等级平均随机TBFL方法

王蓁蓁

(金陵科技学院信息技术学院 南京 211169) (江苏省信息分析工程实验室 南京 211169)

摘 要 运用测试集对程序错误语句定位的算法,现在被统称为TBFL(testing based fault localization)方法。目前通行的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"白白浪费掉。文献[12]引入了一类新的随机TBFL方法,其精神就是在随机理论的框架下,把这些先验知识和实际测试活动结合起来,从而对程序错误语句更好地定位。文献[12]提出的算法可以看成是这种类型算法的一般"模式",人们可以根据这个一般性的模式开发出不同的算法。基于文献[13]的思想,对文献[12]中的算法做了改进。主要是根据测试结果,构造执行矩阵E和功效矩阵F两个工具,并结合测试集和程序先验知识,对程序语句出错可能性引入两个级别的排序,然后对这两个排序进行"平均",得到程序语句出错可能性的平均等级排序,它可以作为程序员改正程序错误的导向。还提出两个有关不同TBFL算法的比较标准,根据这两个标准,在一些具体实例上,将所提算法和其他一般方法以及文献[12]中的方法进行了对比,结果显示所提算法的效果令人满意。

关键词 错误定位,测试为基础的错误定位,随机错误定位方法

中图法分类号 TP311 **文献标识码** A

Average Scale Stochastic TBFL Approach

WANG Zhen-zhen

(School of Information Technology, Jinling Institute of Technology, Nanjing 211169, China)

(Information Analysis Engineering Laboratory of Jiangsu Province, Nanjing 211169, China)

Abstract Approaches for fault localization based on test suites are now collectively called TBFL (testing based fault localization). However, current algorithms have not taken advantage of the prior knowledge about test cases and program so that they waste these valuable "resources". [12] introduces 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 presented in [12] may be regarded as a general patter of this kind of approach, from which people can develop various algorithms. Based on the mind of [13], we performed an improvement of the algorithm in [12]. We mainly constructed two tools-the executive matrix E and the efficient matrix F-from the testing results. Then combined with the prior knowledge of test suite and program, the probability of statement being faulty is rated from two scales. Finally the two scales are "averaged". In this way we got the average rank of program statements about their probability of being faulty, which may help programmers correct program faults. Moreover, this paper presented two standards for comparing different TBFL approaches. And from the investigation of the two standards on some specific instances, the results of the approach presented in this paper are satisfactory.

Keywords Fault localization, Testing based fault localization, Random testing based fault localization

1 引言

运用从软件测试里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[1-13]。

文献[3]讨论了6种涉及动态定位的TBFL方法:Dicing方法(Agrawl et al. 1995)、TARANTULA方法(Jones et al. 2002;Jones and Harrold 2005)、Nearest Neighbor Queries方法(Renieris and Reiss 2003)、CT(Cleve and Zeller 2005)、SoBER(Liu et al. 2005)和Liblit05(liblit et al. 2005),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]主要是运用模糊集合理论提出了SAFL(similarity-aware fault localization)方法。在测试实践中,测试用例的相似性总是难以避免的,所以讨论类似SAFL方法和寻找更有成效的错误定位方法都是有价值的。

然而在测试实践中还存在这样的问题,就是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根源,这不仅是一种"资源浪费",而且有时对程序里隐蔽错误的揭露"无能为力"。为了解决这个问题,文献[12]提出

到稿日期:2013-03-01 返修日期:2013-05-10 本文受国家自然科学基金项目(61170071),金陵科技学院科研基金项目(jit-n-201305)资助。

王蓁蓁(1975—),女,博士后,副教授,CCF会员,主要研究领域为软件测试、人工智能,E-mail:wangzhenzhen@seu.edu.cn。

了一种新的基于随机理论的 TBFL 方法，该方法在减少相似用例的伤害性方面也有成效。

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，因此将整个待测程序看成是一个随机变量，把程序员或测试员在测试前关于程序里每个语句（或各个构件）出错的可能性的估计抽象为该随机变量的先验分布。同样就测试用例捕捉程序错误的能力而言，也视测试用例集为另一个随机变量，其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）和它们覆盖语句的情况，对程序里每个语句的（先验）概率作“综合性”调整，调整后的概率称为后验概率，最后就是根据这个后验概率对错误语句进行排序，为程序员寻找错误提供了导向。

随机 TBFL 算法可以用图 1（参考文献[12]图 3）表示（稍加修改）：

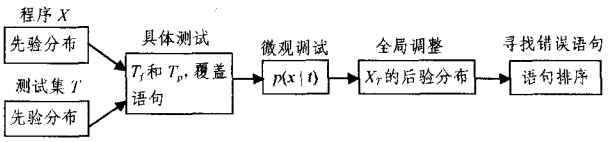

图 1　随机 TBFL 算法流程图

图 1 很好地阐述了文献[12]提出的随机 TBFL 算法的精神。算法的细节参看该文献。文献[12]首先通过条件概率 $p(\cdot|t)$ 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 的先验分布是根据程序“样式”的分析而得到的，测试集的（先验）分布与设计测试集用例的类型、意图有关，它们分别与程序里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，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的帮助。文献[12]在一些具体实例上，把它和前述几个方法进行了对比，证实了这一点。

在运用测试用例对程序覆盖技术进行错误语句定位时，文献[13]讨论了 3 种类型覆盖，它们分别是语句、分支和数据依赖等覆盖。文献[13]认为，不同类型程序应该用不同的类型覆盖技术效果较好。但是考虑到（例如）数据类型覆盖技术“代价”较高，所以文献[13]采用可以从分支类型推出的“近似”方法，进而将 3 种类型覆盖结合起来，并且划归到语句出错可能性上，这是一种较好的技术，特别是当不知道程序应该使用哪种覆盖技术较好时尤其如此。鉴于文献[12]算法计算时比较复杂，受到文献[13]的启发，对文献[12]提出的随机方法进行改良。

改良算法的关键思想是，不管测试用例是失败用例或是通过用例，它里面都包含了程序里语句出错的信息，这一思想是文献[12]首先提出的。利用这个观点，基于测试用例覆盖语句情况和它的结果类型，先后构造了两个矩阵 E,F。程序员用自己的知识（即程序的先验分布）和测试员的知识（即测试集捕获错误的能力）通过 E,F 两个工具对程序里语句错误可能性分布做出两个判断，它们分别对应于运用（类似于）分支、数据依赖类型覆盖技术和语句覆盖技术所做出的判断，进而就这两个判断，运用文献[13]类似做法，对它们加以“平均”，得到综合判断，最后将这个综合判断作为程序员寻找程序错误语句的导向。

上述具体思想可用图 2 表示。

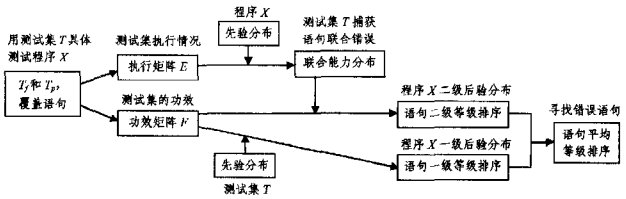

图 2　等级平均随机 TBFL 算法流程图

本文的算法称为等级平均随机 TBFL 算法，为了叙述简洁，下文有时简称它为新方法。

本文第 2 节给出新算法框架及其原理分析；第 3 节给出 TBFL 算法评价标准；第 4 节为实例分析；最后为结论和展望。

2　算法框架及其原理分析

2.1　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。然而由于各种各样不可控制的因素的影响，程序发生错误仍然难免。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“先验分布”。可以根据开发人员的经验、历史资料来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”确定诸 r_k 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，只要它们大体上反映了程序编写时的客观情况，便对于语句错误定位算法都是有用的。如果缺少这方面的资料，在算法里，可以按照统计学上“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。注意，今后我们同样地也用 X 指称它的分布。

值得注意的是，有时对语句出错可能性，人们的经验只能抽象为几个等级，比如 S 是若干个等级集合，语句出错估计序列为：$s_1,s_2,\cdots,s_m$，其中 s_i 是某一个等级，即 $s_i\in S$。我们可以将它们标准化为概率序列，即设 $a=s_1+s_2+\cdots+s_m$，则令 $r_1=\frac{s_1}{a},r_2=\frac{s_2}{a},\cdots,r_m=\frac{s_m}{a}$，它们构成概率序列。但是今后将会看到，我们算法的精神只是根据语句出错可能性大小排序，所以不把它标准化，直接利用 $s_1,s_2,\cdots,s_m$ 即可。今后遇到类似情况，我们可以将它们标准化为概率序列，如果不进行标准化，为了方便，称未标准化的序列为等级分布，例如上面的 $s_1,s_2,\cdots,s_m$ 便是语句出错可能性的等级分布。

2.2　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，以下有时简称为用例。它们是用来对程

序 X 进行测试的。设计测试用例的目的是想捕获程序的错误，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，若没有这方面的资料，在算法里不妨假设 $p_i=\frac{1}{n}$，其中 n 是测试集里用例的总数，这也是统计学中“同等无知”原则的应用。同样，测试集 T 的能力分布也可以是一种等级分布，没有必要把它们标准化为概率分布。

2.3 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也可从测试活动里提取出来。一般来说，从具体测试活动中可以得到上述两类信息。

2.4 执行矩阵 E 和测试集捕获语句关联错误能力分布

首先构造测试用例集 T 的拟执行矩阵 E，以下简称执行矩阵 E。它是 $n\times m$ 阶矩阵，行对应用例，列对应语句。每一行反映一个用例在程序上的执行情况。具体写为：

$$E=\begin{pmatrix} t_{11} & t_{12} & \cdots & t_{1m} \\ t_{21} & t_{22} & \cdots & t_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ t_{n1} & t_{n2} & \cdots & t_{nm} \end{pmatrix} \tag{1}$$

其中，$t_{ij}=\begin{cases} 2, & \text{若用例 } t_i \text{ 是失败用例且覆盖语句 } x_j \\ 1, & \text{若用例 } t_i \text{ 是通过用例但未覆盖语句 } x_j \\ 0, & \text{其他情况} \end{cases}$

令

$$\overline{P}'=E\cdot X',\overline{P}=(\overline{p}_1\overline{p}_2\cdots\overline{p}_n) \tag{2}$$

其中，$X'=(r_1,r_2,\cdots,r_m)'$ 表示 $\{r_k\}$ 构成的列向量，撇号“′”表示向量转置，下面遇到类似记法，意义相同，不再赘述。

以下称 $\overline{P}$ 中元素组成的序列 $\{\overline{p}_i\}$ 为测试集 T 捕获语句联合出错的可能性的等级分布，简称为 T 的联合能力分布。同样为了简洁，有时也用 $\overline{P}$ 表示这个分布。

直观上，当 t_i 是失败用例，则它覆盖的语句关联在一起出错的可能性用它们各自出错概率之和来表示是很自然的，用“2”加权是为了强调相对于通过用例来说，该用例捕获语句组合出错的能力较强。需要说明的是，这里也可以用适当大于 1 的常数 c 加权，不过为了计算简单，选择了 $c=2$。当 t_i 是通过用例时，则对由程序语句之间关联引起程序错误的怀疑自然就转移到它未覆盖的语句上，虽然这些未覆盖的语句也可能会由于与若干个覆盖语句相互作用而导致程序出错，但是做出“它们可能要负主要责任”这个定性判断是合理的，所以该用例捕获语句关联导致程序出错的可能性就用它未覆盖语句出错概率之和来估计。正因为我们对测试集 T 的执行情况做了上述处理，所以称它为拟执行矩阵，只不过为了简洁，省略了“拟”字。

于是 $\overline{P}$ 是测试集 T 捕获程序语句之间关联潜在引起程序出错的能力等级分布，它是类似于文献[13]所讨论的运用分支覆盖和数据覆盖技术对程序出错可能性进行模糊估计的方法。

2.5 功效矩阵 F 和程序后验两级分布

首先构造测试集 T 的功效矩阵 F。

测试集 T 里每一个用例(无论它失败与否)都“独立”提供了有关程序语句出错的信息。现在我们用“等级”表示“出错”可能性大小，并把这些信息抽象为测试集 T 的功效矩阵 F：

$$F=\begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1m} \\ f_{21} & f_{22} & \cdots & f_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ f_{n1} & f_{n2} & \cdots & f_{nm} \end{pmatrix} \tag{3}$$

其中，$f_{ij}=\begin{cases} \text{若 } t_i\in T_f \begin{cases} \text{若 } t_i \text{ 覆盖语句 } x_j, & \text{则为 } c_1 \\ \text{若 } t_i \text{ 未覆盖语句 } x_j, & \text{则为 } c_2 \end{cases} \\ \text{若 } t_i\in T_p \begin{cases} \text{若 } t_i \text{ 覆盖语句 } x_j, & \text{则为 } c_3 \\ \text{若 } t_i \text{ 未覆盖语句 } x_j, & \text{则为 } c_4 \end{cases} \end{cases}$，

$i=1,2,\cdots,n,j=1,2,\cdots,m$。这里 c_1,c_2,c_3,c_4 表示用例 t_i 在语句 x_j 上的功效，它依赖于 t_i 的类型和对 x_j 的覆盖与否。因为设计 F 的目的是挖掘用例里隐含的每个语句对程序错误应负的责任，所以要求不等式 $c_1>c_2\geqslant 0,c_4>c_3\geqslant 0,c_1>c_4$ 成立。我们在引言里说过，遵循文献[12]的精神，不管用例失败与否，它都包含了程序出错的信息。如果 $t_i\in T_f$，则它覆盖的语句比未覆盖的语句应对程序错误负较大责任，所以 $c_1>c_2$，如果 $t_i\in T_p$，则它未覆盖的语句比它覆盖的语句犯错误的可能性要大，所以 $c_4>c_3$，而且根据统计学上常用的原则，失败用例比通过用例更令人对程序有错产生怀疑，所以 $c_1>c_4$。

本文主要目的是阐述新算法的思想，关于 f_{ij} 定义里的常数 c_1,c_2,c_3,c_4 怎样更合理取值不作为研究目标，所以为了使提供的算法计算简单易行，令 $c_1=3,c_4=2,c_2=c_3=1$。并称这一取值原则为“3-2-1”原则，于是 f_{ij} 的定义变为：

$$f_{ij}=\begin{cases} \text{若 } t_i\in T_f \begin{cases} \text{若 } t_i \text{ 覆盖语句 } x_j, & \text{则为 } 3 \\ \text{若 } t_i \text{ 未覆盖语句 } x_j, & \text{则为 } 1 \end{cases} \\ \text{若 } t_i\in T_p \begin{cases} \text{若 } t_i \text{ 覆盖语句 } x_j, & \text{则为 } 1 \\ \text{若 } t_i \text{ 未覆盖语句 } x_j, & \text{则为 } 2 \end{cases} \end{cases}$$

$i=1,2,\cdots,n,j=1,2,\cdots,m$ (4)

下面用到矩阵 F 时，其中元素 f_{ij} 皆由上式定义。

令

$$b=P\cdot F\quad b=(b_1b_2\cdots b_m) \tag{5}$$

其中，$P=(p_1p_2\cdots p_n)$ 为测试集 T 的先验分布，F 是功效矩阵。称 b 为程序 X 语句出错可能性后验第一级等级分布，简称 b 为程序 X 的后验一级等级分布。

令

$$h=\overline{P}\cdot F\quad h=(h_1h_2\cdots h_m) \tag{6}$$

其中，$\overline{P}$ 是由式(2)定义的测试集 T 的联合能力分布，F 是功效矩阵。

称 h 为程序 X 语句出错可能性后验第二级等级分布，简称 h 为程序 X 的后验二级分布。

直观上，功效矩阵 F 每一行代表从相应的测试用例的角度考察程序时，语句出错可能性应该服从的等级分布。因为有 n 个测试用例，所以有 n 个等级分布，即对程序里每一个语句都有 n 个判断。现在为了计算，每一个语句出错可能性必须对这些判断加以综合。然而就综合工作来说，又有偏重于语句本身和偏重于语句之间的关联两种做法。对于前者，我们用测试集 T 先验分布 P 加权估计，因为它仅仅牵涉到测试活动产生的功效矩阵 F，$P\cdot F$ 相当于用语句覆盖技术定位程序错误方法。对于后者，我们用测试集 T 的联合能力分布 $\overline{P}$

加权估计，因为 $\bar{P}\cdot F$ 的计算牵涉到测试活动产生的执行矩阵和功效矩阵，所以计算结果表示每个语句对程序语句之间关联引起的错误问题应“承担的责任”，$\bar{P}\cdot F$ 相当于用分支覆盖技术和数据依赖技术化归到语句出错定位方法。

2.6 两级等级排序和等级平均排序

·语句出错一级等级排序

根据程序后验一级分布 $b_i, i=1,2,\cdots,m$ 的值，从大到小排序，如果有若干个值相等，则按它们相应的语句出现的次序（语句编号较小的排列在前）排序：

$$b_{i_1}\geqslant b_{i_2}\geqslant b_{i_3}\cdots\geqslant b_{i_m}$$

把上述排列中具有相等值的 b_{i_j} 归为一类，然后把 b_{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。假如是 k 个等级，则写成表 1 的格式。

表 1 语句出错一级等级排序

①	②	…	(k)
$x_{i_1}\ x_{i_2}\cdots x_{i_{l_1}}$	$x_{i_{l_1}+1}\cdots x_{i_{l_2}}$	…	$x_{i_{l_{k-1}}+1}\cdots x_{i_m}$

称表 1 为程序语句出错可能性第一级等级排序，简称为语句一级等级排序。

·语句出错二级等级排序

根据程序后验二级分布 $h_i, i=1,2,\cdots,m$ 的值，从大到小排序，如果有若干个值相等，则按它们相应的语句出现的次序（语句编号较小的排列在前）排序：

$$h_{i_1}\geqslant h_{i_2}\geqslant h_{i_3}\cdots\geqslant h_{i_m}$$

把上式中具有相等值的 h_{i_j} 归为一类，然后把 h_{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。假如是 q 个等级，则写成表 2 的格式。

表 2 语句出错二级等级排序

①	②	…	(q)
$x_{i_1}\ x_{i_2}\cdots x_{i_{l_1}}$	$x_{i_{l_1}+1}\cdots x_{i_{l_2}}$	…	$x_{i_{l_{q-1}}+1}\cdots x_{i_m}$

称表 2 为程序语句出错可能性第二级等级排序，简称为语句二级等级排序。

·语句出错平均等级排序

对程序里每一个语句，计算它们在表 1 和表 2 中的等级的平均值，得到程序 X 语句平均等级分布，即：

$$S=\{s_1,s_2,\cdots,s_m\} \tag{7}$$

式中，$s_i=1/2$（x_i 在表 1 中的等级 $+x_i$ 在表 2 中的等级），$i=1,2,\cdots,m$。

根据 $s_i, i=1,2,\cdots,m$ 的值，从小到大排序，如果有若干个值相等，则按它们相应的语句出现的次序（语句编号较小的排列在前）排序：

$$s_{i_1}\leqslant s_{i_2}\leqslant s_{i_3}\cdots\leqslant s_{i_m}$$

把上式中具有相等值的 s_{i_j} 归为一类，然后把 s_{i_j} 换成对应的语句 x_{i_j}，这样就把程序语句出错可能性按平均等级从小到大排列成若干个等级。假如是 a 个等级，则写成表 3 的格式。

表 3 语句出错平均等级排序

①	②	…	(a)
$x_{i_1}\ x_{i_2}\cdots x_{i_{l_1}}$	$x_{i_{l_1}+1}\cdots x_{i_{l_2}}$	…	$x_{i_{l_{a-1}}+1}\cdots x_{i_m}$

称表 3 为程序语句出错可能性平均等级排序，简称为语句平均等级排序。

程序员可以按照表 3 顺序寻找出错语句，正因为它是程序员纠正错误的导向，有时也直接称它是程序语句出错排序。一般地，每找到一个或若干个真实错误并改正以后，就重新用原测试集 T（或者增、删一些用例）测试，假如仍有失败用例，再按表 3 继续往下寻找新的错误语句，如此进行下去，直到用例全部通过为止。

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排列成如表 3 所列的等级，设程序 X 真实错误语句为 $x_{i_1},x_{i_2},\cdots,x_{i_s}$，定义评价标准 D,G 如下：

D=真实错误语句出现在表 3 中的等级之和

G=表 3 中等级的个数 (8)

若有两个 TBFL 算法，可以分别计算它们相应的 D,G 之值，一般地，D 越小越好，G 越大越好。前者是因为大部分纠正错误活动是按表 3 等级顺次进行的，后者是因为 G 越大，表示算法的精度越高，精度高的算法不会使程序在过于庞大的一个等级中浪费太多精力。

4 实例分析

为了和 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，我们采用的实例和文献[12]中的实例一致。

本文主要用文献[12]中图 1（该图来源于文献[3]）里给出的程序和测试用例。为了阅读方便，把它复制于下，在本文中标记为图 3。

		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 A faulty program and its execution traces

在下面的讨论中，称图 3 中的程序为程序 I，该程序的错误语句是 x_2（x_2 应为 $m=z$）和 x_7（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，称这个变体为程序 II，程序 II 中有一个错误语句 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，称这样的变体为程序 III。

虽然上述 3 个程序的真实错误语句不尽相同，但它们“样式”完全相同，所以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 $X=\{x_1,x_2,\cdots,x_{13}\}$ 这个程序变量表示它，并且它的先验分布为：

$$r_2=\frac{4}{20},r_5=r_7=r_{10}=r_{12}=\frac{2}{20}$$
$$r_k=\frac{1}{20},k\neq 2,5,7,10,12 \tag{9}$$

这个先验分布是文献[12]中采用的，关于它的详情可参看该文献。在理论上，上述3个程序语句的出错情况是这个随机变量 X 的3个“具体实现”。

在下面的讨论中，称图3里的全部8个用例(即 Test suitel+Test suite2)组成的测试集为 T，其中前4个用例(即 Test suitel)组成的测试集为 T^*。沿用文献[12]中的术语，T 为冗余测试集，T^* 为无冗余测试集。粗略地说，在文献[12]中，术语冗余指用例覆盖的语句至少几乎是完全相同的。详情请见文献[12]。

测试集随机变量 T 的先验分布为：

$$p_1=p_2=p_4=\frac{2}{11}$$
$$p_3=p_5=p_6=p_7=p_8=\frac{1}{11} \tag{10}$$

测试集随机变量 T^* 的先验分布为：

$$p_1=p_2=p_4=\frac{2}{7}$$
$$p_3=\frac{1}{7} \tag{11}$$

这些分布都是文献[12]采用的，详情也请看文献[12]。

例1　用 T,T^* 两测试集测试程序 I

用 T 测试集测试程序 I，结果是：$T_p=\{t_3,t_4,t_5,t_6,t_7,t_8\}$，$T_f=\{t_1,t_2\}$，它们覆盖语句情况可从图3中看出。

利用式(1)、式(4)分别算出执行矩阵 E 和功效矩阵 F。为了清楚，两个矩阵里的列用语句 x_i 标出，中间的虚线是为了区分 Test suitel 和 Test suite2。

$$E=\begin{array}{c}\begin{matrix}x_1&x_2&x_3&x_4&x_5&x_6&x_7&x_8&x_9&x_{10}&x_{11}&x_{12}&x_{13}\end{matrix}\\\left(\begin{array}{ccccccccccccc}2&2&2&2&0&2&2&0&0&0&0&0&2\\2&2&2&0&0&0&0&2&2&0&2&0&2\\0&0&0&1&1&1&1&0&0&1&0&0&0\\0&0&0&0&0&1&1&1&1&1&1&1&0\\\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots\\0&0&0&0&1&0&0&1&1&1&1&1&0\\0&0&0&0&1&0&0&1&1&1&1&1&0\\0&0&0&0&1&0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&0&0&1&0&0&0\end{array}\right)\end{array} \tag{12}$$

$$F=\begin{array}{c}\begin{matrix}x_1&x_2&x_3&x_4&x_5&x_6&x_7&x_8&x_9&x_{10}&x_{11}&x_{12}&x_{13}\end{matrix}\\\left(\begin{array}{ccccccccccccc}3&3&3&3&1&3&3&1&1&1&1&1&3\\3&3&3&1&1&1&1&3&3&1&3&1&3\\1&1&1&2&2&2&2&1&1&2&1&1&1\\1&1&1&1&1&2&2&2&2&2&2&2&1\\\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots&\cdots\\1&1&1&1&2&1&1&2&2&2&2&2&1\\1&1&1&1&2&1&1&2&2&2&2&2&1\\1&1&1&1&2&1&1&2&2&2&2&2&1\\1&1&1&2&2&2&2&1&1&2&1&1&1\end{array}\right)\end{array} \tag{13}$$

利用式(12)和式(9)，由式(2) $\overline{P}'=E\cdot X'$ 计算测试集 T 的联合能力分布(写成行向量形式)如下：

$$\overline{P}=\left(\frac{22}{20}\ \frac{20}{20}\ \frac{8}{20}\ \frac{10}{20}\ \frac{9}{20}\ \frac{9}{20}\ \frac{9}{20}\ \frac{8}{20}\right) \tag{14}$$

利用式(10)和式(13)，由式(5)$b=P\cdot F$ 计算程序后验一级分布如下(其中因子 $N=1/11$)：

$b_1=19N,b_2=19N,b_3=19N,b_4=17N,$

$b_5=16N,b_6=19N,b_7=19N,b_8=20N,$

$b_9=20N,b_{10}=18N,b_{11}=20N,b_{12}=16N,b_{13}=19N$

把 b_i“改为”相应的 x_i，根据这个分布可得程序语句一级等级排序(按值从大到小排列，并把相等值归为一类而得)，如表4所列。

表4　程序 I 在测试集 T 下的语句一级等级排序

①	②	③	④	⑤
$x_8\ x_9\ x_{11}$	$x_1\ x_2\ x_3\ x_6\ x_7\ x_{13}$	x_{10}	x_4	$x_5\ x_{12}$

利用式(13)和式(14)，由式(6)$h=\overline{P}\cdot F$ 计算程序后验二级分布如下(其中因子 $N=1/20$)：

$h_1=179N,h_2=179N,h_3=179N,h_4=155N,$

$h_5=138N,h_6=165N,h_7=165N,h_8=172N,$

$h_9=172N,h_{10}=148N,h_{11}=172N,h_{12}=132N,h_{13}=179N$

把 h_i“改为”相应的 x_i，根据这个分布可得程序语句二级等级排序(按值从大到小排列，并把相等值归为一类而得)，如表5所列。

表5　程序 I 在测试集 T 下的语句二级等级排序

①	②	③	④	⑤	⑥	⑦
$x_1\ x_2\ x_3\ x_{13}$	$x_8\ x_9\ x_{11}$	$x_6\ x_7$	x_4	x_{10}	x_5	x_{12}

利用表4、表5，由式(7)计算程序语句平均等级分布如下：

$s_1=1.5,s_2=1.5,s_3=1.5,s_4=4,s_5=5.5,s_6=2.5,s_7=2.5,s_8=1.5,s_9=1.5,s_{10}=4,s_{11}=1.5,s_{12}=6,s_{13}=1.5$

把 s_i“改为”相应的 x_i，根据这个分布可得程序语句平均等级排序(按从小到大排列，并把相等值归为一类而得)，如表6所列。

表6　程序 I 在测试集 T 下的语句平均等级排序

①	②	③	④	⑤
$x_1\ x_2\ x_3\ x_8\ x_9\ x_{11}\ x_{13}$	$x_6\ x_7$	$x_4\ x_{10}$	x_5	x_{12}

如果不进行另外测试，一般地，程序员便按照表6给出的线索寻找错误语句。

用测试集 T^* 测试程序 I，T^* 测试结果即为前面的 Test suitel 的测试结果。因此 T^* 测试对程序 I 的执行矩阵和功效矩阵分别由式(12)，式(13)中前4行(虚线以上部分)组成。这时 X 的先验分布仍由式(9)给出，但 T^* 的先验分布由式(11)给出。由这些资料，用上面类似方法计算，最后得到用测试集 T^* 测试程序 I 的语句平均等级排序如表7所列。

表7　程序 I 在测试集 T^* 下的语句平均等级排序

①	②	③	④	⑤	⑥
$x_1\ x_2\ x_3\ x_{13}$	$x_6\ x_7$	$x_4\ x_8\ x_9\ x_{11}$	x_{10}	x_{12}	x_5

如果不进行另外测试，一般地，程序员便按照表7给出的线索寻找错误语句。

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)以及 Dicing 方法、TARANTULA 方法、SAFL 方法进行比较，除了本文算法资料(来自表6和表7)以外，其余算法的资料都摘自文献[12]。

关于可能出错语句的(等级)排序：

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

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

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

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

$$\text{新算法}\begin{cases}T, & x_1x_2x_3x_8x_9x_{11}x_{13}\ x_6x_7\ x_4x_{10}\ x_5\ x_{12}\\ T^*, & x_1x_2x_3x_{13}\ x_6x_7\ x_4x_8x_9x_{11}\ x_{10}\ x_{12}\ x_5\end{cases}\tag{15}$$

分析式(15)中数据，正如文献[12]指出，无论测试集是否冗余，就这个具体例子而言，Wang 方法比前几种方法都较优。它在 T 测试时，把正确语句 x_{10} 排在错误语句 x_7 前，与 T^* 测试时的“正确”的排序有些差异，表明该方法也稍许受到冗余测试的影响。现在，利用式(15)中的数据，用式(8)给出的标准衡量，把文本算法与 Wang 方法进行比较如下：

在测试集 T 下：

Wang：$D=1+3=4, G=7$

新算法：$D=1+2=3, G=5$

在测试 T^* 下：

Wang：$D=1+2=3, G=8$

新算法：$D=1+2=3, G=6$

由 D 值看出，新算法功效稍好一些，但是由 G 值看出，Wang 方法更精细些。然而在这个具体例子中，就两个算法寻找错误语句而言，精度问题并不是主要问题。例如新算法在 T^* 测试下，把错误语句 x_2 放在第一等级“$x_1x_2x_3x_{13}$”里，显然 x_1(是输入语句)，x_{13}(打印语句)不需要检查，x_3(是较早的分支条件语句)也无需花费多少时间检查，因而重点是考察 x_2，它恰好是错误语句。同样在 T^* 下，对于错误语句 x_7，本文算法把它和 x_6 一起放在第二等级“x_6x_7”里，也是因为 x_6(是较早的分支条件语句)无需花费多少时间就可以检查完毕，因而重点也是考察 x_7。从上述的情况来看，本文算法与 Wang 方法相当，由于 Wang 方法计算繁杂，而本文算法简单易行，因此牺牲一些精度换来计算简洁，还是值得的。

例 2　用 T^* 测试集测试程序 II

程序变量 X 和测试集变量 T^* 的先验分布同前，分别由式(9)、式(11)表示。用测试集 T^* 测试程序 II，每个用例覆盖语句的情况和前例中一样，没有变化，但 t_2 在前例中是失败用例，在本例中是通过用例，其他用例的结果类型未变，即在本例中 $T_f=\{t_1\}$；$T_p=\{t_2,t_3,t_4\}$。利用这些资料，进行类似计算，计算过程省略，现在只把得到的语句平均等级排序写出，并与 Wang 方法对比，如表 8 所列。

表 8　程序 II 在测试集 T^* 下语句平均等级排序及和 Wang 方法对比

Wang	x_2	x_7	x_{10}	x_{12}	x_5	x_6	x_4	$x_1x_3x_8x_9x_{11}x_{13}$
新算法	x_6x_7	x_4	$x_1x_2x_3x_{10}x_{13}$	x_{12}	x_5	$x_8x_9x_{11}$		

新算法精度不如 Wang 方法，但是考虑到在本例中精度并不是主要问题(理由和前例类似)，新算法把有错语句 x_7 排在第一等级，显然它比 Wang 方法要好些。

例 3　用测试集 T^* 测试程序 III

程序变量 X 和测试集变量 T^* 的分布分别由式(9)、式(11)表示。用 T^* 测试程序 III，结果类型为：$T_f=\{t_1,t_2,t_3\}$，$T_p=\{t_4\}$。至于覆盖语句方面，t_1，t_4 覆盖的语句和它们在例 1 中覆盖的语句情况一样，t_2，t_3 覆盖语句情况有所变动，t_2 覆盖 $x_1x_2x_3x_8x_9x_{11}x_{12}x_{13}$；$t_3$ 覆盖 $x_1x_2x_3x_8x_9x_{11}x_{13}$。

下面的计算方法同前，故省略，我们只列出语句出错平均等级排序并与 Wang 方法比较，如表 9 所列。

表 9　程序 III 在测试集 T^* 下语句平均等级排序及和 Wang 方法对比

Wang	x_2	x_7 x_{12}	x_{10}	$x_1x_3x_8x_9x_{11}x_{13}$	x_6	x_5	x_4
新算法	x_1x_2 x_3 x_{13}	$x_8x_9x_{11}$	x_{12}	x_6 x_7	x_4	x_{10}	x_5

根据文献[12]的分析，程序 III 有严重的逻辑错误，它是语句 x_{11} 的错误选择所致，具体表现在 x_2 语句的设置上，即程序流程里有的路径要求语句 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}。详细分析见文献[12]。

在这个具有隐蔽逻辑错误的程序上两个算法的功效评价如下：

Wang：$D=1+2+4+2=9, G=7$

新算法：$D=1+4+2+3=10, G=7$

两个算法精度相同，但是在语句出错可能性等级排序上，新算法稍逊一些。虽然如此，但这不是重大差异。因为 x_2 是个“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，所以实质上是要考察 x_2,x_7。现在新算法把 x_2,x_{11},x_{12} 连续排在第一，第二，第三等级，它们一气呵成，并与排在第四等级的错误语句 x_7 分离，因为 x_7 错误“独立于”x_2 的错误，所以这个分离有助于寻找错误语句活动的进行。在这个意义上，新算法可能还优于 Wang 方法，原因是 Wang 方法把 x_7 的检查放在 x_2 之后，可能切断 x_2 与 x_{11},x_{12} 等语句的联系。

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,x_7，程序变量 X 的先验分布由式(9)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，发现它们都是通过用例。它们覆盖语句情况是：o_1 覆盖 6 个语句($x_1,x_2,x_3,x_4,x_5,x_{13}$)；$o_2$ 覆盖 7 个语句($x_1,x_2,x_3,x_8,x_9,x_{10},x_{13}$)；$o_3$ 覆盖 8 个语句($x_1,x_2,x_3,x_8,x_9,x_{11},x_{12},x_{13}$)，$o_4$ 覆盖情况和 o_1 相同。

利用上述资料做类似计算，这里省略，只列出语句出错可能性平均等级排序并与 Wang 方法的结果对比，如表 10 所列。

表 10　程序 I 在测试用例全部通过情况下语句平均等级排序及和 Wang 方法对比

Wang	x_7	$x_{10}x_{12}$	x_2	x_5	x_6	x_{11}	x_4 x_8x_9	$x_1x_3x_{13}$
新算法	x_6x_7	$x_{11}x_{12}$	x_{10}	x_8x_9	x_4x_5	x_1 x_2 x_3 x_{13}		

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。通用的测试在所有用例都通过时，对于软件缺陷都“无话可说”。现在本文算法把真实错误语句 x_7 放在

第一等级，说明至少在理论上，它还是有价值的。可惜的是无论用 D 或是 G 作为标准，本文算法都不如 Wang 方法。

例 5　先验概率未知情况

取程序 I，程序变量 X 的先验分布未知，这时认为 X 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，其先验分布也未知，这时认为 T^* 服从均匀分布，即 $p_j=1/4, j=1,2,3,4$。用测试集 T^* 进行测试，其测试结果和例 1 中一样。现将语句出错平均等级进行排序如表 11 所列(中间计算过程省略)。

表 11　程序和测试集先验概率未知情况下程序 I 出错语句平均等级排序

	①	②	③	④	⑤	⑥	⑦
新算法	x_1 x_2 x_3 x_{13}	x_6 x_7	x_8 x_9 x_{11}	x_4	x_{10}	x_{12}	x_5

下面将上述排序与 Wang 方法在先验概率未知情况的排序以及式(15)里关于 Dicing 方法、TARANTULA 方法、SAFL 方法的排序进行对比。注意后 3 个方法也可看成是在缺乏程序和测试集先验知识上立论的，为了阅读方便，也把它们重新复制于下，如表 12 所列。

表 12　先验概率未知情况下各种方法比较

	①	②	③	④	⑤	⑥	⑦
Dicing 方法	x_6 x_7	x_4 x_8 x_9 x_{11}	其余语句				
TARANTULA 方法	x_6 x_7	x_1 x_2 x_3 x_4 x_8 x_9 x_{11} x_{13}	其余语句				
SAFL 方法	x_6 x_7	x_8 x_9 x_{11}	x_1 x_2 x_3 x_4 x_{13}	其余语句			
Wang 方法	x_6 x_7	x_1 x_2 x_3 x_4 x_8 x_9 x_{10} x_{11} x_{13}	x_5 x_{12}				
新算法	x_1 x_2 x_3 x_{13}	x_6 x_7	x_8 x_9 x_{11}	x_4	x_{10}	x_{12}	x_5

现在用 D,G 标准评价它们(有错语句是 x_2, x_7)

Dicing：$D=1+3=4, G=3$

TARANTULA：$D=1+2=3, G=3$

SAFL：$D=1+3=4, G=4$

Wang：$D=1+2=3, G=3$

新算法：$D=1+2=3, G=7$

用 D,G 标准衡量，就这个具体例子而言，Wang 方法和 TARANTULA 方法相当，它们的 D 值都优于 Dicing 方法和 SAFL 方法，只是 G 值稍逊于 SAFL 方法。本文算法与 Wang 方法相比，D 值一样，而本文算法的 G 值远大于 Wang 方法的 G 值，也比 SAFL 方法高得多，所以总地看来本文算法在这个具体例子上较优。这一点很重要，它说明在“最坏”(即没有挖掘出程序和测试集里的信息)情况下，本文算法和一般算法相比，即使不是较好，也至少相当。

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察软件测试问题。主要是基于文献[13]的思想，本文对文献[12]的算法做了一点改进。本文的贡献是，从测试活动的具体结果里，构造了两个工具，即执行矩阵 E 和功效矩阵 F，它们不仅计算简单，而且通过它们结合测试集和程序的先验知识可以分别近似地挖掘出程序里语句“单独”错误和语句之间的关联错误，并且把程序里语句出错的可能性做了两个级别的排序处理，在某种意义上，它们相当于目前流行的软件测试里运用的语句覆盖、分支覆盖和数据依赖覆盖技术对错误语句做的处理。而且我们在程序的两个级别的排序上做了平均处理，用这种平均等级进行排序，就几个具体实例，和一些流行方法以及文献[12]里的算法(用 D,G 标准)进行对比，效果也是相当令人满意的。

本文提出的算法只是对文献[12]引入的随机 TBFL 方法的一种改进，今后，我们将进一步开发新的随机 TBFL 方法，并努力将它们应用到实际测试活动中。

参 考 文 献

[1] Agrawal H, Horgan J, London S, et al. Fault location using execution slices and dataflow tests[C]// IEEE 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(5): 51601-51609

[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] Schach S R. Object-oriented classical software engineering[M]. Beijing: China Machine Press, 2007: 490-193

[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] 王蔡蓁，徐宝文，周毓明，等. 一种随机 TBFL 方法[J]. 计算机科学，2013，40(1)：5-14

[13] Santelices R, Jones J A, Yu Yan-bing, et al. Lightweight fault-localization using multiple coverage types. Software Engineering, 2009. ICSE 2009[C]// IEEE 31st International Conference on. IEEE, 2009: 56-66

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