一种基于加权软件行为图挖掘的软件错误定位方法

苏小红 王甜甜 杨劭君 马培军
(哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)

摘 要 已有错误定位方法通常仅给出可疑语句排序而缺少必要的上下文信息,导致难于理解软件失效的产生原因.为了解决该问题,定义了加权软件行为图来表示成功和失败的程序执行路径,由于图中边的权重表示了路径的执行频率,因此与 LEAP 方法相比,可以较好地分析与循环和递归等结构相关的软件错误.在此基础上,执行基于分支限界搜索的加权软件行为图挖掘算法,识别成功和失败执行之间最有差异的子图来获得错误签名,不但可以有效定位错误位置,还能输出缺陷语句相关的执行路径,从而提供失效产生的上下文.分析 Siemens 基准测试集和 flex 程序的结果表明,在检查相同百分比的语句的情况下,文中方法可以比 Tarantula 方法和 LEAP 方法定位到更多的错误.特别是对于冗余代码、缺失代码和变量替换,以及会直接改变执行路径类的错误,文中方法具有较高的定位精度.

关键词 错误定位;软件行为图;图挖掘;错误签名;分支限界搜索
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02175

Fault Localization Based on Weighted Software Behavior Graph Mining

SU Xiao-Hong WANG Tian-Tian YANG Shao-Jun MA Pei-Jun
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

Abstract The result of most fault localization approaches are given in the form of suspicious statement sequence ranking in descending order. However, examining a suspicious statement in isolation is hard for developers to understand the cause of failure, often they need more context information associated with the error. To solve this problem, this paper proposed a weighted software behavior graph to represent the path of passed execution or failed execution. It uses the execution frequency as weights, so that it can better handle loops, recursive and other structures than LEAP. Then, branch and bound search graph mining algorithm is performed on the weighted software behavior graphs to identify differences between passed and failed weighted software behavior graph as a bug signature. The proposed approach can not only localize the fault but also provide the failure context to facilitate fault comprehension. Experimental results on Siemens and Flex showed that the proposed approach can localize more faults than Tarantula and LEAP, when examining the same percent of code. In particular, the proposed approach has higher fault localization accuracy, especially in dealing with redundant code, missing codes, variable substitution errors, and errors that will directly change the execution path.

Keywords fault localization; software behavior graphs; graph mining; bug signature; branch and bound search

收稿日期:2015-01-12;在线出版日期:2016-10-08.本课题得到国家自然科学基金(61173021,61202092,61672191)和教育部博士点基金(20112302120052)资助.**苏小红**,女,1966 年生,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为软件缺陷检测、程序分析、信息融合、目标检测与跟踪等.E-mail:sxh@hit.edu.cn.**王甜甜**,女,1980 年生,博士,副教授,主要研究方向为程序分析、软件缺陷检测等.**杨劭君**,男,1990 年生,硕士研究生,主要研究方向为软件错误定位、图挖掘等.**马培军**,男,1963 年生,博士,教授,主要研究领域为信息融合、软件工程、目标检测与跟踪等.

1 引 言

随着计算机行业的不断发展,软件也日趋庞大和复杂,软件失效发生的概率也随之增高,这使得软件的质量越来越难以保证.在软件失效发生以后,调试软件(定位导致错误产生的代码语句并且修复错误)往往需要消耗大量的时间和人力.而自动化错误定位技术,可以显著减少其中的人力和时间消耗,从而提高软件的质量.

目前已存在很多不同的软件错误定位方法,例如基于程序状态修改的方法[1-2]、基于程序行为特征对比的方法[3-6]以及基于程序依赖关系的方法[7-9]等.这些错误定位方法的结果大多数都是一个语句的可疑值序列,其中按照可疑值降序的顺序列出了不同语句的可疑值,以指导开发者对错误进行修改.但是,单独检查给出的可疑语句,通常并不足以令开发者判断该语句是否是正确的.开发者往往需要更多的上下文信息,才能够理解错误是如何产生的,并修正程序.

因此,Hsu 等人[10]提出了能够从成功和失败的程序执行路径中识别错误签名的 RAPID 方法.错误签名是一个程序实体序列,当按顺序执行其中的程序实体时,就很可能会导致错误产生.错误签名不但可以指示错误的所在,还能够提供错误产生的上下文.Cheng 等人[11]根据程序执行路径构造软件行为图,提出了利用图挖掘算法来识别错误签名的 LEAP 方法.

本文则进一步改进了软件行为图,考虑了语句的执行频率对错误定位效果的影响,提出了加权软件行为图,以及适用于挖掘加权图的新函数,得到了错误定位精度更高的错误签名.最后与 Tarantula 方法[3]和 LEAP 方法[11]进行了实验比较,讨论了不同错误类型对错误定位精度的影响.

本文第 2 节介绍目前的相关工作和本文的研究动机;第 3 节介绍本文方法的总体思想;第 4 节介绍加权软件行为图的构造以及相应的图挖掘算法;第 5 节通过实验分析本文的方法;第 6 节讨论本文方法的局限性和可改进之处;最后总结全文.

2 相关工作及研究动机

目前的软件错误定位方法大致可以分为:程序状态修改、程序行为特征对比、程序依赖关系分析[12]和错误签名方法等.

程序状态修改方法,通常在程序执行时收集程序状态信息,并对程序状态进行修改,然后观察修改后的执行结果是否发生变化,据此找到与错误相关的关键语句.Cleve 和 Zeller[1]提出的 Delta Debugging 方法,采用分治的思想找到成功的测试输入和失败的测试输入间的最小差别,再跟踪成功和失败执行之间的状态差异,找到错误语句.Zhang 等人[2]通过在运行时强制修改谓词的取值结果,使得失败的执行变为成功的执行,并依此识别错误所在.这类方法的算法复杂度一般都较大,对小规模程序效果比较理想,但处理复杂程序时效率会大大降低.

程序行为特征对比方法,一般认为成功执行和失败执行的程序行为特征是不同的,它们之间的差异可以用于指导错误定位.这些方法通常统计语句和谓词被每个测试用例覆盖的信息来衡量每条语句出错的可能.Jones 和 Harrold[3]提出的基于语句覆盖的方法 Tarantula,认为错误语句应该在失败的执行中大量的出现,而在成功的执行中少量出现.之后研究人员提出了大量的覆盖分析方法,文献[4-5]对这些方法的等价性和精度进行了理论上的分析比较,文献[6]提出了利用机器学习方法结合多种覆盖分析错误定位方法,期望取得更准确的定位结果.然而此类方法往往只考虑了语句或谓词的覆盖信息,缺少对程序结构和执行信息的分析,会影响错误定位精度,而且难以给出与错误相关的上下文信息,开发者仍然需要做许多工作才能够修复错误.

程序依赖关系分析方法,侧重于根据程序实体间的控制依赖关系或数据依赖关系,给出语句的可疑值,还能提供额外的上下文信息,供开发者理解错误的产生.Baah 等人[7]分析了程序的控制依赖和数据依赖,产生程序依赖图.再通过执行测试,计算节点间的条件概率,建立了概率程序依赖图,可以有效的应用于错误定位和错误理解.苏小红、龚丹丹等人[8-9]定义了联合依赖概率模型,在定位过程中分析程序元素间的控制依赖、数据依赖以及语句执行状态.这类方法的结果仍可能包含大量信息,这些冗余信息会造成难以找到与错误相关的信息.

Hsu 等人[10]提出了识别错误签名的 RAPID 方法,通过增量的分析失败的程序执行路径的公共序列,将挖掘得到可疑语句序列作为错误签名,辅助开发人员定位和理解软件错误.

Cheng 等人[11]对错误签名方法进行了改进,本文将其称为 LEAP 方法,首先利用程序执行路径构

造软件行为图，再通过 LEAP 图挖掘算法，利用信息增益作为目标函数，从成功测试用例和失败测试用例的软件行为图集合中挖掘出 Top-K 个在失败的执行中出现很频繁，而在正确的执行中比较少见的最优局部软件行为图作为可疑语句序列. 该方法能够给出失效产生的上下文且避免了冗余信息，因此有助于定位和理解软件错误. 然而存在的一个不足之处是软件行为图中所有的控制流路径具有相同的重要程度，而没有考虑语句执行频率与软件失效之间的关联.

例如错误的条件表达式或循环体内的某些缺陷语句，可能导致循环执行路径和次数的改变，进而导致软件失效. 对于这种情况如果采用 LEAP 方法进行分析，那么执行了多次循环的测试用例和只执行 1 次循环的测试用例可能具有相同或相似的软件行为图，如果这两个软件行为图分别对应于成功和失败测试用例的软件行为图，则图挖掘算法将因无法在这两个软件行为图中区分成功执行和失败执行的差异而失效，从而不能有效定位到软件错误. 类似地，错误的递归也可能导致函数执行路径的改变. 由于循环和递归是编程语言的基本元素，需要提供一种机制来表示程序执行路径的转移概率，使之可以有效区分成功和失败执行路径. 本文第 4 节中用实例进行了详细分析.

综上所述，目前大多数方法在错误理解方面做的还不够完善，要么只提供可疑语句的列表，缺少可以用于理解错误产生原因的有效上下文信息，要么提供了过多的信息，开发者难以从中找到与错误相关的信息，都不利于开发者理解错误是如何产生的. 错误签名能够较好的平衡过多和过少的信息，但仍需要考虑程序执行路径的统计学意义，而不仅仅是频繁图的挖掘. 本文的研究动机就是如何结合统计信息和图挖掘方法进一步提高错误签名的错误定位精度.

3　方法概述

3.1　方法框架

输入是待测程序 P，以及相应的测试用例集合. 根据程序 P 执行测试用例的实际输出结果，可以将测试用例分为两类：成功的测试用例（P 的实际输出结果与预期输出结果相同）和失败的测试用例（P 的实际输出结果与预期输出结果不同）.

本文方法主要包含 3 个步骤：获取程序执行路径、构造加权软件行为图和挖掘错误签名，其流程如图 1 所示.

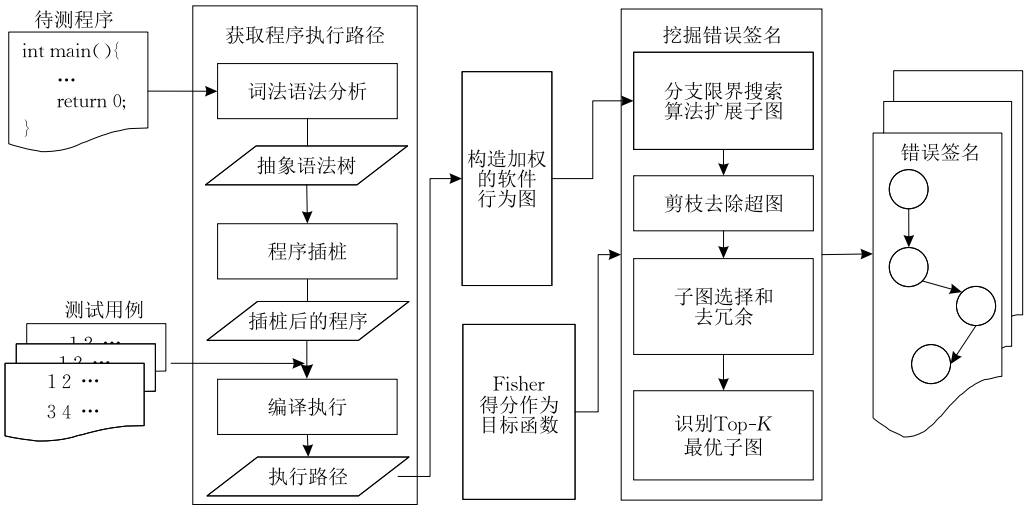


图 1　基于加权软件行为图挖掘的错误定位方法的基本流程

软件测试和调试过程通常协作来检测和定位软件缺陷，如果在测试过程中获得了测试用例的路径信息，则可直接执行构造加权软件行为图和挖掘错误签名. 同大多数错误定位方法一样，本文假设测试用例集提供了充分的用以错误定位的测试覆盖信息，能够保证测试覆盖率，错误定位方法的研究重点主要放在如何在这一前提条件下快速准确地定位错误所在的语句.

3.2　本文方法与 LEAP 方法的差异

LEAP 方法将测试用例的执行路径表示为软件行为图，不区分路径的执行频率，因此对应的图挖掘算法仅是挖掘成功和失败测试用例集合执行路径中

的频繁子图.方法是采用信息增益作为目标函数,用LEAP算法挖掘出区分失败执行和正确执行差异的频繁子图.

而本文定义的加权软件行为图不但表示了程序软件执行路径,还用边的权重表示了程序路径的执行频率,表示了程序执行的统计信息,在本质上区别于软件行为图.此外,本文方法在基于图挖掘定位软件缺陷时,需要考虑路径的权重,即路径执行的统计信息,因此,采用了不同于LEAP的目标函数和图挖掘算法.具体方法如图1所示,用Fisher得分作为目标函数,用分支界限搜索方法挖掘错误签名,并且提出了一系列用来提高挖掘效率和准确率的剪枝和去冗余等算法.

4 关键技术

4.1 获取程序执行路径

为了得到程序的执行路径,需要对待测程序的源代码进行插桩.插桩是在保证待测程序原有逻辑和功能不变的条件下,向程序源代码中插入探针语句,用来在程序被执行时输出程序执行信息.

本文实现语句级别的程序插桩,即跟踪程序中语句的执行信息,每当执行到一条语句,就将被执行语句的行号作为执行路径输出.

首先,需要对待测程序进行词法分析和语法分析,将源代码转换为等价的抽象语法树,然后对抽象语法树进行分析,找到适合插入探针语句的位置,以保证额外的语句不会影响到现有的程序逻辑和功能.例如对于C语言中的条件语句和循环语句,由于它们需要在每次执行条件判断时输出行号信息,因此利用逗号表达式直接将探针语句插入到条件表达式中;对于跳转语句,由于它们会立即改变程序的执行路径,因此必须在其之前插入探针语句.接下来将插入探针的抽象语法树反向生成相应的程序代码,即可得到插桩后的程序.

最后,当执行插桩完毕的待测程序时,插入的探针语句也会被执行,从而输出程序的执行路径.如果探针语句将当前正在执行语句的行号输出,那么得到的程序执行路径中,每个节点就表示待测程序中的一条语句.

以图2所示的findmaxIndex程序为例,它是在数字数组中找到最大值首次出现的位置.main函数中调用了findmaxIndex函数.为了避免冗余说明,本文将main函数中的n和$nums$的读取用了伪代码描述.3组不同的测试用例及其相应的程序执行路径如表1所示.当执行到main函数中的函数调用语句$findmaxIndex(n,nums)$;时,程序的执行路径发生转移,执行findmaxIndex函数中的语句,执行完findmaxIndex后,返回到main函数中第19行.

```
1.  void findmaxIndex(int n, int* nums) {
2.      int i=0, maxIdx=-1;
3.      while (i<n) {
4.          if (maxIdx==-1) {
5.              maxIdx=i;
6.          } else if (nums[i]>=nums[maxIdx]) {
                                        //错误,应该使用>号
7.              maxIdx=i;
8.          }
9.          i++;
10.     }
11.     printf("%d", maxIdx);
12. }
13. void main( )
14. {
15.   int n, *nums;
16.   read(n);
17.   read(nums);
18.   findmaxIndex(n,nums);
19. }
```

图2 findmaxIndex示例程序

表1 findmaxIndex程序的测试用例以及其执行路径

序号	测试用例	分类	程序执行路径
1	$n=3, nums=\{8,20,13\}$	成功	{15,16,17,18,2,3,4,5,9,3,4,6,9,3,4,6,9,3,11,19}
2	$n=3, nums=\{8,18,18\}$	失败	{15,16,17,18,2,3,4,5,9,3,4,6,7,9,3,4,6,7,9,3,11,19}
3	$n=4, nums=\{8,18,18,10\}$	失败	{15,16,17,18,2,3,4,5,9,3,4,6,7,9,3,4,6,7,9,3,4,6,9,3,11,19}

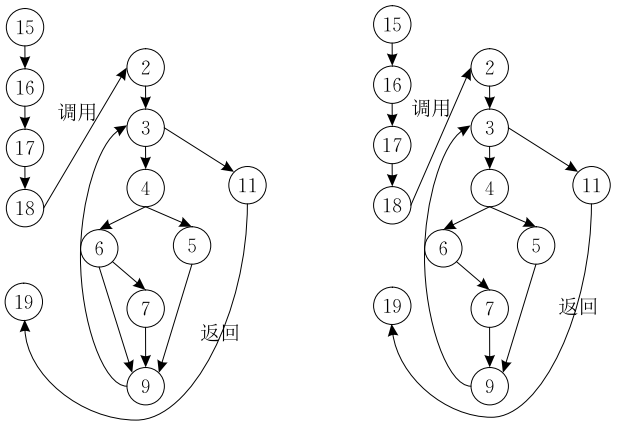
4.2 构造加权软件行为图

表1中的测试用例2执行了两次循环结构,因此路径中包含了两组语句序列3,4,6,7,9,导致其执行路径长于测试用例1的执行路径.实际程序中,循环可能会执行若干次,且如果循环结构中包含较多的执行语句,则会导致过长的执行路径.特别是当程序中包含大量循环结构的时候,分析起来会非常困难,需要消耗大量的空间和时间.而且,由于其中包含的信息过多,很难从中提取关键信息,因此很有必要对程序执行路径进行进一步处理.

为了解决这一问题,Cheng等人[11]提出了软件行为图.软件行为图是一幅有向图,图中的顶点对应于程序执行路径中包含的语句,顶点间的边对应于程序语句执行时的顺序关系.软件行为图有效地压

缩了程序执行路径，在保留了程序执行路径的特征信息的同时，去除了很多冗余信息，可以有效减少数据量，节省时间和空间，而且更有利于进一步的处理.

例如表 1 语句 3，4，6，7，9，在测试用例 2 中被循环执行了 2 遍，但是行为图中通过将其表示为控制流图的循环结构形式，使这些节点只出现一次，有效压缩了该路径的表示，如图 3(b)所示. 同理如果其他测试用例执行多次该循环结构，节点 3，4，6，7，9，仍只需表示一次.



(a) 测试用例1和3　　(b) 测试用例2

图 3　*findmaxIndex* 程序各个执行路径的软件行为图

软件行为图可以认为是与特定执行相关的控制流图. 如果执行路径覆盖到了所有的语句和所有可能的语句执行顺序，那么软件行为图就与控制流图是相同的；如果执行路径仅覆盖到了部分语句，或者未能包含全部可能的语句执行顺序，那么就只包含控制流图的一部分. 当存在函数调用时控制流图可通过增加函数调用边和返回边来表示为过程间控制流图，同样软件行为图也可以类似地表示函数调用关系[11].

但是，软件行为图中只包含了语句是否被执行以及语句执行顺序的信息，而没有考虑到语句执行频率对错误定位结果的影响，不能很好的处理循环和递归等结构. 例如，对于表 1，成功测试用例 1 和失败测试用例 3 的执行路径是不同的，但是却具有相同软件行为图表示，如图 3(a)所示. 如果比较测试用例 1 和 3 的软件行为图则因无法区分成功执行与失败执行之间的差异而无法有效定位缺陷语句.

基于以上分析，本文提出了加权软件行为图.

定义 1(加权软件行为图).　加权软件行为图是一个四元组(V,E,L,W)，其中 V 是顶点集合，包含程序执行路径中出现的所有程序实体，$E\subseteq V\times V$ 是有向边的集合，L 是边与其标签的映射，W 是边与其权重的映射.

边$\langle v_i,v_j\rangle\in E$ 的标签，可能是“调用”、“转移”或“返回”之一：

如果 v_i 调用了 v_j，$\langle v_i,v_j\rangle$ 就被标记为“调用”.

如果从 v_i 返回到了 v_j，则将 $\langle v_i,v_j\rangle$ 标记为“返回”.

否则将 $\langle v_i,v_j\rangle$ 标记为“转移”，表示 v_j 紧跟在 v_i 之后执行.

这 3 种标签表示了程序实体间的 3 种可能的控制流关系. 每条边还附加有一个权重 w. 由于在待测程序的不同执行中，同一条语句的执行次数可能会有很大区别，因此直接将语句执行次数作为权重并不合适，而是采用语句的执行概率作为权重.

若图中某条边 e_i 的起始顶点和结束顶点分别为 p_k 和 p_j，p_k 的出边(以 p_k 为起始顶点的边)集合 $E=\{e_1,e_2,\cdots,e_n\}$，其中每条边在执行路径中出现的次数分别为 c_i，则边 e_i 的权重计算公式为

$$w_i=\frac{c_i}{\sum_{j=1}^{n}c_j} \tag{1}$$

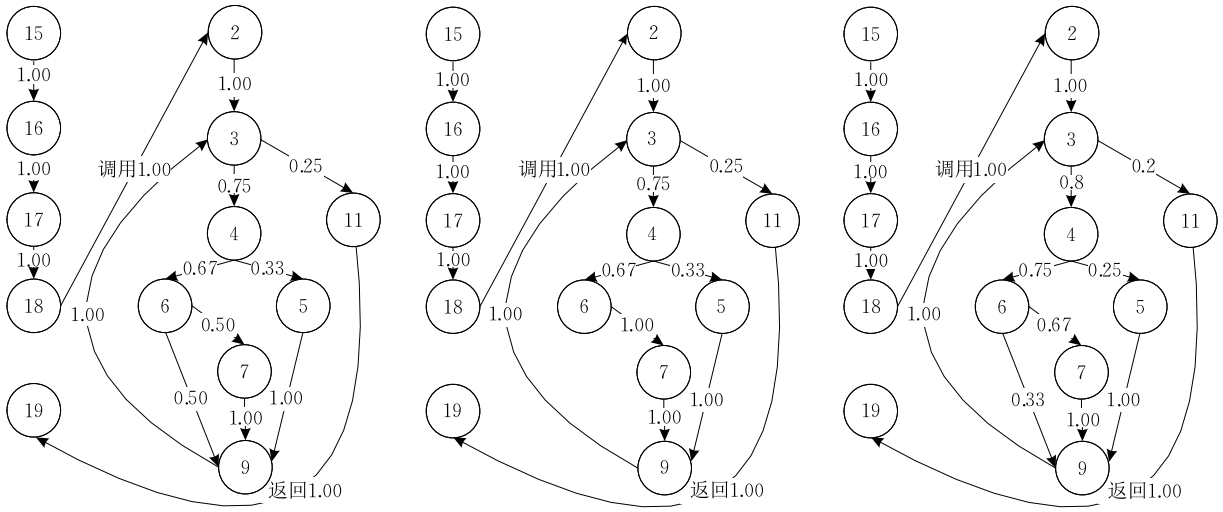
权重 w_i 表示了在此次执行中，顶点 p_k 被执行后，顶点 p_j 会被执行的概率. 对于同一个顶点的出边，它们的权重和为 1，即有

$$\sum_{i=1}^{n}w_i=1 \tag{2}$$

含有条件判定的语句可能有多条执行路径分支，边的权重代表了各个路径的执行频率. 如图 4(a)所示，对于测试用例 1 的路径 1，其语句 3 有两个分支 3→4 和 3→11，它们在循环中的执行次数分别为 3 和 1，因此边 3→4 的权重为 3/(3+1)=0.75，边 3→11 的权重为 0.25.

不同执行频率的路径意味着不同的执行上下文，也可能导致不同的执行结果(成功/失败). 图 4(a)成功测试用例 1 和图 4(c)失败测试用例 3 的加权行为图中的边 3→4，3→11，4→6，4→5，6→9，6→7 的权重不同，有效区分了成功测试用例和失败测试用例的软件行为图，尤其是对与缺陷语句相关的执行路径做出了很好的区分.

有了程序执行路径，只要不断的将执行路径中出现的边添加到有向图中，就可以得到软件行为图. 按照式(1)计算边的权重，就得到了加权软件行为图. 其算法如算法 1 所示，使用 $V(G)$ 表示加权软件行为图 G 的顶点集，使用 $E(G)$ 表示加权软件行为



(a) 测试用例1 (b) 测试用例2 (c) 测试用例3

图 4 findmaxIndex 程序各个执行路径的加权软件行为图

图 G 的边集.

算法 1. 构造加权软件行为图.

输入:待测程序的执行路径

输出:相应的加权软件行为图 G

1. $G=\varnothing$;
2. $last$=NULL;
3. FOR 执行路径中的每条语句 s
4. IF $last$ 不为 NULL
5. $V(G)=V(G)\bigcup\{s\}$;
6. $E(G)=E(G)\bigcup\{\langle last,s\rangle\}$;
7. $Count_{\langle last,s\rangle}=Count_{\langle last,s\rangle}+1$;
8. $Sum_{last}=Sum_{last}++$;
9. END IF
10. $last=s$;
11. END FOR
12. FOR G 中的每个顶点 v
13. FOR 以 v 为起始顶点的每条边$\langle v,k\rangle$
14. $\langle v,k\rangle$ 的权重为 $Count_{\langle last,s\rangle}/Sum_{last}$
15. END FOR
16. END FOR
17. RETURN G;

4.3 识别错误签名

使用不同的测试用例多次执行待测程序,可以得到一个加权软件行为图的集合 G. 根据执行的测试用例是成功的还是失败的,可以将 G 划分为两个子集,一个包含与成功的测试用例对应的加权软件行为图,记为 G^{pass};另一个包含与失败的测试用例对应的加权软件行为图,记为 G^{fail},$G=G^{\text{pass}}\bigcup G^{\text{fail}}$.

在理想情况下,有

$$G^{\text{pass}}\bigcap G^{\text{fail}}=\varnothing \tag{3}$$

在理想情况下,G^{fail} 即为需要分析的可能包含缺陷语句的子图,然而,现实情况下,失败测试用例和成功用例常常执行一些公共的程序语句(既可能包含正确也可能包含缺陷语句). 因此,G^{pass} 与 G^{fail} 的交集不一定为空. 例如图 4 例子中 $G^{\text{pass}}=\{$子图(a)$\}$,$G^{\text{fail}}=\{$子图(b),子图(c) $\}$,$G^{\text{pass}}\bigcap G^{\text{fail}}\neq\varnothing$ 包含了图 4 中所有语句节点. 加权软件行为图的挖掘目标,是在 G 中找到一个最优的子图 g,使得 g 可以指出错误所在,并提供与错误相关的上下文信息. 而这个最优的子图 g,就被称为错误签名. 例如图 4 例子中的最优子图即错误签名为 6→7.

通常认为,错误语句应该被失败测试用例大量的执行,而被成功测试用例少量的执行,即错误语句在 G^{fail} 和 G^{pass} 中的出现频率应当有显著的区别. 那么,可以设计一个目标函数 $F(g)$,来评价子图 g 对成功的执行和失败的执行间差异的辨别能力. 该问题就转化为在 G 中,寻找一个最优子图 g,使得 $F(g)$最大的优化问题.

本文采用 Fisher 得分作为目标函数 $F(g)$的判据,Fisher 得分以最大化类间离散度和最小化类内离散度为目标准则[13]. 在 G^{fail} 中出现频率普遍较高,而在 G^{pass} 中出现频率普遍较低的子图,会具有更高的 Fisher 得分,它也可以更好的指出成功和失败的执行间的差异.

令 $G_1,G_2,\cdots,G_m$ 表示 m 次不同的执行对应的加权软件行为图,对于特定的子图 g,其边集 $E=\{e_1,e_2,\cdots,e_n\}$,使用 f_i^j 表示边 e_i 在 G_j 中出现的频率,其定义为

$$f_i^j=\begin{cases}w_i^j, & g\subseteq G_j\\ 0, & g\not\subset G_j\end{cases} \tag{4}$$

其中，w_i^j表示在图G_j中，与边e_i具有相同起始顶点和结束顶点的边的权重.

使用μ_i^{pass}和μ_i^{fail}表示边e_i在G^{pass}和G^{fail}中出现频率的均值：

$$\mu_i^{\text{pass}}=\frac{1}{|G^{\text{pass}}|}\sum_{G_j\in G^{\text{pass}}}f_i^j,$$

$$\mu_i^{\text{fail}}=\frac{1}{|G^{\text{fail}}|}\sum_{G_j\in G^{\text{fail}}}f_i^j \tag{5}$$

使用$(\sigma_i^{\text{pass}})^2$和$(\sigma_i^{\text{fail}})^2$表示边$e_i$在$G^{\text{pass}}$和$G^{\text{fail}}$中出现频率的方差：

$$(\sigma_i^{\text{pass}})^2=\frac{1}{|G^{\text{pass}}|}\sum_{G_j\in G^{\text{pass}}}(f_i^j-\mu_i^{\text{pass}})^2$$

$$(\sigma_i^{\text{fail}})^2=\frac{1}{|G^{\text{fail}}|}\sum_{G_j\in G^{\text{fail}}}(f_i^j-\mu_i^{\text{fail}})^2 \tag{6}$$

运用 Fisher 准则，得到边e_i的 Fisher 得分为

$$F_i=\frac{(\mu_i^{\text{pass}}-\mu_i^{\text{fail}})^2}{(\sigma_i^{\text{pass}})^2+(\sigma_i^{\text{fail}})^2} \tag{7}$$

最后，将g中所有边的 Fisher 得分求平均数，得到目标函数$F(g)$的计算公式为

$$F(g)=\frac{1}{n}\sum_{i=1}^{n}F_i \tag{8}$$

4.3.1　剪枝策略

由于G中可能包含指数级别的子图，因此直接进行搜索会消耗大量的时间，需要对搜索过程运用剪枝策略.

对于某一子图g，它在G^{pass}和G^{fail}中出现的频率一定不小于其超图的出现次数. 即g的超图在G^{pass}中出现的次数大于等于零，且小于等于g在G^{pass}中出现的次数；在G^{fail}中出现的次数大于等于零，且小于等于g在G^{fail}中出现的次数. 因此，g的超图的得分具有一个上限值，可以通过判断该上限来进行剪枝.

边e_i的 Fisher 得分上限为

$$F_{\max i}=\max\left(\frac{(\mu_i^{\text{pass}})^2}{(\sigma_i^{\text{pass}})^2},\frac{(\mu_i^{\text{fail}})^2}{(\sigma_i^{\text{fail}})^2}\right) \tag{9}$$

g的超图的目标函数值上限为

$$F_{\max}(g)=\frac{1}{n}\sum_{i=1}^{n}F_{\max i} \tag{10}$$

4.3.2　子图的选择和去冗余

在加权软件行为图中，并不是所有子图都可能是错误签名.

首先，错误签名应当是有一定实际意义的子图，能够表示错误的上下文，即如果按照子图给出的顺序执行程序，就很可能导致错误产生. 包含复杂分支的子图，如图 5 所示，反而会让开发者难以理解其表示的意义，不利于理解错误产生的原因. 因此，可以作为错误签名的子图，应当具有执行顺序，需要在选择子图时予以保证.

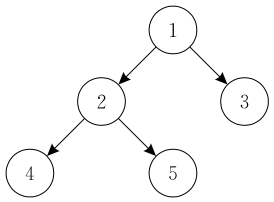


图 5　不利于理解错误产生原因的子图

其次，方法的调用与方法的返回是一一对应的，即如果方法a调用了方法b，那么一定会从方法b中返回到a. 如果方法c也调用了方法b，那么也一定会从b中返回到c. 将该执行过程转换为软件行为图，结果如图 6 所示. 这里有效的子图只有$a\to b\to a$和$c\to b\to c$，而子图$a\to b\to c$和$c\to b\to a$都是无效的子图，不可能在实际运行中出现. 在选择子图的时候必须将无效的子图去除，否则可能会影响错误定位的结果. 过滤子图的算法如算法 2 所示，使用堆栈来检查调用边是否与返回边一一对应. 例如对于有效的方法调用返回序列$a\to b\to c\to b\to a$而言，首先调用边$a\to b$入栈，调用边$b\to c$入栈，接下来分析返回边$c\to b$，此时栈顶是$b\to c$与$c\to b$对应，为有效路径，栈顶$b\to c$出栈，$a\to b$变为栈顶，此时分析$b\to a$与栈顶对应，$a\to b$出栈，判定路径有效. 如果当前分析的返回边e与栈顶对应，则栈顶出栈，否则说明所分析的路径为无效路径，于是在清空堆栈后返回.

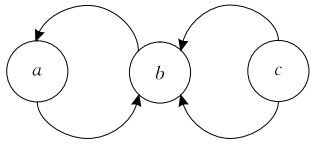


图 6　被两个方法调用的软件行为图

算法 2.　过滤无效子图.

输入：需要检测是否有效的子图g

输出：子图是有效的还是无效的

```
1.  stack=∅;
2.  FOR g 中的每条边 e
3.      IF e 为调用边
4.          stack.push(e);
5.      ELSE IF e 为返回边
6.          IF stack.top 与 e 不对应
7.              stack=∅;
8.              RETURN 无效;
9.          ELSE
```

```
10.        stack.pop();
11.      END IF
12.    END IF
13. END FOR
14. RETURN 有效;
```

最后,在挖掘得到的多个错误签名之间,可能具有包含关系,因此定义下列规则,来合并相互包含的错误签名,减少冗余.

对于两个存在包含关系的错误签名 M 和 N(无论 $M\subset N$ 还是 $M\supset N$):

若 $F(M)\geq F(N)$,则删去 N.

这样,存在包含关系的错误签名中,只会保留得分更高的错误签名,可以有效地减少冗余信息.

4.4 基于分支限界搜索的加权软件行为图挖掘

本文采用如算法 3 所示的分支限界搜索算法来挖掘错误签名.在错误定位中,仅仅报告唯一的最佳错误签名,往往并不足以提供足够的信息.按照可疑度降序给出多个不同的错误签名,更有助于开发者理解错误,因此算法 3 会返回 Top-K 个错误签名.

算法 3. 挖掘 Top-K 错误签名.

输入:加权软件行为图集合 G,结果数 K

输出:Top-K 错误签名 g^*

```
1.  S={G 中只包含一条边的子图};
2.  g*=∅;
3.  WHILE S≠∅
4.    从 S 中选择 g,S=S\{g};
5.    IF g 未检查过且有效
6.      用式(8)和(10)计算 F(g)和 F_max(g);
7.      IF F(g)> min_{g'∈g*} {F(g')}
8.        g*=g*∪{g};
9.        移除 g* 中的冗余结果.
10.       IF g* 包含的错误签名数量大于 K
11.         移除 g* 中得分最小的错误签名
12.       END IF
13.     END IF
14.     IF F_max(g)> min_{g'∈g*} {F(g')}
15.       扩展 g 的最后一个节点并将其加入 S;
16.     END IF
17.   END IF
18. END WHILE
19. RETURN g*;
```

为了选择具有执行顺序的子图,算法 3 每次只会扩展子图的最后一个节点,而不是在任意位置进行扩展.该策略在提高了执行效率的同时,也保证了子图能够具有执行顺序.

最后得到的 Top-K 错误签名,就可以作为错误定位的依据,错误签名中包含的节点对应于可能导致错误发生的语句,可以用于进行错误理解.

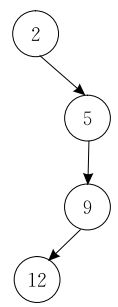
4.5 错误签名实例

以 Siemens Suite[14] 中的 replace 程序版本 9 为例展示本文方法挖掘得到的错误签名.该程序的 *dodash* 函数中,存在如图 7(a)所示的缺失代码错误:

第 3 行代码中包含一个缺失代码错误,if 语句的条件中缺失了一个条件,导致在某些情况下,本应执行第 10 行 else 语句块内的代码,但实际执行的却是第 5 行的 for 循环,从而产生错误.

```
1 if (src[*i]!=DASH){…}
2 else if(isalnum(src[*i-1])&&(isalnum(src[*i+1])))
3    /*&&((src[*i-1]<=src[*i+1]) missing code*/
4 {
5    for(k=src[*i-1]+1; k<=src[*i+1];k++)
6    {
7       junk=addstr(k, dest, j, maxset);
8    }
9    *i=*i+1;
10 }
11 else{…}
12 (*i)=(*i)+1;
```

(a) 缺陷代码



(b) 错误签名

图 7 replace 程序版本 9 的代码片段及其得到的错误签名

本文定位方法得到的错误签名如图 7(b)所示,错误签名中的每个数字都与图 7(a)所示代码片段的语句行号对应.该错误签名表示错误语句就在第 2、5、9 和 12 这 4 行代码之中,而且如果按照语句 2、5、9、12 的顺序执行代码,就很可能会导致错误.开发者就可以根据错误签名给出的执行顺序,去理解为什么会产生错误,找到并修正真正引起错误的语句.

注意到产生错误的原因,是因为没有满足条件 $src[{}^*i-1]<=src[{}^*i+1]$,因该条件缺失导致改变了程序的执行路径,即本应执行语句 10 却执行了语句 5.而此时,必有 $src[{}^*i-1]>src[{}^*i+1]$(因缺失的条件 $src[{}^*i-1]<=src[{}^*i+1]$ 未被满足),所以有 $k=src[{}^*i-1]+1>src[{}^*i+1]$,这样 for 循环的条件 $k<=src[{}^*i+1]$ 一定为假,因此会跳过循环体直接执行语句 9,最后执行语句 12.因此,实际产生错误的执行路径是 2→5→9→12,与图 7(b)所示的错误签名是相同的.

这个例子说明,错误签名与实际产生错误的执行路径是相同的,因此错误签名相当于指出了错误的执行路径,有了这样的错误上下文信息,我们就可

以通过分析为什么会执行这个错误的路径，原因一定是分支的判断条件限制不够严格，从而发现真正产生错误的语句是语句 2. 显然这比单纯地给出最可疑的语句是语句 2 更有助于开发者理解错误产生的原因，也更有利于开发者修正错误.

5　实　验

采用 C# 语言实现了 C 语言词法语法分析工具 CParser、程序插桩工具和本文方法、LEAP、Tarantula 方法.

5.1　实验建立

本文方法采用被广泛应用于评价软件错误定位方法有效性的 Siemens Suite[13] 和 UNIX 程序集 flex(下载于 http://sir. unl. edu)作为测试数据. Siemens 测试集全部采用 C 语言编写，程序规模较小，并且由人工注入错误，每个程序都包含了数千个测试用例. flex 是 UNIX 系统下的一个真实的 C 语言程序，其规模较大，而且其中包含的错误都是生产过程中真实产生的，但测试用例数量较少. 表 2 描述了 Siemens 测试集和 flex 的程序信息，包括程序名称、错误版本数、代码行数、测试用例数及其描述.

表 2　实验数据

程序	错误版本数	代码行数	测试用例数	描述
print_tokens	7	472	4130	Lexical analyzer
print_tokens2	10	399	4115	Lexical analyzer
replace	32	512	5542	Pattern replacement
schedule	9	292	2650	Priority scheduler
schedule2	10	302	2710	Priority scheduler
tcas	41	141	1608	Altitude separation
tot_info	23	440	1052	Information measure
flex	23	8571～10124	567	Lexical parser

表中列出了 132 个 Siemens 错误版本，但由于其中一些版本没有错误测试用例，或者错误位于头文件中，或者运行时产生了段错误，因此只采用了其中的 121 个错误版本. 再加上 flex 程序的 23 个错误版本，本文实验最终使用 144 个错误版本进行实验. 实验的运行环境是 CPU Inter® Pentium® Dua 4-Core 3. 2GHz；内存 2GB.

5.2　评价指标

实验选择与 Tarantula[3] 方法和 LEAP[11] 方法进行比较. Tarantula 是一种实现比较简单的基于覆盖的错误定位方法，它的错误定位效果较好，常被用来作为错误定位方法的比较对象. LEAP 则是一种基于图挖掘的错误定位方法.

由于 Tarantula 方法会产生语句的可疑值列表，因此采用文献[3]使用的错误定位方法精度评价指标 Score. 如果开发者从可疑值最高的语句开始，按照可疑值降序的顺序检查，直到找到了错误所在的语句为止，那么 Score 就表示了在找到错误之前需要检查的代码百分比. 它的计算方法如式(11)所示.

$$Score=\frac{|N|}{|Lines|}\times 100\% \tag{11}$$

其中，$|N|$ 表示在找到错误之前需要检查的语句行数，$|Lines|$ 表示可执行语句的行数.

由于本文方法和 LEAP 方法产生的是 Top-K 错误签名，因此按照前 K 个可疑值降序的顺序检查每个错误签名，将找到错误语句前需要检查的语句百分比作为 *Score*.

代码检查率的计算类似于 *Score*，是按照定位结果可疑度排序从高到低检查的代码行数占程序中可执行语句行数的百分比. 两者的区别是：在计算代码检查率时可能定位到也可能没有定位到缺陷语句，而 *Score* 是在定位到缺陷语句的情况下计算的.

5.3　实验结果总体分析

图 8 展示了 Siemens 测试集上，本文方法与 Tarantula 方法和 LEAP 方法的总体错误定位精度折线图，横轴表示需要检查的语句百分比，纵轴表示检查相应百分比的语句时，能够找到错误的错误版本百分比. 横轴是从 1%开始的，表示只检查 1%的语句，可以找到错误的错误版本百分比. 可以看到，在检查相同百分比的语句时，本文方法总是可以比 Tarantula 方法和 LEAP 方法定位到更多的错误，而且至多只需要检查 65%的代码，本文方法就可以定位到所有的错误.

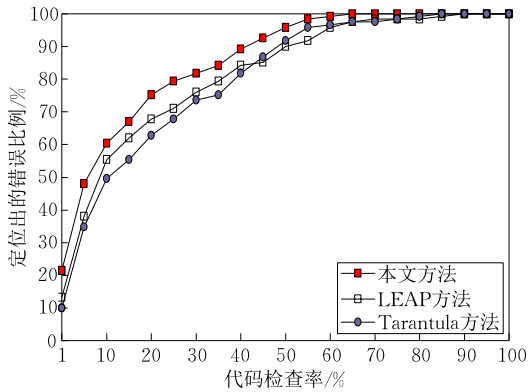
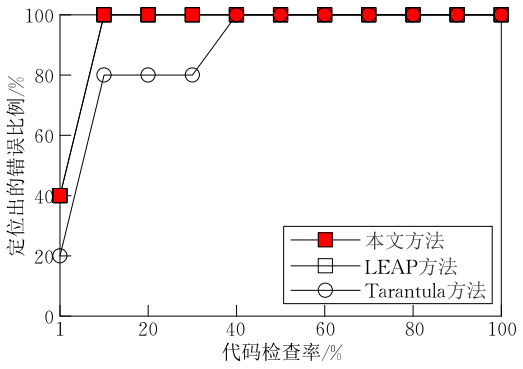
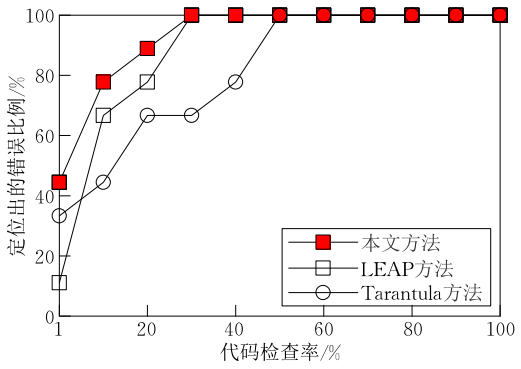


图 8　Siemens 测试集错误定位精度比较

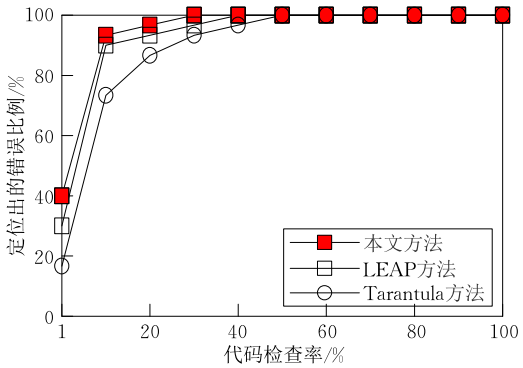
图 9 是 Siemens 测试集中单个程序的错误定位结果. 可以很明显的看到，对于不同的程序，错误定位的精度有较大的差异.



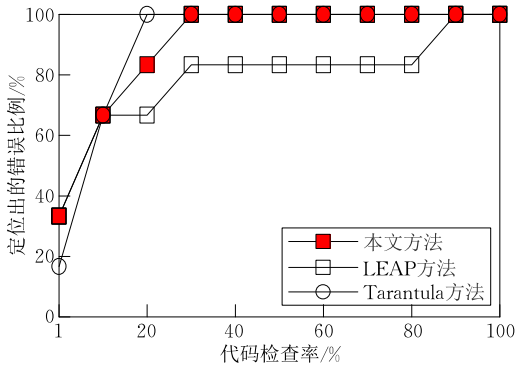
(a) print_tokens



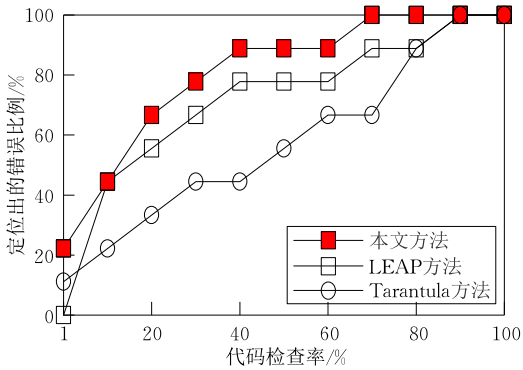
(b) print_tokens2



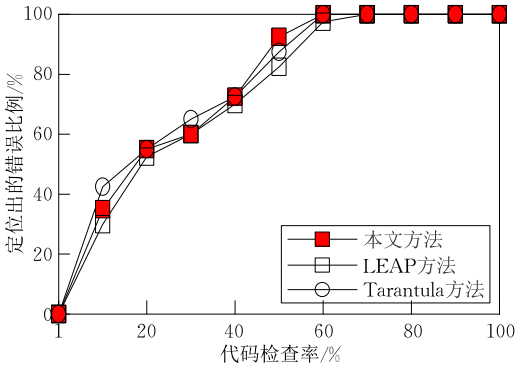
(c) replace



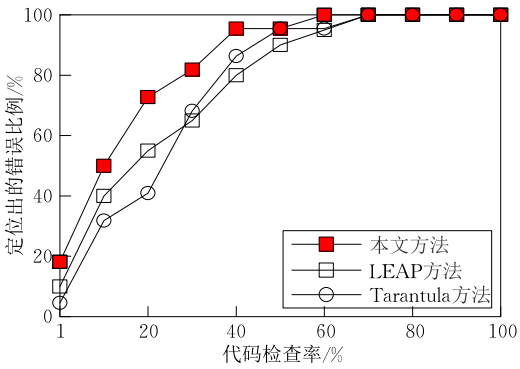
(d) schedule



(e) schedule2



(f) tcas



(g) tot_info

图 9 单个程序的错误定位精度比较

对于 print_tokens、print_tokens2 和 replace 程序，所有错误定位方法的效果都较好，而本文方法效果最好；对于 schedule 程序，错误定位效果也比较好，但是 LEAP 方法表现较差；对于 schedule2 和 tot_info 程序，仍然是本文方法的错误定位精度最高，但明显无法与前 4 个程序的错误定位精度相比；

而对于 tcas 程序，则是 Tarantula 方法的错误定位的精度要更高些，本文方法的错误定位精度略高于 LEAP 方法.

图 10 则是 flex 程序的错误定位精度比较，可以看到 3 种方法的错误定位精度都较高，而且仍然是本文方法效果最好，其次是 LEAP 方法，最后是 Tarantula 方法.

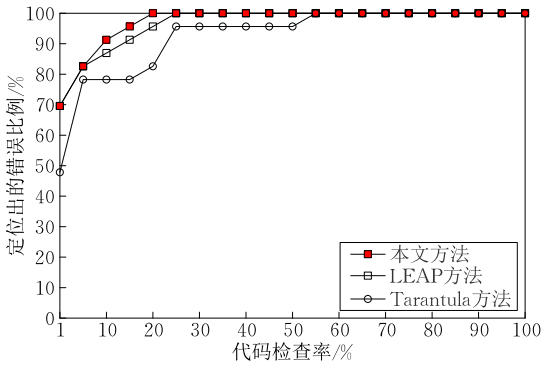


图 10　flex 程序错误定位精度比较

为了找到不同程序上的错误定位精度差异较大的原因，我们在下面的实验中进一步按照不同的错误类型分别对错误定位结果进行分析.

5.4　针对不同错误的结果分析

实验使用的 144 个错误版本，按照错误类型进行分类，总计包含 14 个冗余代码错误(添加了多余的代码)，32 个缺失代码错误(删除了部分代码)，34 个运算符变异错误(运算符发生了改变)，46 个常量变异错误(常量值发生了变化)，11 个变量替换(使用的变量发生了改变)和 7 个其他错误. 采用箱式图显示每种错误类型的错误定位结果. 箱式图描述了数据的统计结果，它由数据样本的五个重要数据统计点组成：最大值，最小值，四分之一位数，四分之三位数和中位数.

这里以 Tarantula 方法作为比较的基准，将 Tarantula 方法与本文方法的 *Score* 之差作为纵轴，记为 *ScoreChange*，如式(12)所示.

$$ScoreChange = Score_{\text{Tarantula}} - Score_{\text{本文方法}} \quad (12)$$

若 *ScoreChange* 大于零，表示本文方法检查更少的代码就能定位到错误，定位精度优于 Tarantula 方法；若 *ScoreChange* 小于零，则表示本文方法需要检查更多的语句才能定位到错误，定位精度比 Tarantula 方法要差.

从图 11 可以看到，对于冗余代码、缺失代码和变量替换错误，本文方法大多都只需要检查更少的语句，就能够定位到错误，而少部分精度变差的，也小于 9%. 而对于运算符变异、常量变异和其他错误，本文方法则有优有劣，约有 47% 的错误定位精度变差了，但变差幅度大于 10% 的只占 9%.

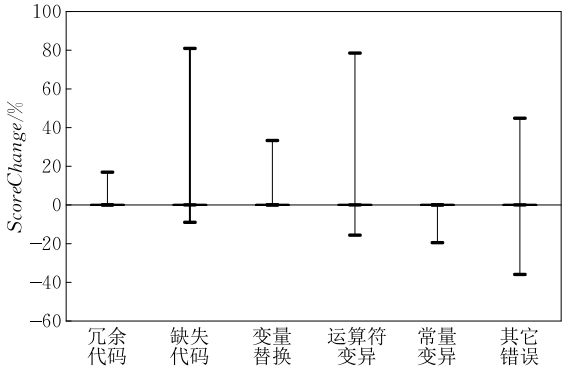


图 11　不同错误类型的 *ScoreChange* 分布

再考虑另一种错误分类方式. 部分错误可能会直接导致程序的执行路径发生改变，例如 if 语句中的条件发生了错误，会直接令程序执行路径发生变化. 而另一些错误则不会直接导致程序执行路径发生改变，例如变量赋值中发生了错误，程序执行路径并未立刻发生变化，只有之后再用到该变量时，程序执行路径才可能发生变化.

按照这一方式来分类，实验使用的 144 个错误版本，包含 59 个会直接导致程序执行路径发生改变的错误，85 个不会直接导致程序执行路径改变的错误，同样使用箱式图显示两类错误的定位结果，如图 12 所示. 可以发现，对于会直接改变程序执行路径的错误，本文方法基本都是优于 Tarantula 方法的，而对于不会直接导致程序执行路径发生变化的错误，本文方法同样是有优有劣，约有 47% 的错误定位精度变差了，而变差幅度大于 10% 的约占 7%.

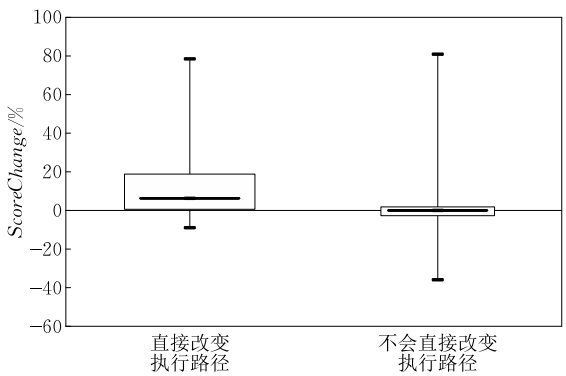


图 12　是否直接改变执行路径的错误的 *ScoreChange* 分布

综合分析以上两个实验，可以发现本文方法对不同的错误类型，定位精度有比较明显的不同：对于冗余代码、缺失代码和变量替换错误，以及会直接改变执行路径的错误的定位精度明显更高. 如果将这

些错误类型记为优势错误,将剩余的错误类型(不会直接改变执行路径的运算符变异、常量变异和其他错误)记为劣势错误,那么优势错误有 92 个,劣势错误有 52 个,优势错误和劣势错误的 *ScoreChange* 对比结果如图 13 所示. 可以看到,对于优势错误,本文方法的定位精度大多都优于 Tarantula 方法,而对于劣势错误,本文的定位精度则普遍要略差于 Tarantula 方法. 图 14 所示的优势错误和劣势错误的错误定位精度比较也证明了这一点,而且LEAP

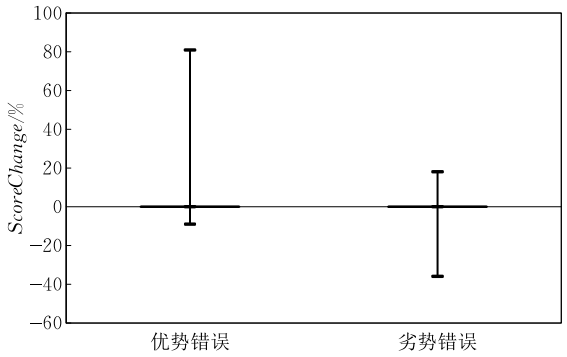
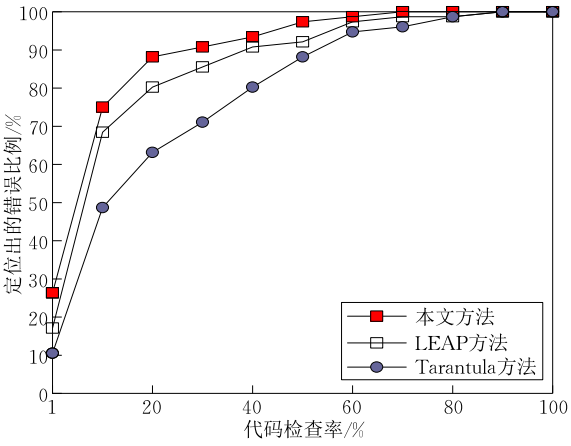
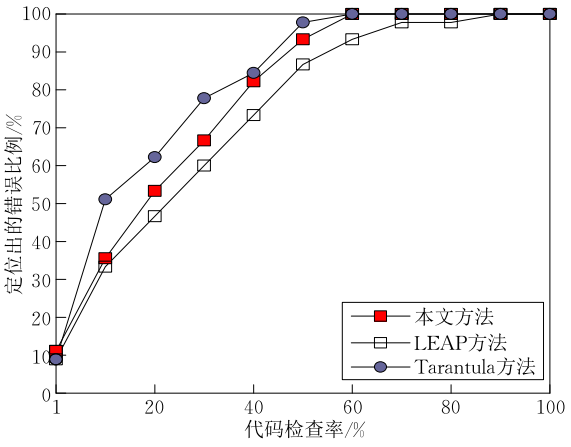


图 13 优势错误和劣势错误的 *ScoreChange* 分布



(a) 优势错误



(b) 劣势错误

图 14 优势错误和劣势错误的错误定位精度比较

方法也具有相同趋势,并且无论是对优势错误还是劣势错误,本文方法都比 LEAP 方法的错误定位精度高.

5.5 性能分析

表 3 给出了 3 种方法定位分析每个错误版本的平均时间. Tarantula 是基于语句覆盖的分析因此具有较高的时间效率,本文方法和 LEAP 方法均涉及到图的分析,两者时间开销相近.

表 3 时间分析

程序	定位每个版本的平均时间/s		
	Tarantula	LEAP	本文方法
print_tokens	2.3	14.2	15.2
print_tokens2	2.2	13.7	13.8
replace	2.3	14.4	15.3
schedule	1.9	12.3	12.4
schedule2	1.9	12.4	12.4
tcas	1.8	12.7	12.6
tot_info	2.2	13.5	13.3
flex	30.5	293.2	285.6

6 讨 论

6.1 本文方法的不足

根据本文方法与 Tarantula 方法对比实验的结果,可以得到以下结论:本文的错误定位方法对冗余代码、缺失代码和变量替换错误,以及会直接改变执行路径的错误的定位精度明显提高,而对不会直接改变执行路径的运算符变异、常量变异错误的定位精度的提高则不明显.

分析产生这种情况的原因,是由于本文方法会在成功的执行和失败的执行之间找到加权软件行为图差异最大的子图,即当程序的执行路径出现明显差异时,本文方法比较有效. 反之,因本文方法的特点不能被有效利用,而使得其错误定位的优势不明显.

文献[15]的研究表明,一条语句的错误可能会影响到程序状态,而随着程序的继续执行,被影响的程序状态可能会被进一步传播. 也就是说,程序的执行路径发生变化的位置,并不一定是真正导致错误的语句,而可能是其他语句的错误,通过程序状态不断传播,直到导致程序的执行路径发生改变.

也就是说,本文方法对于在错误语句附近就能够观察到程序执行路径发生改变的错误,一般都能具有比较好的定位精度. 而对于会将错误传播出去很远才发生执行路径变化的错误,往往只能定位到执行路径发生变化的位置,虽然该位置与实际的错

误语句也具有一定的相关性，但因其并不是真正产生错误的语句，从而导致错误定位的精度下降.

6.2　可改进之处

针对 5.1 节中本文方法的适应性问题，可以尝试捕获更多的程序状态，如谓词的真假等信息，而不仅仅是程序的执行路径. 在进行图挖掘时，可以综合考虑更多的信息或融合其他的错误定位算法，来提高对劣势错误的定位精度.

对于错误传播对错误定位精度的影响，单独的应用图挖掘算法可能难以解决这一问题，因此可以尝试分析程序的控制依赖或数据依赖，并依此对错误的传播进行分析，进而可能减少错误传播带来的影响.

7　讨　论

本文提出了一种新的基于加权软件行为图挖掘的错误定位方法. 与 Tarantula 方法和 LEAP 方法相比，本文方法具有更高的错误定位精度，而且更适合于定位冗余代码、缺失代码和变量替换错误，以及会直接改变执行路径的错误.

本文的主要贡献是：

(1) 提出了加权软件行为图的概念和构造方法，并将其用于错误定位. 与软件行为图相比，加权软件行为图使用语句执行概率作为边的权重，有效地利用了路径执行的统计信息，因此可以更好地分析与循环和递归等结构相关的软件错误.

(2) 提出一种基于分支限界搜索的加权软件行为图挖掘算法，识别成功和失败执行之间最有差异的子图来获得错误签名，不但可以有效定位错误位置，还能输出缺陷语句相关的执行路径，从而提供失效产生的上下文，有助于错误理解.

(3) 从理论和实验两个方面分析了基于加权软件行为图挖掘的错误定位方法的适应性，通过对错误进行分类，讨论了本文方法对不同错误类型的错误定位精度的影响.

参　考　文　献

[1] Cleve H, Zeller A. Locating causes of program failures//Proceedings of the 27th International Conference on Software Engineering. New York, USA, 2005: 342-451

[2] Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching//Proceedings of the 28th International Conference on Software Engineering. New York, USA, 2006: 272-281

[3] Jones J A, Harrold M J. Empirical evaluation of the Tarantula automatic fault-localization technique/Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. New York, USA, 2005: 273-282

[4] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering and Methodology, 2011, 20(3): 11

[5] Xie X, Chen T Y, Kuo F C, et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology, 2013, 22(4): 31

[6] Xuan J, Monperrus M. Learning to combine multiple ranking metrics for fault localization//Proceedings of the IEEE International Conference on Software Maintenance and Evolution. British Columbia, Canada, 2014: 191-200

[7] Baah G K, Podgurski A, Harrold M J. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Transactions on Software Engineering, 2010, 36(4): 528-545

[8] Su Xiao-Hong, Gong Dan-Dan, Wang Tian-Tian, Ma Pei-Jun. Automatic fault localization approach combining test case reduction and joint dependency probabilistic model. Journal of Software, 2014, 25(7): 1492-1504(in Chinese)
(苏小红，龚丹丹，王甜甜，马培军. 结合用例约简与联合依赖概率建模的错误定位. 软件学报，2013，25(7)：1492-1504)

[9] Gong D, Su X, Wang T, et al. State dependency probabilistic model for fault localization. Information and Software Technology, 2015, 57: 430-445

[10] Hsu H Y, Jones J A, Orso A. RAPID: Identifying bug signatures to support debugging activities//Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. New York, USA, 2008: 439-442

[11] Cheng H, Lo D, Zhou Y, et al. Identifying bug signatures using discriminative graph mining//Proceedings of the 18th International Symposium on Software Testing and Analysis. New York, USA, 2009: 141-152

[12] Yu Kai, Lin Meng-Xiang. Advances in automatic fault localization techniques. Chinese Journal of Computers, 2011, 34(8): 1411-1422(in Chinese)
(虞凯，林梦香. 自动化软件错误定位技术研究进展. 计算机学报，2011，34(8)：1411-1422)

[13] Fisher R A. The use of multiple measurements in taxonomic problems. Annals of Eugenics, 1936, 7(2): 179-188

[14] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria//Proceedings of the 16th International Conference on Software Engineering. New York, USA, 1994: 191-200

[15] Voas J M. PIE: A dynamic failure-based technique. IEEE Transactions on Software Engineering, 1992, 18(8): 717-727

SU Xiao-Hong, born in 1966, Ph.D., professor. Her research interests include software bugs detection, software refactoring, information fusing, target detecting and tracking.

WANG Tian-Tian, born in 1980, Ph.D., associate professor. Her research interests include program analysis, and software bugs detection.

YANG Shao-Jun, born in 1990, M.S. candidate. His research interests include fault localization, graph mining.

MA Pei-Jun, born in 1963, Ph.D., professor. His research interests include information fusing, software engineering, target detecting and tracking.

Background

Software debugging, which includes locating and correcting faulty program statements, is an important and expensive activity in the software development and maintenance.

To expedite debugging, researchers have proposed automatic fault localization techniques. However, the fault localization results are usually given in the form of suspicious statement sequence, which is hard for developers to understand the cause of failure.

To solve this problem, this paper proposed weighted software behavior graph to represent the path of passed execution and failed execution. It uses the execution frequency as weights, so that it can better handle loops, recursive and other structures than the LEAP approach. Then graph mining algorithm is performed on the weighted software behavior graphs to identify differences between passed and failed weighted software behavior graph as a bug signature. Our approach can not only localize the fault but also provide the failure context to facilitate fault comprehension. Test results showed that it has higher fault localization accuracy, especially in dealing with redundant code, missing codes, variable substitution errors, and errors will directly change the execution path.

This work is supported by the National Natural Science Foundation of China (Grant Nos. 61173021, 61202092), and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20112302120052).