中山大学硕士学位论文

离散事件系统模型未知时基于日志的故障诊断

Fault Diagnosis based on Event-log with the Model Unknown in Discrete Event Systems

学位申请人: _____

指 导 教 师: _____

专 业 名 称: 工程（软件工程）

答辩委员会主席(签名): _____

答辩委员会成员(签名): _____

_____

_____

_____

_____

二〇一九 年 五 月 二十 日

论文原创性声明

本人郑重声明: 所呈交的学位论文, 是本人在导师的指导下, 独立进行研究工作所取得的成果. 除文中已经注明引用的内容外, 本论文不包含任何其他个人或集体已经发表或撰写过的作品成果. 对本文的研究作出重要贡献的个人和集体, 均已在文中以明确方式标明. 本人完全意识到本声明的法律结果由本人承担.

学位论文作者签名:_____

日期:_____

学位论文使用授权声明

本人完全了解中山大学有关保留、使用学位论文的规定, 即: 学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版, 有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆、院系资料室被查阅, 有权将学位论文的内容编入有关数据库进行检索, 可以采用复印、缩印或其他方法保存学位论文.

学位论文作者签名:　　导师签名:

日期:　　年　　月　　日　　日期:　　年　　月　　日

论文题目: 离散事件系统模型未知时基于日志的故障诊断

专 业: 工程（软件工程）

硕 士 生:

指导教师:

摘 要

离散事件系统的故障诊断问题一直是一个十分重要且被广泛研究的问题。故障诊断问题关注如何在有限步的延迟内把已发生的故障事件诊断并隔离出来。传统的诊断方法分为两类，一类要求对系统模型有充分的认知，此类方法可以给出精确的诊断结果；另一类要求大量完整的系统运行序列以提高系统模型未知时故障诊断的准确率，因此这些传统诊断方法无法被应用于模型未知且含有大量冗余运行序列的系统。

本文基于系统模型未知，仅少量系统前期的运行日志已知的情况，提出利用关键观察消除运行序列中的冗余信息，并通过关键树消除故障无关的冗余序列，从而达到在高准确率的前提下，以多项式的时间复杂度完成对当前系统的故障诊断。其中关键观察可以在多项式时间内通过查找运行序列的最短非公共子序列生成，且不会产生冗余子观察。而利用关键树路径唯一存储了运行序列的序列特征，关键树可以在多项式时间内通过消除故障无关运行序列从而被构造出来。给定系统的运行序列，系统在该序列下的故障状态可以通过匹配关键树的路径得到。

首先我们提出关键树的定义，指出关键树是一颗内部结点标记为子观察，叶子结点标记为运行日志的多叉树，并同时提出关键树结点间层次性、充分性、不相容性、一致性的概念。接着我们给出了正确关键树和$F-$正确关键树的定义及其性质，同时证明了若关键树满足结点间的五条性质，则关键树是正确的。然后我们给出了正确关键树和$F-$正确关键树的构造算法和查询算法，并证明了构造算法具有多项式的时间和空间复杂度，而查询算法具有线性的时间复杂度和常数的空间复杂度。最后我们从空间消耗、时间消耗和准确率上比较了我们的关键树方法与Christopher CJ的日志树方法，实验结果表明我们的算法在效率和精确度上均有大幅度的提升。

关键词: 离散事件系统；故障诊断；关键观察；关键树

Title: Fault Diagnosis based on Event-log with the Model Unknown in Discrete Event Systems
Major: Engineering (Software Engineering)
Name:
Supervisor:

Abstract

The fault diagnosis of Discrete Event Systems (DESs) is an important and widely studied problem in DESs. This problem focuses on the task of detecting and isolating the occurred fault within a finite delay. Traditional diagnostic approaches require full knowledge about the system, or lots of its running sequences to improve diagnostic accuracy, so they cannot be applied to the system whose model is unknown and its available running-sequence are not enough.

In this paper, we propose an approach based on a set of running-logs of the unknown systems, so that we do not need lots of running-sequences, the approach utilizes critical observation to eliminate the redundant information in running sequence and critical tree to filter the relevant sequences. By this way, we can diagnose the faulty mode of systems with high accuracy and polynomial time complexity. A critical observation can be calculated by finding non-common subsequence between two running-sequences in polynomial time complexity. In addition, the critical tree can also be constructed in polynomial time complexity with the help of path, while a path uniquely stores the characteristic of a running-sequence. For a given running-sequence of system, we can diagnose its fault by querying the path of critical tree.

Firstly, we introduce the definition of critical tree, a critical tree is a multi-way tree whose inner node is labeled by a sub-observation and leaf node is labeled by a running-log, meanwhile, we also introduce the concept of hierarchy, sufficiency, incompatibility, similarity and consistency. Secondly, we present the definition of the correctness of critical tree and F-correctness of critical tree, if the critical tree satisfies five properties mentioned above, it is correct. Particularly, we propose the construction algorithm and query algorithm of critical tree, and we transform the problem of critical observation calculation to the problem of the least common subsequence. We prove the construction algorithms

have a polynomial-time complexity over time and space, and we also prove the query algorithm have a linear time complexity and constant space complexity. Finally, a comparative experiment is performed, the results show that our approach has advantages both in efficiency and accuracy over Christopher CJ's approach.

Key Words:Discrete Event System;fault diagnosis;critical observation;critical tree

目 录

第 1 章 引言

本章讨论离散事件系统(DES)及其故障诊断问题的研究背景，和模型未知时基于日志的故障诊断问题的研究意义，并介绍该问题当前的研究现状，同时说明本文的主要研究内容和研究结果，最后给出本文的组织结构。

1.1 研究背景和意义

故障诊断问题在工业自动化、通信系统、大型网络、危机管理等领域有着非常重要的地位。在诸如此类大型复杂系统中，如何在有限步的延迟内诊断并隔离出故障事件一直是一个重要的课题[1, 2]。随着无人化时代、工业定制化时代的到来，故障诊断问题越来越受到重视。举例来说，在大型工厂中，如何确保在生产线出现故障后，管理系统能在尽量短的时间内诊断出该故障，并及时对该故障进行修复是一个十分迫切的问题。特别是随着定制化工业的发展，生产线流程不再固定，生产线的组成随着定制化要求的不同而不同，而一条生产线往往包含若干个子系统，每个子系统又由若干个相互协同的设备组成。在这种情况下，意图只依靠人工就能诊断出故障所在无疑是不切实际的。而借助传感器的帮助，大型系统可以被抽象为一个离散事件系统，通过离散事件系统的状态转移模拟真实系统的运作，诊断器便能迅速诊断出故障所在，甚至能对系统进行自动修复[3–6]。

故障诊断问题早期是在连续变量系统领域内进行研究，此时的故障诊断是通过参数估计、状态估计等方法实现的[7–9]。直到1995年，Sampath M等人[10]首先在离散事件系统领域提出了可诊断性的概念，并给出了正式的定义。该定义指出，若在故障发生后的有限确切步延迟内，系统能诊断出该故障的发生，则称该系统是可诊断的。一般而言，判断一个系统是否可诊断是由诊断器[10–12]或验证器[13]实现的。本文统一将用于验证系统是否是可诊断的模型称为验证器，将用于诊断系统当前故障状态的模型称为诊断器。基于对系统的认知程度，故障诊断方法分为白盒诊断和黑盒诊断[14, 15]。

通常而言，白盒诊断需要对系统有充分的认知，此类方法往往基于模型完全可知的系统。毫无疑问，白盒诊断的诊断结果非常精确，但由于需要具体的系统模型，因此可扩展性不高。另一方面，高度依赖准确的系统模型也意味着白盒诊断对于模型不可知系统、模型部分可知系统和动态系统无能为力。目前在DES领域已有许多白盒诊断方法，其中Sampath M等人[10]首次提出利用验证器和诊断器来进行故障诊断，同时设计了一个指数级的验证器构造方法。之后Jiang S等人[11]和Yoo TS等人[13]分别将Sampath M的方法改进成两个多项式级的验证器构造方法，将复杂度降低到了状态

数的二次方级别。Moreira MV等人[16]则更进一步，提出了目前复杂度最低且支持分散诊断的验证器构造方法。目前在白盒诊断方法中，除基于验证器和诊断器的故障诊断方法外，还有基于解析冗余的故障诊断方法[17, 18]，基于模型的故障诊断方法[19–22]，以及基于可靠性工程的故障树方法[23, 24] 等。另外，白盒诊断方法也被扩展到了模糊离散事件系统上，模糊离散事件系统的监督控制理论首先是由Qiu DW教授建立的[25, 26]，特别是该文提出了一个判定监督者存在性的多项式时间算法。该算法的设计思想一致被后来的DES相关研究应用[27, 28]，如基于模糊离散事件系统的故障诊断问题[29, 30]，随机离散事件系统的故障诊断问题[31–33]等。近期Fabre E[34]，Alban G[35]和Ibrahim H[36]等人也分别将白盒诊断问题扩展到了故障修复、不确定观察、诊断规划等方向。

不同于白盒诊断，黑盒诊断无需获取精确的系统模型，因此受系统模型的影响较小，可延伸性高，目前被广泛应用于混合系统和动态系统的故障诊断问题研究。黑盒诊断方法主要为无模型诊断方法[37–40]。此类方法或根据系统的状态进行分析[41, 42]，或根据系统的可观测事件进行处理[40, 43]，或根据系统的运行序列进行分析[44–46]。本文根据系统前期产生的运行序列进行分析，运行序列是指系统运行时所产生的事件流，这些运行序列通常被记录在运行日志[47]之中。在本文中，我们假设系统在前期的运行中会记录由可观测事件组成的运行序列及系统在该运行序列下的故障状态。

由于系统模型未知，因此黑盒诊断方法的故障诊断结果无法做到完全准确。但毫无疑问，运行序列越多，构造出来的黑盒诊断器越精确。然而，一个完整的运行序列集合往往需要消耗大量的空间，而且该集合中的许多运行序列是与故障无关的。因此，在黑盒诊断方法中，如何消除这些冗余运行序列，从而减少故障诊断所需要的时间和空间便成为一个非常重要的问题。

1.2 研究现状

近期，在讨论如何消除黑盒诊断中的冗余运行序列问题的研究中，最优诊断方法[48, 49]考虑到一个故障往往只与某几个而非所有的可观测事件相联系，从而选择在系统运行前，在确保系统可诊断的前提下，通过最小化系统中的传感器数目从而达到消除故障不相关事件进而减少冗余运行序列的目的。而在序列诊断方法[50, 51]中，只有当故障已被探测到时系统才收集该运行序列，否则丢弃。即序列诊断方法通过仅保留确切的故障序列来提高诊断的精确性。动态观测方法[52–54]与序列诊断方法类似，但其选择在故障被探测到前通过动态开关传感器来消除运行序列中的不相关事件。

以上三类方法中，最优诊断方法考虑的是系统中所有可能发生故障的运行序列，

且和动态观测方法一样都要求系统的传感器必须是集成的且可任意开关，这些要求与本文聚焦于当前状态下系统的运行序列，且系统不可知不可控的要求相违背。而序列诊断方法在消除冗余运行序列时采取的是极端保守的策略，即尽可能的将所有探测到了故障的运行序列和那些没有探测到的运行序列分离。然而在某运行序列下系统没有探测到故障的发生并不代表该运行序列与故障不相关，或者说，无故障发生的运行序列同样可以提供故障序列下系统的运行特征。相比之下，本文选择采取更优的策略，关键观察[55–57]，关键观察选择保留那些可能与故障相关但在当前环境下并未发生故障的运行序列。比如若序列的状态为正常、故障或未知时，序列诊断方法选择忽视未知序列，尽管该序列可能会提供故障相关的运行特征。但关键观察则选择保留，因为尽管系统在该运行序列并发生故障，但并不代表该运行序列与故障无关。但另一方面，关键观察会忽视运行序列中那些可能与故障相关但在本运行序列下并未提供故障信息的可观测事件，仅保留运行序列中的系统运行特征。

关键观察的概念由Christopher CJ等人[55]提出，其最早提出了一个基于系统状态的关键观察，这里的关键观察是一个最小化的状态集合，对于一个故障诊断问题，系统能通过该状态集合充分推断出当下的故障状态。在这之后，关键观察被推广到基于事件的系统中[56]，同时，子观察的概念也被提出。子观察被形式化为一个由软事件和硬事件组成的序列。硬事件是指故障发生时，运行序列中那些一定会发生的事件，硬事件表示系统的运行特征。而软事件则是故障发生时运行序列中那些可能会发生，也可能不会发生的事件的集合。在这种框架下，关键观察被定义为最简洁的子观察，即子观察中的硬事件在充分保留系统的运行特征的同时长度最短。之后又在引入了日志[47]，故障标签[58]和事件模式[59, 60]的概念后，提出了一个基于日志树的关键观察生成算法[57]。该算法将传统的故障诊断问题转换为了更为常见的模式匹配问题，使得关键观察可以被运用于实际的系统中。然而该算法具有着指数级的时间复杂度，因此无法适应于一些低延迟的系统。如何在较少的时间和空间消耗内找到所有的关键观察并将之应用于故障诊断问题便成为本文的主要研究对象。

1.3 主要研究内容和研究结果

本文基于Christopher CJ等人[57]的框架，通过关键观察来消除冗余运行序列及运行序列中的冗余事件，从而达到在系统的模型完全未知，仅系统前期的运行日志集合已知的情况下，以较低的时间和空间复杂度完成对离散事件系统的故障诊断。关键观察通过仅保留运行序列中的故障相关事件消除运行序列中的冗余事件。关键观察表示运行序列的序列特征，若某运行序列符合关键观察，就认为该运行序列符合故障发生时系统的运行特征，因此推测系统在该运行序列下的故障状态为关键观察对应的故障状态。若能找出给定运行序列集合下所有的关键观察，系统就能以较高的准确率推测

出系统当前的故障状态。因此如何在较少的时间和空间消耗内找到所有的关键观察便成为主要的研究问题。

不同于以往的研究，本文提出通过查找非公共子序列来生成关键观察，利用关键树存储关键观察，同时消除故障无关序列，而系统的故障状态则通过查询关键树得到。在本文中，关键树即为故障诊断所需的诊断器。简单来说，关键树是一颗多叉树，其内部结点存储的是子观察，叶子结点存储的是运行日志。为避免冗余序列和序列中的冗余事件的干扰，关键树路径存储着关键观察运算过程中产生的子观察。而为保证关键树能符合这个结构，关键树的结点间需要满足特定的性质，将满足这些性质的关键树称为正确关键树，将只有部分特定路径满足这些性质的关键树称为$F-$正确关键树。若关键树是正确的或$F-$正确的，则认为基于关键树的故障诊断结果是可信的。

由于正确关键树与$F-$正确关键树的唯一区别在于前者要求关键树的所有结点都满足性质，因此通过利用运行日志对关键树进行更新而非一次性生成整个关键树，构造算法可以只需要多项式的时间和空间复杂度。同时，为防止当给定的运行日志集发生变化时已构造的关键树会失效的现象，通过保证每次更新后的关键树是正确的/$F-$正确的，运行日志集的改变对关键树造成的影响可以降低到最小。

由于关键观察表示故障序列的序列特征，因此若某系统运行序列匹配关键观察，则认为该序列符合对应故障发生时系统的运行特征。关键树使得系统无需查询所有的关键观察，若系统运行序列不匹配某内部结点上的子观察，则该系统运行序列一定不会匹配该结点所有子孙结点的子观察以及关键观察。通过查询关键树，系统的故障状态便能在线性级别的时间复杂度和常数级别的空间复杂度内被诊断出。

比较关键树方法与Christopher CJ等人[57]的日志树方法，实验结果表明无论是时间消耗、空间消耗，还是故障诊断的准确率，关键树方法都有较大幅度的优势。特别是$F-$正确关键树方法，在仅损失少量准确率的情况下，$F-$正确关键树方法仅需少量的时间和空间即可构造出所需的诊断器。

1.4 论文结构

本文的组织结构如下：

第2章简要介绍离散事件系统的基本概念，以及子观察、关键观察以及运行日志的概念和性质。

第3章给出关键树的定义，并介绍结点间层次性、充分性、不相容性、相似性和一致性的概念，然后提出了基于关键树的诊断问题的定义及相关定理。

特别的，在第4章中，我们给出了关键树的构造算法和查询算法，详细解释了程序的原理、实现及其作用，并证明了算法的复杂度。

第5章我们展示了关键树方法与Christopher CJ等人[57]的日志树方法的比较实验，并对实验结果进行了分析。

最后第6章总结本文，说明了本文存在的不足之处，以及对未来工作的展望。

第 2 章 预备知识

本章主要介绍离散事件系统及其诊断问题的相关符号、定义和定理。首先我们给出离散事件系统的基本概念，然后介绍经典诊断问题，接着说明子观察的基本框架、形式化定义以及基于子观察的诊断问题，之后给出关键观察的定义，最后介绍运行日志及基于日志的诊断问题。由于篇幅的关系，本章只对这些概念作简要介绍，想要详细了解这方面的知识则建议阅读Cassandras CG等人的教材[61]和Sampath M等人的经典论文[10]，以及国内的经典教材[62]。

2.1 基本概念

离散事件系统（DES）是状态空间离散，且状态转移是由事件驱动的系统。其中事件驱动是指仅当有事件发生时系统从一个状态转移到另一个状态（包括当前状态），否则系统将永远停留在当前状态[61]。一个离散事件系统通常由一个**有限状态自动机**模型表示。

$$G = (X, \Sigma, \delta, x_0) \tag{2.1}$$

其中X为自动机的**状态空间**，Σ为自动机的**事件集合**，δ:$X \times \Sigma \to X$为**部分转移函数**，x_0为自动机的**初始状态**。系统从初始状态x_0 出发，在Σ中的某个事件发生后，经由δ决定的转移函数转移到下一个状态[10]。系统的行为由G生成的前缀封闭语言$\mathcal{L}_G$描述[63]，**语言**$\mathcal{L}$被定义为事件集Σ的Kleene 闭包Σ^* 的子集。

事件集Σ分为**可观事件**Σ_o和**不可观事件**Σ_{uo}，即$\Sigma = \Sigma_o \cup \Sigma_{uo}$[64]。**故障事件**是指那些待诊断的事件，本文假设所有的故障事件都是不可观事件，即$\Sigma_f \subseteq \Sigma_{uo}$。

由于系统从一个状态转移到另一个状态的行为是由部分转移函数描述的，因此部分转移函数δ通常被扩展为δ^*:$X \times \Sigma^* \to X$。定义ε为空串，δ^*的定义如下：

$$\begin{aligned} &\delta^*(x, \varepsilon) = x && ;x \in X; \\ &\delta^*(x, s\sigma) = \delta^*(\delta^*(x, s), \sigma) && ;x \in X, s\sigma \in \mathcal{L}_G. \end{aligned} \tag{2.2}$$

无论事件是否是可观事件，只要存在当前状态下该事件的部分转移，则系统行为就会随之发生改变。本文假设系统G中的所有状态都存在转移，因此系统不可能在到达某个状态后无法继续运行，我们称由这样的系统G生成的语言都是**活语言**。同时我们假设系统从初始状态x_0出发，经若干转移可到达G中的任意状态，即

$$\forall x \in X, \exists s \in \mathcal{L}_G : \delta^*(x_0, s) = x \tag{2.3}$$

系统语言描述了该系统的行为，把引起状态转移的事件所组成的序列称为**串**。特别的，空串为ε。

对于$\mathcal{L}$中的一个串s，本文假设s的长度$\|s\|$都是有限的。若$s = tuv$，我们称t为s的前缀，u 为s 的子串，v 为s 的后缀。定义$\boldsymbol{s/t}$为串s中位于前缀t后的子串，即$s/t = uv$。定义串s的前缀封闭集合$\overline{s}$为串s的所有前缀的集合，即

$$\overline{s} = \{t \in \Sigma^* | \exists v \in \Sigma^* : tv = s\} \tag{2.4}$$

空串ε属于任何一个串的前缀集合。定义s_f为串s的最后一个事件，定义$\Psi(\sigma)$为所有以事件σ结尾的串的集合，即

$$\Psi(\sigma) = \{s \in \mathcal{L} | s_f = \sigma\} \tag{2.5}$$

定义语言$\mathcal{L}/s$为$\mathcal{L}$关于s的后语言，

$$\mathcal{L}/s = \{t \in \Sigma^* | st \in \mathcal{L}\} \tag{2.6}$$

定义$\overline{\mathcal{L}}$为$\mathcal{L}$的前缀语言，

$$\overline{\mathcal{L}} = \{s \in \Sigma^* | \exists t \in \Sigma^* : st \in \mathcal{L}\} \tag{2.7}$$

若$\mathcal{L} = \overline{\mathcal{L}}$，则称语言$\mathcal{L}$是**前缀封闭**的。

串可经由自然投影P_{Σ_o}投影为**运行序列**（或简称**序列**），特别的，空序列为ε。自然投影P_{Σ_o}的定义如下：

$$\begin{aligned}
&P_{\Sigma_o}(\varepsilon) = \varepsilon \\
&P_{\Sigma_o}(\sigma) = \sigma && ; \text{if } \sigma \in \Sigma_o \\
&P_{\Sigma_o}(\sigma) = \varepsilon && ; \text{if } \sigma \in \Sigma_{uo} \\
&P_{\Sigma_o}(s\sigma) = P_{\Sigma_o}(s)P_{\Sigma_o}(\sigma) && ; s \in \Sigma^*, \sigma \in \Sigma
\end{aligned} \tag{2.8}$$

简单来说，自然投影P_{Σ_o}忽略串中的不可观事件，仅保留可观事件。对于系统G，若$s \in \mathcal{L}_G, o = P_{\Sigma_o}(S)$，则称$s$为系统运行串，称$o$ 为系统运行序列。逆投影P^{-1}为所有可能产生序列o的串的集合，

$$P_{\Sigma_o}^{-1}(o) = \{s \in \Sigma^* | P_{\Sigma_o}(s) = o\} \tag{2.9}$$

将序列o逆投影后的语言标记为L_o。

为方便区分，后文统一用s表示串，o表示序列，同时用$f \in \Sigma_f$表示故障事件，$F \subseteq 2^{\Sigma_f}$表示故障事件的集合，其中2^{Σ_f}表示集合Σ_f的幂集。简写$\sigma \in o$ 表示序列o对应的串$s = P_{\Sigma_o}^{-1}(o)$ 中发生了事件σ，即$\overline{P_{\Sigma_o}^{-1}(o)} \cap \Psi(\sigma) \neq \emptyset$，特别的，当$f \in o$，称$o$为**故障序列**。简写$\eta = \Sigma_f \cap o$ 表示序列o**恰好**发生了故障事件集η，即$\eta = \{f | f \in \Sigma_f \wedge f \in o\}$。

2.2 经典诊断问题

离散事件系统下可诊断性的概念由Sampath M等人[10]提出，一个系统语言是可诊断的当可区分的不可观事件（通常指故障事件）可以在有限确切步延迟内，利用已知的可观事件序列被诊断出来。正式定义如下：

定义 2.1 一个前缀封闭的活语言$\mathcal{L}$是关于自然投影P_{Σ_o}，故障事件集Σ_f可诊断的当

$$(\forall i)(\exists n_i \in \mathbb{N})[\forall s \in \Psi(f_i)](\forall t \in \mathcal{L}/s)(\|t\| \geq n_i \Rightarrow D)$$

其中诊断条件D为

$$\omega \in P_{\Sigma_o}^{-1}[P_{\Sigma_o}(st)] \Rightarrow f_i \in \omega$$

在定义2.1中，串s为以故障事件f_i结尾，串t为s的充分长的后缀，诊断条件D表示对于语言$\mathcal{L}$中任何一个与串st产生相同投影序列的串，该串中一定包含故障事件f_i。诊断性需要在每个故障事件发生后有足够多的可观事件，这些可观事件使得系统可以在有限步的延迟内去诊断该故障的发生。

定义序列o的诊断信息$\Delta(o)$为系统在序列o下所有可能发生的故障事件集合的集合，

$$\Delta(o) = \{\eta \subseteq \Sigma_f | \exists s \in \mathcal{L}_G, P_{\Sigma_o}(s) = o \wedge s \cap \Sigma_f = \eta\} \tag{2.10}$$

定义语言$\mathcal{L}_\eta$为所有恰好发生了故障事件集η的串的集合，

$$\mathcal{L}_\eta = \{s \in \Sigma^* | s \cap \Sigma_f = \eta\} = \bigcap_{f \in \eta} \Sigma^* f \Sigma^* \cap \bigcap_{f \in \Sigma_f \setminus \eta} (\Sigma \setminus \{f\})^* \tag{2.11}$$

诊断信息$\Delta(o)$也可以用语言$\mathcal{L}_\eta$的空集问题来描述，即

$$\eta \in \Delta(o) \Leftrightarrow \mathcal{L}_G \cap \mathcal{L}_o \cap \mathcal{L}_\eta \neq \emptyset \tag{2.12}$$

若存在一个串，其投影序列为o且其中的故障事件集合为η，则η属于o的诊断信息。空集问题可以通过构造三个语言$\mathcal{L}_G$，$\mathcal{L}_\eta$和$\mathcal{L}_o$的自动机并同步这三个自动机来验证，类似的，空集问题也可以弱化为规划问题[65, 66]，或模型检查问题[67]。

2.3 子观察

不同于基于状态的诊断问题，基于事件的诊断问题中，系统运行序列中往往蕴含着许多隐藏信息，子观察捕捉这些隐藏信息并将之用于解决诊断问题。

2.3.1 子观察的定义

为区分序列中的冗余信息和有效信息，引入**硬事件**和**软事件**的概念。一个硬事件是一个单独的事件，记为$x \in \Sigma_o$，该事件必须在序列中按序发生，硬事件是序列中与故障相关的关键信息，硬事件序列描述了系统的运行特征（或者说**序列特征**）。一个软事件是一个可观事件集合，记为$y \subseteq \Sigma_o$，该集合中的事件在序列中可以以任意顺序发生任意次，软事件是序列中与故障不相关或相关性较弱的非关键信息。利用硬事件和软事件的概念，结合定义2.5，定义子观察如下：

定义 2.2 一个子观察θ是一个严格按序的，硬事件和软事件交替发生的，以软事件开始和结尾的序列，

$$\theta = y_0 x_1 y_1 \dots x_n y_n \tag{2.13}$$

或以逗号分隔的列表，

$$\theta = (y_0, x_1, y_1, \dots, x_n, y_n) \tag{2.14}$$

其中，$x_1, \dots, x_n$均为硬事件，$y_0, \dots, y_n$均为软事件。θ的长度为$\|\theta\| = n$。

特别的，定义空子观察为$(\emptyset) \in \mathbb{O}$，该子观察只被空序列$\varepsilon$匹配。定义$\mathbb{O}(o) \subseteq \mathbb{O}$表示序列$o$匹配的所有子观察组成的子观察子空间，$\mathbb{O} = \bigcup_{o \in \Sigma_o^*} \mathbb{O}(o)$。公式2.18 中语言$\mathcal{L}_\theta$ 可以被表示为：

$$\mathcal{L}_\theta = (y_0 \cup \Sigma_{uo})^* x_1 (y_1 \cup \Sigma_{uo})^* \dots x_n (y_n \cup \Sigma_{uo})^* \tag{2.15}$$

$\mathcal{L}_\theta$说明子观察θ允许系统在硬事件间发生若干次软事件和不可观事件，但硬事件必须得按序发生。子观察主要保留序列的序列特征，这个序列特征被子观察形式化为按序发生的硬事件序列。只要在序列中，子观察的硬事件都是按序发生的，且相邻硬事件间发生的事件都在对应软事件中，该子观察就被认为是该序列的抽象。

例 2.1 对于子观察$\theta = (\{b, d\}, a, \emptyset, c, \{a\})$，长度$\|\theta\| = 2$。$\theta$中$x_1 = a$和$x_2 = c$是硬事件，这表明在序列中，$a$和$c$必须发生且必须按序发生。而事件集合$y_0 = \{b, d\}$、$y_1 = \emptyset$和$y_2 = \{a\}$是软事件，软事件$y_0 = \{b, d\}$ 这表明事件b和d在序列中可以以任意顺序发生任意次，但所有这些事件必须在硬事件$x_1 = a$ 之前发生。类似的，$y_1 = \emptyset$意味着在硬事件$x_1 = a$和$x_2 = c$之间不能有可观事件发生，$y_2 = \{a\}$意味着在硬事件$x_2 = c$后，序列中只允许有任意数目的事件a发生。对于θ而言，有许多序列可以匹配，最简单的莫过于序列ac，而其它如bac、$bdbdbacaaa$或其它符合要求的任意长序列（包括无限长序列）。

定义序列的最具体子观察函数sub如下：

定义 2.3 单射函数$sub:\Sigma_o^*\to\mathbb{O}$通过将空集软事件$\emptyset$插入到序列$o$中每个事件的前后从而生成$\mathbb{O}$中关于序列$o$的最具体子观察，假定$o=e_1e_2\dots e_n$，则

$$sub(o)=\emptyset x_1\emptyset x_2\emptyset\dots x_n\emptyset\in\mathbb{O} \tag{2.16}$$

其中$\forall j:x_j=e_j$

子观察$sub(o)$的硬事件为序列o的所有事件，软事件则为空集，这意味着硬事件间不允许任何可观事件发生，换而言之，除了序列o本身，其它任何序列都不匹配$sub(o)$。这也就是说不存在另一个比$sub(o)$更具体的子观察能被序列o所匹配，$sub(o)$是序列o最具体的子观察。另一方面子观察(Σ_o^*)匹配所有序列，故而(Σ_o^*)为所有序列最抽象子观察。抽象运算的形式化定义如下：

定义 2.4 $\theta'\preceq\theta$当且仅当存在一个单射$f:\mathbb{N}\to\mathbb{N}$:

$$\begin{aligned}
&\text{假定}\|\theta'\|=n,\|\theta\|=l\\
&f:\{0,\dots,n+1\}\to\{0,\dots,l+1\}\text{满足}\\
&f(j)\geq j,f(j)<f(j+1),f(0)=0,f(n+1)=l+1\\
&x'_j=x_{f(j)}\\
&y'_j\supseteq\bigcup_{f(j)\leq k\leq f(j+1)-1}y_k\cup\bigcup_{f(j)<k<f(j+1)}\{x_k\}
\end{aligned}$$

若$\exists j\in\{0,\dots,n+1\}$满足$f(j)>j$，则$\theta'\prec\theta$。

分析定义2.4，$\theta'\preceq\theta$当存在映射f，f将θ'的硬事件映射到θ中时，仍然保持硬事件发生的顺序，且每个θ'中的软事件y'_i为映射后的两硬事件间所有事件的集合。因此不可能存在这样一个映射使得一个子观察匹配o的同时比$sub(o)$更具体，故而称$sub(o)$是序列o最具体子观察。序列o 匹配子观察θ同样可以用$\preceq$和sub 表示，即$o\in\theta$当$\theta\preceq sub(o)$。

例 2.2 对于子观察$\theta=(\{ac\},b,\{cd\},a,\{c\},d,\{c\},a,\emptyset)$和$\theta'=(\{abcd\},a,\{bcd\},a,\emptyset)$，$\theta'$中的硬事件$x'_1=a$和$x'_2=a$能通过函数$f:f(1)=2,f(2)=4$映射到$\theta$中的硬事件$x_2=a$和$x_4=a$，且软事件$y'_0=\{abcd\}=\{ac\}\cup\{b\}\cup\{cd\}=y_0\cup\{x_1\}\cup y_1$，同理对于$y'_1=\{bcd\}$和$y'_2=\emptyset$，详细对应关系如图2-1。故而$\theta'\preceq\theta$。

若系统模型用有限状态自动机表示，则系统从初始状态出发经子观察的每个硬事件和软事件后一定能到达信念状态–信念状态包含了系统的故障诊断信息。

序列抽象已被应用于许多方面，如处理可观测事件会丢失的情况[14]，传感器可能会失效的情况[68, 69]，可观测事件发生的顺序只是部分可知的情况[70–72]，可观测事件可以变化的情况[52]等。

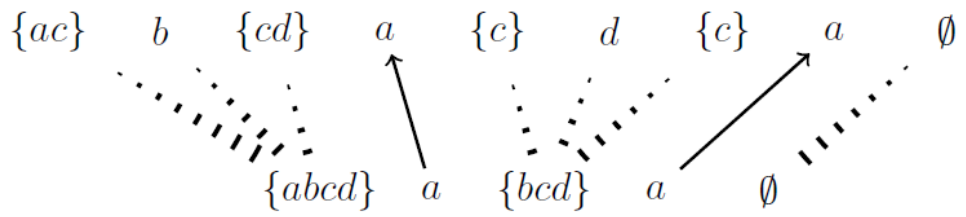


图 2-1 θ'与θ映射关系

2.3.2 子观察的框架

定义 2.5 一个**子观察**θ是一个序列的抽象表示，其通过有意忽略序列中的冗余信息从而把有效信息抽象出来。子观察定义在三元组$(\mathbb{O}, \preceq, sub)$框架下，具体地

1 $\mathbb{O}$表示子观察空间。

2 符号$\preceq$是一个基于$\mathbb{O}$的二元偏序关系。对于两个子观察θ和θ'，$\theta' \preceq \theta$表示θ'比θ更抽象，反之θ比θ'更具体。

3 $sub : \Sigma_o^* \to \mathbb{O}$ 是一个单射函数，该函数将序列映射为该序列最具体的子观察。

定义$\chi(\theta)$表示所有可抽象为子观察θ的序列的集合，

$$\chi(\theta) = \{o \in \Sigma_o^* | \theta \preceq sub(o)\} \tag{2.17}$$

定义序列o匹配子观察θ表示o可以被抽象为θ，

定义 2.6 序列o匹配子观察θ，表示为$o \in \theta$，当$o \in \chi(\theta)$。

定义子观察的语言$\mathcal{L}_\theta$表示所有可以抽象为子观察θ 的串的集合。$\mathcal{L}_\theta$ 可以被表示为所有可被抽象为子观察θ的序列的语言的并集，即

$$\mathcal{L}_\theta = \bigcup_{o \in \chi(\theta)} \mathcal{L}_o \tag{2.18}$$

序列o匹配θ也可以用$\mathcal{L}_\theta$表示，即$o \in \theta$当$o \in P_{\Sigma_o}(\mathcal{L}_\theta)$。

毫无疑问，子观察越抽象，匹配该子观察的序列就越多；且若子观察θ' 比θ更抽象，则匹配θ 的序列一定可以匹配θ'，即

定义 2.7 对任意两个子观察$\theta, \theta' \in \mathbb{O}$而言，$\theta'$比$\theta$ **抽象**，表示为$\theta' \preceq \theta$，当且仅当$\forall o \in \theta$都有$o \in \theta'$，换而言之：

$$\theta' \preceq \theta \Leftrightarrow \chi(\theta) \subseteq \chi(\theta')$$

若$\exists o \in \theta'$使得$o \notin \theta$，则称θ'比θ**严格抽象**的，表示为$\theta' \prec \theta$。

抽象运算也可以被表示为$\theta' \preceq \theta \Leftrightarrow \mathcal{L}_\theta \subseteq \mathcal{L}'_\theta$。抽象关系$\preceq$为偏序关系，其满足自反性、反对称性和传递性。

2.3.3 基于子观察的诊断问题

定义子观察下的诊断信息为匹配子观察的序列的诊断信息的并集：

定义 2.8 子观察θ的诊断信息是所有匹配θ的序列的诊断信息的并集，

$$\Delta(\theta) = \bigcup_{o \in \theta} \Delta(o) \tag{2.19}$$

若故障事件集$\eta \in \Delta(o)$且$o \in \theta$，则$\eta \in \Delta(\theta)$。子观察的诊断信息同样可以用语言$\mathcal{L}_\theta$的空集问题表示，

$$\eta \in \Delta(\theta) \Leftrightarrow \mathcal{L}_G \cap \mathcal{L}_\eta \cap \mathcal{L}_\theta \neq \emptyset \tag{2.20}$$

若能从子观察中推断出给定的诊断信息，则称该子观察是足够充分的，

定义 2.9 给定一个诊断信息D，一个子观察θ关于D是**充分**的当$\Delta(\theta) = D$。类似的，给定一个序列o，一个子观察$\theta \preceq sub(o)$关于o是充分的当$\Delta(\theta) = \Delta(o)$。

子观察的充分性保证了子观察在忽略序列中冗余信息的同时不会影响序列蕴含的故障信息。显然$sub(o)$关于o充分，即

$$\Delta(sub(o)) = \bigcup_{o' \in sub(o)} \Delta(o') = \Delta(o) \tag{2.21}$$

子观察还具有着有限性和单调性[56]：

引理 2.1 (**有限性**) 给定一个序列o，由o匹配的子观察θ组成的子观察子空间$\mathbb{O}(o)$的大小是有限的。

引理 2.2 (**单调性**) 给定一个序列o和两个子观察θ_1, θ_2，其中$\theta_1 \preceq \theta_2 \preceq sub(o)$，若$\theta_1$是关于$o$充分的，则$\theta_2$关于$o$也是充分的。

2.4 关键观察

我们的目标是找到关于序列充分中的最抽象的子观察，并称该子观察为关键观察：

定义 2.10 给定序列o，一个子观察$\theta \preceq sub(o)$是关于序列o**关键**的当：

1 子观察θ关于序列o充分；

2 不存在一个比θ严格抽象的子观察关于o充分，即

$$\forall \theta' \in \mathbb{O}, (\theta' \preceq \theta) \wedge (\Delta(\theta') = \Delta(o)) \Rightarrow (\theta' = \theta) \tag{2.22}$$

一个序列的关键观察是关于该序列充分，且无法在损失故障信息的前提下进行进一步抽象的子观察。显然，空子观察($\emptyset$)关于空序列ε是关键的。对于一个序列来说存在着多个关键观察，例如对于图2-2所示系统模式而言，$\theta_1 = (\Sigma_o, c, \Sigma_o, a, \Sigma_o)$和$\theta_2 = (\Sigma_o \setminus \{a\}, d, \Sigma_o, a, \Sigma_o)$都是序列$o = cda$的关键观察。

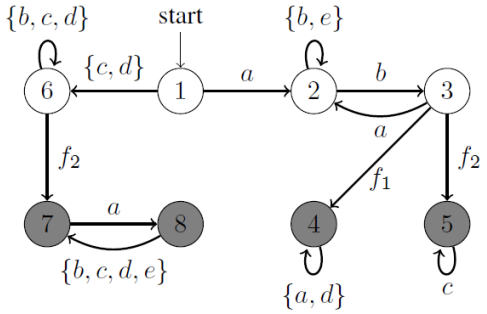


图 2-2 一个DES系统模型

由子观察的有限性（引理2.1）可知：

命题 2.1 序列o至少会存在一个关键观察。

命题 2.2 对于任意比序列o抽象的子观察θ而言，一定存在一个比θ抽象的子观察（包括θ本身）是o的关键观察。

命题 2.3 一个关键观察的“深度”是有限的。这里的深度是指$sub(o)$与θ之间严格抽象的子观察的数目，即$\theta \prec \theta_1 \prec \cdots \prec \theta_k \prec sub(o)$的$k$值大小。

另一方面，子观察的单调性（引理2.2）保证所有比序列o抽象的子观察都可以由$sub(o)$抽象得到的。利用子观察的这两个性质，可以提出子观察的孩子子观察概念如下：

定义 2.11 一个子观察θ的**孩子**子观察θ'是一个严格抽象于θ，且不存在另一个子观察“位于”θ和θ'之间：

$$\theta' \in children(\theta) \Longleftrightarrow (\theta' \prec \theta) \wedge (\nexists \theta'' \in \mathbb{O} : \theta' \prec \theta'' \prec \theta) \tag{2.23}$$

其中$children(\theta)$表示θ的孩子子观察的集合。

我们也称θ'是θ的**父**子观察。一个关于序列o充分的子观察是关键观察当且仅当没有孩子子观察是关于o充分的。

引理 2.3 一个子观察θ的孩子子观察的数目是有限的。

引理 2.4 一个子观察的诊断信息等于其所有父子观察的诊断信息的并集，

$$\Delta(\theta) = \bigcup_{\theta' \in \mathbb{O} \wedge \theta \in children(\theta')} \Delta(\theta') \tag{2.24}$$

2.5 运行日志

根据系统运行串中可能发生的故障事件的数目，我们将系统分为单故障系统和多故障系统。**单故障系统**是指系统的一次运行中最多只发生一个故障事件，反之**多故障系统**是指系统的一次运行中可发生多个故障事件。为方便表示，设定单故障系统下的每个故障事件或多故障系统下的每个故障事件集合一一对应于一个故障模式。假设故障事件集大小为$\|\Sigma_f\| = m$，单故障系统下故障事件$f_i, i \in \{1, 2, ..., m\}$或多故障系统下故障事件集合$F_i, i \in \{1, 2, ..., 2^m - 1\}$被唯一标记为故障模式$T_i$，特别的，将没有故障发生时的故障模式记为$T_0$。因此，若系统处于故障模式$T_i$则意味则系统的运行串中发生了故障事件$f_i$/故障事件集$F_i$。

2.5.1 运行日志的定义

定义 2.12 一个**运行日志**（简称**日志**）l是一个二元组(o, t)，其中：

- 存在一个系统运行串$s \in \mathcal{L}_G$ 满足$o = P_{\Sigma_o}(s)$；
- t为故障模式标签，表示串s中发生了模式t对应的故障事件f_t/故障事件集合F_t。

标注 2.1 我们将运行日志$l = (o_l, t_l)$中的序列o_l称为l的日志序列，t_l称为l的日志标签。定义$o_l = \rho_o(l)$取l的日志序列，$t_l = \rho_t(l)$取l的日志标签。并设定u表示故障模式的数目，对于单故障模式$u = m$，对于多故障模式$u = 2^m - 1$。

在本文中，系统模型是完全未知的，仅系统记录的运行日志集合是可知的。根据定义2.12，运行日志记录了系统的运行序列及系统当时的故障模式标签。对于单故障模式，模式标签$t \in \{T_0, T_1, ..., T_m\}$，其分别对应于故障事件$f_t \in \{\emptyset, f_1, f_2, ..., f_m\}$。而对于多故障模式，模式标签$t \in \{T_0, T_1, ..., T_{2^m-1}\}$，其分别对应于故障事件集$F_t \in \{\emptyset, F_1, F_2, ..., F_{2^m-1}\}$。模式标签表示系统在运行序列下恰好发生了故障模式对应的故障事件。

定义 2.13　一个运行日志集L是**正确**的当对于L中的任意一个运行日志$l = (o, t) \in L$，$\forall i \in \{1, \ldots, u\}$，$l$满足以下两个条件：

1 $t = T_i \Rightarrow \exists s \in \mathcal{L}_G, (P_{\Sigma_o}(s) = o) \wedge (\Sigma_f \cap s = (\{f_i\}\ or\ F_i))$

2 $t = T_0 \Rightarrow \exists s \in \mathcal{L}_G, (P_{\Sigma_o}(s) = o) \wedge (\Sigma_f \cap s = \emptyset)$

为确保故障诊断的准确性，系统必须保证其所记录的运行日志集是正确的。若模式标签非T_0，则系统在日志所记录的运行序列下一定恰好发生了模式标签所对应的故障事件，反之若模式标签为T_0，则系统一定没有发生任何故障事件。显然，对任意的运行日志集合L而言，$(\varepsilon, T_0) \in L$成立，即在系统没有运行的情况下，一定没有发生任何故障事件。

记录序列o对于系统而言是个简单问题，然而序列o和故障模式T_i的联系却不是如此直观的。对于系统而言，一般而言可以通过系统传感器或数据分类技术标记序列对应的故障模式，但这不属于本文的讨论范围。

2.5.2 基于日志的诊断问题

在引入日志后，定义运行日志$l = (o_l, t_l)$匹配子观察θ，表示为$l \in \theta$，当日志序列o_l匹配θ。定义运行日志集L中所有匹配θ的运行日志子集为L_θ，即

$$L_\theta = \{l \in L | l \in \theta\} \tag{2.25}$$

类似的，定义所有被运行日志集中的运行日志匹配的子观察集合为L_L，即

$$L_L = \{\theta \in \mathbb{O} | \exists l \in L : l \in \theta\} \tag{2.26}$$

基于日志的诊断信息$\Delta_L(\theta)$是运行日志集L中那些匹配θ的日志的日志标签的集合。

定义 2.14　给定一个运行日志集合$L = \{l_1, l_2, \ldots, l_n\}$，其中$l_j = (o_j, t_j)$，$j \in \{1, 2, \ldots, n\}$，定义基于日志的子观察诊断信息$\Delta_L(\theta)$为故障模式的集合：

$$\Delta_L(\theta) = \bigcup_{j=1}^{n} \{T_i | t_j = T_i \wedge o_j \in \theta\}, i \in \{1, \ldots, u\} \tag{2.27}$$

特别的，若不存在运行日志满足条件$o_j \in \theta$，则$\Delta_L(\theta) = T_0$。其中u表示故障模式的数目。

$\Delta_L(\theta)$高度依赖运行日志集L，当L变化时，基于日志的诊断信息也随之变化。同样的，基于日志的诊断信息同样满足引理2.4，即

命题 2.4　一个子观察的基于日志的诊断信息等于其所有父子观察的基于日志诊断信息的并集：

$$\Delta_L(\theta) = \bigcup_{\theta' \in L_L \wedge \theta \in children(\theta')} \Delta_L(\theta') \tag{2.28}$$

基于日志的子观察同样具有充分性的概念，即：

定义 2.15　一个子观察θ是关于日志$l = (o, T_i), i \in \{0, 1, \ldots, u\}$**充分**的当$o \in \theta \wedge \Delta_L(\theta) = T_i$。

一个子观察是否关于运行日志充分高度依赖于给定的运行日志集，当运行日志集L变化时，子观察关于日志的充分性也可能随之变化。

定义$\mathbb{O}(l) \subseteq \mathbb{O}$ 表示所有被运行日志l匹配的子观察组成的子观察子空间。基于日志的子观察依然具备有限性和单调性。

性质 2.1 (**有限性**)　给定一个运行日志$l = (o_l, t_l)$，被l匹配的子观察组成的子观察子空间$\mathbb{O}(l)$的大小是有限的。

性质 2.2 (**单调性**)　给定一个运行日志$l = (o_l, t_l)$和两个子观察θ_1, θ_2，其中$\theta_1 \preceq \theta_2 \preceq sub(o_l)$，若$\theta_1$是关于$l$ 充分的，则θ_2关于l也是充分的。

最后，定义基于日志的关键观察如下：

定义 2.16　给定一个运行日志$l = (o, T_i), i \in \{0, 1, \ldots, u\}$，一个子观察$\theta$且$o \in \theta$，称$\theta$关于$l$是**关键**的当且仅当：

1 子观察θ 关于运行日志l充分；

2 存在一个θ的孩子子观察关于l不充分，即

$$\exists \theta' \in children(\theta) : \Delta_L(\theta') \neq T_i \tag{2.29}$$

同样的，一个子观察关于日志是否是关键也会随着运行日志集L的变化而变化。特别的，无论L怎么变化，空子观察$(\emptyset)$始终关于空序列ε是关键的。

第 3 章 关键树的定义和性质

本文的目标是在模型未知的系统上，借助系统前期的运行日志，推断出系统当前的故障状态。为了完成这个目标，我们提出了一个名为关键树的数据结构。简单来说，关键树是一个多叉树，其内部结点的标签为子观察，叶子结点的标签为运行日志。关键树存储了运行日志的关键观察及其抽象运算过程中的子观察，及运行日志本身。首先我们给出关键树的定义，并介绍了关键树结点间的层次性、充分性、不相容性、相似性和一致性的定义；然后我们给出关键树的一个示例来直观的描述关键树；接着我们在关键树的基础上提出了基于关键树的诊断问题；最后我们针对前者的不足又提出了基于首次匹配的诊断问题。

3.1 关键树的定义及其性质

关键树是一颗多叉树，其内部结点标记为子观察，叶子结点标记为运行日志，叶子结点与内部结点不会共有同一个父结点，即叶子结点的兄弟结点为叶子结点，内部结点的兄弟结点为内部结点。除此之外，每个内部结点的父结点子观察是该结点子观察的孩子子观察，若其孩子结点为叶子结点，则该结点子观察是叶子结点日志的关键子观察。关键树的定义如下：

定义 3.1 给定运行日志集合L，**关键树**是个多叉树：

$$CtT(L) = (Q, S, \psi, \phi, q_0)$$

其中：

- $Q = Q_I \cup Q_L$是一个有限大小的结点空间。Q_I是内部结点空间，其中每个结点标记为子观察θ；Q_L是叶子结点空间，其中每个结点标记为运行日志$l \in L$；
- $S \subseteq \Sigma_o^*$是系统运行序列空间，$S = \{o | o = P_{\Sigma_o}(s) \wedge s \in \mathcal{L}_G\}$；
- $\psi : Q_I \to \mathbb{O}$是提取内部结点子观察的标签函数；
- $\phi : Q_L \to L$是提取叶子结点运行日志的标签函数；
- $q_0 \in Q_I$是关键树的根结点，其被标记为(Σ_o^*)。

关键树满足多叉树的性质，如根结点只有一个，除根结点外的任何结点都只有一个父结点，叶子结点无孩子结点等。除此之外，关键树还要求树中的每个结点带有一个标

签，内部结点（包括根结点）标记为一个子观察，叶子结点标记为一个运行日志。特别的，根结点标签始终为(Σ_o^*)，该子观察被所有序列所匹配。关键树存储了运行日志的关键观察及其抽象运算过程中的子观察，利用这些子观察，我们可以在尽可能少的时间和空间消耗内把关键树构造出来。未达到这个目的，对于关键树中任意两个结点$q, q' \in Q$，要求q, q'满足层次性、充分性、不相容性、相似性和一致性。

3.1.1 层次性

性质 3.1 (层次性) *若$q_I, q_I' \in Q_I$，且结点q_I是结点q_I'的父结点，则子观察$\psi(q_I)$比子观察$\psi(q_I')$严格抽象。*

层次性要求对于树中的任意一个内部结点$q_I \in Q_I$，结点q_I的祖先结点的子观察都比$\psi(q_I)$严格抽象，而q_I的子孙结点的子观察都比$\psi(q_I)$严格具体。这意味着对于关键树中从根节点到叶子结点路径上的所有内部结点，其子观察是从抽象到具体的。越接近叶子结点的内部结点，其子观察的匹配范围越局限，这也就意味着该子观察保留的序列特征越多。因此，在关键树的查询匹配过程中，我们便能跳过不符合序列特征的子观察，即通过查询运行序列匹配哪个孩子结点子观察决定下一个要查询的结点。

据引理2.1可知子观察的数目是有限的，因此关键树的大小也是有限的，

推论 3.1 *一个关键树的结点数目是有限的。*

命题 3.1 *一个非空关键树至少存在一个内部结点和叶子结点。*

证明 关键树的根结点为内部结点，而运行日志(ε, T_0) 属于所有系统运行日志，故而关键树中至少存在一个叶子结点。

命题 3.2 *一个关键树的深度是有限的。*

证明 关键树的任意一条路径存储着运行日志的关键观察及其抽象运算过程中的子观察，根据命题2.3，任意一个关键观察的深度都是有限的，而每个关键结点的孩子结点都为叶子结点，故而一个关键树的深度是有限的。 ■

3.1.2 充分性

性质 3.2 (充分性) *若$q_I \in Q_I, q_l' \in Q_L$，且内部结点$q_I$是叶子结点$q_l'$的父结点，则子观察$\psi(q_I)$关于$\phi(q_l')$的运行日志是关键的。并称结点$q_I$ 是结点q_l'的**关键结点**，表示为q_C。*

从关键树的根结点q_0到任意一个关键结点q_C的**路径**，记为$P_C = q_0 \to q_1 \to \cdots \to q_C$，路径$P_C$记录着关键观察$\psi(q_C)$抽象运算过程中产生的子观察。关键结点的孩子结点为叶子结点，该关键结点记录的关键观察关于叶子结点日志是充分的。若某运行序列匹配该关键观察，则该运行序列符合该叶子结点日志的序列特征，因此我们有把握认为系统在该运行序列下的故障状态与叶子结点日志标签相同。因此，该路径对应的故障模式即为叶子结点的日志标签。

标注 3.1　序列o匹配结点q_I表示序列o匹配结点q_I的子观察，即$o \in \psi(q_I)$。称序列o匹配路径P_C表示序列o匹配路径P_C上的所有子观察，记为$\boldsymbol{o \in P_C}$。标记符号T_{P_C}表示路径P_C对应的故障模式。

3.1.3 不相容性

性质 3.3 (**不相容性**)　若$q_I, q_I' \in Q_I$，且结点q_I与结点q_I'互为兄弟结点，则$\psi(q_I) \not\subseteq \psi(q_I')$且$\psi(q_I') \not\subseteq \psi(q_I)$。

不相容性要求存在兄弟关系的内部结点子观察之间不存在祖孙关系，更不会相等。不相容性通过避免兄弟结点的子观察的匹配范围重复，来减少关键树中冗余结点数目，并保证关键树中不存在两相同的路径。因此对于关键树中的任意两条路径，它们所匹配的序列范围都是不同的。引理2.3表明一个子观察的孩子子观察的数目是有限的，因此比一个子观察严格抽象且与该子观察匹配范围不重复的数目也是有限的。又因为运行日志集的大小是有限的，因此关键树中任意一个内部结点的孩子结点的数目也是有限的。

推论 3.2　关键树中任意一个结点的孩子结点数目是有限的。

根据命题2.4，即一个子观察的诊断信息等于其所有父子观察的诊断信息的并集，因此一个内部结点子观察的诊断信息等于其孩子结点子观察的诊断信息的并集：

推论 3.3　若$q_I \in Q_I$且$q_I \to children \subseteq Q_I$，则$\Delta_L(\psi(q_I)) = \bigcup_{q_I' \in q_I \to children} \Delta_L(\psi(q_I'))$。其中$q_I \to children$表示结点$q_I$的孩子结点集合。

3.1.4 相似性

性质 3.4 (**相似性**)　若$q_l, q_l' \in Q_L$，且结点q_l是结点q_l'的兄弟结点，则$\phi(q_l)$与$\phi(q_l')$有共同的故障模式和关键观察。

关键树结点间的相似性是关键树的核心所在。首先为了说明关键树的相似性是合理的，这里我们先证明关键观察是可以共享的，

命题 3.3 *对于两运行日志l, l'，日志l和l'的日志标签相同。若l'匹配l的关键观察θ，即$l' \in \theta$，则θ也是l'的关键观察。*

证明 据定义2.16知子观察θ关于日志$l = (o_l, t_l)$关键当$o_l \in \theta \wedge \Delta_L(\theta) = t_l$。由于运行日志$l$和$l'$的日志标签相同，因此$\Delta_L(\theta) = t_l$依然成立。又因为$o' \in \theta$，因此$\theta$关于$o'$也是充分的。另外序列增加不会改变$\theta$中存在关于序列$o_l$不充分的孩子子观察的结论，即$\exists \theta' \in children(\theta) : \Delta(\theta') \neq t_l$仍然成立。因此命题成立。 ■

一方面，叶子结点与其兄弟结点的父结点相同，根据关键树的充分性，叶子结点的父结点为关键结点，也就是说它们拥有相同的关键观察，所以它们的日志标签一定是相同的。也就是说，叶子结点和其兄弟结点拥有相同的故障模式和关键观察。另一方面，对于每条运行日志，当其匹配某路径且故障模式与原叶子结点日志标签一致时，据命题3.3，该路径关键结点的关键观察亦是该运行日志的关键观察，因此该运行日志就会成为该路径关键结点新的叶子结点。由命题3.3知关键观察是可以共享的，所以一个子观察可以是多个运行日志的关键观察，在关键树中则表现为一个关键结点可以有多个孩子叶子结点，且它们的日志标签是相同的。

如命题2.1所言，对于任意序列o，o至少会存在一个关键观察，因此对于L中的任一运行日志$l \in L$，在关键树中至少存在一个关键结点，该结点的标签为l的关键观察。同理，由于o可能会存在多个关键观察，所以关键树中可能存在多个关键结点，这些关键结点的标签为l的关键观察。总的来说，关键树中可能存在多个关键结点，它们的孩子叶子结点的标签都是l。同样的，一个关键结点可以拥有多个叶子结点。

3.1.5 一致性

性质 3.5 (**一致性**) *若$q_I \in Q_I, q'_l \in Q_L$，则内部结点$q_I$和叶子结点$q'_l$不拥有共同的父结点。*

一致性要求一个内部结点的兄弟结点一定是内部结点，一个叶子结点的兄弟结点也一定是叶子结点。很显然，若一个叶子结点的兄弟结点是内部结点，根据充分性和相似性，两结点的父结点子观察即要严格抽象于内部结点的子观察，又要是叶子结点的关键观察，而这是相互矛盾的。

一致性保证了关键树中的任意路径都是不可延伸的，这也就保证了当序列匹配某条路径时系统就可以得到确切的诊断结果，而不存在另一条延伸路径，该路径同样匹

配该序列但诊断信息不同。总的来说，一致性保证了关键树中的每条路径蕴含的诊断信息都是独一无二的。

3.2 关键树示例

给定$\Sigma_o = \{a, b, c\}$，图3-1展示了运行日志集$L = \{(aab, T_1), (aaab, T_1), (aac, T_2)\}$的关键树$CtT(L)$，接下来我们将以此关键树为例，详细解释关键树的性质。

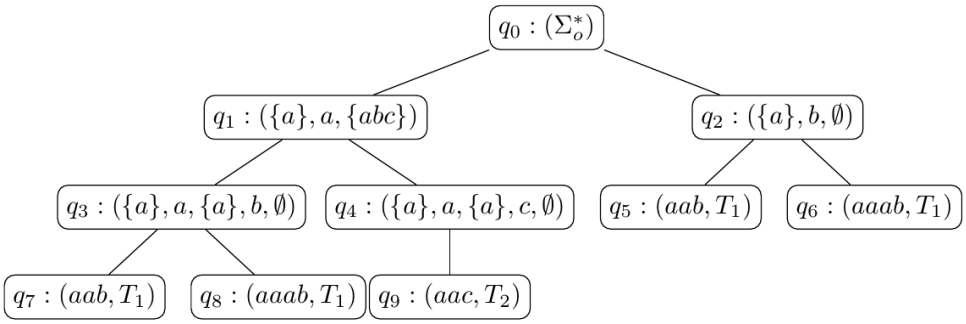


图 3-1 运行日志集$L = \{(aab, T_1), (aaab, T_1), (aac, T_2)\}$的关键树$CtT(L)$

显然，关键树中共有10个结点，从上到下从左至右标号为q_0到q_9，树的深度为4。树中共有5个内部结点，$Q_I = \{q_0, q_1, q_2, q_3, q_4\}$，每个结点标记为一个子观察；有5 个叶子结点，$Q_I = \{q_5, q_6, q_7, q_8, q_9\}$，每个结点标记为一个运行日志。$q_0$为根结点，其被标记为子观察$(\Sigma_o^*)$。接下来我们分析该关键树是否满足结点间的层次性、充分性、不相容性、相似性和一致性。

$CtT(L)$中的任意两个父子关系的内部结点都满足结点间的层次性，这里以两内部结点q_1和q_3为例，$\psi(q_1) = (\{a\}, a, \{abc\})$，而$\psi(q_3) = (\{a\}, a, \{a\}, b, \emptyset)$，$q_1$为$q_3$的父结点。显然，子观察$\psi(q_3)$与子观察$\psi(q_1)$之间存在一个单射$f : \{0, 1, 2\} \to \{0, 1, 2, 3\}$，这里$\|\psi(q_3)\| = 2$且$\|\psi(q_1)\| = 1$，使得$\psi(q_3)$ 和$\psi(q_1)$满足定义2.4，即$\psi(q_1)$比$\psi(q_3)$严格抽象。具体而言，单射f的映射为$f(0) = 0, f(1) = 1, f(2) = 3$，令$\psi(q_1) = (y_0', x_1', y_1')$且$\psi(q_3) = (y_0, x_1, y_1, x_2, y_2)$，则$x_1' = a = x_{f(1)}$；且满足$y_0 = \{a\} \subseteq \{a\} = y_0'$ 和$y_1 \cup y_2 = \{a\} \cup \emptyset = \{a\} \subseteq \{abc\} = y_1'$；又因为$f(2) = 3 > 2$，故而$\psi(q_1) \prec \psi(q_3)$成立，因此$\psi(q_1)$比$\psi(q_3)$严格抽象。确切的说，由于$\nexists \theta'' \in \mathbb{O} : \psi(q_1) \prec \theta'' \prec \psi(q_3)$，因此$\psi(q_1)$是$\psi(q_3)$ 的孩子子观察。同理，对于其它存在父子关系的内部结点q_0, q_1，q_0, q_2和q_1, q_4也都满足结点间的层次性。

$CtT(L)$中的任意一个叶子结点和其父结点满足结点间的充分性，这里以叶子结点q_9和其父结点q_4为例，$\psi(q_4) = (\{a\}, a, \{a\}, c, \emptyset)$，而$\phi(q_9) = (aac, T_2)$。首先$\psi(q_4)$关于$\phi(q_9)$是充分的，因为在运行日志集$L$ 中，$aab \notin \psi(q_4)$且$aaab \notin \psi(q_4)$，仅有$aac \in$

$\psi(q_4)$，故而$\Delta_L(\psi(q_4)) = T_2 = \rho_t(\phi(q_9))$。另一方面，仅$\psi(q_4)$和$\psi(q_3)$是集合$L_L$中的$\psi(q_1)$的父子观察，据命题2.4，$\Delta_L(\psi(q_1)) = \Delta_L(\psi(q_3)) \cup \Delta_L(\psi(q_4)) = \{T_1, T_2\}$，即存在$\psi(q_4)$的孩子结点子观察$\psi(q_1)$关于$\phi(q_9)$不充分。根据定义2.16，$\psi(q_4)$关于$\phi(q_9)$关键，即$q_4$是$q_9$的关键结点。同理对于$q_3, q_7$，$q_3, q_8$和$q_2, q_5$，$q_2, q_6$也满足结点间的充分性。

$CtT(L)$中的任意两个兄弟关系的内部结点满足结点间的不相容性，这里以q_3和q_4为例，$\psi(q_3) = (\{a\}, a, \{a\}, b, \emptyset)$，而$\psi(q_4) = (\{a\}, a, \{a\}, c, \emptyset)$。根据定义2.4，显然$\psi(q_3)$与$\psi(q_4)$之间不存在一个单射$f$使得$\psi(q_3) \preceq \psi(q_4)$或$\psi(q_4) \preceq \psi(q_3)$；或根据定义2.7，存在一个日志序列$aac \in \psi(q_4)$ 但$aac \notin \psi(q_3)$，且又存在另一个日志序列$aab \in \psi(q_3)$但$aab \notin \psi(q_4)$，因此$\psi(q_3) \not\preceq \psi(q_4)$且$\psi(q_4) \not\preceq \psi(q_3)$。同理对于$q_1, q_2$也满足结点间的不相容性。

显然，$CtT(L)$满足相似性和一致性，这里不再分析。

接下来我们分析路径$P_C = q_0 \to q_1 \to q_3$，$\psi(q_0)$为$\psi(q_1)$的孩子子观察，$\psi(q_1)$是$\psi(q_3)$的孩子子观察，路径$P_C$记录着基于$L$的关键观察$\psi(q_3)$生成过程中产生的所有子观察。另一方面，此路径两叶子结点的故障模式都为T_1，且关键观察都为$\psi(q_3) = (\{a\}, a, \{a\}, b, \emptyset)$，因此我们有把握认为故障模式$T_1$发生时具备事件$a$先发生事件$b$最后发生的特征。由此若某序列同样匹配路径$P_C$上所有子观察，这也就意味着该序列符合故障模式$T_1$发生时系统的运行特征，故而推测该序列下系统的故障模式为T_1。

3.3 基于关键树的诊断问题

关键树中存储着关键观察及其抽象运算过程中产生的子观察，通过匹配关键树的路径，我们可以推测出系统的故障模式。定义基于关键树的诊断信息如下：

定义 3.2 给定运行日志集合L及其关键树$CtT(L)$，对于任意的序列o，若关键树中存在路径P_C使得序列o匹配路径P_C上所有子观察，则Δ_C返回P_C对应的故障模式的集合：

$$\Delta_C(o) = \bigcup \{T_i | \exists P_C \in CtT(L) : o \in P_C \wedge T_{P_C} = T_i\}, i \in \{1, 2, \ldots, u\} \tag{3.1}$$

特别的，当$CtT(L)$中不存在任何一条路径被o匹配，则$\Delta_C(o) = T_0$。其中u表示故障模式的数目。

对于系统而言，仅可观事件是可以被观测到的，而运行日志记录的也是系统的运行序列而非系统的运行串。因此系统中可能存在着投影相同的不同串，即$\exists s_1, s_2 \in \mathcal{L}_G$: $P_{\Sigma_o}(s_1) = P_{\Sigma_o}(s_2)$，此时若系统在串$s_1, s_2$下的故障模式不同，则两运行日志是冲突的，即

定义 3.3 两条运行日志$l = (o_l, t_l)$和$l' = (o'_l, t'_l)$是**冲突**的当$o_l = o'_l$但$t_l \neq t'_l$。

冲突日志的出现意味着我们无法从该日志序列中提取出关键观察，因为对任意的子观察$\theta \preceq sub(o_l)$而言，θ都不会关于冲突日志充分。我们把不包含冲突日志的运行日志集合表示为L_C，并定义关键树$CtT(L_C)$关于运行日志集L_C正确。

定义 3.4　一个关键树$CtT(L_C)$关于运行日志集L_C是正确的当L_C中所有日志序列的诊断信息都是正确的，即

$$\forall l = (o_l, t_l) \in L_C : \Delta_C(o_l) = t_l$$

当运行日志集非完整集合时—可能在诊断过程中仍有新的运行日志生成，也可能该运行日志集是无限的，我们也可以利用关键树中已有的运行日志来进行判断。

定义 3.5　一个关键树$CtT(L)$是正确的当$CtT(L)$中所有的叶子结点运行日志的诊断信息都是正确的，即

$$\forall q_l \in Q_L, \phi(q_l) = (o_l, t_l) : \Delta_C(o_l) = t_l$$

由于在构建过程中$CtT(L)$会自动去除冲突日志，故而此定义适用范围更为广泛，无需确保L不包含冲突日志。

一个关键树是正确的意味着在诊断给定的运行日志集时，关键树的诊断结果是正确的。因此该关键树中的所有路径正确存储了系统在该故障模式下的序列特征。正确性的定义使关键树的构造与运行日志集中运行日志的顺序无关，在关键树本身正确的情况下，运行日志的顺序不同只会改变关键树的结点位置，但基于关键树的诊断信息不会被改变。同时关键树的正确性也弱化了运行日志集的变化对关键树造成的影响，在关键树本身正确的情况下，运行日志集变化只会使相关的关键结点和叶子结点发生改变，对其它结点则不会有任何影响。因此系统可以在较少的时间和空间消耗内完成关键树的构造和更新，并同时保证路径的诊断信息正确。

接下来我们证明满足结点间五个性质的关键树一定是正确的，

定理 3.1　满足结点间的层次性、充分性、不相容性、相似性和一致性的关键树$CtT(L_C)$关于运行日志集L_C一定是正确的。

证明　我们将从两方面来证明$\forall l = (o_l, t_l) \in L_C : \Delta_C(o_l) = t_l$：

1. 首先运行日志集L_C中的每条日志$l \in L_C$都存储在关键树的叶子结点中。充分性保证了每个叶子结点的父结点子观察都关于l关键。层次性保证了关键结点的祖先结点的子观察都比关键观察严格抽象，因此日志l匹配关键结点到根结点的所有子观察，即l匹配叶子结点所在路径。不相容性保证了关键结点的子观察之间都是不相容的，即

不存在两相同的路径。相似性保证了该路径只对应一个故障模式。一致性则保证该路径不会再延伸。因此l的故障模式t_l一定在诊断信息内，即$t_l \in \Delta_C(o_l)$。

2. 其次我们假设存在另一条路径使得l匹配给路径，但该路径对应的故障模式非t_l。则该路径的关键结点即匹配l又匹配本身的叶子日志，根据定义2.14，该路径的诊断信息应该为t_l和叶子结点日志标签的集合。这显然与关键观察的定义相违背，故一定不存在路径即被o_l匹配但故障模式又非t_l。

最后我们得到$t_l \in \Delta_C(o_l)$且$\nexists P_C \in CtT(L_C) : o_l \in P_C \wedge T_{P_C} \neq t_l$。根据$\Delta_C(o_l)$的定义2.14，$\Delta_C(o_l) = t_l$成立。因此满足结点间层次性、充分性、不相容性、相似性和一致性的关键树$CtT(L_C)$关于运行日志集L_C一定是正确的。 ■

定理3.1使得关键树在构造过程中无需时刻验证正确性，只需保证构造出来的关键树满足以上五个性质即可。

例 3.1 以3.2节给定的关键树为例解释说明基于关键树的诊断问题，示例关键树如下：首先，毫无疑问的是运行日志集L_C中并不存在冲突日志。其次对于日

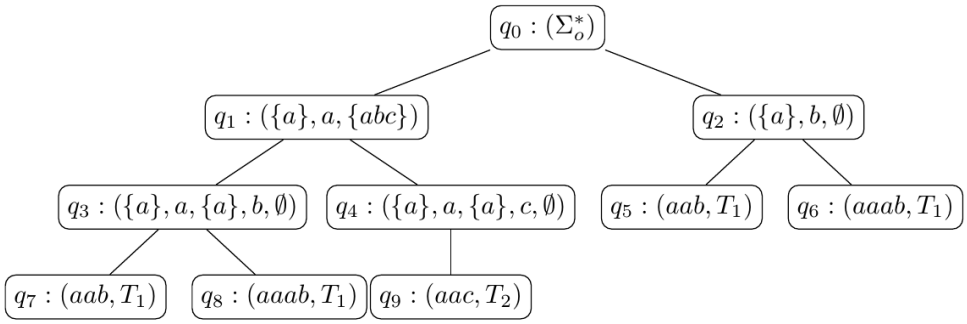


图 3-2 运行日志集$L_C = \{(aab, T_1), (aaab, T_1), (aac, T_2)\}$的关键树$CtT(L_C)$

志序列$aab \in S$，$CtT(L_C)$中存在路径$P_{C1} = q_0 \to q_1 \to q_3$和$P_{C2} = q_0 \to q_2$被$aab$匹配，且其它路径都不被$aab$匹配；而$P_{C1}$对应的故障模式为$T_1$—即路径叶子结点$q_7, q_8$的日志标签，$P_{C2}$对应的故障模式也为$T_1$，故而运行序列$aab$基于关键树的诊断信息为$\Delta_C(aab) = T_1$。同理对于日志序列$aaab$和$aac$有$\Delta_C(aaab) = T_1$，$\Delta_C(aac) = T_2$，根据定义3.4，$L_C$中所有日志序列的诊断信息都是正确的，因此关键树$CtT(L_C)$是正确的。3.2节详细证明了$CtT(L_C)$满足结点间的五个性质，由此同样也可以推测出$CtT(L_C)$是正确的。

3.4 基于关键树首次匹配路径的诊断问题

由于关键树仅保证兄弟关系的内部结点间不存在祖孙关系，因此在关键树中可能存在两内部结点，它们位于两条不同的路径，但它们的子观察是存在祖孙关系的。这

些冗余路径的诊断信息是相同的，它们所蕴含的序列特征也是相同的，因此这些路径所匹配的序列范围也是相同的。由于在查询关键树时需要对这些冗余路径进行若干次重复且无效的查询，因此这些冗余路径在保证关键树正确性的同时也给关键树的构建和查询造成了大量不必要的时间和空间消耗。对于这些冗余子观察，我们只需保留第一个及其路径即可，其余的可以忽略。基于此原因，我们提出基于首次匹配的关键树诊断信息如下：

定义 3.6 给定运行日志集合L及其关键树$CtT(L)$，假设关键树中的路径数目为v，将关键树中由左至右的路径分别标记为$P_C^1, P_C^2, \ldots, P_C^v$。对任意的系统运行序列$o$，若关键树中存在路径$P_C^j$，$j \in \{1, 2, \ldots, v\}$使得序列$o$匹配路径$P_C^j$上所有子观察，则$\Delta_C^1$返回第一条匹配路径对应的故障模式：

$$\begin{aligned}\Delta_C^1(o) = T_i \Leftrightarrow (\exists j \in \{1, \ldots, v\} : o \in P_C^j) \\ \wedge (\nexists k \in \{1, \ldots, j-1\} : o \in P_C^k) \\ \wedge T_{P_C^j} = T_i, i \in \{1, 2, \ldots, u\}\end{aligned} \tag{3.2}$$

特别的，当$CtT(L)$中不存在任何一条路径被o匹配，则$\Delta_C^1(o) = T_0$。

相比定义2.14，基于首次匹配的诊断信息仅返回第一条匹配路径所对应的故障模式，而忽略之后所有的匹配路径的故障模式。

命题 3.4 若一个关键树$CtT(L)$是正确的，则$CtT(L)$中所有的叶子结点日志序列基于首次匹配的诊断信息都是正确的，即

$$\forall q_l \in Q_L, \phi(q_l) = (o_l, t_l) : \Delta_C^1(o_l) = t_l$$

证明 若关键树是正确的，则所有的叶子日志序列的诊断信息都是正确的，即$\forall q_l \in Q_L, \phi(q_l) = (o_l, t_l) : \Delta_C(o_l) = t_l$，根据定义3.2，$\Delta_C$返回的所有匹配路径的故障模式，因此$\Delta_C^1(o_l) = t_l$成立。 ■

更进一步，我们定义关键树关于L_C是$F-$正确如下：

定义 3.7 一个关键树$CtT(L_C)$关于运行日志集L_C是$F-$正确的，当L_C中的所有日志序列基于首次匹配的诊断信息都是正确的，即

$$\forall l = (o_l, t_l) \in L_C : \Delta_C^1(o_l) = t_l$$

定义关键树是$F-$正确如下：

定义 3.8 一个关键树$CtT(L)$是$F-$正确的当$CtT(L)$中所有叶子结点日志序列基于首次匹配的诊断信息都是正确的，即

$$\forall q_l \in Q_L, \phi(q_l) = (o_l, t_l) : \Delta_C^1(o_l) = t_l$$

不同于正确性要求对于每条日志序列o_l在关键树中的所有被o_l匹配的路径对应的故障模式必须都是日志标签，$F-$正确性仅要求每条日志序列o_l在关键树中的第一条被o_l匹配的路径对应的故障模式为日志标签。因此满足正确性的关键树一定是$F-$正确的，但反之不亦然。

推论 3.4 $CtT(L)$的$F-$正确性是正确性的必要条件。

$F-$正确性在忽略冗余子观察的同时也遗失了部分诊断信息，但正如前面所言，冗余子观察所蕴含的序列特征往往是相同的，也就是说$F-$正确性同样保证了关键树保留了大部分的序列特征，故而利用$F-$正确性的关键树推测运行序列下系统的故障模式是完全可信的。

类似定义3.4和定义3.5，接下来我们给出$F-$正确关键树的判定方法。

命题 3.5 若对于运行日志集L_C中的每一个运行日志l,若l匹配的第一条路径上的所有结点都满足层次性、充分性、不相容性、相似性和一致性，则关键树$CtT(L_C)$关于运行日志集L_C是$F-$正确的。

命题 3.6 若对于关键树$CtT(L)$中所有叶子结点日志l，若l匹配的第一条路径上的所有结点都满足层次性、充分性、不相容性、相似性和一致性，则关键树$CtT(L)$是$F-$正确的。

3.5 本章小结

本章主要提出并介绍了关键树这样一个数据结构，并详细解释了层次性、充分性、不相容性、相似性和一致性的概念。并介绍了正确关键树和$F-$正确关键树的定义及相关定理。关键树是本文最重要的一个数据结构，本文提出的故障诊断方法也主要是围绕关键树的构造及其查询展开。

第 4 章　基于关键树的故障诊断算法

如何设计基于关键树的故障诊断算法是本文的主要研究内容，本章给出了我们的解决方法—基于关键树的故障诊断算法，包括CTT算法和$FCTT$算法。CTT算法构造出正确关键树并通过查询算法得到故障诊断结果，而$FCTT$算法构造出$F-$正确关键树，并同样通过查询算法得到故障诊断结果，CTT算法与$FCTT$算法的唯一区别在于CTT算法更新整个关键树而$FCTT$算法只更新运行日志的首次匹配路径。首先我们给出构造算法的主程序；接着介绍正确关键树更新程序的流程及其实现，还有$F-$正确关键树的更新程序；然后介绍结点更新程序的流程及其实现；再之后给出关键观察生成程序的流程及其实现的。当构造算法介绍完后，我们给出关键树查询算法的实现。最后我们给出例子来说明正确关键树和$F-$正确关键树是如何构造的。

4.1　关键树的构造算法

构造算法从一个初始的关键树出发，删除运行日志集中的所有冲突日志后对其中的每条日志查询现有关键树，找到违反性质的结点（主要是充分性），然后更新该结点使得该结点符合要求，待运行日志集中的所有运行日志查询完毕即可得到要求的关键树。值得注意的是，构造算法在查询关键树时并非基于完整的运行日志集，而是仅基于已查询的运行日志组成的集合。

初始的关键树由两个结点组成，标签为子观察(Σ_o^*)的根节点，和标签为运行日志(ε, T_0)的叶子结点，如图4-1所示。毫无疑问，初始的关键树满足五条性质，对于运行日志集$L = \{(\varepsilon, T_0)\}$，子观察$(\Sigma_o^*)$关于$(\varepsilon, T_0)$是关键的。

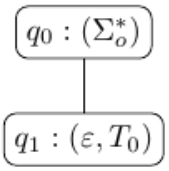


图 4-1 初始关键树

在初始关键树的基础上，利用运行日志集中的每个运行日志对关键树进行更新，得到构造算法的主程序（算法1）。

构造算法接受一个运行日志集合L作为输入，最后输出L对应的关键树$CtT(L_C)$。$initialize$函数初始化关键树。$removeConflict$函数删除给定运行日志集中的冲突日志，同时也删除集合中重复的日志，最后返回结果L_C。程序$update$利用运行日志对

Algorithm 1 关键树的构造算法

Input: 运行日志集L

Output: 关键树$CtT(L_C)$

1: set $L_C = removeConflict(L)$
2: $initialize(CtT(L_C))$
3: **for all** $l \in L_C$ **do**
4: $\quad update(CtT(L_C), l)$
5: **end for**
6: **return** $CtT(L_C)$

现有关键树进行更新。当L_C所有的运行日志都已被用于更新后，算法便得到了关键树$CtT(L_C)$。

$initialize$函数和$removeConflict$函数的实现非常简单。接下来我们只介绍构造算法的核心程序—关键树的更新程序$update$。根据定义3.5和定义3.8，$F-$正确关键树与正确关键树的区别在于对于关键树中所有的日志序列，前者仅要求首次匹配路径的诊断信息是正确的，而后者要求所有匹配路径的诊断信息都是正确的。因此$F-$正确关键树的构造与正确关键树的构造除更新程序外其它程序和函数都相同。更新程序$update$也因此分为正确关键树的更新程序和$F-$正确关键树的更新程序。

4.1.1 正确关键树的更新程序

正确关键树更新程序确保在初始关键树的基础上，每次更新后得到的关键树依然满足五条性质。由于初始关键树是正确的，因此运行日志集的变化只会影响关键树中那些与改变的运行日志相关的关键结点和叶子结点，对其它结点不会有任何影响。所以更新程序只需保证现有关键树中那些与改变的运行日志相关的结点在更新后满足五条性质即可。一般的，这些相关结点是指当运行日志集改变时，现有关键树中那些不再满足充分性和相似性的结点及其父结点、孩子结点。事实上，运行日志集的改变不会对层次性、不相容性和一致性造成影响，后者只与关键树结构的改变有关。总而言之，更新程序必须保证在更新完成后，这些相关结点以及新生成的结点满足五条性质，从而使得关键树在更新完成后保持正确。为方便区分，后文将这些相关结点称为待更新结点。

1. 硬匹配

在正式给出正确关键树的更新程序之前，我们还需要介绍**硬匹配**的概念。正如2.3.1节所言，子观察中的硬事件描述了系统的序列特征。因此我们便在匹配的基础

上提出硬匹配的概念，即若子观察的所有硬事件在序列中是按序发生的，则称该序列硬匹配该子观察。

定义 4.1 *对于子观察$\theta = (y_0, x_1, y_1, \ldots, x_n, y_n)$，标注由$\theta$的硬事件序列组成的子观察为$\theta_x = (\Sigma_o^*, x_1, \Sigma_o^*, \ldots, x_n, \Sigma_o^*)$。若序列$o$满足$o \in \theta_x$，则称序列$o$硬匹配子观察$\theta$。*

运行序列o硬匹配某子观察，则o一定硬匹配所有与该子观察硬事件序列相同的子观察。硬匹配忽略子观察中的所有软事件，当运行序列按序发生了子观察的硬事件序列时，就认为该运行序列符合子观察存储的序列特征。为方便区别硬匹配与匹配的概念，我们将需要符合子观察的软事件的匹配称为**全匹配**。显然，子观察被运行序列硬匹配是被该序列全匹配的必要条件。

推论 4.1 *子观察被运行序列硬匹配是被该运行序列全匹配的必要条件。*

若日志序列硬匹配某子观察，且该子观察的诊断信息包含该日志的日志标签，则说明该子观察存储了该日志下系统的运行特征，尽管该日志序列可能不全匹配该子观察，但我们可以通过完善软事件的方式完善该子观察，最终使得该日志序列同样可以全匹配该子观察。

采用硬匹配而非全匹配的方式来查找待更新结点可以避免大量冗余子观察的产生。若我们要以全匹配的方式查找待更新结点，会导致尽管运行序列符合子观察的序列特征，但由于不满足软事件要求，该序列就无法以该子观察为基础生成新的关键观察，从而使得关键树中存在许多子观察拥有相同的硬事件序列，造成大量冗余。另一方面，这些冗余子观察之间往往只有软事件有细微差别，对于给定的运行日志集而言，它们所匹配的运行日志往往是相同的，含有的诊断信息往往也是相同的。采用硬匹配使得我们在更新关键树时既能保证子观察不会遗失运行日志的序列特征，还能避免冗余子观察的存在。

2. 程序流程

正确关键树更新程序如程序1

*update*的大致流程如下：程序首先找到第一个被日志序列o_l硬匹配的路径并返回路径的最后一个结点q_{cur}，由于关键树至少拥有一个内部结点和叶子结点（命题3.1），故此时关键树中一定存在这样一个路径。接着判断该关键结点及其孩子结点是否违背了结点间的充分性和相似性，若是则对这些结点进行更新，在更新完这些结点后返回q_{cur}。由于我们采用的是硬匹配的方式查找路径，因此子观察$\psi(q_{cur})$可能存在着不完善的情况，即日志序列硬匹配但不全匹配该子观察，此时我们需要通过完善软事件的方式完善该子观察。之后程序继续以q_{cur}为起点查找下一个被o_l硬匹配的路径，然后更

Procedure 1 正确关键树更新程序

Input: 关键树$CtT(L_C)=(Q,S,\psi,\phi,q_0)$,待更新的运行日志$l=(o_l,t_l)$

Output: 更新后的$CtT(L_C)$

```
1: function UPDATE(CtT(L_C), l)
2:     set q_cur = matchPath(q_0, o_l)
3:     while q_cur ∈ Q do
4:         updateNode(q_cur, l)
5:         optimize(q_cur, o_l)
6:         set q_cur = nextPath(q_cur, o_l)
7:     end while
8:     return CtT(L_C)
9: end function
```

新下一个结点。程序循环查找过程和更新过程直到现有关键树中的所有待更新结点更新完成，此时我们得到的关键树便是运行日志l更新后的关键树。

3. 函数的作用及其流程

*update*程序中函数作用和实现大多十分简单，这里只作简单说明。

*matchPath*函数从根结点q_0出发查找第一个被o_l硬匹配的路径，返回路径的最后一个结点。流程如下：从根结点出发对路径上的每一个内部结点判断序列o_l是否硬匹配其子观察，若匹配则继续查询第一个孩子结点，否则查询下一个兄弟结点。若当前结点非内部结点，则说明已达到路径末尾。此时需要区分当前结点是叶子结点还是不属于结点空间Q的空结点，若前者直接返回叶子结点的父结点即可；若后者，由于只有当上一个结点为兄弟结点且o_l不硬匹配其结点子观察时，当前结点才为空结点，因此需要先退回到上一个结点再返回其父结点。无论是哪种情况，当前结点非内部结点都表示路径结束，不同之处在于当前结点为叶子结点时，运行序列o_l符合当前路径存储的序列特征；而当前结点为空结点时，当前路径缺少o_l的序列特征，或者说，缺少运行序列o_l的关键观察。

*optimize*函数利用日志序列o_l完善子观察$\psi(q_{cur})$，使得o_l能既能硬匹配也能全匹配该子观察。流程如下：首先判断o_l是否全匹配子观察，若能则说明该子观察是完善的，直接返回。否则收集两相邻硬事件间发生的所有事件的集合，接着并入对应的软事件中，可通过收集每个硬事件在序列o_l中首次/最后一次按序发生的位置，接着将两位置间的所有事件的集合并入对应的软事件中实现。

与*matchPath*函数类似，*nextPath*函数同样是查询被o_l硬匹配的路径并返回路径的

最后一个结点，不同之处在于$nextPath$函数函数是从当前结点q_{cur}出发，通过回溯查询下一条硬匹配路径。流程如下：从当前结点q_{cur}出发查找下一个兄弟结点，若下一个兄弟结点存在且序列o_l硬匹配该结点，则查询兄弟结点的第一个孩子结点。否则若下一个兄弟结点存在但序列o_l不硬匹配该结点，则继续查询下一个兄弟结点。若下一个兄弟结点不存在，则回溯到父结点并查询父结点的下一个兄弟结点。直到查询结点非内部结点，新的被o_l硬匹配的路径就找到了，接下来的处理与$matchPath$函数类似。若已查询到根结点无法再继续查询时，返回空结点表示更新结束。

$updateNode$程序对每个待更新的结点进行更新，从而使得这些不满足充分性和相似性的结点在更新完毕后满足五条性质，4.1.3节我们将详细解释该程序是如何处理这些待更新结点的。

4.1.2 $F-$正确关键树的更新程序

回顾第3.4节提及的内容，关键树中存在着冗余子观察，这些子观察在保证关键树正确性的同时也给关键树的构建和查询造成了大量的空间和时间冗余，另一方面也导致我们在查询关键树时需要对关键树进行若干次重复无效的查询匹配。因此我们提出了基于首次匹配的正确关键树的概念，当所有叶子结点日志序列在关键树中的第一条匹配路径的诊断信息为日志标签时，我们就说该关键树是$F-$正确的。$F-$正确关键树在减少关键树的时间空间消耗的同时也保留了故障序列大部分的序列特征。

不同于正确关键树的更新程序，$F-$正确关键树只需保证所有叶子结点日志序列在关键树中的第一条匹配路径的诊断信息为日志标签，因此$F-$正确关键树的更新程序第一个待更新结点即可。

Procedure 4 $F-$正确关键树更新程序

Input: 关键树$CtT(L_C)=(Q,S,\psi,\phi,q_0)$,待更新的运行日志$l=(o_l,t_l)$

Output: 更新后的$CtT(L_C)$

1: **function** UPDATE($CtT(L_C),l$)
2: set $q_{cur}=matchPath(q_0,o_l)$
3: $updateNode(q_{cur},l)$
4: **return** $CtT(L_C)$
5: **end function**

$F-$正确关键树更新程序函数的作用及流程与正确关键树更新程序相同，这里不再赘述。

4.1.3 关键树的结点更新程序

对待更新结点q_u，结点更新程序判断q_u关于运行日志$l = (o_l, t_l)$是否违背充分性和相似性，若是则更新q_u及其孩子结点，最终使得这些结点满足五条性质。

1. 程序流程

关键树的结点更新程序流程如下：首先程序接受待更新的结点q_u和待更新的运行日志l，接着根据q_u的性质分为如下几种处理方法：

1 若q_u为关键结点且孩子结点中存在于与日志l冲突的标签，则不仅说明冲突标签的叶子结点是无效的，还说明该关键观察不再关于q_u叶子结点标注的运行日志充分。因此我们需要先删除冲突的叶子结点；接着生成原非冲突叶子结点运行日志的新关键观察；然后新建标注为新关键观察的内部结点并新建标注为对应的运行日志叶子结点；最后将它们按结点间性质的要求插入到关键树待更新结点的位置处。

2 若q_u为关键结点且其孩子结点标签非日志标签，则说明结点q_u违背了结点间的充分性。因此需要在旧关键结点的基础上生成q_u原叶子结点的新关键观察，以及l的新关键观察。由于原叶子结点的日志标签都是相同的，因此只需确保原叶子结点的新关键观察不被o_l匹配即可。最后新建结点并插入待更新结点的位置处。

3 若q_u为关键结点且其孩子结点标签等于日志标签t_l，但该结点没有被标注为l的孩子结点，说明该路径遗失了具有相同特征的运行日志。因此将标注为l的叶子结点加入到关键结点q_u的孩子结点中即可。

4 若q_u为关键结点且存在孩子结点标注为l，则无需更新。

5 若q_u非关键结点—即其孩子结点非叶子结点，则说明该路径虽然被l匹配但q_u的孩子结点中不存在被l匹配的子观察，也就是说路径中不存在l的关键结点，因此需要生成l新关键观察并将对应结点插入到待更新结点的处置处。

$updateNode$的具体实现程序如程序5，程序中$q_u \to child$表示取结点q_u的第一个孩子结点，$q_u \to children$表示取结点q_u的所有孩子结点，$q_u \to leaf$表示以q_u为根结点的子树的第一个叶子子孙结点。

2. 程序分析

本小节我们对程序5进行具体分析。首先分析第一种情况。第3行$q_u \to child \in Q_L$表示结点q_u的孩子结点是叶子结点，因此q_u一定为关键结点，而第4行$t_l \neq \rho_t(q_u \to child)$表示$l$的日志标签与叶子结点的日志标签不一致，此时根据是否存在冲突日志分为如下两种情况：

Procedure 5 关键树结点更新程序

Input: 待更新的结点q_u,待更新的运行日志$l = (o_l, t_l)$

Output: 更新后的结点q_u

function UPDATENODE(q_u, l)
  set $\mathbb{O} = \emptyset$
  if $q_u \to child \in Q_L$ **then**
    if $t_l \neq \rho_t(\phi(q_u \to child))$ **then**
      if $(\exists q_l \in q_u \to children : o_l = \rho_o(\phi(q_l))$ **then**
        $removeNode(q_l)$
      else
        $\mathbb{O} = \mathbb{O} \cup \bigcup_{q_l \in q_u \to children} genCritical(o_l, \psi(q_u), \rho_o(\phi(q_l))$
      end if
      $\mathbb{O} = \mathbb{O} \cup \bigcup_{q_l \in q_u \to children} genCritical(\rho_o(\phi(q_l)), \psi(q_u), o_l)$
      $distinct(\mathbb{O})$
      $createAndInsert(q_u, \mathbb{O}, \bigcup_{q_l \in q_u \to children} \phi(q_l) \cup l)$
    else if $(\forall q_l \in q_u \to children : l \neq \phi(q_l)$ **then**
      set $q_{newl} = createLeaf(l)$
      $insertChild(q_u, q_{newl})$
    end if
  else
    set $\mathbb{O} = genCritical(o_l, \psi(q_u), \rho_o(\phi(q_u \to leaf)))$
    $distinct(\mathbb{O})$
    $createAndInsert(q_u, \mathbb{O}, \{o_l\})$
  end if
  return q_u
end function

1 若如5所示，叶子结点中存在与o_l相同的日志序列，即$o_l = \rho_o(\phi(q_l)) \wedge t_l \neq \rho_o(\phi(q_t))$。根据定义3.3，$l$和$\phi(q_t)$为冲突日志。因此$removeNode$函数首先移除这些冲突叶子结点，而后对$q_u$的每个孩子叶子结点$q_l$调用$genCritical$函数在$\psi(q_u)$的基础上生成关于$\phi(q_l)$关键的关键观察，这里只需找到与序列$o_l$不同的序列特征即可;

2 若非冲突日志，此时我们不仅需要生成关于$\rho_o(\phi(q_l))$关键的关键观察，如第8行所示，还需要生成关于o_l关键的关键观察。同样的，这里只需找到与序列$\rho_o(\phi(q_l))$不同的序列特征即可。

找到所有的关键观察后，第12行的$createAndInsert$函数就可以生成关键结点和匹配日志标注的叶子结点，然后删除原叶子结点，最后将关键结点插入成q_u的孩子结点并将相应的叶子结点插入成关键结点的孩子结点，此时q_u及其子孙满足一致性和充分性。另一方面，$genCritical$函数保证新关键观察一定不重复且互不存在祖孙关系，因此不相容性得到了满足。再者，新关键观察是在$\psi(q_u)$的基础上通过添加新的序列特征得到的，故而新关键观察一定比$\psi(q_u)$严格具体，因此满足了层次性。最后充分性保证了关键结点的孩子叶子结点拥有相同的关键观察和日志标签，故而满足相似性。此种情况的处理能保证结点更新后能满足五条性质。

然后分析第二种情况。若q_u为关键结点且l的日志标签与叶子结点的日志标签一致，并且q_u的孩子结点中不存在被l标注的叶子结点，如13行所示。根据命题3.3，由于o_l匹配q_u且l标签与$\rho_t(\phi(q_l))$相同，所以关键观察$\psi(q_u)$同样是l的关键观察。因此我们只需调用$createLeaf$函数创建一个标注为l的叶子结点并利用$insertChild$函数将该叶子结点插入成q_u的孩子结点。由于l与原叶子结点拥有相同的关键观察和日志标签，故不违背结点间的相似性，原有的其它四条性质更不会被违背。

最后分析第三种情况。若$q_u \rightarrow child$非叶子结点，这意味着q_u非关键结点。如之前所述，该路径虽然被l匹配但路径中不存在l的关键结点。因此我们需要利用$genCritical$函数生成关于l的关键观察，如18行所示。值得注意的是，由于l不匹配任何一个已知的比$\psi(q_u)$严格具体的子观察，所以该运行日志蕴含的新的关键观察与现有的关键观察即不重复又没有祖孙关系（否则一定匹配现有子观察），因此这里我们只需与q_u的一个子孙叶子结点的日志序列进行比较即可。$createAndInsert$函数将生成的关键观察标志的关键结点和它的运行日志l标注的孩子叶子结点插入成q_u的孩子结点。由于q_u的原孩子结点非叶子结点，因此插入关键结点后并不会违背一致性。新关键结点的孩子结点为l标注的叶子结点也保证了充分性。新关键结点的孩子结点的运行日志都为l，因此同样满足相似。至于不相容性和层次性同上。

值得注意的是，在利用$genCritical$函数得到新关键观察集合$\mathbb{O}$后，为防止关键树

中出现重复子观察，我们需要调用*distinct*函数删除$\mathbb{O}$中那些已存储在关键树中的子观察。也就是说，只有当新关键观察不在关键树中时，我们方才将之插入树中。

综上所述，*updateNode*程序保证了待更新结点及其子孙结点更新完毕后满足结点间的五条性质。

推论 4.2 *待更新结点及其子孙结点在执行updateNode程序完毕后满足结点间的层次性、充分性、不相容性、相似性和一致性。*

由于*update*程序是通过对每一个需更新结点调用*updateNode*来实现的，因此*update*后的所有的需更新结点都满足结点间的五条性质，由于关键树中的其它结点不违反性质，因此：

推论 4.3 *执行update程序后的关键树满足结点间的层次性、充分性、不相容性、相似性和一致性。*

3. **函数的作用及其实现**

同样的，*updateNode*程序中函数的作用及其实现大都比较简单，这里只作简单说明。

首先是*removeNode*函数，这个函数接收一个结点q并删除该结点，若该结点有孩子结点，则一并删除该结点的所有子孙结点。若该结点的父结点及连续的祖先结点都只有一个孩子结点，则一并删除。若删除到了根结点，则保留根结点并添加标签为(ε, T_0)的叶子结点。*removeNode*函数因此便保证了关键树中至少存在一个内部结点和叶子结点，同时还能确保关键树中不会存在孤立结点（即从根结点出发查询不到的结点）。

*createLeaf*函数接收一个运行日志并返回标注为该运行日志的叶子结点。*insertChild*函数将第二个结点插入为第一个结点的孩子结点，若该父结点已存在孩子结点，则插入到最后一个孩子结点之后。

*createAndInsert*函数对关键观察集合中的每个关键观察调用*createInner*函数建立标注为该关键观察的内部结点，接着对运行日志集合中所有匹配该关键观察的运行日志调用*createLeaf*函数建立标注为此运行日志的叶子结点，然后调用*insertChild*将所有叶子结点插入成该内部结点的孩子结点，最后调用*insertChild*函数将所有内部结点插入成父结点的孩子结点。

*distinct*函数删除$\mathbb{O}$中那些已在关键树中的子观察。该函数从根结点开始查询关键树，若当前结点子观察比查询子观察严格抽象，则查询当前结点的孩子结点，否则查询当前结点的兄弟结点。若兄弟结点不存在，则退回到当前结点的父结点，并查询父

结点的兄弟结点。当查询到关键树中的某个子观察与查询子观察相同时，则从$\mathbb{O}$中删除该子观察，查询结束。否则若回到了根结点，则说明无重复子观察，保留该子观察，查询结束。

$genCritical$函数在给定子观察的基础上生成关于给定运行日志的互不为祖孙关系的关键观察。$genCritical$函数通过在子观察的基础上查找属于运行日志且不属于比较运行日志的非公共子序列来实现。分析结点更新程序第10行，对与q_u的每个孩子叶子结点，调用$genCritical$函数在子观察$\psi(q_u)$的基础上生成$\rho_o(\phi(q_l))$的新关键观察，该关键观察可以通过与序列o_l比较来找到，并集$\cup$保证不存在重复的关键观察。在所有新关键观察找到后，调用$createAndInsert$函数将关键结点和叶子结点插入为q_u的子孙结点。相比其它仅仅是对关键树的结构进行调整的函数，生成关键观察的$genCritical$函数需要在4.1.4节中详细说明。

4.1.4 关键观察生成程序

回顾关键观察的定义2.16，一个子观察θ是l的关键观察当θ关于l是充分的，同时存在θ的孩子子观察关于l不是充分的。对于一个运行日志而言，若某子观察匹配该运行日志但不匹配任何非相同日志标签的运行日志，则该子观察关于该运行日志充分。更进一步，对于一个关于运行日志l不充分的子观察θ'，我们可以在比θ'严格具体的子观察中找到匹配l但不匹配其它匹配θ'的子观察，若该子观察是所有严格具体的子观察中最短的子观察，则该子观察是关键观察（因为该子观察一定存在一个孩子子观察关于l是不充分的，否则它与最短的子观察相矛盾）。因此，如何在最少的时间和空间内找到这些符合要求的子观察便成为了关键观察生成程序中需要重点解决的问题。

1. 程序流程

根据定义2.4，假设$\|\theta\| = n, \|\theta'\| = l$，$\theta'$严格具体于$\theta$当至少存在一个$j \in \{0, \ldots, l\}, k \in \{0, \ldots, n\}$满足$y'_j \supseteq y_k \cup \{x_k\} \cup y_{k+1}$。严格具体的实现在于提取出原子观察中被忽视的硬事件，因此我们可以通过查找仅运行日志能匹配的硬事件序列来生成关键观察。

对于运行日志$l = (o_l, t_l)$，若子观察能被日志标签非t_l的日志硬匹配，则该子观察关于l是不充分的。为找到一个关于l充分的子观察，我们先找到一个能被l匹配但不能被非t_l的运行日志匹配的硬事件序列，而后以日志序列o_l中发生在两硬事件间的所有事件填充软事件。如此便一定能确保该子观察能被l全匹配，同时不会被日志标签非t_l的日志匹配，根据定义2.15，该子观察关于l是充分的。

这样如何找到符合要求的子观察的问题便能转换为如何找到符合要求的最短硬事件序列的问题。首先在原子观察中查找一个在日志序列o_l中发生了但在原子观察中不存在的事件，由于该事件是o_l特有的，根据定义2.2，该事件是一个硬事件。这类新硬事

件可以通过比较原子观察的软事件和序列o_l在两相邻硬事件间发生的事件集合找到，若无法从中找到，则进一步比较日志序列o_l与原日志序列，从而找到仅发生在o_l新硬事件序列。程序的具体实现见程序6，其中比较序列是指匹配原子观察的日志序列。

Procedure 6 关键观察生成程序

Input: 运行序列$o_l = e_1e_2\dots e_m$,原子观察$\theta = (y_0, x_1, \dots, x_n, y_n)$,比较序列$o' = e'_1e'_2\dots e'_z$

Output: o的关键观察集合$\mathbb{O}$

1: **function** GENCRITICAL($o_l, \theta, \mathcal{O}$)
2: set $h = (x_1, x_2, \dots, x_n), H = \emptyset$
3: set $lpos = lfind(o_l, h), rpos = rfind(o_l, h)$
4: **while** $j \in \{0, 1, \dots, n\}$ **do**
5: set $E = \bigcup_{k=lpos_j}^{rpos_j} e_k$
6: **for all** $e_{new} \in \{e | e \in E \wedge e \notin y_j\}$ **do**
7: set $H = H \cup (x_1, \dots, x_j, e_{new}, x_{j+1}, \dots, x_n)$
8: **end for**
9: **end while**
10: **if** $H \neq \emptyset$ **then**
11: **return** $\mathbb{O} = \bigcup_{h_{new} \in H} fillSoft(h_{new}, o_l)$
12: **end if**
13: set $lpos' = lfind(o', h), rpos' = rfind(o', h)$
14: **while** $j \in \{0, 1, \dots, n\}$ **do**
15: set $o_{sub} = e_{lpos_j}e_{lpos_j+1}\dots e_{rpos_j}, o'_{sub} = e'_{lpos'_j}e'_{lpos'_j+1}\dots e'_{rpos'_j}$
16: set $SEQ = diffSubSeq(o_{sub}, o'_{sub})$
17: set $H = H \cup \bigcup_{seq_{new} \in SEQ} insertSeq(h, seq_{new}, j)$
18: **end while**
19: **return** $\mathbb{O} = \bigcup_{h_{new} \in H} fillSoft(h_{new}, o_l)$
20: **end function**

2. **程序分析**

接下来我们详对关键观察生成程序进行具体分析。

第2行到第7行比较原子观察与日志序列，并从中找到新硬事件。首先我们提取出原子观察的硬事件序列$h = (x_1, x_2, \dots, x_n)$。然后找到硬事件序列$h$在序列$o_l$发生的位置，函数$lfind$返回首次发生的位置，函数$rfind$返回最后一次发生的位置，两函数返回的都是一个位置序列$pos = (pos_0, pos_1, \dots, pos_n)$，$pos$的第$j \in \{0, 1, \dots, n)$个值表示事

件x_j在序列o_l中首次/最后一次按序发生的位置。确定硬事件在序列o_l中发生的位置后，由于匹配子观察只要求硬事件一定发生且两相邻硬事件之间软事件中的事件可以发生任意次，所以我们需要收集两相邻硬事件间可能发生的所有事件的集合。如第5行所示，其中$lpos_j$指前一个硬事件首次按序发生的位置，$rpos_j$指后一个硬事件最后一次按序发生的位置。

第2.3.1节指出软事件中的事件在序列中可发生任意次，这其实也意味着不允许发生不在软事件内的事件，因此子观察软事件一定包含所有匹配该子观察的序列在两硬事件间发生的事件的集合。

推论 4.4 *若子观察$\theta = (y_0, x_1, y_1, \ldots, x_n, y_n)$，则$\forall j \in \{0, \ldots, n), \forall o \in \Sigma_o^* \wedge o \in \theta$，序列$o$发生在硬事件$x_j$和$x_{j+1}$之间所有可能的事件的集合一定是软事件$y_j$的子集。*

第5行中的事件集合E即为序列o_l发生在硬事件x_j和x_{j+1}之间所有可能的事件的集合，若事件在集合E中但不在对应的子观察软事件中，则该事件一定是新硬事件，因为加上该硬事件后的硬事件序列一定只被日志序列o_l匹配但不会被比较序列匹配。对每个找到的硬事件，将其插入到原硬事件序列h，由于该新硬事件是在硬事件x_j和x_{j+1}之间被发现的，正如第7行所示，将新硬事件插入到h的x_j和x_{j+1}之间生成新硬事件序列，最后并入新硬事件序列集合H。

此时若找到了新硬事件，即$H \neq \emptyset$，则只需增加一个新硬事件即可得到新硬事件序列，而无需再比较日志序列与比较序列。最后如第11行所示，调用$fillSoft$函数对每个新硬件序列$h_{new} \in H$以序列o_l相邻两硬事件间发生的事件填充软事件即可得到新的关键观察，最后返回得到的新关键观察集合。

若无法找到新硬事件，即$H = \emptyset$，这说明日志序列o_l在所有相邻硬事件间发生的事件同样在比较序列中发生了，因此若想要得到符合要求的硬事件序列，我们必须比较日志序列与比较序列并从中找到最短非公共子序列。对于比较序列o'，我们先找到硬事件序列h在序列o'中发生的位置，然后提取出日志序列o_l和比较序列o'中位于两相邻硬事件的最长子串（这里的串指字节串，下同），如第15行所示。得到两子串后，调用$diffSubSeq$函数找到属于o_l但不属于比较序列o'的最短子序列。最后调用$insertSeq$函数将该子序列插入原硬事件序列即得到新硬事件序列。由于两序列非相同序列，即$o_l \neq o'$，因此一定能找到至少一个新硬事件序列。最后调用$fillSoft$函数填充软事件得到新关键观察。

命题 4.1 *令日志序列为o_l，原子观察为θ，比较序列集合为$\mathcal{O}$，则对与任意一个由程序6生成的子观察θ_{new}，θ_{new}都满足$\theta \prec \theta_{new}$；且基于运行序列集合$\mathcal{O} \cup o_l$，$\theta_{new}$是序列$o_l$的关键观察。*

证明 新子观察是通过将新硬事件或非公共子序列插入原硬事件序列，然后填充软事件得到的。显然，根据定义2.4，$\theta \prec \theta_{new}$。

从程序$updateNode$可知，$genCritical$函数接收的第二个参数要么为旧关键观察，要么为无孩子结点被日志序列匹配的内部结点的子观察，而第三个参数为旧关键结点的原叶子结点日志序列集合。

1 若原子观察为旧关键观察，则在现关键树中，原子观察现只匹配原叶子结点日志序列和日志序列o_l，其中原叶子结点日志的日志标签都相同。由于新生成的子观察仅匹配日志序列o_l，因此该子观察一定关于o_l充分；

2 若原子观察非关键观察，则说明该子观察所在结点的所有孩子结点子观察都不被日志序列o_l匹配。因此$genCritical$函数生成的子观察一定是关于o_l充分的，否则若该子观察匹配其它非相同标签叶子日志序列。由于$genCritical$是生成序列的所有关键观察，且对任意一个日志序列，$update$函数查找所有的匹配路径，故而一定存在一个孩子结点子观察被日志序列o_l所匹配，这与所有孩子结点子观察都不被匹配相矛盾。

另一方面，由于$genCritical$程序通过查找最短非公共子序列方法得到要求的子观察，因此该子观察一定是$\psi(q_u)$中关于o_l充分的子观察中的最短子观察。根据关键观察的定义2.16，该子观察为关键观察。 ■

3. 函数作用及其实现

$lfind$函数和$rfind$函数的作用类似，两者都接受序列o_l和硬事件序列h，只不过前者查找关键事件序列首次按序出现的位置，后者查找关键事件序列最后一次按序出现的位置。$lfind$函数首先从序列第一个事件开始查找硬事件序列的第一个硬事件，找到后记录第一个硬事件首次发生的位置，接着从该位置开始查找第二个硬事件首次发生的位置，直到找到最后一个硬事件发生的位置。$lfind$函数则相反，其首先从序列最后一个事件开始反向查找硬事件序列的最后一个硬事件，记录第一次找到的位置，然后从该位置开始查找倒数第二个硬事件发生的位置，直到找到第一个硬事件。注意两函数返回的位置序列长度都为$n+1$而非n，因为位置序列需要加上序列起始位置或结束位置以方便得到第一个硬事件发生前/最后一个硬事件发生后的子串。

$insertSeq$函数将子序列seq_{new}插入原硬事件序列h的第j个硬事件之后，最后返回新硬事件序列h_{new}。而$fillSoft$函数函数返回一个硬事件为h_{new}，且能恰好被o_l匹配的子观察。该函数首先调用$lfind$函数和$rfind$函数找到硬事件序列h_{new}在序列o_l发生的位置。假设$o_l = e_1e_2\ldots e_m$，$lfind$和$rfind$返回的结果分别为$lpos$和$rpos$，则新子观察的

第j个软事件被设置为事件集合$\bigcup_{k=lpos_j}^{rpos_j} e_k$。待所有的软事件都被设置完毕，我们就得到了$o_l$的一个新关键观察。

$diffSubSeq$函数接收两个序列，并返回这两个序列的非公共子序列，且该子序列属于第一个序列但不属于第二个序列。该函数利用子序列自动机方法实现，该方法属于后缀自动机方法。子序列自动机是一个能够跑出一个串的所有子序列的有限状态机，基本思路是每个点存每一种字符下一次出现位置的点，从自动机的根出发到自动机上任意一点为原字符串的一个子序列。我们首先构造两个序列的子序列自动机，接着搜索两子序列自动机，查找是否存在一个状态在第一个子序列自动机中是可达的但在第二个子序列自动机中是不可达的，若存在则第一个子序列自动机中能到达该状态的串即为所求的非公共子序列。关于后缀自动机的构造算法详见M. Mohri发表与2009年发表的论文[73]，由于篇幅的关系这里不再详细解释。假设两序列的最长序列长为m，可观事件集的大小为l，则该函数的时间和空间复杂度都为$O(ml)$。

4. 示例

例 4.1 假设运行序列为$o_l = aac$，原子观察为$\theta = (\{a\}, a, \{ab\})$，比较序列为$o' = aab$，接下来我们展示$o_l$的关键观察生成流程。

1 原子观察硬事件序列为$h_\theta = (a)$；
2 查找h在o_l中发生的位置得到$lpos = (1, 1)$，$rpos = (2, 3)$；
3 统计相邻硬事件间所有可能发生的事件，在硬事件a之前可能发生的事件集合为$E_0 = \bigcup_{k=1}^{1} = \{a\}$；
4 比较E_0与y_0，而$E_0 = y_0 = \{a\}$，无新事件；
5 继续统计，$E_1 = \bigcup_{k=2}^{3} = \{ac\}$，比较$E_1$与$y_1$，得到一个在$E_1$中但没在$y_1$中的新事件$c$；
6 将新事件c插入h中，得到新硬事件序列$h_{new} = (a, c)$；
7 以aac填充软事件得到新关键观察$(\{a\}, a, \{a\}, c, \emptyset\}$。

例 4.2 若原子观察改为$\theta = (\{a\}, a, \{abc\})$，我们无法从原子观察中找到新的硬事件序列。此时我们需要比较运行序列aac和比较序列aab，找到最短非公共子序列。

1 第1、2步同上；
2 同第2步，h在o'中发生的位置为$lpos' = (1, 1), rpos' = (2, 3)$；
3 取硬事件a发生前的最长子串得$o_{sub} = a$，$o'_{sub} = a$，无非公共子序列；
4 取硬事件a发生后的最长子串$o_{sub} = ac$，$o'_{sub} = ab$，得到一个属于o_l但不属于o'的最短非公共子序列c；

5 将c插入h中得到新硬事件序列$h_{new}=(a,c)$;

6 最后填充软事件得到新的关键观察$(\{a\},a,\{a\},c,\emptyset\}$。

4.1.5 构造算法总结

本节详细介绍了关键树的构造算法。第4.1.1节详细解释了正确关键树的更新程序，4.1.2节给出了$F-$正确关键树的更新程序。构造函数对运行日志集中的每条运行日志调用$update$函数，$update$函数首先查询现关键树，接着对每个待更新结点调用$updateNode$函数。4.1.3详细介绍了$updateNode$函数的流程及其实现，$updateNode$函数根据待更新的结点的性质，通过删除冗余结点，添加新的关键结点和叶子结点等操作，使得待更新结点及其子孙结点在执行$updateNode$程序完毕后满足层次性、充分性、不相容性、相似性和一致性（推论4.2），进而使得执行$update$程序后的关键树也满足这些性质（推论4.3）。相较于$updateNode$程序中那些仅仅是对关键树的结构进行调整的函数，4.1.4节则详细描述了关键观察生成程序$genCritical$，在该节中，我们将查找运行序列的关键观察的问题装换为查找非公共子序列问题，并证明了由$genCritical$函数生成的子观察都是日志序列的关键观察（命题4.1）。

定理 4.1　给定运行日志集合L_C，正确关键树构造算法构造出的关键树$CtT(L_C)$关于L_C是正确的。

证明　由于$genCritical$函数生成的子观察都是关键观察（命题4.1），因此待更新结点在更新后都一定满足层次性、充分性、不相容性、相似性和一致性（推论4.2）。$update$函数对于现关键树中的每个待更新结点都调用$updateNode$函数，因此关键树在更新后也满足这五条性质（推论4.3）。所以当构造算法（算法1）对运行日志集L_C中的所有日志都调用$update$程序后，关键树仍然是满足五条性质的。根据定理3.1，关键树$CtT(L_C)$关于L_C是正确的。 ■

定理 4.2　给定运行日志集合L_C，$F-$正确关键树构造算法构造出的关键树$CtT(L_C)$关于L_C是$F-$正确的。

证明　算法4找到关键树中第一个被序列o_l匹配的路径，接着对路径上最后一个结点执行结点更新程序，从而保证了首次匹配路径上所有的结点都满足结点结点间的性质，根据命题3.5，$CtT(L_C)$是$F-$正确的。 ■

既然已经证明了算法1构造出的关键树是正确的，因此我们可以确定算法1为关键树的构造算法，接下来我们详细分析构造算法的算法复杂度。

定理 4.3　给定运行日志集L的大小为$\|L\| = n$，日志序列的最大长度为$\max(\|o\|) = m$，可观事件集Σ_o大小为$\|\Sigma_o\| = l$，正确关键树的构造算法在最差情况下的时间复杂度为$O(n^3m^4l^3)$，空间复杂度为$O(n^2ml)$。

证明　首先我们讨论$genCritical$函数的复杂度。由于可观事件集大小为l，因此新事件集合E的大小最大为l。考虑最差的情况，即$\|E\| = l, \|o_l\| = m$，则此时h的长度最大为$m/l - 1$，因此从h中最多生成$m - l$个新的硬事件序列。考虑到填充全部软事件需要遍历整个序列，因此$fillSoft$函数的时间复杂度为$O(m)$。最终得到从原子观察中生成关键观察所需的时间复杂度为$O(m^2 - ml)$。

再考虑需要比较日志序列与比较序列的情况。由于$\|L\| = n$，因此比较序列集合$\mathcal{O}$大小最大为n。假设此时h的长度为k，对于硬事件间的每一个子序列$o^j_{sub}, j \in \{0, \ldots, k+1\}$，假设其长度为$m_j$，则非公共子序列查找函数$diffSubSeq$时间复杂度为$O(m_jl)$，且每次调用能找到最多$l^2$个最短非公共子序列，考虑到对每个比较序列都会调用$diffSubSeq$函数$k+1$次，因此最多得到$\max((k+1)l^2) = ml$个新硬事件序列。又因为$\Sigma_{j=0}^{k+1} m_j = m$，因此在程序6第13行的$while$循环中，$diffSubSeq$所消耗的时间复杂度为$O(ml)$，而$insertSeq$函数至多被调用$ml$次。最终从比较序列中生成关键观察所需的时间复杂度为$O(m^2l)$。综上，最多有ml个新关键观察被生成，且$genCritical$函数的时间复杂度为$O(m^2l)$。

接下来分析$updateNode$函数的复杂度。相比其它几种情况，q_u是关键结点且日志标签不一致显然需要耗费更多的时间和空间。由于一个关键结点至多有n个叶子结点，因此程序5中第8行和第10行最多需要执行$genCritical$函数n次，因此$updateNode$函数的时间复杂度为$O(nm^2l)$。

$update$程序对于每个违背性质的结点调用$updateNode$函数使得每个结点满足结点间的五条性质。因为对于长度为m的序列而言，其最多产生ml个关键观察，因此关键树中至多存在着ml个关键结点，这意味着最多存在着ml个结点需要更新，程序1中第3行的$updateNode$函数循环最多执行ml次，因此$update$程序的时间复杂度为$O(nm^3l^2)$。

最后由于算法1是通过对运行日志集L_C中的每一个运行日志调用$update$程序实现的，因此算法1在最差情况下的时间复杂度为$O(n^2m^3l^2)$。

算法1的空间复杂度分析较为简单，因为算法1中，存储关键树结点所需的空间最大，而关键树结点数目最多不超过ml个，每个关键结点最多有n个叶子结点，故算法1所用空间大小为$O(nml)$。 ■

定理 4.4　给定运行日志集L的大小为$\|L\| = n$，日志序列的最大长度为$\max(\|o\|) =$

m，可观事件集Σ_o大小为$\|\Sigma_o\| = l$，$F-$正确关键树构造算法在最差情况下的时间复杂度为$O(n^2m^2l)$，空间复杂度为$O(n+ml)$。

证明 相比正确关键树的构造算法，$F-$正确关键树构造算法只需找到第一条匹配路径，因此只需执行$updateNode$程序（程序5）一次。其它程序的具体分析见定理4.3的证明。 ■

4.2 关键树的查询算法

当我们利用系统已知的运行日志集合构造出关键树后，接下来的问题便是如何利用该关键树和当前的系统运行序列诊断出系统的故障状态。由于关键树满足层次性，因此对于关键树中的每一条路径，路径存储着从最抽象子观察Σ_o^*到关键观察的抽象运算过程中产生的所有子观察。从根节点出发，越靠近关键结点，该内部结点所存储的子观察越具体，子观察存储的序列特征越详细。

当运行序列查询关键树时，我们通过查询该运行序列全匹配当前结点的哪个孩子子观察来确定下一个查询结点，从而避免查询无意义的子观察。由于关键观察存储着能精确描述故障发生时系统的运行特征，因此若运行序列全匹配某个关键观察，则说明该运行序列符合该故障发生时的序列特征，因此我们推测系统在该运行序列下的故障状态与关键观察对应的故障状态相同。而若关键树是正确的，这意味着所有全匹配某序列的路径的诊断信息一致，因此我们只需找到一条匹配路径即可。为使得查询算法同样可以被应用于$F-$正确关键树，这里我们选择第一条全匹配路径，并返回该路径的诊断信息。

不同于关键树的更新程序1和4，查询算法需要运行序列全匹配子观察时方才确定下一步的查询结点。这样做的主要原因在于更新程序的重点在于生成运行序列的关键观察，因此只要运行序列符合子观察的主要特征即可，在结点更新完成后我们还可以完善该子观察。但查询算法不同，为保证查询的准确度，我们必须要求运行序列完全符合子观察的要求，仅要求运行序列符合序列特征会造成许多不必要的误差。

关键树的查询算法具体见算法2，其中$q \to child$指代结点q的第一个孩子结点，$q \to sibling$指代结点q的下一个兄弟结点，$q \to parent$指代结点q的父结点。

关键树的查询算法流程较为简单。我们先从根结点开始查询，若序列o全匹配当前结点子观察，则继续查询孩子结点，否则查询兄弟结点。若当前结点无下一个兄弟结点，则说明当前结点无符合运行序列特征的子观察，因此我们回溯到父结点并查询父结点的下一个兄弟结点。如第7行所示，若父结点也无下一个兄弟结点，继续回溯直到找到兄弟结点。当回溯到了一个不存在的结点，即$q \notin Q$，则说明关键树中不存在路径

Algorithm 2 关键树的查询算法

Input: 关键树$CtT(L)=(Q,S,\psi,\phi,q_0)$,运行序列$o$

Output: 故障模式

```
1: set q = q_0
2: while q ∈ Q_I do
3:     if o ∈ ψ(q) then
4:         set q = q → child
5:     else
6:         while q → sibling ∉ Q do
7:             set q = q → parent
8:         end while
9:         if q ∉ Q then
10:            return T_0
11:        else
12:            set q = q → sibling
13:        end if
14:    end if
15: end while
16: return ρ_t(φ(q))
```

符合运行序列的序列特征，也就是说，当前运行日志集中不存在日志序列与运行序列有相同的序列特征，因此我们推测该运行序列中并无故障发生，返回T_0。当我们能找到一个路径被运行序列全匹配，则结点q为叶子结点，由于该结点的日志标签即为该路径对应的故障模式，因此我们返回叶子结点的日志标签$\rho_t(\phi(q))$。

查询算法的核心思想是查询是否存在路径被运行序列全匹配，若存在，则说明该运行序列符合路径所记录的序列特征，因此我们可以推测系统在该运行序列下的故障状态为路径对应的故障模式。因为路径存储了关键观察由抽象到具体的运算过程，因此我们可以通过匹配路径来避免查询所有的子观察。当序列不匹配某个子观察时，该序列同样不会匹配该子观察结点下的所有关键观察。

事实上，查询算法返回的是基于首次匹配的关键树诊断信息，即$\Delta_C^1(o)$，不论关键树是正确还是$F-$正确的，首次匹配路径的诊断信息是正确的，查询算法的推测结果是完全可信的。因此关键树的查询算法（算法2）可以被用来查询正确关键树和$F-$正确关键树。

定理 4.5　给定关键树$CtT(L)=(Q,S,\psi,\phi,q_0)$，关键树的结点数目为$\|Q\|=n$，则算法2的时间复杂度为$O(n)$，空间复杂度为$O(1)$。

4.3 算法运行流程示例

4.3.1 正确关键树的构造及其查询

给定可观事件集为$\Sigma_o=\{a,b,c\}$，运行日志集合为$L=\{(aba,T_1),(aba,T2),(aab,T_1),(aaab,T_1),(aac,T_2)\}$，接下来我们详细讲述基于$L$的正确关键树$CtT(L)$的构造过程和查询过程。

首先移除L中的冲突日志(aba,T_1)和$(aba,T2)$，得到运行日志集$L_C=\{(aab,T_1),(aaab,T_1),(aac,T_2)\}$。接下来初始化关键树$CtT(L_C)$，初始关键树如图4-2。

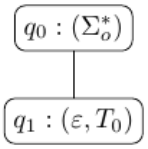


图 4-2 初始关键树

接着利用日志集L_C的第一个日志(aab,T_1)对初始关键树进行更新，毫无疑问，日志序列aab硬匹配关键观察(Σ_o^*)，但该关键观察对应的故障模式为T_0，因此该关键结点违背了结点间的充分性。为使得该结点满足性质，我们需要生成新日志和旧叶子结点日志的关键观察。比较日志序列aab和原子观察(Σ_o^*)，我们得到了两个新硬事件序列(a)和(b)，利用aab填充软事件得到新关键观察$(\{a\},a,\{ab\})$和$(\{a\},b,\emptyset)$，建立以这些关键观察为标签的内部结点，并建立以运行日志(aab,T_1)为标签的叶子结点，最后将它们插入到关键树中。由于原叶子结点日志序列ε找不到新关键观察，因此我们得到运行日志(aab,T_1)更新后的关键树为图4-3。

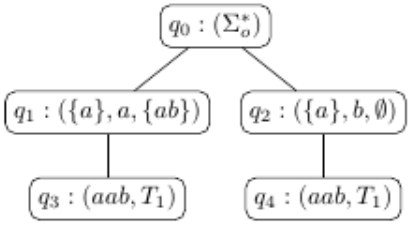


图 4-3 (aab,T_1)更新后的正确关键树

继续利用日志集L_C的第二个日志$(aaab, T_1)$对关键树进行更新。毫无疑问，序列$aaab$硬匹配路径$q_0 \to q_1$和$q_0 \to q_1$，而这两路径对应的故障模式都为T_1，因此我们只需新建以该日志为标签的叶子结点，接着将叶子结点插入对应的关键结点下即可，运行日志$(aaab, T_1)$更新后的关键树为图4-4

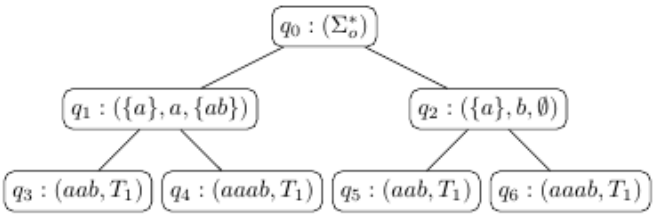


图 4-4 $(aaab, T_1)$更新后的正确关键树

最后利用日志集L_C的最后一个日志(aac, T_2)对关键树进行更新。序列aac硬匹配子观察(Σ_o^*)和子观察$(\{a\}, a, \{ab\})$，即硬匹配路径$q_0 \to q_1$，然而该路径的诊断信息为T_1，与日志标签不同，违背了充分性。比较日志序列aac和原子观察$(\{a\}, a, \{ab\})$，我们得到了一个新硬事件序列(a, c)，利用aac填充软事件得到新关键观察$(\{a\}, a, \{a\}, c, \emptyset)$；同样的，比较叶子结点日志序列$aab$、$aaab$和日志序列$aac$，得到了一个新硬事件序列$(a, b)$，利用$aab$和$aaab$填充软事件得到新关键观察$(\{a\}, a, \{a\}, b, \emptyset)$。需要注意的是由于序列$aac$仅硬匹配原子观察$(\{a\}, a, \{ab\})$，因此我们需要通过填加软事件完善该子观察，最终得到更新后的原子观察$(\{a\}, a, \{abc\})$。接着我们继续寻找下一个匹配路径，我们发现结点q_2无兄弟结点，或者说，关键树中丢失了一些比子观察(Σ_o^*)具体的关键观察，因此我们比较日志序列aac和原子观察(Σ_o^*)，得到新硬事件序列c，利用aac填充软事件得到$(\{a\}, c, \emptyset)$。同上，将所有关键结点和叶子结点插入先关键树中。

最终我们得到运行日志集L_C的关键树如图4-5。

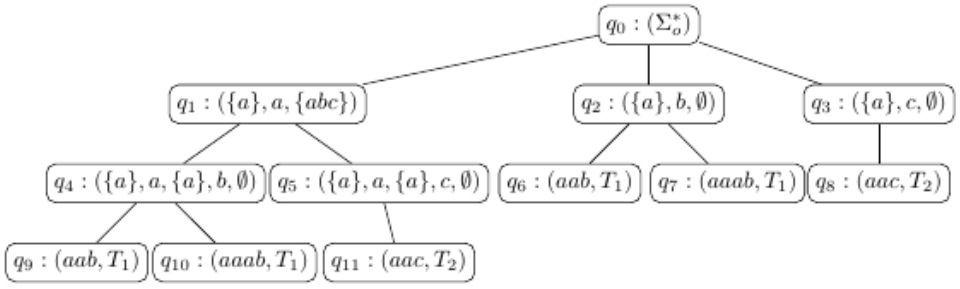


图 4-5 运行日志集L_C的正确关键树$CtT(L_C)$

现在我们得到了由正确关键树的构造算法（算法1）构造出的关键树$CtT(L_C)$，且由定理4.1可知该关键树$CtT(L_C)$是关于运行日志集L_C正确的。因此我们可以利

用$CtT(L_C)$对系统序列下系统的故障状态进行诊断。首先我们查询运行日志集L_C中已有的日志序列aac，很显然，aac全匹配路径$q_0 \rightarrow q_1 \rightarrow q_5$上的所有子观察，特别是$aac$全匹配关键观察$(\{a\}, a, \{a\}, c, \emptyset)$，因此我们推测序列$aac$下系统的故障模式为$T_2$。

然后我们查询一个新的系统运行序列aba，该序列全匹配子观察(Σ_o^*)和其孩子结点q_1的子观察$(\{a\}, a, \{abc\})$，然而由于aba即不全匹配$(\{a\}, a, \{a\}, b, \emptyset)$又不全匹配$(\{a\}, a, \{a\}, c, \emptyset)$，因此我们继续查询$q_1$的兄弟结点的子观察$(\{a\}, b, \emptyset)$和$(\{a\}, c, \emptyset)$，显然序列$aba$都不全匹配。因此该关键树中并无关键观察符合序列$aba$的序列特征，我们推测序列$aba$下系统的故障模式为$T_0$。

4.3.2 $F-$正确关键树的构造及其查询

同样的，以可观事件集$\Sigma_o = \{a, b, c\}$，运行日志集合$L = \{(aba, T_1), (aba, T2), (aab, T_1), (aaab, T_1), (aac, T_2)\}$为例，接下来我们解释基于$L$的$F-$正确关键树$CtT(L)$的构造过程。

移除冲突日志和利用第一条日志更新初始关键树的流程与上一节相同。但从第二条日志开始$F-$正确关键树就只需更新首次匹配路径上的待更新结点。利用L_C的第二条日志$(aaab, T_1)$对关键树进行更新，得到图4-6。 与图4-4比较，由于日志序列$aaab$的第

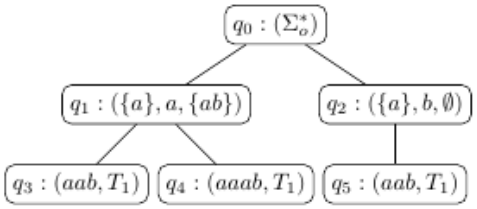


图 4-6 $(aaab, T_1)$更新后的$F-$正确关键树

一条匹配路径为$q_0 \rightarrow q_1$，而该路径的诊断信息为T_1，因此新建以该日志为标签的叶子结点，接着将叶子结点插入q_1下即可，更新结束，无需继续寻找下一条匹配路径。继续利用L_C的第三条日志(aac, T_2)对关键树进行更新，得到图4-7。 同样的，日志序列aac匹配的第一条匹配路径为$q_0 \rightarrow q_1$，然而路径诊断信息不等于T_2，因此需要生成新的关键观察，最后将新关键结点和叶子结点插入关键树中，更新结束。

相比上一节的正确关键树，$F-$正确关键树只保留首次匹配路径，消除了如$(\{a\}, c, \emptyset)$这样的冗余子观察和如$(aaab, T_1)$这样的重复日志，可以看出匹配子观察$(\{a\}, c, \emptyset)$的序列一定也匹配$(\{a\}, a, \{a\}, c, \emptyset)$，因此$F-$正确关键树仍保留了故障序列的序列特征。

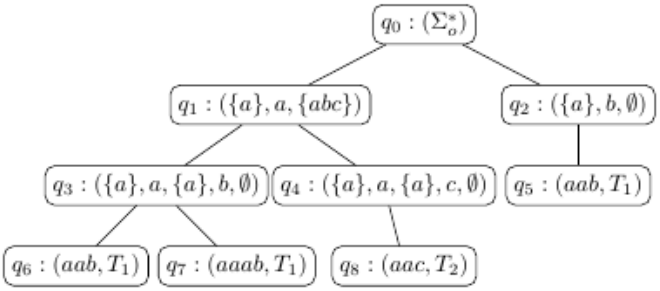


图 4-7 运行日志集L_C的$F-$正确关键树$CtT(L_C)$

4.4 本章小结

本章给出了正确关键树和$F-$正确关键树的构造算法和查询算法，并详细解释了程序的流程，还有程序中每个函数的实现和作用。在介绍算法的同时，我们还证明了由构造算法构造出的关键树的正确的/$F-$正确的，最后分析了算法的时间复杂度和空间复杂度。

第 5 章 实验及分析

为直观的说明本文提出的关键树方法在时间、空间以及准确率上的优势，本章给出了关键树方法与Christopher CJ的日志树方法[57]的比较实验。

5.1 实验方案

首先我们对两方法进行理论分析。在第4章中我们已经分析了正确关键树的构造算法的时间和空间复杂度，与Christopher CJ等人的日志树方法[57]和Cassez F等人的动态观测法[52]的对比如表5-1。

表 5-1 关键树方法与其它方法的复杂度对比

算法	时间复杂度	空间复杂度
正确关键树法	$O(n^3m^4l^3)$	$O(n^2ml)$
F-正确关键树法	$O(n^2m^2l)$	$O(n+ml)$
日志树方法	$O(n2^{ml})$	$O(nm^3l^3)$
动态观测法	$O(\|\Sigma\|m2^{n^2}2^k2^{2^{\|\Sigma\|}})$	$O(2^{n^2}2^k2^{2^{\|\Sigma\|}})$

不同于关键树方法将故障序列的序列特征刻画成树的唯一路径，Christopher CJ在17年提出的日志树方法[57]将运行日志刻画成一个树形的有限状态自动机，每个日志序列在日志树中都拥有唯一的路径。从理论上分析，关键树方法相较于日志树方法大幅度减少了求取关键观察的时间和空间消耗。

由于日志树方法具有指数级的时间复杂度，当离散事件系统规模较大时，该方法需要过长的时间和过大的空间来完成对系统的故障诊断，这显然不符合本文对系统故障进行快速诊断的目标不同。为保证实验的顺利进行，在与日志树的比较实验中我们采用较小规模的离散事件系统模型。

另一方面，算法的时间和空间消耗只与运行日志集L的大小n，日志序列的最大长度m，可观事件集的大小l有关，此外尽管故障事件集的大小（后文用k指代，$k=\|\Sigma_f\|$无法被量化到复杂度的分析之中，但它毫无疑问也会影响到算法的效率及其准确率。因此在本实验中，我们将令其它变量（如系统的状态数目）恒定不变，分析当以上四个变量变化时算法消耗的时间、空间以及准确率的变化情况。

值得注意的是，由于实验中的空间消耗受程序本身的影响较大，而算法的主要空间消耗在于需要存储运算过程中产生的子观察，因此我们将实验过程中生成的所有子观察数目近似为算法的空间消耗。

同时，为保证算法的普适性，我们利用随机生成的自动机模型来模拟离散事件系统。并确保随机生成的自动机模型拥有以下特征：

1 自动机的状态数目恒定不变，在日志树比较实验中，恒定系统有100个状态；
2 自动机不会处于一个不存在的状态；
3 自动机中的所有状态均存在转移；
4 在生成自动机的部分转移函数时不会区别对待可观事件、不可观事件以及故障事件；
5 自动机生成的运行序列长度都是有限的，当运行序列长度达到m时自动机运行结束（或称为本次运行日志记录周期结束更为准确）；
6 自动机在运行结束后会记录此次运行的运行序列和故障状态，并将之添加到运行日志集中。

此外，由于多故障系统与单故障系统对于关键树方法而言并无任何区别，关键树方法只针对故障模式进行处理，因此本实验采用更接近于现实的多故障系统，即系统的一次运行中可能会发生多个故障事件。因此在实验中可能出现的故障模式共有2^k个。

最后，由于关键树方法与日志树方法利用已构造完毕的关键树或日志树推测系统当时的故障状态上所花费的时间和空间都极少，因此下文只讨论构造算法所消耗的时间和空间。

接下来我们将从时间、子观察数目以及准确率方面分析关键树方法与日志树方法的优劣之处。

5.2 实验结果及其分析

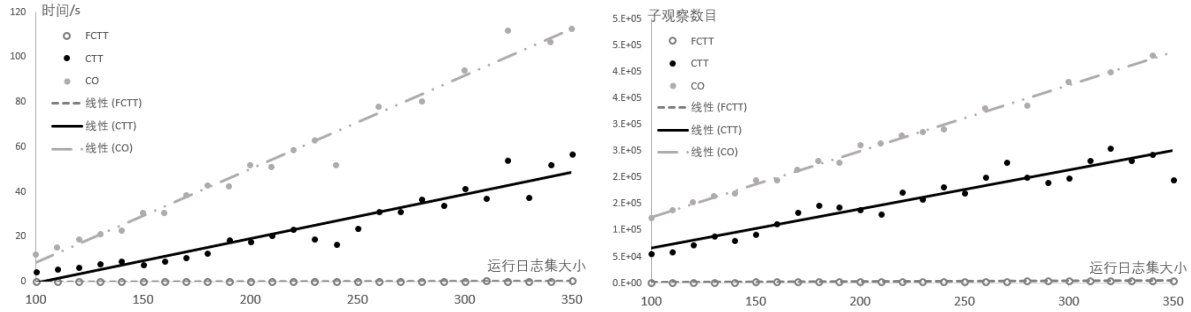
本节我们给出正确关键树方法（后文以CTT算法指代）、$F-$正确关键树方法(后文$FCTT$算法指代)与日志树方法（后文以CO算法指代）在时间消耗、空间消耗以及准确率方面的实验结果，并对结果进行分析。

本实验在x64架构，Intel Core i7-7700 cpu，双核3.6GHz，8GB的RAM环境下运行。

5.2.1 时间消耗和空间消耗

毫无疑问，当运行日志集大小、运行序列的最大长度、可观事件集的大小、故障事件集大小变大时，三算法所消耗的时间和空间都会随之增加。图5.2.1展示了在其它环境不变（$m = 200, l = 20, k = 5$）的情况下，仅运行日志集大小变化时三个算法在时

间消耗、生成的子观察数目上的变化情况。其它三变量变化时，三算法的时间和空间消耗情况类似。



(a) 时间消耗 (b) 子观察数目

图 5-1 运行日志集大小变化下的时间和空间消耗情况

显然，当运行日志集大小n增加时，三算法所消耗的时间和运算过程中产生的子观察数目都会随之增加。三个算法中，$F-$正确关键树构造算法$FCTT$的用时最短，冗余子观察的数目也最少，而日志树算法CO需要的时间最多，冗余子观察的数目也最多。并且相比CTT和CO算法随着n的增加，消耗时间和空间急剧增加，$FCTT$算法在n增加到350时仍然能在1秒内将该运行日志集对应的$F-$正确关键树构造出来。

为直观的说明三个算法的时间和空间消耗情况，表5-2展示了运行日志集的大小为350时，三算法所消耗的时间和子观察数目。

表 5-2 运行日志集的大小为350时，CTT、$FCTT$与CO算法的时间和空间消耗

算法	时间	子观察数目
$FCTT$	0.28s	3962
CTT	56.85s	196148
CO	112.56s	430726

通过表格可以很显然的看出，$FCTT$算法的时间和空间消耗均远低于其它两个算法，只需0.28s即可生成所需要的$F-$正确关键树，相反的，CO算法则需要112.56s才能生成所有运行日志的关键观察。然而，从实验结果中我们也发现了一个不同寻常的问题，即尽管日志树方法CO的时间复杂度为指数级，远高于关键树方法CTT，然而两者所花费的时间和空间却只相差一倍。认真分析后我们认为主要有三个原因：

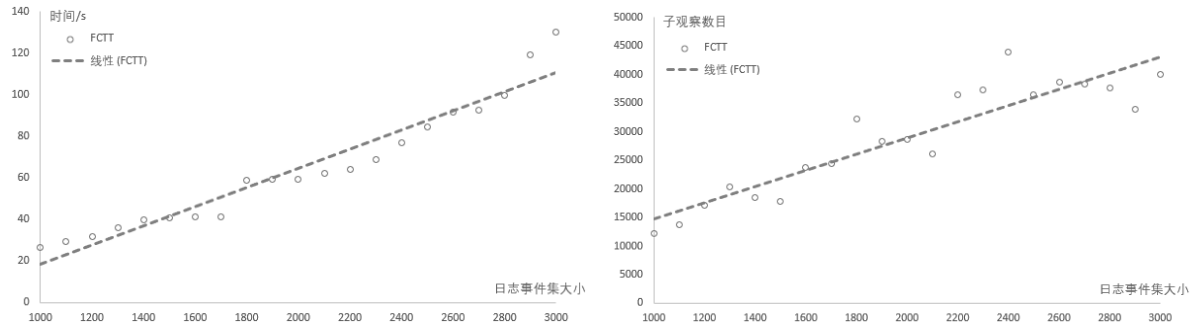
1 一是因为对于每个运行日志，日志树方法只会生成一个关键观察，相比之下CTT算法会生成所有的关键观察；

2 二是因为日志树方法采用$event-softening$运算和$collapse$运算生成关键观察，尽管运算过程中产生了大量的冗余子观察，但可以保证生成的关键观察一定关于序列充分的最短的子观察，而关键树方法使用子序列自动机方法尽管只需要少量的时间即可得到所需要的所有关键观察，且不会产生额外的子观察，但无法保证得到关键观察是最短的；

3 三则是日志树方法不采用硬匹配来对运行日志进行处理，因此无需在产生关键观察时填充软事件。

但这并不代表硬匹配和子序列自动机是冗余操作，相反的，从$FCTT$算法实验结果可以看出，硬匹配在花费少量时间填充关键观察的软事件的同时帮助我们节省了大量的空间，而节省的空间实际上又反过来帮助我们节省了用于查找匹配子观察的时间。子序列自动机方法更是使得关键观察可以直接通过查找非公共子序列的方法得到，无需生成额外的子观察。

尽管如此，由于需要数百秒的时间来进行故障诊断，关键观察日志树方法CO仍然是无法适应于较大规模的离散事件系统，相比之下关键树方法，特别是$FCTT$在较大规模的系统下依旧可以保持较高的效率，图5.2.1展示了当运行序列的最大长度为1000，可观事件集大小为50，故障事件数目为10时，即$m=1000, l=50, k=10$时，运行日志集大小从1000到3000时，$F-$正确关键树构造算法$FCTT$的时间、空间消耗情况。

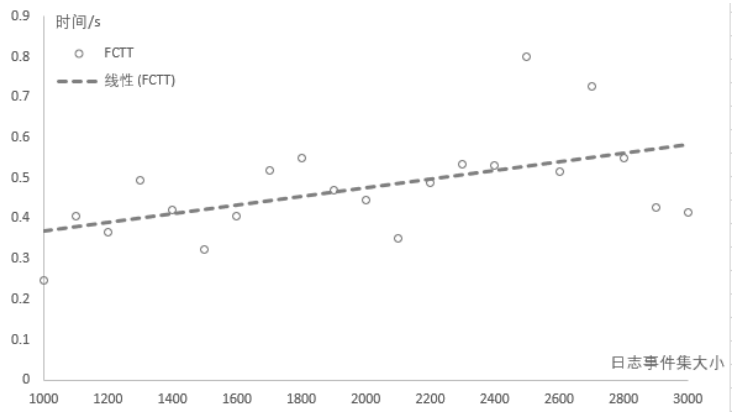


(a) 时间消耗 (b) 子观察数目

图 5-2 运行日志集大小变化下$FCTT$算法的时间和空间消耗

可以看出，尽管离散事件系统的规模相对于之前有了大幅度的提高，但$FCTT$算法仍只需要数十秒的时间即可得到需要的$F-$正确关键树，且过程中只会产生几万个子观察。在$n=3000, m=1000, l=50, k=10$下$FCTT$构造算法需消耗的时间和空间仅相当于$n=300, m=200, l=20, k=5$时$CO$算法所消耗的时间和空间。而此时查询关键树

得到故障诊断的结果所需的时间仍只需不到1秒，如图5.2.1所示。



(a) 查询时间

图 5-3 运行日志集大小变化时$FCTT$查询算法的时间消耗

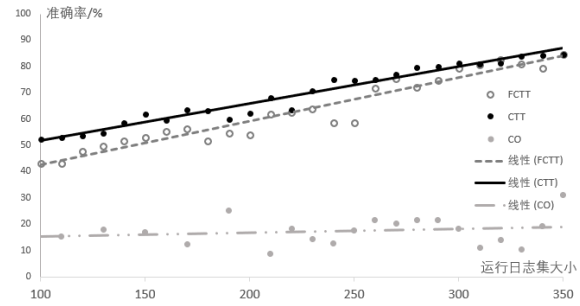
5.2.2 故障诊断的准确率

尽管之前的实验结果表明了关键树方法，特别是$F-$正确关键树方法构造关键树需要的时间远少于日志树方法，然而对于黑盒诊断方法而言，故障诊断的准确率才是最值得关注的问题。若某黑盒诊断方法虽然只需要少量的时间和空间，但故障诊断的正确率却低于其它方法，则该方法也是无意义的。

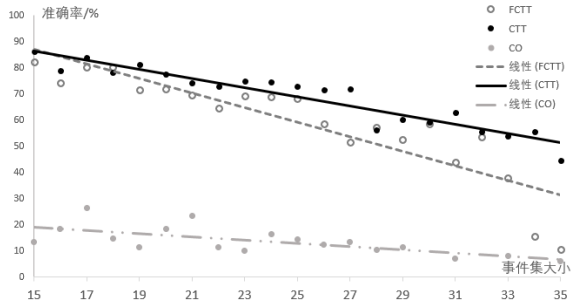
图5.2.2展示了当运行日志集大小n、运行序列的最大长度m、可观事件集大小l、故障事件集大小k变化时，三算法故障诊断的正确率变化情况。

图5.2.2表明，当运行日志集和运行序列的长度增加时，故障诊断结果的准确率也随之增加，相反的，当可观事件集和故障事件集增多时，准确率随之降低。很显然，运行日志数目越多，得到的关键观察越多也越具体，这也就意味着得到的故障序列的序列特征越丰富，同时也越详细，因此准确率也会随之增加。同样的，当运行序列的长度增加时，序列所蕴含的序列特征越丰富，我们可以从序列中得到的信息也就越多，准确率也因此而增加。而可观事件集的增多导致序列中的冗余信息变多，从运行日志中得到的关键观察也越抽象，存储的序列特征也就越少，准确率也因为关键观察无法再详细的描述系统的运行特征而降低。同样的道理，故障事件集的增多使得单个故障的关键观察数目变少，关于该故障下系统的运行特征越少，因此相比较少的故障事件集，更难准确的诊断出系统的故障状态。

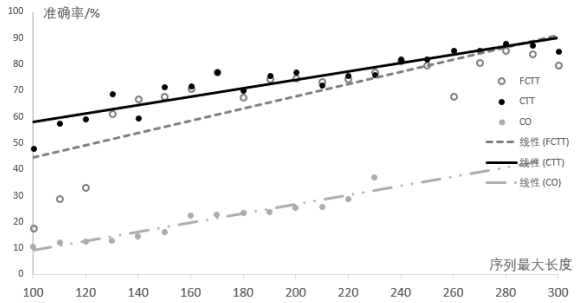
从实验结果中可以看出，相对于对每个运行序列只生成一个关键观察的日志树方法，关键树方法在故障诊断的正确率上有着巨大的优势。当运行序列的最大长度达



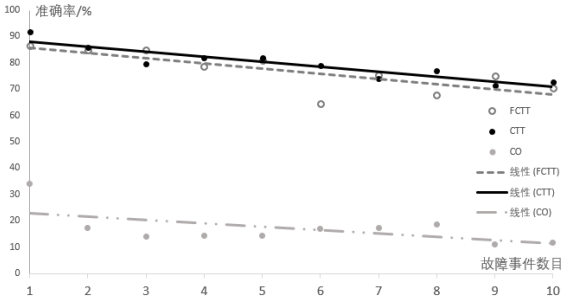
(a) 运行日志集大小变化时准确率变化情况



(b) 可观事件集大小变化时准确率变化情况



(c) 运行序列的长度变化时准确率变化情况



(d) 故障事件集大小变化时准确率变化情况

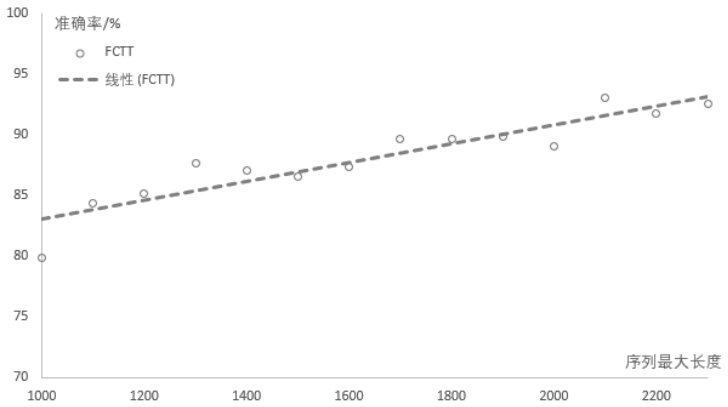
图 5-4 算法故障诊断的准确率比较

到300时，关键树方法CTT和$FCTT$算法的准确率甚至可以达到90%以上，而日志树方法准确率最高时也只有39%。经过分析，我们认为日志树方法准确率如此之低的原因有二，一是由于对于日志树方法只会对每个运行日志生成一个关键观察，而关键树方法会生成所有的关键观察，显然，关键观察越多，保留的序列特征越多，故障诊断的结果也就越精确。二则是日志树方法通过查询所有的关键观察来得到所有可能发生的故障事件的并集，进而得到系统的故障模式，而一个故障序列往往不止匹配该故障的关键观察，该序列含有的冗余信息往往也匹配其它关键观察，因而导致日志树下该序列的故障诊断结果含有与该序列的序列特征不符的诊断信息，相比之下，关键树只返回首次匹配路径的诊断信息，而关键树的路径可以帮助我们在故障诊断过程中忽略无效子观察。

除此之外，实验结果显示尽管正确关键树在构造时会对树中的所有路径进行更新，而$F-$正确关键树只会更新首次匹配路径，然而正确关键树方法CTT的准确率却仅略高于$F-$正确关键树方法。这充分证明了关键树中存在着大量的冗余路径，即关键树中被同一运行序列匹配的路径所存储的序列特征也往往是相同的。对于$F-$正确关键树而言，尽管因为只保留首次匹配路径而遗失了一些有效的序列特征，但相对于极少的时间和空间消耗，损失的那点准确率对于对故障诊断准确率要求不那么严格的系统而

言显然是可以接受的。

同样的，图5.2.2展示了较大规模离散事件系统下，当序列长度变化时，$FCTT$算法故障诊断的准确率变化情况，实验参数为$n = 1000, l = 50, k = 20$。



(a) 准确率

图 5-5 运行序列长度变化时$FCTT$的准确率变化情况

实验结果表明当序列长度高于1000时，故障诊断的准确率便能稳定在80%以上，而当序列序列长度为1700时，准确率甚至可以大于90%。

5.3 本章小结

本章展示了在系统模型未知，仅系统前期的运行日志集已知的情况下，关键树方法（包括正确关键树方法与$F-$正确关键树方法）与Christopher CJ的日志树方法[57]对系统进行故障诊断的比较实验极其结果。实验结果表明相对于日志树方法，关键树方法可以在较少的时间和空间内，以较高的准确率诊断出系统的故障状态。特别是$F-$正确关键树方法，可以在仅损失少量准确率的情况下，以高效率完成对系统进行故障诊断，因此适应于对故障诊断准确率要求并非非常严格的系统。

第 6 章 总结

本章将总结本文的内容，指出当前方法的不足之处，并展望未来的工作。

6.1 论文总结

在本文中，我们提出了一个在模型未知的系统上，借助系统前期的运行日志集合，在多项式的时间和空间复杂度内构造出诊断器，并利用诊断器在线性的时间复杂度和常数级的空间复杂度内诊断出离散事件系统故障状态的关键树方法。本文利用关键观察来消除冗余序列及序列中的冗余信息，提出通过查找非公共子序列来生成关键观察，关键树存储运算过程中产生的子观察，最后通过查询关键树得到系统当前的故障状态。关键树是一颗多叉树，其内部结点标记为子观察，叶子结点标记为运行日志。若关键树中的每个结点都满足结点间的层次性、充分性、不相容性、相似性和一致性，则该关键树是正确的，正确关键树存储了给定运行日志集下的所有关键观察。利用关键树路径，我们可以在关键观察的生成过程中，排除不相关日志的影响，从而能在较短的时间及较少的空间内找到所有的关键观察。另一方面，由于正确关键树中存在着冗余子观察，因此我们又在正确关键树的基础上提出了$F-$正确关键树的概念。$F-$正确关键树消除了正确关键树中的冗余子观察，大大减少了构造关键树所需的时间和空间消耗，但同时$F-$正确关键树也遗失了部分有效子观察，使得故障诊断的准确率有所降低。

最重要的是，本文给出了正确关键树和$F-$正确关键树的构造算法及其查询算法，并证明了正确关键树构造算法具有$O(n^3m^4l^3)$的时间复杂度和$O(n^2ml)$的空间复杂度，$F-$正确关键树构造算法具有$O(n^2m^2l)$的时间复杂度和$O(n+ml)$的空间复杂度，而关键树查询算法的时间复杂度为$O(n)$，空间复杂度为$O(1)$。在构造算法中我们将关键观察生成问题转换为最短非公共子序列问题，利用子序列自动机，我们可以在较短的时间和较少的空间消耗内生成关键观察。在关键树的构造过程中，我们通过利用运行日志对关键树进行更新的方法构建出完整的关键树，使得正确关键树与$F-$正确关键树无需设计两套完全不同的构造算法。另一方面，我们保证在每一次更新过程中关键树都能满足要求的性质，从而使得关键树在每次更新后都能保持正确性，这样即使系统运行日志集发生了变化，我们也无需改动现有关键树。

最后，本文还与Christopher CJ[57]的日志树方法进行了比较，与日志树方法聚焦于对序列本身进行处理不同，本文提出的关键树方法主要在于优化关键观察的生成。最后我们给出的实验结果表明关键树方法与日志树方法相比，无论是在时间和空间消耗上，还是在故障诊断的准确率上，都有着较大幅度的优势。特别是$F-$正确关键树方

法，在仅损失少量准确率的情况下，$F-$正确关键树方法仅需少量的时间和空间即可构造出给定运行日志集的$F-$正确关键树。

6.2 对未来工作的展望

为保证充分性，关键观察的硬事件被赋值为故障序列的非公共子序列，因此无法保证关键观察完整且正确的保留了故障序列的序列特征。诊断性的定义指出系统是可诊断的当故障可以在发生后的有限步延迟内被诊断出，因此我们可以提出一个更完善的数据结构来进行序列特征提取，该数据结构能完整且正确的保留序列中的有效信息，并同时能消除序列中的无效信息。另一方面，正确关键树中存在着冗余子观察而$F-$正确关键树在消除冗余子观察的同时也遗失了部分有效子观察，因此我们可以找到一个更优的关键树结构，该结构保留了所有的有效子观察，同时也不存在冗余子观察。

除了用于模型完全未知的系统中外，关键树方法还可以扩展到以下方向：

1 动态系统的故障诊断问题。关键树方法的核心在于系统前期的运行日志集合，尽管动态系统具有随机性的特性，但系统的故障序列仍然是存在着共同的序列特征的。

2 模型学习。尽管系统的模型未知，但关键树路径和路径存储的序列特征可以帮助我们重建该模型。

参考文献

[1] Lin F. Diagnosability of discrete event systems and its applications [J]. Discrete Event Dynamic Systems, 1994, 4 (2): 197–212.

[2] Traore M, Sayed-Mouchaweh M, Billaudel P. Learning diagnoser and supervision pattern in discrete event system: Application to crisis management [C]. In Annual Conference of the Prognostics and Health Management Society, 2013: 694–701.

[3] Sülek A N, Schmidt K W. Computation of supervisors for fault-recovery and repair for discrete event systems [J]. IFAC Proceedings Volumes, 2014, 47 (2): 428–433.

[4] Wen Q, Kumar R, Huang J, *et al.* A framework for fault-tolerant control of discrete event systems [J]. IEEE Transactions on Automatic Control, 2008, 53 (8): 1839–1849.

[5] Isermann R. Fault-diagnosis systems: an introduction from fault detection to fault tolerance [M]. Springer Science & Business Media, 2006.

[6] Son H I, Lee S. Failure diagnosis and recovery based on DES framework [J]. Journal of Intelligent Manufacturing, 2007, 18 (2): 249–260.

[7] Tzafestas S G, Singh M, Schmidt G. System Fault Diagnostics, Reliability and Related Knowledge-Based Approaches: Volume 1 Fault Diagnostics and Reliability Proceedings of the First European Workshop on Fault Diagnostics, Reliability and Related Knowledge-Based Approaches, Island of Rhodes, Greece, August 31–September 3, 1986 [M]. Springer Science & Business Media, 1987.

[8] Patton R J, Frank P M, Clarke R N. Fault diagnosis in dynamic systems: theory and application [M]. Prentice-Hall, Inc., 1989.

[9] Roychoudhury I, Biswas G, Koutsoukos X. Designing distributed diagnosers for complex continuous systems [J]. IEEE Transactions on Automation Science and Engineering, 2009, 6 (2): 277–290.

[10] Sampath M, Sengupta R, Lafortune S, *et al.* Diagnosability of discrete-event systems [J]. IEEE Transactions on automatic control, 1995, 40 (9): 1555–1575.

[11] Jiang S, Huang Z, Chandra V, *et al*. A polynomial algorithm for testing diagnosability of discrete-event systems [J]. IEEE Transactions on Automatic Control, 2001, 46 (8): 1318–1321.

[12] Vento J, Sarrate R, *et al*. Fault detection and isolation of hybrid system using diagnosers that combine discrete and continuous dynamics [C]. In 2010 Conference on Control and Fault-Tolerant Systems (SysTol), 2010: 149–154.

[13] Yoo T-S, Lafortune S. Polynomial-time verification of diagnosability of partially observed discrete-event systems [J]. IEEE Transactions on automatic control, 2002, 47 (9): 1491–1495.

[14] Lamperti G, Zanella M. Diagnosis of active systems: principles and techniques [M]. Springer Science & Business Media, 2003.

[15] Zaytoon J, Lafortune S. Overview of fault diagnosis methods for discrete event systems [J]. Annual Reviews in Control, 2013, 37 (2): 308–320.

[16] Moreira M V, Jesus T C, Basilio J C. Polynomial time verification of decentralized diagnosability of discrete event systems [J]. IEEE Transactions on Automatic Control, 2011, 56 (7): 1679–1684.

[17] Frank P M. Analytical and qualitative model-based fault diagnosis–a survey and some new results [J]. European Journal of control, 1996, 2 (1): 6–28.

[18] Gertler Janos J. Fault Detection and Diagnosis in Engineering Systems. 1998.

[19] Hamscher W, Console L, De Kleer J. Readings in model-based diagnosis [M]. Morgan Kaufmann Publishers Inc., 1992.

[20] Williams B C, Nayak P P. A model-based approach to reactive self-configuring systems [C]. In Proceedings of the national conference on artificial intelligence, 1996: 971–978.

[21] Darwiche A, Provan G. Exploiting system structure in model-based diagnosis of discrete-event systems [C]. In Proc. 7th Intl. Workshop on Principles of Diagnosis, 1996.

[22] Puig V, Quevedo J, Escobet T, *et al*. On the integration of fault detection and isolation in model-based fault diagnosis [C]. In Proceedings of the 16th International Workshop on Principles of Diagnosis (DX-05), 2005: 227–232.

[23] De Vries R C. An automated methodology for generating a fault tree [J]. IEEE transactions on reliability, 1990, 39 (1): 76–86.

[24] Milde H, Hotz L. Facing diagnosis reality-model-based fault tree generation in industrial application [C]. In Proc. DX-00, 11th International Workshop on Principles of Diagnosis, 2000.

[25] Qiu D. Supervisory control of fuzzy discrete event systems: a formal approach [J]. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 2005, 35 (1): 72–88.

[26] Qiu D, Liu F. Fuzzy discrete-event systems under fuzzy observability and a test algorithm [J]. IEEE Transactions on Fuzzy Systems, 2009, 17 (3): 578–589.

[27] Deng W, Qiu D. Supervisory control of fuzzy discrete-event systems for simulation equivalence [J]. IEEE Transactions on Fuzzy Systems, 2015, 23 (1): 178–192.

[28] Deng W, Qiu D. Bifuzzy discrete event systems and their supervisory control theory [J]. IEEE Transactions on Fuzzy Systems, 2015, 23 (6): 2107–2121.

[29] Deng W, Qiu D. State-based decentralized diagnosis of bi-fuzzy discrete event systems [J]. IEEE Transactions on Fuzzy Systems, 2017, 25 (4): 854–867.

[30] Liu F, Qiu D. Diagnosability of fuzzy discrete-event systems: A fuzzy approach [J]. IEEE Transactions on Fuzzy Systems, 2009, 17 (2): 372–384.

[31] Deng W, Yang J, Qiu D. Supervisory Control of Probabilistic Discrete Event Systems under Partial Observation [J]. IEEE Transactions on Automatic Control, 2019.

[32] Liu F, Qiu D. Safe diagnosability of stochastic discrete event systems [J]. IEEE Transactions on Automatic Control, 2008, 53 (5): 1291–1296.

[33] Liu F, Qiu D, Xing H, *et al*. Decentralized diagnosis of stochastic discrete event systems [J]. IEEE Transactions on Automatic Control, 2008, 53 (2): 535–546.

[34] Fabre E, Hélouët L, Lefaucheux E, *et al*. Diagnosability of repairable faults [J]. Discrete Event Dynamic Systems, 2018, 28 (2): 183–213.

[35] Grastien A, Zanella M. Diagnosability of Discrete Faults with Uncertain Observations [M] // Grastien A, Zanella M. Diagnosability, Security and Safety of Hybrid Dynamic and Cyber-Physical Systems. Springer, 2018: 2018: 253–278.

[36] Ibrahim H, Dague P, Grastien A, *et al*. Diagnosability Planning for Controllable Discrete Event Systems [C]. In Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[37] Pouliezos A, Stavrakakis G S. Real time fault monitoring of industrial processes [M]. Springer Science & Business Media, 2013.

[38] Mah R S, Stanley G M, Downing D M. Reconcillation and rectification of process flow and inventory data [J]. Industrial & Engineering Chemistry Process Design and Development, 1976, 15 (1): 175–183.

[39] Romagnoli J, Stephanopoulos G. Rectification of process measurement data in the presence of gross errors [J]. Chemical Engineering Science, 1981, 36 (11): 1849–1863.

[40] Tong H, Crowe C M. Detection of gross erros in data reconciliation by principal component analysis [J]. AIChE Journal, 1995, 41 (7): 1712–1722.

[41] Zad S H, Kwong R H, Wonham W M. Fault diagnosis in discrete-event systems: Framework and model reduction [J]. IEEE Transactions on Automatic Control, 2003, 48 (7): 1199–1212.

[42] Lin F. Diagnosability of discrete event systems and its applications [J]. Discrete Event Dynamic Systems, 1994, 4 (2): 197–212.

[43] Behzad H, Casavola A, Tedesco F, *et al*. Fault-tolerant sensor reconciliation schemes based on unknown input observers [J]. International Journal of Control, 2018: 1–11.

[44] Cabasino M P, Giua A, Seatzu C. Fault detection for discrete event systems using Petri nets with unobservable transitions [J]. Automatica, 2010, 46 (9): 1531–1539.

[45] Cabasino M P, Giua A, Pocci M, *et al*. Discrete event diagnosis using labeled Petri nets. An application to manufacturing systems [J]. Control Engineering Practice, 2011, 19 (9): 989–1001.

[46] Mahulea C, Seatzu C, Cabasino M P, *et al*. Fault diagnosis of discrete-event systems using continuous Petri nets [J]. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, 2012, 42 (4): 970–984.

[47] Fessant F, Clérot F. An efficient SOM-based pre-processing to improve the discovery of frequent patterns in alarm logs [C]. In Conference on Data Mining— DMIN, 2006: 277.

[48] Briones L B, Lazovik A, Dague P. Optimal observability for diagnosability [C]. In Nineteenth International Workshop on Principles of Diagnosis (DX-08), 2008: 31–38.

[49] Grastien A, Travé-Massuyès L, Puig V. Solving diagnosability of hybrid systems via abstraction and discrete event techniques [J]. IFAC-PapersOnLine, 2017, 50 (1): 5023–5028.

[50] De Kleer J. Using crude probability estimates to guide diagnosis [J]. Artificial Intelligence, 1990, 45 (3): 381–391.

[51] Felfernig A, Walter R, Galindo J A, *et al*. Anytime diagnosis for reconfiguration [J]. Journal of Intelligent Information Systems, 2018, 51 (1): 161–182.

[52] Cassez F, Tripakis S. Fault diagnosis with static and dynamic observers [J]. Fundamenta Informaticae, 2008, 88 (4): 497–540.

[53] Sears D, Rudie K. Minimal sensor activation and minimal communication in discrete-event systems [J]. Discrete Event Dynamic Systems, 2016, 26 (2): 295–349.

[54] Yin X, Lafortune S. A uniform approach for synthesizing property-enforcing supervisors for partially-observed discrete-event systems [J]. IEEE Transactions on Automatic Control, 2016, 61 (8): 2140–2154.

[55] Christopher C J, Cordier M-O, Grastien A. Critical observations in a diagnostic problem [C]. In 53rd IEEE Conference on Decision and Control, 2014: 382–387.

[56] Christopher C J, Grastien A. Formulating event-based critical observations in diagnostic problems [C]. In 2015 54th IEEE Conference on Decision and Control (CDC), 2015: 4462–4467.

[57] Christopher C J, Pencolé Y, Grastien A. Inference of fault signatures of discrete-event systems from event logs. [C]. In DX, 2017: 219–233.

[58] Cordier M-O, Travé-Massuyes L, Pucel X, *et al.* Comparing diagnosability in continuous and discrete-event systems [C]. In Proceedings of the 17th International Workshop on Principles of Diagnosis (DX-06), 2006: 55–60.

[59] Jéron T, Marchand H, Pinchinat S, *et al.* Supervision patterns in discrete event systems diagnosis [C]. In 2006 8th International Workshop on Discrete Event Systems, 2006: 262–268.

[60] Gougam H-E, Pencolé Y, Subias A. Diagnosability analysis of patterns on bounded labeled prioritized Petri nets [J]. Discrete Event Dynamic Systems, 2017, 27 (1): 143–180.

[61] Cassandras C G, Lafortune S. Introduction to discrete event systems [M]. Springer Science & Business Media, 2009.

[62] 郑大钟, 赵千川. 离散事件动态系统 [M]. 清华大学出版社, 2001.

[63] Ramadge P J, Wonham W M. The control of discrete event systems [J]. Proceedings of the IEEE, 1989, 77 (1): 81–98.

[64] Sampath M, Sengupta R, Lafortune S, *et al.* Failure diagnosis using discrete-event models [J]. IEEE transactions on control systems technology, 1996, 4 (2): 105–124.

[65] Sohrabi S, Baier J A, McIlraith S A. Diagnosis as planning revisited [C]. In Twelfth International Conference on the Principles of Knowledge Representation and Reasoning, 2010.

[66] Haslum P, Grastien A, *et al.* Diagnosis as planning: Two case studies [J], 2011.

[67] Cordier M-O, Largouët C. Using model-checking techniques for diagnosing discrete-event systems [C]. In 12th International Workshop on Principles of Diagnosis (DX’ 01), 2001: 39–46.

[68] Carvalho L K, Moreira M V, Basilio J C, *et al.* Robust diagnosis of discrete-event systems against permanent loss of observations [J]. Automatica, 2013, 49 (1): 223–231.

[69] Carvalho L K, Basilio J C, Moreira M V. Robust diagnosis of discrete event systems against intermittent loss of observations [J]. Automatica, 2012, 48 (9): 2068–2078.

[70] Debouk R, Lafortune S, Teneketzis D. Coordinated decentralized protocols for failure diagnosis of discrete event systems [J]. Discrete Event Dynamic Systems, 2000, 10 (1-2): 33–86.

[71] Pencolé Y, Cordier M O. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks [J]. Artificial Intelligence, 2005, 164 (1-2): 121–170.

[72] Su R, Wonham W M. Global and local consistencies in distributed fault diagnosis for discrete-event systems [J]. IEEE Transactions on Automatic Control, 2005, 50 (12): 1923–1935.

[73] Mohri M, Moreno P, Weinstein E. General suffix automaton construction algorithm and space bounds [J]. Theoretical Computer Science, 2009, 410 (37): 3553–3562.

攻读硕士学位期间发表学术论文情况

1 Fault Diagnosis in Unknown Discrete Event Systems via Critical Tree [C]. The 31st Chinese Control and Decision Conference. 2019, Nanchang, China.（EI已录用)(第一作者)

致　　谢

感谢我尊敬的导师***教授。在我攻读硕士的两年间，***老师在我的学业和科研工作给予悉心的指导，在生活上给予了许多无私的帮助。***老师严谨的治学态度、广阔的学术视野以及对学术的不懈追求给我留下了及其深刻的印象，令我敬佩！他的谆谆教诲使我受益终身。值此论文成稿之际，谨向***老师表示我衷心的感想和崇高的敬意！

感想***师兄对本文提出的许多富有建设性的意见和建议，同时也感想***师兄在科研工作和学习生活中给我提供的巨大帮助。感谢***等师兄师姐在生活、学习和科研上的帮助，与他们的讨论让我受益匪浅。

最后感谢我的家人对我的全力支持，正是有了他们的支持我才没有后顾之忧。同时也感谢所有在我成长过程中关心和帮助过我的朋友们，祝愿所有人在未来的学习、工作和生活中一切顺利！