下分析，直到所有的输入事件都不需要继续分析为止（此时故障机理或概率分布都是已知的）。
3. 掌握系统：包括系统的设计资源（如说明书、原理图、结构图）、试验资料（试验报告、试验记录等）、使用维护资料以及用户信息等。
4. 选择顶事件：顶事件选取根据分析目的不同，可分别考虑对系统技术性能、可靠性和安全性、经济性等影响显著的故障事件。例如，“飞机起落架放不下来”将直接危及飞机安全。当对起落架进行安全性分析时，可以选取“起落架放不下来”这一顶事件进行故障树分析。
5. 建造故障树：对于复杂系统，建树时按系统层次由上到下逐级展开。例如，“飞机起落架放不下来”这一事件，其原因包括收放机构发生机构卡死故障（上位锁故障、收放作动筒故障、连杆机构故障）、液压系统故障（如管路泄漏造成动力不足）、电磁控制系统故障等。

构建故障树过程中应注意以下事项：

1. 明确建树边界条件：建模前应对分析作出合理的假设。例如，导线不会故障、暂不考虑人为故障、软件故障等的一些假设；应在FHA（故障危害分析）或FMEA（故障模式和影响分析）的基础上，将那些不重要的因素舍去。从而减少树的规模及突出重点；
2. 故障树应严格定义：如果故障树不严格定义，则将难以得到正确的故障树，复杂系统的FTA 式往往由许多人共同完成，如定义不统一，将会建出不一致的故障树。
3. 应从上向下逐级建模：这样可以防止建树时发生事件的遗漏；
4. 建树时不允许门与门直接相连：为了防止不对中间事件严格定义就仓促建树，从而导致难以进行评审，或导致逻辑混乱使后续建模时出错。
5. 用直接事件代替间接事件：使事件具有明确的定义且便于进一步向下发展。
6. 重视共因事件：共同的故障原因会引起不同的部件故障甚至不同的系统故障；共因事件对系统故障发生概率影响很大，帮建树时必须妥善处理共因事件；若某个故障事件是共因事件，则对故障树的不同分支中出现该事件必须使用同一事件符号。

表3 故障树分析法的逻辑门符号及描述



顶端事件的概率：

逻辑或门的计算公式：

逻辑与门的计算公式：

结构重要度：

概率重要度：

关键重要度：

# 3.算法实现

## 3.1 算法流程

构建故障树流程如下：

首先选择顶事件，接着查找顶事件（通过查询 FMEA 表中的 “最终影响”），然后查找并画出所有对应的底事件（FMEA 表中的 “故障模式”）。

之后检索每个 “故障模式” 对应的 “代码”，将该 “代码” 末位数字截去，得到上一级部件对应的 “代码\*”。接着判断 “代码\*” 是否为 “1”，如果是，则流程结束；如果不是，则检索 “代码\*” 对应的上一级部件的名称并在故障树中画出（在系统结构代码表中检索），最后流程结束。

构建故障树流程图如图2所示：

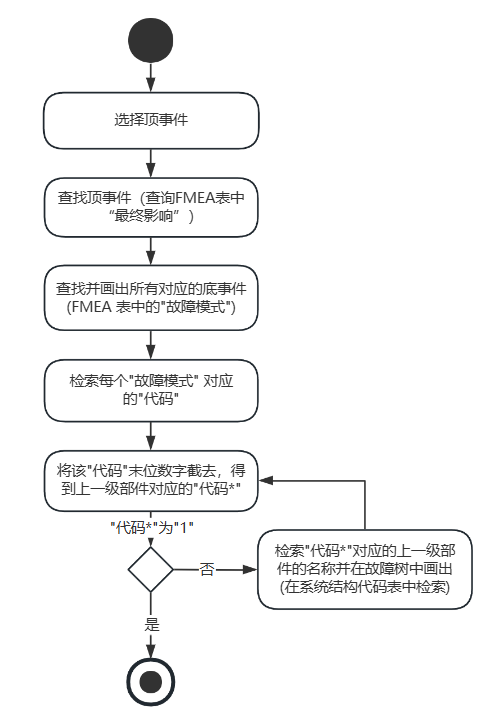


图2 构建故障树流程图

## 3.2 算法实现

本次实验采用python语言编写，运行的IDE是VScode。

|  |
| --- |
| 故障树代码如下 |
| import sympy as sp  import json  # 导入符号计算库SymPy和JSON库  class Node():      def \_\_init\_\_(self):          """          Node类的构造函数，用于初始化节点对象。          :return: 无返回值          """          self.list = []  class TreeCreator():      def \_\_init\_\_(self):          """          TreeCreator类的构造函数，用于初始化树对象。          :return: 无返回值          """          self.tree = []      def search(self, parentNode):          """          在树中搜索父节点的索引。          :param parentNode: 要搜索的父节点。          :return: 父节点在树中的索引，如果未找到则返回None。          """          length = len(self.tree)          # 获取树的长度          for i in range(length):              # 遍历树中的每个节点              if self.tree[i][0] == parentNode:                  # 如果当前节点的第一个元素等于父节点                  return i                  # 返回当前节点的索引      def addChild(self, parentNode, childNode):          """          将子节点添加到父节点的子节点列表中。          :param parentNode: 父节点的值。          :param childNode: 要添加的子节点的值。          :return: 无返回值。          """          if (parentNode == -1):              # 如果父节点的值为-1，表示没有父节点，直接返回              return          parentIdx = self.search(parentNode)          # 在树中搜索父节点的索引          self.tree[parentIdx][-1].append(childNode)          # 将子节点添加到父节点的子节点列表中      def jsonToTree(self, path):          """          从JSON文件中读取数据并构建树结构。          :param path: JSON文件的路径。          :return: 构建好的树结构。          """          with open(path, "r", encoding="utf-8") as file:              content = json.load(file)          for i in content:              node = []              node.append(i["code"])  # 节点编号              node.append(len(i["children"]))  # 子节点数量              node.append(i["sign"])  # 符号              if (i["children"] == []):  # 是否是基本事件                  node.append(True)              else:                  node.append(False)              node.append(i["possiblity"])  # 概率              node.append(-1)  # 原素数位置，现轮空              node.append(i["children"])  # 插入子节点列表              self.tree.append(node)  # 将该节点插入树中              # print(node)          return self.tree  class Queue():      def \_\_init\_\_(self, Tree, list=[]):          """          Queue类的构造函数，用于初始化队列对象。          :param Tree: 树对象。          :param list: 初始列表，默认为空列表。          :return: 无返回值。          """          self.list = list          # 初始化队列的列表          self.flag = False          # 初始化标志位          self.Tree = Tree          # 初始化树对象      def length(self):          """          返回队列的长度。          :return: 队列的长度。          """          if not self.list:              # 如果队列为空              return 0              # 返回0          else:              # 如果队列不为空              return len(self.list)              # 返回队列的长度      def is\_Empty(self):          """          检查队列是否为空。          :return: 如果队列为空，返回True；否则返回False。          """          return self.list == []        def push(self, value):          """          将元素添加到队列的末尾。          :param value: 要添加到队列的元素。          :return: 无返回值。          """          self.list.append(value)      def pop(self):          """          移除并返回队列头部的元素。          :return: 队列头部的元素。          """          top = self.list[0]          # 获取队列头部的元素          self.list.pop(0)          # 移除队列头部的元素          return top        def top(self):          """          返回队列头部的元素，但不移除它。          :return: 队列头部的元素。          """          return self.list[0]        def queue(self):          """          返回队列中的所有元素。          :return: 队列中的所有元素。          """          return self.list        def travel(self):          """          打印队列中的所有元素。          :return: 无返回值。          """          print(self.list)        def isALLBase(self):          """          检查队列中的所有节点是否都是基本事件。          :return: 如果队列中的所有节点都是基本事件，返回True；否则返回False。          """          self.flag = True  # 初始化标志位为True          for node1 in self.list:  # 遍历队列中的每个节点              if not node1 == [-1]:  # 如果节点不是[-1]（表示根节点）                  for node2 in node1:  # 遍历节点的子节点                      if not self.Tree[node2][3]:  # 如果子节点不是基本事件                          self.flag = False  # 将标志位设置为False                          return self.flag  # 返回False          return self.flag  # 如果所有节点都是基本事件，返回True    def getCutSet(Tree):      """      从故障树中获取割集的函数。      :param Tree: 故障树的结构，以列表形式表示。      :return: 包含所有割集的列表。      """      Q = Queue(Tree)      cutSet = Queue(Tree)  # 存放割集状态      # getCutSet函数，用于从树中获取割集      # 设置cutSet初始状态      if Tree[0][3]:          return []      else:          if Tree[0][2] == '+':              # 展开列表              for item in Tree[0][6]:                  itemList = list()                  itemList.append(item)                  cutSet.push(itemList)          else:              # 将child直接插到cutSet中去              cutSet.push(Tree[0][6])      cutSet.push([-1])      # 对cutSet进行循环判断      while (1):          # cutSet.travel()          # 1.          if cutSet.top() == [-1]:              if (cutSet.isALLBase()):                  break              else:                  tail = cutSet.pop()                  cutSet.push(tail)          # 3.2.1.          flag = True          for item1 in cutSet.top():              if not Tree[item1][3]:                  flag = False                  break          if flag:              node = cutSet.pop()              cutSet.push(node)              continue          else:              # 3              # 首先先找到在top中第一个不是基本事件的元素              firstNotBase = -1              nodeSetTop = []              for item2 in cutSet.top():                  if not Tree[item2][3]:                      cutSet.top().remove(item2)                      nodeSetTop = cutSet.pop()[:]                      # 将非基本事件的节点从队列顶部移除，并记录其索引                      firstNotBase = item2                      break              # 4              if Tree[firstNotBase][2] == '+':                  for item3 in Tree[firstNotBase][6]:                      nodeset = nodeSetTop[:]                      nodeset.append(item3)                      cutSet.push(nodeset)              else:                  nodeset = nodeSetTop[:]                  for item4 in Tree[firstNotBase][6]:                      nodeset.append(item4)                  cutSet.push(nodeset)      # 将临时队列中的割集转换为最终结果      cutSetAns = []      for item in cutSet.queue():          if item!= [-1]:              cutSetAns.append(item)      return cutSetAns  def flatten\_list(nested\_list):      """      将嵌套列表展平成单层列表。      :param nested\_list: 嵌套列表。      :return: 展平后的单层列表。      """      return [item for sublist in nested\_list for item in              (flatten\_list(sublist) if isinstance(sublist, list) else [sublist])]  # 定义一个质数列表，用于后续的最小割集计算  prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]  def getMinCutSet(cutSet=[]):      """      从割集中找到最小割集的函数。      :param cutSet: 包含所有割集的列表，默认为空列表。      :return: 包含所有最小割集的列表。      """      length = len(cutSet)  # 获取割集列表的长度      tempSet = set(flatten\_list(cutSet))  # 将割集列表展平并转换为集合，去除重复元素      primeSet = prime[:len(tempSet)]  # 获取与tempSet长度相同的质数列表      code2prime = dict(zip(tempSet, primeSet))  # 创建一个字典，将tempSet中的元素映射到primeSet中的质数      prime2code = dict(zip(primeSet, tempSet))  # 创建一个字典，将primeSet中的质数映射到tempSet中的元素      mulSet = []  # 用于存储每个割集的乘积      flagSet = []  # 用于标记每个割集是否为最小割集      for cut in cutSet:  # 遍历每个割集          mul = 1  # 初始化乘积为1          for i in cut:  # 遍历割集中的每个元素              mul \*= code2prime[i]  # 计算乘积          mulSet.append(mul)  # 将乘积添加到mulSet列表中          flagSet.append(True)  # 初始时，将每个割集标记为最小割集      for i in range(length):  # 遍历每个割集          for j in range(length):  # 遍历每个割集              if i == j:  # 如果是同一个割集，跳过                  continue              if mulSet[i] % mulSet[j] == 0:  # 如果一个割集的乘积能被另一个割集的乘积整除                  flagSet[i] = False  # 则将该割集标记为非最小割集      ans = []  # 用于存储最小割集      for i in range(length):  # 遍历每个割集          if flagSet[i]:  # 如果该割集是最小割集              ans.append(cutSet[i])  # 将其添加到ans列表中      return ans  def calProb(minCutSet, Tree):      """      计算最小割集的概率。      :param minCutSet: 最小割集的列表。      :param Tree: 故障树的结构，以列表形式表示。      :return: 最小割集的概率。      """      Prob = 0  # 初始化概率为0      sum = 1  # 初始化总和为1      for item in minCutSet:  # 遍历最小割集中的每个割集          prob1 = 1  # 初始化割集的概率为1          for item2 in item:  # 遍历割集中的每个元素              prob1 = Tree[item2][4] \* prob1  # 计算割集的概率          sum = (1 - prob1) \* sum  # 计算总和      Prob = 1 - sum  # 计算最小割集的概率      return Prob  H = []  # 存储所有基本事件的结构重要度  def Structural\_importance(Tree, cutSet):      """      计算结构重要度的函数。      :param Tree: 故障树的结构，以列表形式表示。      :param cutSet: 包含所有割集的列表。      :return: 包含所有最小割集的列表、基本事件的结构重要度列表和基本事件列表。      """      x = []  # 存储基本事件的索引      # 计算基本事件个数      for i in Tree:  # 遍历故障树的每个节点          if (i[3] == True):  # 如果节点是基本事件              if not i[0] in x:  # 如果基本事件的索引不在x列表中                  x.append(i[0])  # 将基本事件的索引添加到x列表中                  H.append(0)  # 将基本事件的结构重要度初始化为0      clear\_cutSet = []  # 存储清理后的割集      for i in cutSet:  # 遍历每个割集          temp = []  # 临时存储清理后的割集          for j in i:  # 遍历割集中的每个元素              temp.append(x.index(j))  # 将基本事件的索引转换为x列表中的索引          clear\_cutSet.append(temp)  # 将清理后的割集添加到clear\_cutSet列表中      generate\_permutations(len(x), clear\_cutSet)  # 生成所有可能的排列      return clear\_cutSet, H, x  # 返回清理后的割集、基本事件的结构重要度列表和基本事件列表  # 定义一个函数，用递归的方式生成所有可能的排列  def generate\_permutations(base\_events, cutSet, result=[]):      """      生成所有可能的排列。      :param base\_events: 基本事件的数量。      :param cutSet: 割集列表。      :param result: 当前生成的排列，默认为空列表。      :return: 无返回值。      """      # 如果当前列表的长度等于base\_events，说明已经生成了一个排列，将其添加到结果列表中      if len(result) == base\_events:          calculate\_structuralImportance(cutSet, result)          return      else:          for i in [0, 1]:              result.append(i)              generate\_permutations(base\_events, cutSet, result)              result.pop()  # 回溯，删除最后一个元素      return  def calculate\_structuralImportance(cutSet, event\_count):      """      计算结构重要度的具体值。      :param cutSet: 割集列表。      :param event\_count: 事件计数列表。      :return: 无返回值。      """      # 检查所有数组中为1的元素，与最小割集的集合进行判断，如果元素在某个某个中，检测这个割集的其他元素，如果都为1，说明该事件发生，Pxi1=1,      # 如果元素不在某个割集中，则判断该割集的元素，如果都为1，说明当这个元素为0时，顶事件也发生，Pxi1=Pxi0=1,结果为(Pxi1-Pxi0)/pow(2, len(event\_count)-1)      for i in range(len(event\_count)):  # 遍历事件计数列表          if (event\_count[i] == 0):  # 如果事件计数为0，跳过该事件              continue          Pxi1 = 0  # 初始化Pxi1为0          Pxi0 = 0  # 初始化Pxi0为0          for cutSet\_cell in cutSet:  # 遍历割集列表              if i in cutSet\_cell:  # 如果事件i在割集中                  isTop = True  # 初始化isTop为True                  for j in cutSet\_cell:  # 遍历割集中的元素                      if event\_count[j] == 0:  # 如果割集中的元素计数为0                          isTop = False  # 将isTop设置为False                          break                  if isTop:  # 如果isTop为True，说明该事件发生                      Pxi1 = 1  # 将Pxi1设置为1              else:  # 如果事件i不在割集中                  isTop = True  # 初始化isTop为True                  for j in cutSet\_cell:  # 遍历割集中的元素                      if event\_count[j] == 0:  # 如果割集中的元素计数为0                          isTop = False  # 将isTop设置为False                          break                  if isTop:  # 如果isTop为True，说明当该事件为0时，顶事件也发生                      # 当一个最小割集中不包含xi也导致顶事件发生则Pxi1和Pxi0都为1                      Pxi1 = 1  # 将Pxi1设置为1                      Pxi0 = 1  # 将Pxi0设置为1          H[i] += (Pxi1 - Pxi0) / pow(2, len(event\_count) - 1)  # 计算结构重要度并累加到H列表中  def Probabilistic\_importance(t, cutSet, oldcutSet):      """      计算概率重要度的函数。      :param t: 时间点。      :param cutSet: 最小割集。      :param oldcutSet: 旧的割集。      :return: 概率重要性列表以及对应的节点顺序。      """      proImportanceNode = []  # 存储概率重要性节点的列表      # 遍历旧的割集，找到所有不重复的节点并添加到proImportanceNode列表中      for i in range(len(oldcutSet)):          for j in range(len(oldcutSet[i])):              if not oldcutSet[i][j] in proImportanceNode:                  proImportanceNode.append(oldcutSet[i][j])      parameter = []  # 存储参数的列表      # 记录最小割集中互不相同的元素个数，方便为后面的Lamda输入      exp = "1-"  # 初始化表达式字符串      # 遍历最小割集，构建表达式字符串      for i in range(len(cutSet)):          if (i == 0):              exp += "("          else:              exp += "\*("          for j in range(len(cutSet[i])):              if j == 0:                  exp += "1-"              else:                  exp += "\*"              temp = chr(ord('a') + cutSet[i][j])  # 将节点索引转换为字符              exp += temp  # 将字符添加到表达式字符串中              if not temp in parameter:                  parameter.append(temp)  # 将字符添加到参数列表中          exp += ")"      # 读取指数分布的Lamda参数      #print('请输入各组件指数分布Lamda参数：')      Lama = []  # 存储Lamda参数的列表      Lama = [0.004,0.002,0.003,0.001,0.001]      # 遍历参数列表，读取用户输入的Lamda参数      # for i in range(len(parameter)):      #     temp = input()      #     Lama.append(float(temp))      f = []  # 存储符号变量的列表      # 遍历参数列表，创建符号变量并添加到f列表中      for i in range(len(parameter)):          temp = sp.symbols(parameter[i])          f.append(temp)      func = sp.sympify(exp)  # 将表达式字符串转换为sympy表达式      result = []  # 存储结果的列表      # 遍历Lamda参数列表，计算概率重要性并添加到result列表中      for i in range(len(Lama)):          temp = sp.diff(func, f[i])  # 对表达式求导          # 遍历Lamda参数列表，将符号变量替换为具体数值          for j in range(len(Lama)):              temp = temp.subs(f[j], 1 - sp.exp(-Lama[j] \* t))  # 带入Fj(t)=1-e^(-Lamda[j]\*t)          result.append(temp.evalf())  # 计算表达式的值并添加到result列表中      return result, proImportanceNode  # 返回结果列表和概率重要性节点列表  def Critical\_importance(t, cutSet, Tree):      """      计算关键重要度的函数      :param t: 时间点      :param cutSet: 最小割集      :param Tree: 故障树结构      :return: 关键重要度列表以及对应的节点顺序      """      # 先计算顶事件发生概率F\_s(t)      F\_s = calProb(cutSet, Tree)      proImportanceNode = []      for cut in cutSet:          for node in cut:              if node not in proImportanceNode:                  proImportanceNode.append(node)      # 计算每个基本事件的Fi(t)和Δgi(t)并进而计算关键重要度      critical\_importance\_list = []      for node in proImportanceNode:          # 模拟该基本事件发生，计算此时顶事件发生概率作为Fi(t)          temp\_cutSet = []          for cut in cutSet:              if node in cut:                  temp\_cut = cut.copy()                  temp\_cut.remove(node)                  if temp\_cut:                      temp\_cutSet.append(temp\_cut)                  else:                      continue              else:                  temp\_cutSet.append(cut)          # 计算此时顶事件发生概率作为Fi(t)          F\_i = calProb(temp\_cutSet, Tree)          # 计算Δgi(t)，这里简单示例为概率变化率（实际可能需要更精确的定义和计算方式，根据具体情况调整）          delta\_g\_i = (F\_i - F\_s) / F\_s if F\_s!= 0 else 0          # 计算关键重要度          critical\_importance = (F\_i / F\_s) \* delta\_g\_i if F\_s!= 0 else 0          # 将关键重要度添加到critical\_importance\_list列表中          critical\_importance\_list.append(critical\_importance)      # 返回关键重要度列表以及对应的节点顺序      return critical\_importance\_list, proImportanceNode |

|  |
| --- |
| 计算重要度代码如下 |
| import fault\_tree as ft  filePath = "ex3/test.json"  TreeCreator = ft.TreeCreator()  Tree = TreeCreator.json2tree(filePath)  cutSet = ft.getCutSet(Tree)  print("故障树表:", Tree)  cutSet = ft.getMinCutSet(cutSet)  print("最小割集:", cutSet)  Prob = ft.calProb(cutSet, Tree)  print("顶事件发生概率:", Prob)  newcutSet, H, x = ft.Structural\_importance(Tree, cutSet)  Lamda = [0.004,0.002,0.003,0.001,0.001]  print("各组件指数分布Lamda参数:", Lamda)  print("结果重要度节点顺序:", x)  print("结构重要度:", H)  t = 100  # 表示要求的时间点  probImportance, probImportanceNode = ft.Probabilistic\_importance(t, newcutSet, cutSet)  print("概率重要度节点顺序", probImportanceNode)  print("概率重要度:", probImportance)  # 新增计算关键重要度的调用  critical\_importance\_list, critical\_importance\_node = ft.Critical\_importance(t, cutSet, Tree)  print("关键重要度节点顺序:", critical\_importance\_node)  print("关键重要度:", critical\_importance\_list) |

|  |
| --- |
| 故障树可视化代码如下 |
| import fault\_tree as ft  import networkx as nx  import matplotlib.pyplot as plt  filePath = "ex3/test.json"  TreeCreator = ft.TreeCreator()  Tree = TreeCreator.json2tree(filePath)  def plot\_node(node, x\_pos, y\_pos, level, ax, base\_spacing=0.5, level\_factor=0.4):      """      绘制故障树节点及其子节点的函数。      :param node: 当前节点的信息列表，包含节点编号、符号、子节点列表等。      :param x\_pos: 当前节点的x坐标。      :param y\_pos: 当前节点的y坐标。      :param level: 当前节点的层级。      :param ax: 用于绘制的matplotlib轴对象。      :param base\_spacing: 子节点之间的基本间距，默认为0.5。      :param level\_factor: 随着层级增加，子节点间距的缩放因子，默认为0.4。      """      node\_code = node[0]  # 节点编号      sign = node[2]  # 节点的符号（+ 或 \*）        # 在指定位置绘制节点文本，包括节点编号和符号，并设置文本框样式      ax.text(x\_pos, y\_pos, f"{node\_code}\n({sign})", ha='center', va='center', fontsize=10, bbox=dict(facecolor='skyblue', edgecolor='black'))        # 如果当前节点有子节点，递归绘制子节点      if node[6]:  # 子节点列表          num\_children = len(node[6])          child\_spacing = base\_spacing \* level\_factor \*\* level  # 根据层级调整子节点间距          child\_x = x\_pos - (num\_children - 1) \* child\_spacing / 2  # 为子节点分配x坐标位置          for i, child in enumerate(node[6]):              # 查找子节点在故障树中的信息              child\_node = next(item for item in Tree if item[0] == child)              # 绘制当前节点与子节点之间的连线              ax.plot([x\_pos, child\_x + i \* child\_spacing], [y\_pos, y\_pos - 1], 'k-', lw=1)              # 递归调用plot\_node函数绘制子节点及其子节点              plot\_node(child\_node, child\_x + i \* child\_spacing, y\_pos - 1, level + 1, ax, base\_spacing, level\_factor)  def draw\_tree(Tree):      """      绘制故障树的函数。      :param Tree: 故障树的结构，以列表形式表示。      """      # 创建一个新的图形和轴对象，设置图形大小为10x6英寸      fig, ax = plt.subplots(figsize=(10, 6))      # 关闭轴显示，只显示绘制的图形内容      ax.axis('off')      # 从根节点开始绘制故障树，根节点位于(0.5, 0)位置，层级为0      plot\_node(Tree[0], 0.5, 0, 0, ax)      # 显示绘制的故障树图形      plt.show()  draw\_tree(Tree) |

|  |
| --- |
| 可视化显示代码如下 |
| import tkinter as tk  from tkinter import messagebox, simpledialog  import fault\_tree as ft  import matplotlib.pyplot as plt  from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  class FaultTreeGUI:      def \_\_init\_\_(self, master):          self.master = master          master.title("故障树分析工具")          master.geometry("400x200")  # 设置窗口大小为800x600像素          # 创建按钮          self.show\_tree\_button = tk.Button(master, text="显示故障树", command=self.show\_fault\_tree)          self.show\_tree\_button.pack(side=tk.TOP, padx=20, pady=30)  # 将按钮放在顶部，          self.calculate\_importance\_button = tk.Button(master, text="计算重要度", command=self.calculate\_importance)          self.calculate\_importance\_button.pack(side=tk.TOP, padx=20, pady=30)  # 将按钮放在顶部，并设置垂直间距为30像素      def show\_fault\_tree(self):          try:              filePath = "ex3/test.json"              TreeCreator = ft.TreeCreator()              Tree = TreeCreator.jsonToTree(filePath)              self.draw\_tree(Tree)          except Exception as e:              messagebox.showerror("错误", f"显示故障树时发生错误: {str(e)}")      def calculate\_importance(self):          try:              filePath = "ex3/test.json"              TreeCreator = ft.TreeCreator()              Tree = TreeCreator.jsonToTree(filePath)              cutSet = ft.getCutSet(Tree)              cutSet = ft.getMinCutSet(cutSet)              # 获取各组件的lambda参数              #lambda\_params = self.get\_lambda\_params(Tree)              # 检查 calProb0 函数的调用，确保只传递两个参数              Prob = ft.calProb(cutSet, Tree)              newcutSet, H, x = ft.Structural\_importance(Tree, cutSet)              t = 100              # 调用 Probabilistic\_importance 函数时，只传递三个参数              probImportance, probImportanceNode = ft.Probabilistic\_importance(t, newcutSet, cutSet)              # 调用 Critical\_importance 函数时，也只传递三个参数              critical\_importance\_list, critical\_importance\_node = ft.Critical\_importance(t, cutSet, Tree)              Lamda = [0.004,0.002,0.003,0.001,0.001]              messagebox.showinfo("重要度计算结果",                                  f"顶事件发生概率: {Prob}\n"                                  f"各组件指数分布Lamda参数: {Lamda}\n"                                  f"结构重要度节点顺序: {x}\n结构重要度: {H}\n"                                  f"概率重要度节点顺序: {probImportanceNode}\n概率重要度: {probImportance}\n"                                  f"关键重要度节点顺序: {critical\_importance\_node}\n关键重要度: {critical\_importance\_list}")          except Exception as e:              messagebox.showerror("错误", f"计算重要度时发生错误: {str(e)}")      def get\_lambda\_params(self, Tree):          """          获取用户输入的各组件lambda参数。          :param Tree: 故障树的结构，以列表形式表示。          :return: 包含各组件lambda参数的字典。          """          lambda\_params = {}          for node in Tree:              node\_code = node[0]              # 使用tkinter的simpledialog获取用户输入              lambda\_param = simpledialog.askfloat(f"输入组件 {node\_code} 的lambda参数", f"请输入组件 {node\_code} 的lambda参数:")              if lambda\_param is not None:                  lambda\_params[node\_code] = lambda\_param          return lambda\_params      def plot\_node(self, node, x\_pos, y\_pos, level, ax, Tree, base\_spacing=0.5, level\_factor=0.4):          """          绘制故障树节点及其子节点的函数。          :param node: 当前节点的信息列表，包含节点编号、符号、子节点列表等。          :param x\_pos: 当前节点的x坐标。          :param y\_pos: 当前节点的y坐标。          :param level: 当前节点的层级。          :param ax: 用于绘制的matplotlib轴对象。          :param Tree: 故障树的结构，以列表形式表示。          :param base\_spacing: 子节点之间的基本间距，默认为0.5。          :param level\_factor: 随着层级增加，子节点间距的缩放因子，默认为0.4。          """          node\_code = node[0]  # 节点编号          sign = node[2]  # 节点的符号（+ 或 \*）            # 在指定位置绘制节点文本，包括节点编号和符号，并设置文本框样式          ax.text(x\_pos, y\_pos, f"{node\_code}\n({sign})", ha='center', va='center', fontsize=10, bbox=dict(facecolor='skyblue', edgecolor='black'))            # 如果当前节点有子节点，递归绘制子节点          if node[6]:  # 子节点列表              num\_children = len(node[6])              child\_spacing = base\_spacing \* level\_factor \*\* level  # 根据层级调整子节点间距              child\_x = x\_pos - (num\_children - 1) \* child\_spacing / 2  # 为子节点分配x坐标位置              for i, child in enumerate(node[6]):                  # 查找子节点在故障树中的信息                  child\_node = next(item for item in Tree if item[0] == child)                  # 绘制当前节点与子节点之间的连线                  ax.plot([x\_pos, child\_x + i \* child\_spacing], [y\_pos, y\_pos - 1], 'k-', lw=1)                  # 递归调用plot\_node函数绘制子节点及其子节点                  self.plot\_node(child\_node, child\_x + i \* child\_spacing, y\_pos - 1, level + 1, ax, Tree, base\_spacing, level\_factor)      def draw\_tree(self, Tree):          fig, ax = plt.subplots(figsize=(10, 6))          ax.axis('off')          self.plot\_node(Tree[0], 0.5, 0, 0, ax, Tree)          plt.show()  # 使用plt.show()在弹出窗口中显示图形  # 创建主窗口  root = tk.Tk()  gui = FaultTreeGUI(root)  root.mainloop() |

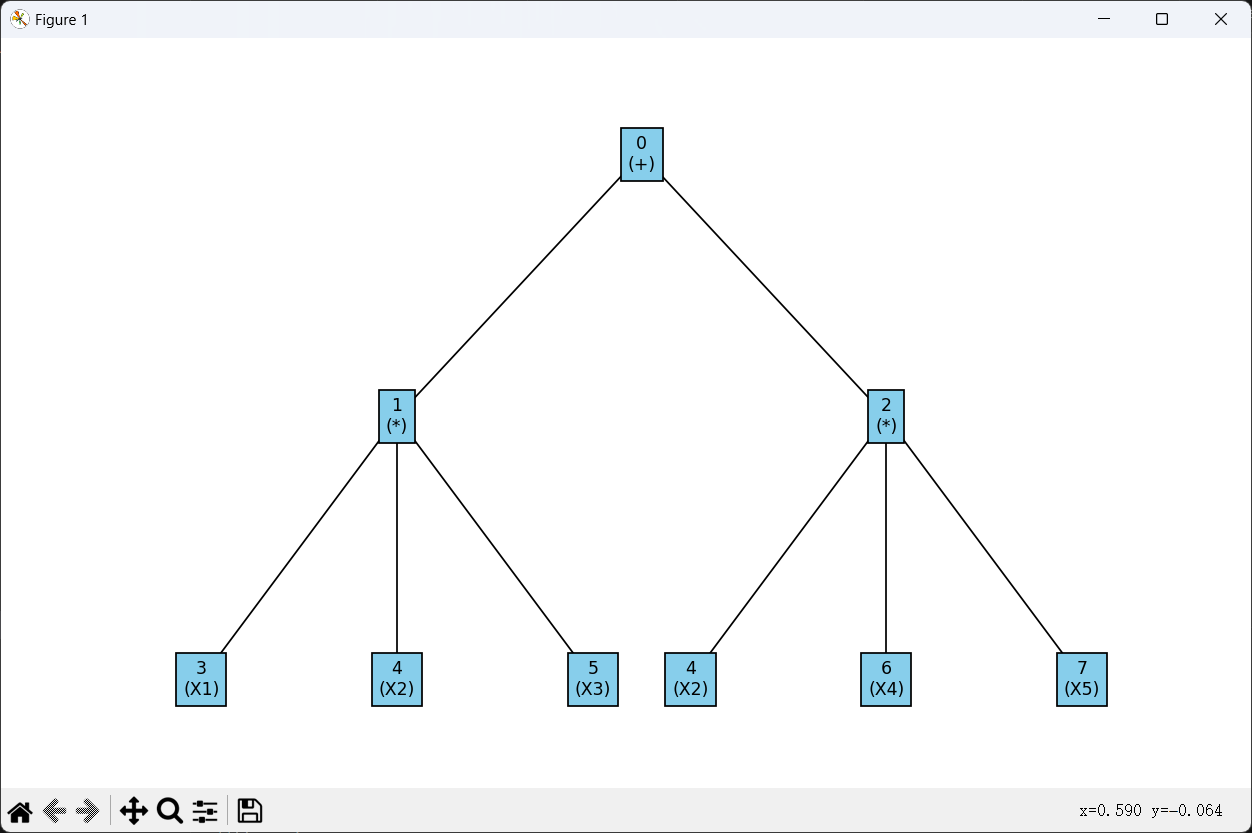
## 3.3 数据来源

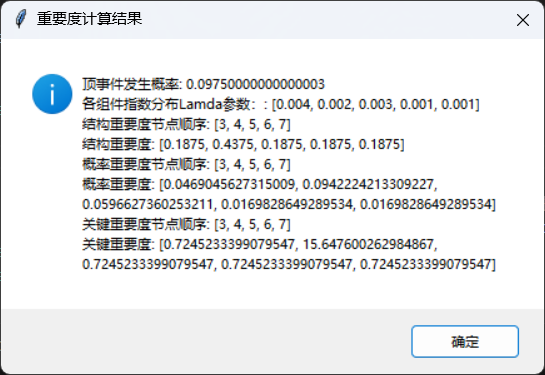
数据来源是自己编写的test.json文件。

## 3.4 结果展示

故障树可视化和重要度计算结果显示如下：







# 4.算法结果分析

从结构重要度、概率重要度和关键重要度结果可以看出，节点4（X2）在整个故障分析体系中占据着极为关键的地位。

在结构重要度方面，虽然各节点都有一定的数值，但节点4的0.4375远高于其他节点的0.1875，这表明从故障树的结构逻辑构成来看，节点4所代表的事件在导致顶事件发生的逻辑架构中起到了更为核心的作用。其状态的改变对整个系统故障发生逻辑链条的完整性和连续性影响更大，一旦节点4出现问题，整个系统的故障逻辑传导路径将受到极为关键的冲击。

对于概率重要度，节点4的0.0942224213309227同样高于其他节点。这意味着在考虑各基本事件发生概率的情况下，节点4发生故障对顶事件发生概率的提升幅度最大。即当节点4发生故障时，系统发生故障的可能性相较于其他节点故障时的情况会显著增加，它在众多因素中对最终故障结果的概率贡献最为突出。

而关键重要度进一步强化了节点4的重要性，其数值15.647600262984867远超其他节点。这不仅体现了节点4自身发生概率对顶事件的影响程度，还综合考虑了其在结构中的关键地位以及其他因素的相互作用。在整个系统的故障风险管控中，节点4无疑是重中之重，对其进行重点监测、维护以及风险防范措施的制定，将对系统整体的可靠性和安全性提升产生最为显著的效果。