基于校正因子的随机 TBFL 方法

王蓁蓁[1,2] 刘 嘉[3]

(金陵科技学院软件工程学院 南京 211169)[1] (江苏省软件测试工程实验室 南京 211169)[2]

(南京大学计算机软件新技术国家重点实验室 南京 210093)[3]

摘 要 运用测试集对程序错误语句定位的算法被统称为 TBFL(Testing Based Fault Localization)方法。目前通用算法一般都没有利用测试员、程序员关于测试用例和程序的先验知识,致使这些"资源"被浪费。随机 TBFL 方法是一类新型 TBFL 方法,其精神就是在随机理论的框架下,把这些先验知识(抽象为先验分布)和实际测试活动结合起来,从而更好地定位程序错误语句。事实上,随机 TBFL 算法可以看成这类算法的一般"模式",人们可以从这个一般框架里开发出不同的算法。文中方法就是将随机 TBFL 算法加以简化得到的,主要是从各个测试用例的具体测试活动着手,对程序变量 X 的先验概率加以校正,如果测试集里有 n 个用例,便可以得到程序变量 X 的 n 个校正值,将 n 个校正值效应迭加,并且标准化,即得到程序变量 X 的后验概率,用它作为寻找错误语句的向导。由于提出的简化算法是借助一个校正因子矩阵而得到的,因此所提算法被称为基于校正因子的随机 TBFL 方法。文中还提出了 3 个有关不同 TBFL 算法的比较标准,并依据它们在一些具体实例上的表现证实所提算法的有效性。

关键词 错误定位,软件测试,随机错误定位方法,校正因子

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/jsjkx.191100503C

Stochastic TBFL Approach Based on Calibration Factor

WANG Zhen-zhen[1,2] LIU Jia[3]

(School of Software Engineering, Jinling Institute of Technology, Nanjing 211169, China)[1]

(Software Testing Engineering Laboratory of Jiangsu Province, Nanjing 211169, China)[2]

(State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210093, China)[3]

Abstract Approaches for fault localization based on test suites are now collectively called TBFL (Testing Based Fault Localization). However, current algorithms have not taken advantages of the prior knowledge about test cases and program, so that they waste these valuable "resources". Literature [12] introduced a new kind of stochastic TBFL approach whose spirit is to combine the prior knowledge with actual testing activities under stochastic theory, so as to locate program faults. This algorithm may be regarded as a general pattern of this kind of approach, from which people can deve-lop various algorithms. The approach presented in this paper was simplifying the TBFL algorithm. It mainly revises the prior probability of program variable X from separate testing activity of each test case. If there are n test cases, n calibration factors can be obtained. These n calibration factors are then added and standardized, finally the posterior probability of the program is obtained. The approach proposed in this paper is called stochastic TBFL approach just because it depends on a calibration factor matrix. This paper presented three standards for comparing different TBFL approaches. Based on these standards, the improved approach is feasible for some instances.

Keywords Fault localization, Software test, Stochastic testing based fault localization, Calibration factor

1 引言

运用从软件测试中获得的信息自动确定错误位置是很重要的一种技术手段,有关方法可以统称为 TBFL(Testing-Based Fault Localization)方法[1-17]。

文献[3]讨论了 6 种涉及动态定位的 TBFL 方法,包括 Dicing 方法[1]、TARANTULA 方法[5-6]、Nearest Neighbor Queries 方法[10]、CT[2]、SoBER[9]和[7],发现它们都忽略了实施相似性的测试用例可能产生的问题。同样,文献[6]也指出,相似性的测试用例可能会损害 TBFL 方法的功效。为了解决该问题,文献[3]主要运用模糊集合理论,提出了 SAFL (Similarity-Aware Fault Localization)方法。在测试实践方法时,测试用例的相似性总是难以避免的,因此讨论类似 SAFL 方法和寻找更有成效的错误定位方法都是有价值的。

然而,测试实践中还存在这样的问题,即没有充分利用原程序和测试用例本身所包含的信息去辅助测试结果寻找错误

到稿日期:2018-10-16 返修日期:2018-12-20 本文受国家自然科学基金项目(61772014)资助。

王蓁蓁(1975—),女,博士,副教授,CCF 会员,主要研究方向为软件测试、人工智能,E-mail:wangzhenzhen@seu.edu.cn(通信作者);刘 嘉(1976—),男,博士,副教授,主要研究方向为软件测试,E-mail:liujia@nju.edu.cn。

根源，这不仅是一种“资源”浪费，而且有时对程序中隐蔽错误的揭露“无能为力”。为了解决这个问题，文献[12]提出了一种新的基于随机理论的 TBFL 方法。该方法对减少相似用例的伤害性也有成效。文献[12]认为软件虽然是由高度负责和有丰富专业知识的程序员编写的，但是由于软件的复杂性和“非物质性”，一些偶然因素所导致的错误是无法避免的。因此，文献[12]将整个待测程序看成一个随机变量，并把程序员或测试员在测试前关于程序中每个语句(或每个部件)出错的可能性的估计抽象为该随机变量的先验分布。同样设计测试用例的目的是想捕获程序错误，它们捕获错误的可能性也是随机现象，因而也视测试用例集为另一个随机变量，测试用例集的分布就是它们捕捉错误能力的抽象表示。在这些信息的基础上，利用每个测试用例的测试结果类型(失败或通过)及其覆盖语句的情况，对程序中每个语句的(先验)概率做综合性调整，调整后的概率被称为后验概率。最后根据该后验概率对错误语句进行排序，为程序员寻找错误提供向导。

随机 TBFL 算法的流程图如图 1 所示(对文献[12]中的图 3 稍作改进)。

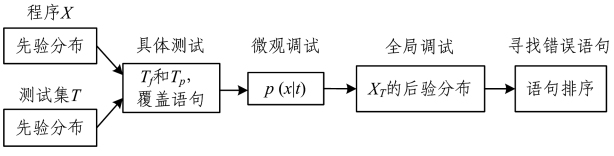

图 1 文献[12]的随机 TBFL 算法的流程图

Fig. 1 Flowchart of stochastic TBFL algorithm in ref. [12]

图 1 很好地阐述了文献[12]提出的随机 TBFL 算法的思想，算法的细节请参见该文献。文献[12]首先是通过条件概率 $p(\cdot|t)$，利用每个测试用例的功效，即固定用例 t，根据 t 的类型(失败或通过)和覆盖的语句，将程序 X 的先验分布单独地进行微观调整。然后就每一个语句 x，按测试集 T 的先验分布对各个用例得到的条件概率 $p(x|\cdot)$ 加权综合，由这种类型的全面调整而得到的概率分布，文献[12]称之为程序变量的后验分布，并记为 X_T 的后验分布。

文献[12]指出，X 的先验分布是根据对程序“样式”的分析得到的，测试集的(先验)分布与设计测试用例的类型、意图有关，它们分别与程序中语句的真实错误和测试用例具体实施的结果无关，并且这些先验知识即使粗糙，但只要大体上“正确”，当它们和测试实践活动产生的结果从微观和宏观两个层次上进行关联以后，就会对语句的错误定位有较大帮助。文献[12]在一些具体实例上把提出的随机 TBFL 算法和前述几个方法进行对比，证实了这一点。

本文受到减少样本空间的研究[13-14]的启发，发现文献[12]所提算法合理的原因是，测试用例 t 后，原样本空间就缩小了范围，于是条件概率 $p(\cdot|t)$ 是在一个由 t 用例产生的一个“范围较小”的子空间上讨论的。本文即在这一个关键点上立论。

在这个缩减了范围的子空间上，文献[12]提出的概率分布主要由因子 β 决定，β 因子的设立请参见文献[12]。β 因子决定的算法实质上是把程序先验分布和具体用例的测试效果相结合，它们与测试集的先验分布无关。在子空间上，本文考虑测试集先验分布，在一些合理假设下(其中有些是文献[12]中的隐蔽假设，例如独立性)，这个概率分布涉及测试功效、程序先验分布、测试集的先验分布 3 个方面，是它们的某种简单组合，而这种组合在文献[12]里是通过微观、全局两个层次上的运作达到的，计算比较复杂。最后本文把这些子空间“拼凑”在一起，得到程序的后验分布。本文提出的算法称为基于子空间概率分布的语句出错概率算法(后文有时简称新算法)，其思想如图 2 所示。

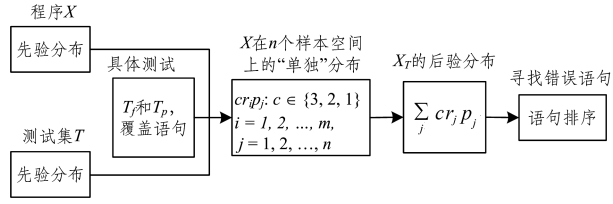

图 2 基于校正因子的随机 TBFL 方法的流程图

Fig. 2 Flowchart of stochastic TBFL based on calibration factor

本文第 2 节给出新算法的框架；第 3 节给出 TBFL 算法的评价标准；第 4 节进行实例分析；最后总结全文并展望未来。

2 算法框架

(1) 程序随机变量 X

用 $X=\{x_1,x_2,\cdots,x_m\}=\{x|x$ 是程序语句$\}$表示程序，它是语句的集合，其中语句 x_i 的下标 i 可按照程序的书写方式编码。假定程序是“精心”编码的，即由有责任心且有熟练技能的开发人员编写的。然而由于受到各种不可控制的因素的影响，程序仍可能发生错误。把程序错误归咎到语句层次上，认为它的每个语句出错是一种偶然现象。因此，视程序 X 为(出错)随机变量，用 $r_k=P(x_k$ 出错$)$表示语句 x_k 出错的概率。$r_k,k=1,2,\cdots,m$，是程序 X 随机变量的先验概率质量函数。可以根据开发人员的经验、历史资料分析各种语句类型通常犯错误的可能性，也就是说可以根据程序 X 的“样式”确定各个 r_k 的值。当然不同的人可能有不同的估计值，然而即使这些估计值粗糙、不精确，但只要它大体上反映了程序编写时的客观情况，那么它们对于语句错误定位算法就是有用的。如果缺少这方面的资料，在算法中可以按照统计学上的“同等无知”原则，令 $r_k=\frac{1}{m}$，$k=1,2,\cdots,m$，其中 m 是程序 X 的语句总数。

值得注意的是，对于语句出错可能性，人们的经验只能抽象为几个等级，令 S 是若干个等级集合，这时语句出错的估计序列为：$s_1,s_2,\cdots,s_m$，其中 s_i 是某一等级，即 $s_i\in S$。我们可以将它标准化为概率序列，即设 $a=s_1+s_2+\cdots+s_m$，令 $r_1=s_1/a,r_2=s_2/a,\cdots,r_m=s_m/a$，它们构成了概率序列。但是所提算法只是根据语句出错的可能性大小排序，因此不把它标准化，直接利用 $s_1,s_2,\cdots,s_m$ 也可。为了便于阐述，称未标准化的序列为准概率分布(有时省略“准”字)。

(2)测试集随机变量 T

用 $T=\{t_1,t_2,\cdots,t_n\}$ 表示测试用例集，其中 $t_j,j=1,2,\cdots,n$ 表示测试用例(有时简称为用例)。它们是用来对程序 X 进行测试的。设计测试用例的目的是捕获程序的错误，它们捕获错误的可能性也是随机现象，可以根据测试人员的经验和软件测试理论确定每个用例捕获错误的概率，用 $p_i=$

$P(t_i$ 发现错误$)$表示用例 t_i 能检测出错误的概率。同样地,若没有这方面的资料,不妨假设 $p_i=\frac{1}{n}$,其中 n 是测试集中测试用例的总数,这也是统计学中"同等无知"原则的应用。

值得注意的是,如果人们根据经验用若干个等级表示用例捕获错误能力,并令 K 为这些等级集合,那么这时用例捕获错误序列为:$k_1,k_2,\cdots,k_n$,其中 $k_i\in K$,则设 $a=k_1+k_2+\cdots+k_n$,令 $p_1=\frac{k_1}{a},p_2=\frac{k_2}{a},\cdots,p_n=\frac{k_n}{a}$,它们构成了概率序列。然而对于语句排序来说,即使不标准化,直接用 $k_1,k_2,\cdots,k_n$ 进行计算也可。同样地,未标准化的序列称为准概率分布(有时省略"准"字)。

(3)具体测试

用 T 测试 X 后,可以把 T 中用例分为两类:T_p 和 T_f。T_p 表示测试程序时没有发现错误的所有用例组成的子集合,T_f 表示测试程序时发现错误的所有测试用例组成的子集合。T_p 和 T_f 中的用例分别称为通过用例和失败用例。每个用例覆盖语句的具体信息可以从测试活动中提取出来。一般来说,从具体测试活动中可以得到上述两类信息。

(4)对应每个用例测试结果修正 X 的概率分布

测试集 T 里每一个用例都"独立"地提供有关程序语句出错的信息,因此可以对程序 X 的先验概率分布分别进行相应的修正。该修正涉及 X 原有概率、用例能力和用例类型及其覆盖 3 个方面。用矩阵 $\boldsymbol{F}$ 表示所有修正的信息:

$$\boldsymbol{F}=(t_{ij})_{n\times m}$$

其中:

$$t_{ij}=\begin{cases}c_1p_ir_j, & t_i\text{ 覆盖语句 }x_j,t_j\in T_f\\ c_2p_ir_j, & t_i\text{ 未覆盖语句 }x_j,t_j\in T_f\\ c_3p_ir_j, & t_i\text{ 覆盖语句 }x_j,t_j\in T_p\\ c_4p_ir_j, & t_i\text{ 未覆盖语句 }x_j,t_j\in T_p\end{cases}\tag{1}$$

其中,$i=1,2,\cdots,n,j=1,2,\cdots,m$。这里,$c_1,c_2,c_3,c_4$ 满足 $c_1>c_2\geqslant 0,c_4>c_3\geqslant 0,c_1>c_4$ 等不等式,它们也可能与 t_i 有关,即不同的用例有不同的 c_1,c_2,c_3,c_4 值。

$\boldsymbol{F}$ 中每一行代表基于一个用例修正后的程序 X 的准概率分布,即 $\forall i\in\{1,2,\cdots,n\}$,$X$ 的概率分布修正为:$t_{i1},t_{i2},\cdots,t_{im}$,它是原分布 $r_1,r_2,\cdots,r_m$ 关于第 i 个用例的校正。例如对于语句 x_j,原先估计的概率为 r_j,现在用因子 $c\in\{c_1,c_2,c_3,c_4\}$和因子 p_i 校正它为 cp_ir_j。用 p_i 校正 r_j 很容易理解,而系数 c 反映了人们的通常思维定势。如果 $t_i\in T_f$,那么相比未覆盖的语句,其覆盖的语句应对程序错误负较大责任,因此有 $c_1>c_2$;如果 $t_i\in T_p$,那么它未覆盖的语句比覆盖的语句犯错误的可能性要大,因此有 $c_4>c_3$;而且根据统计学上常用的原则,失败用例比通过用例更容易令人怀疑程序有错,因此有 $c_1>c_4$。

本文主要的目的是阐述新算法的思想,并不把与每一个用例有关的系数 c_1,c_2,c_3,c_4 的合理取值作为研究目标,因此为了使算法计算简单易行,对于所有用例,都令 $c_1=3,c_4=2,c_2=c_3=1$,并称之为"3-2-1"原则。于是 t_{ij} 的定义变为:

$$t_{ij}=\begin{cases}3p_ir_j, & t_i\text{ 覆盖语句 }x_j,t_i\in T_f\\ p_ir_j, & t_i\text{ 未覆盖语句 }x_j,t_i\in T_f\\ p_ir_j, & t_i\text{ 覆盖语句 }x_j,t_i\in T_p\\ 2p_ir_j, & t_i\text{ 未覆盖语句 }x_j,t_i\in T_p\end{cases}\tag{2}$$

其中,$i=1,2,\cdots,n,j=1,2,\cdots,m$

下文将用到矩阵 $\boldsymbol{F}$,它里面的元素 t_{ij} 皆由式(2)定义。为了计算简单,不把每一行标准化为真正概率分布,后文称 $\boldsymbol{F}$ 为概率校正矩阵。

(5)后验概率

$\forall j\in\{1,2,\cdots,m\}$,语句 x_j 的出错概率分别被每个用例校正,共有 n 个校正值,令:

$$f_j=\sum_{i=1}^{n}c^{ij}p_ir_j=(\sum_{i=1}^{n}c^{ij}p_i)r_j\tag{3}$$

其中,$c^{ij}\in\{3,2,1\}$,$j=1,2,\cdots,m$。

称 $f_1,f_2,\cdots,f_m$ 为程序变量 X 的(准)后验概率。

由式(3)可以看出,如果设 $\boldsymbol{I}=(1\ 1\ 1\cdots 1)$ 为 n 维元素都是单位 1 的向量,则后验概率组成的向量 $\boldsymbol{f}=(f_1,f_2,\cdots,f_m)$可以由下式计算得到(后文并不应用这个公式):

$$\boldsymbol{f}=\boldsymbol{I}\cdot\boldsymbol{F}$$

另外,由式(3)也可以看出,我们可以将矩阵 $\boldsymbol{F}$ 简化为矩阵 $\boldsymbol{F}'$,其中 $\boldsymbol{F}'$ 里的元素 t'_{ij} 由 $\boldsymbol{F}$ 里的元素 $t_{ij}=c^{ij}p_ir_j$(其中 $c^{ij}\in\{3,2,1\}$)省略因子 p_i 和 r_j 得到。这样,后验概率可以由式(4)计算:

$$\begin{aligned}&\boldsymbol{b}=(b_1b_2\cdots b_m)=\boldsymbol{P}\cdot\boldsymbol{F}'\\ &f_j=b_jr_j,\ j=1,2,\cdots,m\end{aligned}\tag{4}$$

其中,$\boldsymbol{P}=(p_1p_2\cdots p_n)$为测试集 T 的先验分布组成的向量。

$\boldsymbol{F}'$称为校正因子矩阵,向量 $\boldsymbol{b}$ 称为修正系数向量,其中元素 $b_j(j=1,2,\cdots,m)$为修正系数,于是后验概率 f_j 即由 b_j 修正先验概率 r_j 而得,即 $f_j=b_jr_j$。下面的计算采用上述简洁算法,引入矩阵 $\boldsymbol{F}$ 是为了更好地说明算法的思想。

(6)语句出错排序

根据 $f_i(i=1,2,\cdots,m)$的值对其从大到小排序,如果有若干个值相等,那么按它们相应的语句出现的次序排序(语句编号较小的排列在前),得到:

$$f_{i_1}\geqslant f_{i_2}\geqslant f_{i_3}\geqslant\cdots\geqslant f_{i_m}\tag{5}$$

把式(5)中具有相等值的 f_{i_j} 归为一类,然后把 f_{i_j} 换成对应的语句 x_{i_j},这样就把程序语句按出错可能性从大到小排列成若干个等级。假如有 k 个等级,则出错语句排列等级如表 1 所列。

表 1　出错语句排列等级

Table 1　Rank of suspicious statements

①	②	…	ⓚ
$x_{i_1}x_{i_1}\cdots x_{i_{t_2}}$	$x_{i_{t_1+1}}\cdots x_{i_{t_2}}$	…	$x_{i_{t_{k-1}+1}}\cdots x_{i_m}$

程序员可以按照上面次序寻找出错语句,一般地,每找到一个或若干个真实错误并改正以后,就重新用测试集 T(或者增、删一些用例)测试。假如仍有失败用例,则再按表 1 继续往下寻找新的错误语句,如此进行,直到用例全部通过为止。

(7)算法总结(符号同前一致)

1)设定程序变量 X 的先验分布向量:$\boldsymbol{r}=(r_1r_2\cdots r_m)$。

2)设定测试集 T 的先验分布向量:$\boldsymbol{P}=(p_1p_2\cdots p_n)$。

3)具体测试:求 T_f 和 T_p 以及每一个用例覆盖的语句。

4)校正因子矩阵 $\boldsymbol{F}'=(t'_{ij})_{n\times m}$,其中:

$$t'_{ij}=\begin{cases}3, & t_i\text{ 覆盖语句 }x_j,t_i\in T_f\\ 1, & t_i\text{ 未覆盖语句 }x_j,t_i\in T_f\\ 1, & t_i\text{ 覆盖语句 }x_j,t_i\in T_p\\ 2, & t_i\text{ 未覆盖语句 }x_j,t_i\in T_p\end{cases}\tag{6}$$

5)修正系数向量 $\boldsymbol{b}=\boldsymbol{P}\cdot\ \boldsymbol{F}'$即修正系数：

$$b_j=\sum_{i=1}^{n}p_i\cdot\ t'_{ij},j=1,2,\cdots,m \tag{7}$$

6)后验概率向量 $\boldsymbol{f}=(f_1,f_2,\cdots,f_m)$,其中

$$f_j=b_jr_j,j=1,2,\cdots,m \tag{8}$$

7)语句出错的可能性按从大到小进行等级排序。

3 TBFL 算法评价标准

假如某个 TBFL 算法已把程序语句出错可能性按等级排列成表 1 的形式,注意它也可看作语句出错可能性按从大到小相应的线性排序,如式(5)所示。

设程序 X 真实错误语句为 $x_{i_1},x_{i_2},\cdots,x_{i_s}$,下面定义 3 个评价标准：

$D_1=$ 真实错误语句出现在表 1 中的等级之和

$D_2=$ 真实错误语句出现在表 1 中的次序(即 x_{i_1} 次序为 $1,\cdots,x_{i_m}$ 次序为 m)之和

$D_3=$表 1 中等级的个数 (9)

如果有两个 *TBFL* 算法,可以分别计算它们相应的 D_1,D_2,D_3 的值,一般地,D_1 与 D_2 越小越好,D_3 越大越好。很明显 D_1,D_2 的值越小,而 D_3 越大,则表示该 TBFL 算法的辨别能力越强,区分语句的精度越高,因此相应的算法就比较优越。

4 实例分析

为了与 Dicing 方法、TARANTULA 方法、SAFL 方法以及文献[12]提出的方法(下面简称 Wang 方法)作比较,实验采用的实例与文献[12]中的实例一致。

本文主要使用文献[3]中给出的程序和测试用例。为了阅读方便,这里给出该程序和测试用例,如图 3 所示。

		Test suite 1				Test suite 2			
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Mid ( ) {									
int x, y, z, m;	statements	2,1,3	1,3,2	5,5,3	1,2,3	3,3,5	1,1,2	2,2,3	5,5,2
read ("Enter 3 numbers:", x, y, z);	x_1	●	●	●	●	●	●	●	●
m = x;	x_2	●	●	●	●	●	●	●	●
if (y < z)	x_3	●	●	●	●	●	●	●	●
if (x < y)	x_4	●			●	●	●	●	
m = y;	x_5				●				
else if (x < z)	x_6	●				●	●	●	
m = y;	x_7	●				●	●	●	
else	x_8		●	●					●
if (x > y)	x_9		●	●					●
m = y;	x_{10}								
else if (x > z)	x_{11}		●	●					●
m = x;	x_{12}			●					●
printf ("Middle number is:" m);	x_{13}	●	●	●	●	●	●	●	●
}		F	F	P	P	P	P	P	P

图 3 一个错误程序及其执行轨迹

Fig. 3 Faulty program and its execution traces

下文中称图 3 中的程序为程序 I,该程序的错误语句是 x_2(应为:$m=z$)和 x_7(应为:$m=x$)。

把程序 I 中的错误语句 x_2:"$m=x$" 改为正确语句 x_2:"$m=z$",其余语句保留不变,称该变体为程序 II,程序 II 中只有一个错误语句 $x_7(m=y)$。

将程序 I 的 $x_2(m=x)$改为 $x_2(m=z)$,x_{11}(else if ($x>z$))改为 x_{11}(else if ($x<z$)),其余语句不变,称该变体为程序 III。

将程序 I 的错误语句 $x_7(m=y)$改为正确语句 $x_7(m=x)$,其余语句不变,称该变体为程序 IV,它只有一个错误语句 x_2。

显然上述 4 个程序的真实错误语句并不相同,但它们的"样式"完全相同,因此估计每个语句出错的(先验)可能性是一样的,它们都是客观存在的。现在不管是哪个程序,我们都用 $X=\{x_1,x_2,\cdots,x_{13}\}$这一程序变量来表示,并且它的先验分布如式(10)所示：

$$r_2=4/20,r_5=r_7=r_{10}=r_{12}=2/20$$
$$r_k=1/20,k\neq2,5,7,10,12 \tag{10}$$

理论上,上述 4 个程序语句出错的情况是这个随机变量 X 的 4 个"具体实现"。

下文中称图 3 中的全部 8 个用例(即 Test suite1+Test suite2)组成的测试集为 T,其中前 4 个用例(即 Test suite1)组成的测试集为 T^*。T 为冗余测试集,T^* 为无冗余测试集。粗略地说,冗余指用例覆盖的语句几乎完全相同(详情请见文献[12])。

测试集随机变量 T 的先验分布为：

$$p_1=p_2=p_4=\frac{2}{11},p_3=p_5=p_6=p_7=p_8=\frac{1}{11} \tag{11}$$

测试集随机变量 $T*$ 的先验分布为：

$$p_1=p_2=p_4=\frac{2}{7},p_3=\frac{1}{7} \tag{12}$$

例 1 用 T,T^* 两个测试集来测试程序 I。

用 T 测试集测试程序 I,它们覆盖语句的情况如图 3 所示。

利用式(6),计算出校正因子矩阵 $\boldsymbol{F}'=(t'_{ij})_{8\times13}$,由 $\boldsymbol{b}=\boldsymbol{P}\cdot\boldsymbol{F}'$,即由式(7)所示的 $b_j=\sum_{i=1}^{8}t'_{ij}\,p_i,j=1,2,\cdots,13$,利用式(11)计算出修正系数,再由式(8),即 $f_j=b_jr_j,j=1,2,\cdots,13$,计算出后验分布。最后,根据 $f_i(i=1,2,\cdots,13)$的大小相应地把语句出错的可能性按等级排列,如表 2(其中的标号为等级标号)所列。

表 2 程序 I 在测试集 T 下的出错语句排序

Table 2 Rank of suspicious statements of program I under test suite T

①	②	③	④	⑤	⑥	⑦
x_2	x_7	x_{10}	x_5 x_{12}	$x_8x_9x_{11}$	x_1x_3 x_6 x_{13}	x_4

用测试集 $T*$ 测试程序 I,$T*$ 测试结果即为前面的 Test suite1 的测试结果。相应地,用式(12)给出的 $T*$ 分布和校正因子矩阵 $\boldsymbol{F}'$的前 4 行,通过式(7)计算出修正系数 $\boldsymbol{b}$,再通过式(8)计算出后验概率,最后按 $f_i(i=1,2,\cdots,13)$的大小相应地把语句出错可能性按等级排列,如表 3 所列。

表 3 程序 I 在测试集 T^* 下的出错语句排序

Table 3 Rank of suspicious statements of program I under test suite T^*

①	②	③	④	⑤	⑥	⑦	⑧	⑨
x_2	x_7	x_{10}	x_{12}	x_5	x_1x_3 x_{13}	x_6	$x_8x_9x_{11}$	x_4

现将本文算法(下面简称为新算法)与文献[12]中的算法(记为 Wang 方法)、Dicing 方法、TARANTULA 方法、SAFL

方法进行比较,除了本文算法的相关资料(来自表 2 和表 3)以外,其余算法的资料都来自于文献[12]。

关于可能出错语句的(等级)排序如下。

$$\text{Dicing}\begin{cases} T & x_8x_9x_{11}\ x_6x_7\ x_4 \quad \text{其余语句} \\ T^* & x_6x_7\ x_4x_8x_9x_{11} \quad \text{其余语句} \end{cases}$$

$$\text{TARANTULA}\begin{cases} T & x_8x_9x_{11}\ x_1x_2x_3x_6x_7x_{13}\ x_4 \quad \text{其余语句} \\ T^* & x_6x_7\ x_1x_2x_3x_4x_8x_9x_{11}x_{13} \quad \text{其余语句} \end{cases}$$

$$\text{SAFL}\begin{cases} T & x_6x_7\ x_8x_9x_{11}\ x_1x_2x_3x_4x_{13} \quad \text{其余语句} \\ T^* & x_6x_7\ x_8x_9x_{11}\ x_1x_2x_3x_4x_{13} \quad \text{其余语句} \end{cases}$$

$$\text{Wang}\begin{cases} T & x_2\ x_{10}\ x_7\ x_5x_{12}\ x_8x_9x_{11}\ x_6\ x_1x_3x_4x_{13} \\ T^* & x_2\ x_7\ x_{10}\ x_{12}\ x_5\ x_6\ x_1x_3x_8x_9x_{11}x_{13}\ x_4 \end{cases}$$

$$\text{新算法}\begin{cases} T & x_2\ x_7\ x_{10}\ x_5x_{12}\ x_8x_9x_{11}\ x_1x_3x_6x_{13}\ x_4 \\ T^* & x_2\ x_7\ x_{10}\ x_{12}\ x_5\ x_1x_3x_{13}\ x_6\ x_8x_9x_{11}\ x_4 \end{cases} \quad (13)$$

分析式(13)中的数据,正如文献[12]所述,无论测试集是否冗余,就这个具体例子而言,Wang 方法相比之前几种方法都较优。虽然它在 T 测试时把正确语句 x_{10} 排在错误语句 x_7 前,但是与 T^* 测试时"正确"的排序有些差异,似乎受到了冗余测试的伤害。同样,从式(13)中也可以看出,无论测试集是否有冗余,本文算法都比 Wang 方法要好,而且在 T 测试和 T^* 测试时,新算法并没有受到冗余测试的伤害,两个算法都把有错语句 x_2,x_7 排在最前两位,且新算法在精细度方面也与 Wang 方法相当。

例 2　用 T^* 测试集测试程序 II。

用测试集 T^* 测试程序 II 时,每个用例覆盖语句的情况与例 1 中一样,但是 t_2 在例 1 中是失败用例,而在本例中是通过用例,其他用例的结果类型未变。

利用上述资料,可以构造校正因子矩阵 $\boldsymbol{F}'$,修正系数 b_i ($i=1,2,\cdots,13$),以及(准)后验概率分布。下面根据出错可能性把程序语句从大到小排成等级,并将其与 Wang 方法进行对比,结果如表 4 所列。

表 4　程序 II 在测试集 T^* 下的语句排序及新算法与 Wang 方法的对比

Table 4　Rank of program II's suspicious statements under test suite T^* and comparison between propose method and Wang approach

Wang	x_2	x_7	x_{10}	x_{12}	x_5	x_6	x_4	$x_1x_3x_{13}$ $x_8x_9x_{11}$	
新算法	x_2	x_7	x_{10}	x_{12}	x_5	x_6	x_4	$x_1x_3\ x_{13}$	$x_8x_9x_{11}$

除了新算法把 Wang 方法中最后一个等级分为两个等级外,两种算法的排序完全一样,这说明两算法的 D_1 和 D_2 相等,且新算法比 Wang 算法更精细。

例 3　用测试集 $T*$ 测试程序 III。

用 T^* 测试程序 III,结果类型为:$T_f=\{t_1,t_2,t_3\}$,$T_p=\{t_4\}$。至于覆盖语句,t_1,t_4 覆盖的语句与它们在例 1 中覆盖的语句一样,而 t_2,t_3 覆盖的语句有所变动。按照前面的算法流程,得到程序的后验概率,根据后验概率大小把程序 III 中的语句按出错可能性大小排成等级,并将所提方法与 Wang 方法进行比较,结果如表 5 所列。

表 5　程序 III 在测试集 $T*$ 下的语句排序及新方法和 Wang 方法的对比

Table 5　Rank of program III's suspicious statements under test suite T^* and comparison between proposed method and Wang approach

Wang	x_2	x_7	x_{12}	x_{10}	$x_1x_3\ x_8x_9$ $x_{11}\ x_{13}$	x_6	x_5	x_4
新算法	x_2	x_7	x_{12}	x_{10}	$x_1x_3\ x_{13}$	$x_8x_9x_{11}$	x_5	x_6	x_4

根据文献[12]的分析,程序 III 有严重的逻辑错误,这是由语句 x_{11} 的错误选择导致的,具体表现在 x_2 语句的设置上,即程序流程中有的路径要求语句 x_2 为"$m=z$",有的路径要求语句 x_2 为"$m=x$"。因此程序 III 的错误语句最后可以归结为 x_2,x_7,x_{11},x_{12}。详细分析请见文献[12]。

下面评价在这个具有隐蔽逻辑错误的程序上两个算法的功效。Wang 方法的 D_1 为 9,D_2 为 15,D_3 为 7,而新算法的 D_1,D_2,D_3 分别为 10,16,8。从语句排列情况来看,两个算法只有细微差别,这反映在它们对应的 D_1,D_2,D_3 值上。虽然这些值略有不同,但却不是本质的。因为 x_2 是"设置语句",要求程序员有较高的编程技巧,但是一旦检查出 x_2 语句在设置上的"逻辑"困境,就可以发现是语句 x_{11} 的选择问题以及由此连带产生的语句 x_{12} 的错误,所以其实质上是要考查 x_2,x_7。然而就这种考查而言,两个算法的功效基本一样。如果就 x_2,x_7,x_{12} 的考查而言(因为考查了这 3 个语句,肯定会发现 x_{11} 有错),两个算法是一模一样的,如果考虑到新算法计算的简洁性,我们倾向于认为新算法较好。

例 4　用例全部通过的测试。

仍然考虑程序 I,它有两个错误语句 x_2 和 x_7,程序变量 X 的先验分布由式(10)表示。

现在用测试集 $T_o=\{o_1,o_2,o_3,o_4\}$ 对程序 I 进行测试,其中用例的设计为:$o_1=\{10,13,15\}$,$o_2=\{8,6,4\}$,$o_3=\{5,9,2\}$,$o_4=\{17,19,21\}$。T_o 的先验分布为均匀分布,即 $p_1=p_2=p_3=p_4=1/4$,也就是说每个用例捕获错误的可能性相同。

用 T_o 测试,可以发现它们都是通过用例。利用上述资料做类似计算,现把语句排序结果以及 Wang 方法的结果对比列出,如表 6 所列。

表 6　程序 I 在测试用例全部通过的情况下的出错语句排序及新方法和 Wang 方法的比较

Table 6　Rank of program I's statements when test cases are passed and comparison between proposed method and Wang approach

Wang	x_7	$x_{10}x_{12}$	x_2	x_5	x_6	x_{11}	$x_4x_8x_9$	$x_1x_3\ x_{13}$
新算法	x_7	$x_{10}x_{12}$	x_2	x_5	x_6	x_{11}	$x_4x_8x_9$	$x_1x_3\ x_{13}$

两种算法的(线性)排序一样,但 Wang 方法较精细些。

一般来说,测试员只能报告软件缺陷存在,却不能报告软件缺陷不存在。但是通用的测试在所有用例都通过时,对于软件缺陷都无法暗示缺陷存在与否。因此文献[12]与本文算法在这种情况下,至少在理论上是有价值的。

例 5　先验概率未知情况。

程序 I 是待测程序,程序变量 X 的先验分布未知,这时认为它服从均匀分布,即 $r_k=1/13$,$k=1,2,\cdots,13$。用测试集

T^* 进行测试，T^* 的先验分布也未知，这时也认为它服从均匀分布，即 $p_j=1/4, j=1,2,3,4$。

现将语句出错可能性按等级排列，如表 7 所列。

表 7 先验概率未知情况下的出错语句排序

Table 7 Rank of statements under unkonwn prior probability

	①	②	③	④
新方法	x_1x_2 x_3x_6 x_7 x_{13}	x_4 $x_8x_9x_{11}$	x_{10}	x_5 x_{12}

下面将上述排序和 Wang 方法在先验概率未知情况的排序以及式(14)中关于 Dicing、TARANTULA 方法、SAFL 方法的排序进行对比。注意后 3 个方法也可看成在缺乏程序和测试集的先验知识(即分布)上立论的，为了阅读方便，将其列于表 8 中。

表 8 先验概率未知情况下各种方法的比较

Table 8 Comparisons of various approaches under unknown prior probability

	①	②	③	④
Dicing 方法	x_6x_7	$x_4x_8x_9x_{11}$	其余语句	
TARANTULA	x_6x_7	$x_1x_2x_3x_4$ x_8 $x_9x_{11}x_{13}$	其余语句	
SAFL	x_6x_7	$x_8x_9x_{11}$	$x_1x_2x_3x_4$ x_{13}	其余语句
Wang	x_6x_7	$x_1x_2x_3x_4$ x_8x_9 $x_{10}x_{11}x_{13}$	x_5x_{12}	
新方法	x_1x_2 x_3 x_6 x_7 x_{13}	$x_4x_8x_9x_{11}$	x_{10}	x_5x_{12}

现在用 D_1, D_2, D_3 标准评价各个算法的性能(有错语句 x_2, x_7)：

Dicing：$D_1=1+3=4, D_2=2+8=10, D_3=3$；

TARANTULA：$D_1=1+2=3, D_2=2+4=6, D_3=3$；

SAFL：$D_1=1+3=4, D_2=2+7=9, D_3=4$；

Wang：$D_1=1+2=3, D_2=2+4=6, D_3=3$；

新算法：$D_1=1+1=2, D_2=2+5=7, D_3=4$；

从上述评价中可以看出，就这个具体例子而言，Wang 方法和 TARANTULA 方法相当。与这两种方法相比，由于新算法的 D_1 较低，而 D_2 较大，如果主要关注 D_1 和 D_2 标准，则可以认为这 3 个方法也是相当的，而且新算法优于 Dicing 方法和 SAFL 方法。因此在“最坏”(即没有挖掘出程序和测试集里的信息)的情况下，Wang 方法和新算法也至少与普通方法相当。

例 6 测试用例设计较好的情况。

在前面的实例中，对于程序 I 等的测试所用的测试集，无论是 T, T^* 或是 T_o 都设计得不好。现在根据软件测试理论，应该设计用例集，使得它们执行时能覆盖程序 I 等的每一条路径。为此引入新的测试集：

$T'=\{e_1, e_2, e_3, e_4, e_5, e_6\}$

其中，$e_1=\{6,7,9\}$，即输入 $x=6, y=7, z=9$，$e_2=\{8,7,9\}$，$e_3=\{10,7,9\}$，$e_4=\{11,10,7\}$，$e_5=\{9,10,7\}$，$e_6=\{8,10,9\}$。因为程序 I 和它的变体程序都是求 3 个输入的整数中的中位数，而 3 个数按“小、中、大”不同的方式排列，共有 6 种排列，所以如上设计的测试用例集在理论上可以通过程序 I 等的每条路径，它是比较好的用例设计。当然，对于现在讨论的程序来说，好的测试集至少应该保证输入的 3 个数中有若干个相等的用例，以及比较大的整数用例。关于后者，它并不是本质问题，至于前者，由于前面的测试集 T 和 T^* 都包含了这种用例，我们已经看到了算法对这些用例的适应程度，因此为了明确说明问题，设计了 T'。

下面用 T' 测试程序 I、程序 II 和程序 IV，无论用 T' 测试哪个程序，根据设计它的意图来看，每个用例捕获错误的可能性相同，即 $p_i=P(e_i$ 捕获到错误$)=1/6, i=1,2,\cdots,6$。注意 T' 的均匀分布是“客观”的先验分布，并不是由于“同等无知”得到的。

在 T' 测试下，新算法和文献[12]中算法对语句排序的结果对比如表 9 所列。

表 9 程序 I 在测试集 T' 下的语句可能出错的排序及新算法和 Wang 方法的比较

Table 9 Rank of program I's statements under test suite T' and comparison between proposed method and Wang approach

Wang	x_2	x_7	$x_5x_{10}x_{12}$	x_6	x_4	$x_1x_3x_{11}$ x_{13}	x_8x_9
新算法	x_2	x_7	x_5 $x_{10}x_{12}$	x_6	x_1x_3 x_4 x_{13}	x_{11}	x_8x_9

两种算法的 D_1, D_2, D_3 等值都相等，但新算法的计算简单些。

用 T' 测试程序 II，语句可能出错的排序如表 10 所列。

表 10 程序 II 在测试集 T' 下的语句可能出错的排序及新方法和 Wang 方法的比较

Table 10 Rank of program II's statements under test suite T' and comparison between proposed method and Wang approach

Wang	x_2	x_7	$x_5x_{10}x_{12}$	x_6	x_4	x_{11}	x_8x_9	$x_1x_3x_{13}$
新算法	x_2	x_7	$x_5x_{10}x_{12}$	x_6	x_4	x_{11}	$x_1x_3x_8x_9x_{13}$	

在这个实例中，由于 x_7 是错误语句，因此新算法比 Wang 方法差，而且其精细度也不如 Wang 方法。但是许多语句排序一致，因此差异不大。

用 T' 测试程序 IV，语句可能出错的排序如表 11 所列。

表 11 程序 IV 在测试集 T' 下的语句可能出错的排序及新算法和 Wang 方法的比较

Table 11 Rank of program IV's statements under test suite T' and comparison between proposed method and Wang approach

Wang	x_2	$x_5x_7x_{10}x_{12}$	x_6x_{11}	$x_4x_8x_9$	$x_1x_3x_{13}$
新算法	x_2	$x_5x_7x_{10}x_{12}$	x_6x_{11}	$x_1x_3x_4x_8x_9x_{13}$	

两个算法的 D_1, D_2 值相等，但 Wang 方法较精细，而新算法计算简单，因此两者相当。

结束语 就我们所知，文献[12]为 TBFL 方法引入了一个新类型，其精神是用随机理论考查软件的测试问题。本文受到文献[13-14]的启发，对文献[12]的算法机理做了直观、定性的分析，并在此基础上提出了一个比文献[12]算法更简单的算法，然后通过上面的一些实例，证实了新算法与 Wang 方法相当甚至在某些情况下更优。

实际上，文献[12]视程序变量 X 为在语句集上取值的离散型随机变量，X 的取值 x_k 的概率为语句 x_k 有错的概率，它们由有关人员用经验确定，并称之为程序变量 X 的先验分布。先验分布用大家熟知的符号表示：$r_k=P(x_k$ 有错$), k=1,2,\cdots,m$，其中 m 是程序语句总数。先验分布的确定依赖于

有关人员对程序样式的识别。待测程序的“样式”是一个客观存在,在软件世界里,它代表了一大类具有这种样式的“实际”程序,而现在待测的程序究竟是哪一个“实体”程序,我们不得而知,因此只有靠经验猜测它是哪一个“实体”,即该实体里面哪些语句有问题。这个猜测是用先验分布表示的,但是对先验分布的要求很低,只要 $0\leqslant r_k\leqslant 1,\sum_{k=1}^{m} r_k=1$ 即可,因此很不可靠。为了使估计更精确,软件界采取的一般做法是设计测试用例集进行测试,文献[12]就是通过测试活动中获得的信息来校正这个先验分布。一个用例的测试活动结果实际上是对由程序样式决定的庞杂的软件实体集进行缩减,也就是缩小猜测范围,因而在这个缩减的实体集中更容易猜测我们正在研究的程序的属性。该属性是用条件概率 $p(x|t)$表达的。文献[12]用 β 因子构造 $p(x|t)$,然后用 $p(t)\cdot p(x|t)$表示 (X,T)的联合发布,由 $\sum_t p(t),p(x|t)$表示 (X,T)的边缘分布 X_T,它就是程序变量 X 的后验概率,是测试活动对程序变量 X 的先验分布的总体校正,用它作为寻找错误真实语句的依据(上述符号的精确解释请参考文献[12])。

鉴于文献[12]算法的复杂性,我们直接在用例 t 的活动结果上对 X 的先验分布进行校正,该校正值是用例 t 的捕捉错误的能力、X 在语句 x 的概率以及用例 t 在语句 x 上的功效 c 三者之间的乘积组合,前文已对这种乘积组合的合理性做了常识性说明。如果测试集有 n 个用例,那么就有 n 个校正分布,将这 n 个校正效应迭加,就是程序变量 X 的(准)后验概率分布,标准化即为 X 的后验概率,无论标准与否,它们都可以作为寻找程序语句错误的向导。我们知道,直观和定性分析是人类思维的重要特征,也是人类思维的一种重要方式[18-19],因此上述推理是合理的。

由于从上述推理得到的简洁算法主要由正交矩阵 $\boldsymbol{F}'$ 所表征,因此将本文算法命名为基于校正因子的随机 TBFL 算法。文献[12]算法的机理不仅具有理论价值,而且对其进行深入分析还能开发新的算法,今后我们将继续深入研究,努力将这种类型的 TBFL 方法从理论和实践两个方面进行完善。实质上,基于校正因子的随机 TBFL 算法可视为一种原型方法,因此将来的工作是进一步细化该算法并把它推广到实际大规模复杂软件的错误定位问题上。

致谢　衷心感谢徐宝文教授、邢汉承教授、周毓明教授、陈振宇教授对本文工作的支持和指导。

参 考 文 献

[1] AGRAWAL H,HORGAN J R,LONDON S,et al. Fault Localization Using Execution Slices and Dataflow Tests[C]// IEEE International Symposium on Software Reliability Engineering. 1995:143-151.

[2] CLEVE H,ZELLER A. Locating Causes of Program Failures C]// Proceedings of the 27th International Conference on Software Engineering. 2005:342-351.

[3] HAO D,ZHANG L,PAN Y,et al. On Similarity-awareness in Testing-based Fault Localization [J]. Automated Software Engineering,2008,15(2):207-249.

[4] KYRIAZIS A,MATHIOUDAKIS K. Enhance of Fault Localization Using Probabilistic Fusion with Gas Path Analysis Algorithms [J]. Journal of Engineering for Gas Turbines and Power,2009,131(131):239-247.

[5] JONES J A,HARROLD M J,STASKO J. Visualization of Test Information to Assist Fault Localization[C]// Proceeding of the 24th International Conference on Software Engineering. 2002:467-477.

[6] JONES J A, HARROLD M J. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique[C] // Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. 2005:273-282.

[7] LIBLIT B,NAIK M,ZHENG A X,et al. Scalable Statistical Bug Isolation[C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2005:15-16.

[8] LEI Y,MAO X G,CHEN T Y,et al. Slice-Based Statistical Fault Localization [J]. Journal of Systems & Software,2014,89(2):51-62.

[9] LIU C,YAN X,FEI L,et al. SOBER:Statistical Model-based Bug Localization[C]// Proceedings of the 13the ACM SIGSOFT Symposium on Foundations of Software Engineering. 2005:286-295.

[10] RENIERIS M,REISS S P. Fault Localization with Nearest Neighbor Queries[C] // Proceedings of the 18th International Conference on Automated Software Engineering. 2003:30-39.

[11] ZELLER A. Isolating Cause-effect Chains from Computer Programs[C] // Proceedings of the 10th ACM SIGFOFT Symposium on Foundations of Software Engineering. 2002:1-10.

[12] WANG Z Z,XU B W,ZHOU Y M,et al. New Random Testing-based Fault Localization Approach [J]. Computer Science,2013,40(1):5-14.

[13] SINGH P K,GARG S,KAUR M,et al. Fault Localization in Software Testing Using Soft Computing Approaches [C]// 2017 4th International Conference on Signal Processing,Computing and Control (ISPCC). Solan,2017:627-631.

[14] WONG W E,GAO R,LI Y,et al. A Survey on Software Fault Localization [J]. IEEE Transactions on Software Engineering,2016,42(8):707-740.

[15] WONG W E,DEBROY V,GAO R,et al. The Dstar Method for Effective Software Fault Localization [J]. IEEE Transaction on Reliability,2014,63(1):290-308.

[16] STEIMANN F,FRENKEL M,ABREU R. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators[C] // Proceeding of the 2013 Int'l Symposium on Software Testing and Analysis. ACM Press,2013:314-324.

[17] ALEXANDRE P,RUI A,ANDRE R. A Dynamic Code Coverage Approach to Maximize Fault Localization Efficiency [J]. The Journal of Systmes & Software,2014,90:18-28.

[18] 王健吾. 数学思维方法引论[M]. 合肥:安徽教育出版社,1996.

[19] WANG Z Z. Construction of Knowledge Database and Naïve Reasoning of Naïve Fuzzy Description Logic [J]. Applied Science and Technology,2012,39(6):18-29.